LNAI 4483

Chitta Baral
Gerhard Brewka
John Schlipf (Eds.)

Logic Programming and Nonmonotonic Reasoning

9th International Conference, LPNMR 2007
Tempe, AZ, USA, May 2007
Proceedings

Springer

Lecture Notes in Artificial Intelligence 4483

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Chitta Baral   Gerhard Brewka
John Schlipf (Eds.)

Logic Programming and Nonmonotonic Reasoning

9th International Conference, LPNMR 2007
Tempe, AZ, USA, May 15-17, 2007
Proceedings

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Chitta Baral
Arizona State University, Department of Computer Science and Engineering
Box 875406, Tempe, AZ 85287-5406, USA
E-mail: chitta@asu.edu

Gerhard Brewka
University of Leipzig, Department of Computer Science
Postfach 100920, 04009 Leipzig, Germany
E-mail: brewka@informatik.uni-leipzig.de

John Schlipf
University of Cincinnati, Department of Computer Science
892 Rhodes Hall, Cincinnati, OH 45221-0030, USA
E-mail: schlipf@ececs.uc.edu

Library of Congress Control Number: 2007925046

CR Subject Classification (1998): I.2.3-4, I.2, F.4.1, D.1.6

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN	0302-9743
ISBN-10	3-540-72199-1 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-72199-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12056606 06/3180 5 4 3 2 1 0

Preface

These are the proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). LPNMR is a forum for exchanging ideas on declarative logic programming, nonmonotonic reasoning, and knowledge representation. LPNMR encompasses theoretical studies, design and implementation of logic-based programming languages and database systems, and development of experimental systems.

LPNMR 2007 was held in Tempe Arizona, USA, May 15–17, 2007, with workshops on May 14. Previous conferences were organized in Washington D.C., USA (1991), Lisbon, Portugal (1993), Lexington Kentucky, USA (1995), Schloß Dagstuhl, Germany (1997), El Paso Texas, USA (1999), Vienna, Austria (2001), Fort Lauderdale Florida, USA (2004), and Diamante, Italy (2005).

The conference included invited talks by Nicola Leone and Jorge Lobo, 18 technical papers, 7 system descriptions, and 5 posters. Keynote speaker was Jack Minker who gave a talk entitled

Reminiscences on Logic Programming and Nonmonotonic Reasoning and Future Directions

There was again an answer set programming competition (Torsten Schaub and Mirek Truszczynski, organizers), whose results are reported in these proceedings. Prior to the conference there were three workshops: Software Engineering and Answer Set Programming (Marina de Vos and Torsten Schaub, organizers), Correspondence and Equivalence for Nonmonotonic Theories (David Pearce), and Argumentation and Nonmonotonic Reasoning (Paolo Torroni and Guillermo Simari).

The organizers of LPNMR 2007 thank the Program Committee and all reviewers for their commitment and hard work on the program. Special thanks for support are due to Arizona State University and to its School of Computing and Informatics.

LPNMR 2007 was dedicated to the memory of Marco Cadoli, who died November 21, 2006. He was a professor at the University of Rome La Sapienza and a long-time leader in the LPNMR research areas, as well as a Program Committee member for this conference. He was a brilliant researcher, whose work ranged from nonmonotonic logic and knowledge compilation to databases, software development tools, and to computational complexity. Our community has lost an excellent scientist and a wonderful person.

February 2007

Chitta Baral
Gerhard Brewka
John Schlipf

Conference Organization

Program Chairs

Gerhard Brewka   John Schlipf

Local Chair

Chitta Baral

Competition Chairs

Torsten Schaub   Mirosław Truszczyński

Program Committee

José Alferes
Chitta Baral
Marco Cadoli
James Delgrande
Marc Denecker
Jürgen Dix
Thomas Eiter
Wolfgang Faber
Paolo Ferraris
Norman Foo
Michael Gelfond
Enrico Giunchiglia
Katsumi Inoue
Tomi Janhunen
Tony Kakas
Nicola Leone
Vladimir Lifschitz
Fangzhen Lin
Jorge Lobo
Ilkka Niemelä
David Pearce
Chiaki Sakama
Torsten Schaub
Tran Cao Son
Evgenia Ternovska
Hans Tompits
Mirosław Truszczyński
Dirk Vermeir
Marina de Vos
Yan Zhang

Additional Reviewers

Ofer Arieli
Marcello Balduccini
Federico Banti
Pedro Cabalar
Francesco Calimeri
Michael Fink
Alfredo Gabaldon
Martin Gebser
Gianluigi Greco
Aaron Hunter
Matthias Knorr
Ninghui Li
Marco Maratea
Maarten Marien
Veena Mellarkod
Thomas Meyer
David Mitchell
Johannes Oetsch
Magdalena Ortiz
Luigi Palopoli
Axel Polleres
Enrico Pontelli
Davy Van Nieuwenborgh
Joost Vennekens
Richard Watson
Johan Wittocx
Stefan Woltran

Table of Contents

Logic Programming and Nonmonotonic Reasoning: From Theory to Systems and Applications

Nicola Leone

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
`leone@mat.unical.it`

Abstract. LPNMR is based on very solid theoretical foundations. After nearly twenty years of research, LPNMR languages are expressively rich, and their semantic and computational properties are well understood today. Moreover, in the last decade the LPNMR community has focused also on systems implementation, and, after the pioneering work carried out in DLV and Smodels, many efficient LPNMR systems are now available.

The main challenge of LPNMR now concerns applications. Two key questions are:

- Is LPNMR technology mature enough for the development of industrial applications?
- Can LPNMR be attractive also for the market?

In this talk, we will address the above questions. We will report also some feedback that we have got ”from the field”, in the collaboration with a Calabrian spin-off company, which is developing three Knowledge Management products based on the DLV system, in a joint venture with a US company. We will present our vision on the future development of LPNMR, pointing out promising application areas where LPNMR can be profitably exploited, and specifying the key theoretical and practical issues to be addressed in order to enhance the applicability of LPNMR.

Policy-Based Computing: From Systems and Applications to Theory

Jorge Lobo

IBM T. J. Watson Research Center
jlobo@us.ibm.com

Abstract. The need for a more autonomous management of distributed systems and networks has driven research and industry to look for management frameworks that go beyond the direct manipulation of network devices and systems. One approach towards this aim is to build policy-based management systems. Policy-based computing refers to a software paradigm developed around the concept of building autonomous systems that provide system administrators and decision makers with interfaces that let them set general guiding principles and policies to govern the behavior and interactions of the managed systems. Although many of the tasks are still carried out manually and *ad hoc*, instances of limited policy-based systems can be found in areas such as Internet service management, privacy, security and access management, management of quality of service and service level agreements in networks.

Policies can be specified at many levels of abstraction, from natural language specifications to more elementary condition-action rule specifications. From these specifications policy systems need to come up with implementations. Some of these implementations can be done automatically, others require manual steps. In some cases policies impose legal commitments and systems should be able to demonstrate compliance. There are also situations in which policies are in conflict with each other and a system cannot implement them simultaneously without providing methods for conflict resolution. In this presentation I will review a few policy systems, applications and specification languages. Then I will provide a more formal characterization of policies and their computational model. I will show a simple policy language in the style of the action description language $\mathcal{A}$. I will discuss current solutions to policy conflicts, discuss the problem of policy refinement, i.e. transformations from high level specifications to lower level specifications, current approaches to refinement and provide a partial formalization of the general problem. I will discuss limitations of current systems and directions of research.

The First Answer Set Programming System Competition

Martin Gebser[1], Lengning Liu[2], Gayathri Namasivayam[2], André Neumann[1], Torsten Schaub[1,⋆], and Mirosław Truszczyński[2]

[1] Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany
{gebser,aneumann,torsten}@cs.uni-potsdam.de

[2] Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA
{gayathri,lliu1,mirek}@cs.uky.edu

Abstract. This paper gives a summary of the *First Answer Set Programming System Competition* that was held in conjunction with the *Ninth International Conference on Logic Programming and Nonmonotonic Reasoning*. The aims of the competition were twofold: first, to collect challenging benchmark problems, and second, to provide a platform to assess a broad variety of Answer Set Programming systems. The competition was inspired by similar events in neighboring fields, where regular benchmarking has been a major factor behind improvements in the developed systems and their ability to address practical applications.

1 Introduction

Answer Set Programming (ASP) is an area of knowledge representation concerned with logic-based languages for modeling computational problems in terms of constraints [1,2,3,4]. Its origins are in logic programming [5,6] and nonmonotonic reasoning [7,8]. The two areas merged when Gelfond and Lifschitz proposed the *answer set semantics* for logic programs (also known as the *stable model semantics*) [9,10]. On the one hand, the answer set semantics provided what is now commonly viewed to be the correct treatment of the negation connective in logic programs. On the other hand, with the answer set semantics, logic programming turned out to be a special case of Reiter's default logic [8], with answer sets corresponding to default extensions [11,12].

Answer Set Programming was born when researchers proposed a new paradigm for modeling application domains and problems with logic programs under the answer set semantics: a problem is modeled by a program so that answer sets of the program directly correspond to solutions of the problem [1,2]. At about the same time, first software systems to compute answer sets of logic programs were developed: *dlv* [13] and *lparse/smodels* [14]. They demonstrated that the answer set programming paradigm has a potential to be the basis for practical declarative computing.

These two software systems, their descendants, and essentially all other ASP systems that have been developed and implemented so far contain two major components. The first of them, a *grounder*, grounds an input program, that is, produces its propositional

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and IIIS at Griffith University, Brisbane, Australia.

equivalent. The second one, a *solver*, accepts the ground program and actually computes its answer sets (which happen to be the answer sets of the original program).

The emergence of practical software for computing answer sets has been a major impetus behind the rapid growth of ASP in the past decade. Believing that the ultimate success of ASP depends on the continued advances in the performance of ASP software, the organizers of the *Ninth International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'07) asked us to design and run a contest for ASP software systems. It was more than fitting, given that the first two ASP systems, *dlv* and *lparse/smodels*, were introduced exactly a decade ago at the *Fourth International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'97). We agreed, of course, convinced that as in the case of propositional SATisfiability, where solver competitions have been run for years in conjunction with SAT conferences, this initiative will stimulate research on and development of ASP software, and will bring about dramatic improvements in its capabilities.

In this paper, we report on the project — the *First Answer Set Programming System Competition* — conducted as part of LPNMR'07. When designing it, we built on our experiences from running preliminary versions of this competition at two Dagstuhl meetings on ASP in 2002 and 2005 [15]. We were also inspired by the approach of and the framework developed for SAT competitions [16], along with the related competitions in solving Quantified Boolean Formulas and Pseudo-Boolean constraints.

The First Answer Set Programming System Competition was run *prior* to the LPNMR'07 conference. The results are summarized in Section 6 and can be found in full detail at [17]. The competition was run on the *Asparagus* platform [18], relying on benchmarks stored there before the competition as well as on many new ones submitted by the members of the ASP community (cf. Section 4).

The paper is organized as follows. In the next section, we explain the format of the competition. Section 3 provides a brief overview of the Asparagus platform. In Section 4 and 5, we survey the benchmark problems and the competitors that took part in the competition. The competition results are announced in Section 6. Finally, we discuss our experiences and outline potential future improvements.

2 Format

The competition was run in three different categories:

MGS (Modeling, Grounding, Solving) In this category, benchmarks consist of a problem statement, a set of instances (specified in terms of ground facts), and the names of the predicates and their arguments to be used by programmers to encode solutions. The overall performance of software (including both the grounding of input programs and the solving of their ground instantiations) is measured. Success in this category depends on the quality of the input program modeling a problem (the problem encoding), the efficiency of a grounder, and the speed of a solver.

SCore (Solver, Core language) Benchmarks in this category are ground programs in the format common to *dlv* [19] and *lparse* [20]. In particular, aggregates are not allowed. Instances are classified further into two subgroups: *normal* (SCore) and *disjunctive* (SCore$^{\vee}$). The time needed by *solvers* to compute answer sets of ground programs is measured. Thus, this category is concerned *only* with the performance of solvers on ground programs in the basic logic programming syntax.

SLparse (Solver, Lparse language) Benchmarks in this category are ground programs in the lparse format *with* aggregates allowed. The performance of solvers on ground programs in lparse format is measured. This category is similar to SCore in that it focuses entirely on solvers. Unlike SCore, though, it assesses the ability of solvers to take advantage of and efficiently process aggregates.

Only decision problems were considered in this first edition of the competition. Thus, every solver had to indicate whether a benchmark instance has an answer set (SAT) or not (UNSAT). Moreover, for instances that are SAT, solvers were required to output a certificate of the existence of an answer set (that is, an answer set itself). This certificate was used to check the correctness of a solution.

The output of solvers had to conform to the following formats (in `typewriter` font):

SAT: `Answer Set: atom1 atom2 ... atomN`
The output is one line containing the keywords '`Answer Set:`' and the names of the atoms in the answer set. Each atom's name is preceded by a single space. Spaces must not occur within atom names.

UNSAT: `No Answer Set`
The output is one line containing the keywords '`No Answer Set`'.

The competition imposed on each software system (grounder plus solver) the same time and space limits. The number of instances solved within the allocated time and space was used as the **primary** measure of the performance of a software system. Average running time is only used as a tie breaker.

3 Platform

The competition was run on the *Asparagus* platform [18], providing a web-based benchmarking environment for ASP. The principal goals of Asparagus are (1) to provide an infrastructure for accumulating challenging benchmarks and (2) to facilitate executing ASP software systems under comparable conditions, guaranteeing reproducible and reliable performance results. Asparagus is a continuously running benchmarking environment that combines several internal machines running the benchmarks with an external server for the remaining functionalities including interaction and storage. The internal machines are clones having a modified Linux kernel to guarantee a strict limitation of time and memory resources. This is important in view of controlling heterogeneous ASP solvers that run in multiple processes (e.g., by invoking a stand-alone SAT solver). A more detailed description of the Asparagus platform can be found in [15].

4 Benchmarks

The benchmarks for the competition were collected on the Asparagus platform. For all competition categories (MGS, SCore, and SLparse), we asked for submissions of non-ground problem encodings and ground problem instances in separate files.

To add a problem to the MGS category, the author of the problem had to provide

– a textual problem description that also specified the names and arguments of input and output predicates and
– a set of problem instances in terms of ground facts (using only input predicates).

The submission of a problem encoding was optional for benchmark problems submitted to the MGS category. In most cases, however, the authors provided it, too. In all remaining cases, the competition team provided an encoding. From all benchmark classes already stored on Asparagus or submitted for the competition, we selected several

Table 1. Benchmarks submitted by the ASP community

Benchmark Class	#Instances	Contributors	M	C	L
15-Puzzle	15	Lengning Liu and Mirosław Truszczyński	×	–	×
15-Puzzle	11	Asparagus	–	×	–
Blocked N-Queens	40	Gayathri Namasivayam and Mirosław Truszczyński	×	–	×
Blocked N-Queens	400	Asparagus	–	×	–
Bounded Spanning Tree	30	Gayathri Namasivayam and Mirosław Truszczyński	×	–	×
Car Sequencing	54	Marco Cadoli	×	–	×
Disjunctive Loops	9	Marco Maratea	–	×	–
EqTest	10	Asparagus	–	×	–
Factoring	10	Asparagus	–	×	×
Fast Food	221	Wolfgang Faber	×	–	×
Gebser Suite	202	Martin Gebser	–	×	×
Grammar-Based Information Extraction	102	Marco Manna	–	×	–
Hamiltonian Cycle	30	Lengning Liu and Mirosław Truszczyński	×	–	×
Hamiltonian Path	58	Asparagus	–	×	×
Hashiwokakero	211	Martin Brain	–	–	×
Hitori	211	Martin Brain	–	–	×
Knight's Tour	165	Martin Brain	–	–	×
Mutex	7	Marco Maratea	–	×	–
Random Non-Tight	120	Enno Schultz, Martin Gebser	–	×	×
Random Quantified Boolean Formulas	40	Marco Maratea	–	×	–
Reachability	53	Giorgio Terracina	×	×	×
RLP	573	Yuting Zhao and Fangzhen Lin	–	×	×
Schur Numbers	33	Lengning Liu and Mirosław Truszczyński	×	–	×
Schur Numbers	5	Asparagus	–	×	–
Social Golfer	175	Marco Cadoli	×	–	×
Sokoban	131	Wolfgang Faber	×	–	–
Solitaire Backward	36	Martin Brain	–	–	×
Solitaire Backward (2)	10	Lengning Liu and Mirosław Truszczyński	–	–	×
Solitaire Forward	41	Martin Brain	–	–	×
Strategic Companies	35	Nicola Leone	–	×	–
Su-Doku	8	Martin Brain	–	–	×
TOAST	54	Martin Brain	–	–	×
Towers of Hanoi	29	Gayathri Namasivayam and Mirosław Truszczyński	×	–	×
Traveling Salesperson	30	Lengning Liu and Mirosław Truszczyński	×	–	×
Weight-Bounded Dominating Set	30	Lengning Liu and Mirosław Truszczyński	×	–	×
Weighted Latin Square	35	Gayathri Namasivayam and Mirosław Truszczyński	×	–	×
Weighted Spanning Tree	30	Gayathri Namasivayam and Mirosław Truszczyński	×	–	×
Word Design DNA	5	Marco Cadoli	×	–	×

Algorithm 1. Semiautomatic Benchmarking Procedure

```
Input : $Classes$ — set of benchmark classes
        $Used$ — set of benchmark instances run already
        $Fresh$ — set of benchmark instances not run so far
        $max$ — maximum number of suitable benchmark instances per benchmark class
1 repeat
2     $ToRun \leftarrow \emptyset$                                   /* no runs scheduled yet */
3     foreach $C$ in $Classes$ do
4         $done \leftarrow |\textsc{Suitable}(Used[C])|$
5         if $done < max$ then $ToRun \leftarrow ToRun \cup \textsc{Select}(max - done, Fresh[C])$
6     $\textsc{Run}(ToRun)$                                          /* execute the scheduled runs */
7     $Used \leftarrow Used \cup ToRun$
8     $Fresh \leftarrow Fresh \setminus ToRun$
9 until $ToRun = \emptyset$
```

instances for use in the competition (we describe our selection criteria below). For SCore and SLparse, we relied on the availability of encodings to produce ground instances according to the input format of the respective category.

It is important to note that competitors in the MGS category did not have to use the default Asparagus encodings. Instead, they had the option to provide their own problem encodings, presumably better than the default ones, as the MGS category was also about assessing the modeling capabilities of grounders and solvers.

The collected benchmarks constitute the result of efforts of the broad ASP community. Table 1 gives an overview of all benchmarks gathered for the competition on the Asparagus platform, listing problems forming benchmark classes, the accompanying number of instances, the names of the contributors, and the associated competition categories (M stands for MGS, C for SCore, and L for SLparse, respectively).

For each competition category, benchmarks were selected by a fixed yet random scheme, shown in Algorithm 1. The available benchmark classes are predefined in $Classes$. Their instances that have been run already are in $Used$, the ones not run so far are in $Fresh$. As an invariant, we impose $Used \cap Fresh = \emptyset$. For a benchmark class C in $Classes$, we access used and fresh instances via $Used[C]$ and $Fresh[C]$, respectively. The maximum number max of instances per class aims at having approximately 100 benchmark instances overall in the evaluation, that is, $|Classes| * max \approx 100$ (if feasible). A benchmark instance in $Used[C]$ is considered suitable, that is, it is in $\textsc{Suitable}(Used[C])$, if at least one call script (see Section 5) was able to solve it and at most three call scripts solved it in less than one second (in other words, some system can solve it, yet it is not too easy). Function $\textsc{Select}(n, Fresh[C])$ randomly determines n fresh instances from class C if available (if $n \leq |Fresh[C]|$) in order to eventually obtain max suitable instances of class C. Procedure $\textsc{Run}(ToRun)$ runs all call scripts on the benchmark instances scheduled in $ToRun$. When $ToRun$ runs empty, no fresh benchmark instances are selectable. If a benchmark class yields less than max suitable instances, Algorithm 1 needs to be re-invoked with an increased max value for the other benchmark classes; thus, Algorithm 1 is “only” semiautomatic.

5 Competitors

As with benchmarks, all executable programs (grounders and solvers) were installed and run on Asparagus. To this end, participants had to obtain Asparagus accounts, unless they already had one, and to register for the respective competition categories.

Different variants of a system were allowed to run in the competition by using different *call scripts*; per competitor, up to three of them could be registered for each competition category. The list of participating solvers and corresponding call scripts can be found in Table 2.

Table 2. Participating solvers and corresponding call scripts ($\cdot^\star$ used in both SCore and SCore$^\vee$; $\cdot^\vee$ used in SCore$^\vee$ only)

Solver	Affiliation	MGS	SCore	SLparse
asper	Angers		ASPeR-call-script ASPeRS20-call-script ASPeRS30-call-script	
assat	Hongkong	script.assat.normal		script.assat.lparse-output
clasp	Potsdam	clasp_cmp_score clasp_cmp_score2 clasp_score_def	clasp_cmp_score_glp clasp_cmp_score_glp2 clasp_score_glp_def	clasp_cmp_slparse clasp_cmp_slparse2 clasp_slparse_def
cmodels	Texas	default scriptAtomreasonLp scriptEloopLp	defaultGlparse.sh scriptAtomreasonGlparse scriptEloopGlparse disjGlparseDefault$^\vee$ disjGparseEloop$^\vee$ disjGparseVerMin$^\vee$	groundedDefault scriptAtomreasonGr scriptEloopGr
dlv	Vienna/ Calabria	dlv-contest-special dlv-contest	dlv-contest-special* dlv-contest*	
gnt	Helsinki	gnt dencode+gnt dencode_bc+gnt	gnt_score* dencode+gnt_score* dencode_bc+gnt_score*	gnt_slparse dencode+gnt_slparse dencode_bc+gnt_slparse
lp2sat	Helsinki		lp2sat+minisat wf+lp2sat+minisat lp2sat+siege	
nomore	Potsdam	nomore-default nomore-localprop nomore-D	nomore-default-SCore nomore-localprop-SCore nomore-D-SCore	nomore-default-slparse nomore-localprop-slparse nomore-D-slparse
pbmodels	Kentucky	pbmodels-minisat+-MGS pbmodels-pueblo-MGS pbmodels-wsatcc-MGS	pbmodels-minisat+-SCore pbmodels-pueblo-SCore pbmodels-wsatcc-SCore	pbmodels-minisat+-SLparse pbmodels-pueble-SLparse pbmodels-wsatcc-SLparse
smodels	Helsinki	smodels smodels_rs smodels_rsn	smodels_score smodels_rs_score smodels_rsn_score	smodels_slparse smodels_rs_slparse smodels_rsn_slparse

6 Results

This section presents the results of the *First Answer Set Programming System Competition*. The placement of systems was determined according to the number of instances solved within the allocated time and space as the primary measure of performance. Run times were used as tie breakers. Given that each system was allowed to participate in each competition category in three variants, we decided to allocate only one place to each system. The place of a system is determined by its best performing variant, represented by a call script.

Each single run was limited to 600 seconds execution time and 448 MB RAM memory usage. For each competition category, we give below tables providing a complete placement of all participating call scripts. We report for each call script the absolute and relative number of solved instances ('Solved' and '%'), its minimum, maximum, and average run times, where 'avg' gives the average run time on all solved instances and 'avg^t' gives the average run time on all instances with a timeout taken as 600 seconds. The last column gives the Euclidean distance between the vector of all run times of a call script and the virtually best solver taken to be the vector of all minimum run times.

For each competition category, we also provide similar data for the used benchmark classes. These tables give the number of instances selected from each class ('#'), the absolute and relative number of successfully completed runs ('Solved' and '%') aggregated over all instances and call scripts, the same separately for satisfiable ('SAT') and unsatisfiable ('UNSAT') instances, and finally, the minimum, maximum, and average times over all successfully completed runs on the instances of a class.

Finally, we chart the numbers of solved instances and the solving times of participating call scripts for each competition category. See Section 6.1 for an exemplary description of these graphics.

More statistics and details are available at [17].

6.1 Results of the MGS Competition

The winners of the MGS competition are:

FIRST PLACE WINNER	*dlv*
SECOND PLACE WINNER	*pbmodels*
THIRD PLACE WINNER	*clasp*

The detailed placement of call scripts is given in Table 3. Table 4 gives statistics about the benchmark classes used in the MGS competition. The performance of all participat-

Table 3. Placing of call scripts in the MGS competition

Place	Call Script	Solved	%	min	max	avg	avg^t	EuclDist
1	dlv-contest-special	76/119	63.87	0.07	565.38	54.31	251.49	3542.79
2	dlv-contest	66/119	55.46	0.06	579.09	49.73	294.81	3948.3
3	pbmodels-minisat+-MGS	65/111	58.56	0.52	563.39	83.27	297.41	4142.88
4	clasp_cmp_score2	64/111	57.66	0.87	579.14	115.35	320.56	4285.59
5	clasp_score_def	60/111	54.05	0.91	542.64	80.09	318.97	4277.69
6	clasp_cmp_score	58/111	52.25	0.83	469.46	87.40	332.16	4280.92
7	pbmodels-pueble-MGS	54/111	48.65	0.34	584.31	80.94	347.49	4491.85
8	default	51/119	42.86	0.23	453.88	64.96	370.7	4543.12
9	smodels_rs	34/118	28.81	0.30	539.33	153.86	471.45	5349.11
10	smodels	34/104	32.69	1.14	584.06	173.60	460.6	4974.14
11	pbmodels-wsatcc-MGS	23/111	20.72	1.05	563.52	136.97	504.06	5409.17
12	smodels_rsn	22/111	19.82	0.12	579.10	163.14	513.42	5471.81
13	nomore-default	13/111	11.71	22.04	521.11	315.78	566.71	5714.67
14	scriptAtomreasonLp	12/24	50.00	3.59	259.88	91.18	345.59	2121.13
15	nomore-localprop	10/111	9.01	19.54	521.03	324.83	575.21	5787.71
16	nomore-D	9/111	8.11	48.50	473.89	161.99	564.49	5763.96
17	scriptEloopLp	4/16	25.00	49.92	223.03	106.09	476.52	2088.98
18	gnt	0/8	0.00				600	1696.31
19	dencode+gnt	0/8	0.00				600	1696.31
20	dencode_bc+gnt	0/8	0.00				600	1696.31
21	script.assat.normal	0/50	0.00				600	4101.66

Table 4. Benchmarks used in the MGS competition

Benchmark Class	#	Solved	%	SAT	%	UNSAT	%	min	max	avg
Sokoban	8	16/16	100.00	6/6	100.00	10/10	100.00	12.66	109.52	54.39
Weighted Spanning Tree	8	89/127	70.08	89/127	70.08	0/0		0.06	579.10	115.27
Social Golfer	8	75/120	62.50	59/75	78.67	16/45	35.56	0.23	579.09	40.34
Bounded Spanning Tree	8	73/127	57.48	73/127	57.48	0/0		0.23	519.09	65.47
Towers of Hanoi	8	71/136	52.21	71/136	52.21	0/0		1.74	584.31	101.10
Blocked N-Queens	8	60/120	50.00	44/75	58.67	16/45	35.56	6.50	542.64	181.69
Hamiltonian Cycle	8	64/150	42.67	64/150	42.67	0/0		0.88	453.88	57.46
Weighted Latin Square	8	41/120	34.17	22/45	48.89	19/75	25.33	0.12	477.67	93.04
Weight-Bounded Dominating Set	8	42/127	33.07	42/127	33.07	0/0		0.83	563.39	127.87
Schur Numbers	8	32/120	26.67	18/90	20.00	14/30	46.67	3.27	468.79	120.96
15-Puzzle	7	24/105	22.86	24/105	22.86	0/0		52.91	579.14	291.20
Car Sequencing	8	25/120	20.83	24/105	22.86	1/15	6.67	0.69	563.52	106.08
Fast Food	8	19/127	14.96	8/64	12.50	11/63	17.46	5.30	352.11	113.08
Traveling Salesperson	8	16/128	12.50	16/128	12.50	0/0		0.72	11.18	2.17
Reachability	8	8/160	5.00	6/120	5.00	2/40	5.00	0.26	169.90	49.48

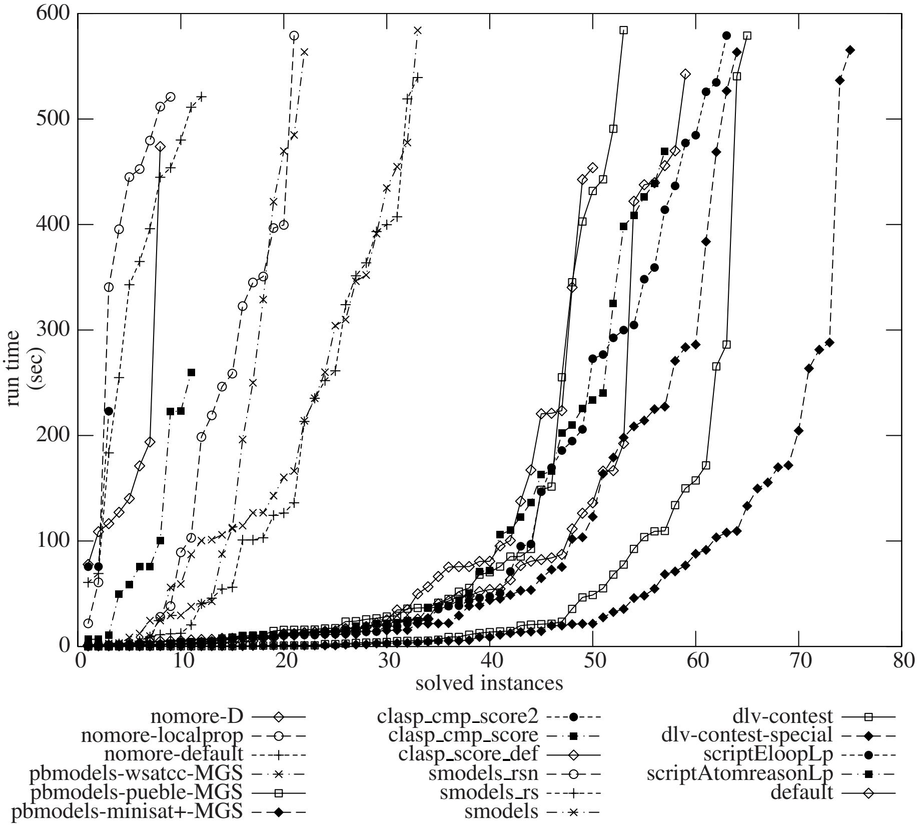


Fig. 1. Chart of the MGS competition

ing call scripts is also given in graphical form in Figure 1. Thereby, the x-axis represents the number of (solved) benchmark instances, and the y-axis represents time. For a given call script, the solved instances are ordered by run times, and a point (x, y) in the chart expresses that the xth instance was solved in y seconds. The more to the right the curve of a call script ends, the more benchmark instances were solved within the allocated time and space. Since the number of solved instances is our primary measure of performance, the rightmost call script is the winner of the MGS competition.

6.2 Results of the SCore Competition

The winners of the SCore competition are:

FIRST PLACE WINNER	*clasp*
SECOND PLACE WINNER	*smodels*
THIRD PLACE WINNER	*cmodels*

The detailed placement of call scripts is given in Table 5. Table 6 gives statistics about the benchmark classes used in the SCore competition. The performance of all participating call scripts is charted in Figure 2.

Table 5. Placing of call scripts in the SCore competition

Place	Call Script	Solved	%	min	max	avg	avg^t	EuclDist
1	clasp_cmp_score_glp2	89/95	93.68	0.56	530.21	29.81	65.82	1080.46
2	clasp_cmp_score_glp	89/95	93.68	0.75	504.49	30.36	66.34	1099.14
3	clasp_score_glp_def	86/95	90.53	0.75	431.66	25.20	79.66	1386.63
4	smodels_rs_score	81/95	85.26	1.21	346.36	38.93	121.61	1872.81
5	defaultGlparse.sh	81/95	85.26	1.35	597.97	46.86	128.38	2089.18
6	scriptAtomreasonGlparse	80/95	84.21	1.30	576.80	42.40	130.44	2107.83
7	pbmodels-minisat+-SCore	80/95	84.21	0.72	436.11	57.18	142.89	2170.4
8	pbmodels-pueblo-SCore	78/95	82.11	0.34	452.84	41.00	141.03	2210.39
9	dencode+gnt_score	78/95	82.11	1.27	363.19	42.80	142.51	2162.64
10	smodels_score	77/95	81.05	1.28	352.41	40.40	146.43	2217.61
11	dencode_bc+gnt_score	77/95	81.05	1.27	360.70	42.52	148.15	2228.65
12	gnt_score	77/95	81.05	1.27	359.77	42.56	148.18	2228.83
13	scriptEloopGlparse	75/95	78.95	1.36	598.20	42.86	160.15	2493.41
14	smodels_rsn_score	75/95	78.95	1.21	486.23	63.00	176.05	2503.32
15	lp2sat+minisat	75/95	78.95	1.10	561.06	79.89	189.39	2621.13
16	wf+lp2sat+minisat	73/95	76.84	1.56	587.40	86.42	205.35	2792.51
17	dlv-contest-special	69/95	72.63	0.24	586.62	102.47	238.64	3090.71
18	dlv-contest	68/95	71.58	0.24	587.83	96.69	239.74	3110.36
19	lp2sat+siege	68/95	71.58	1.11	471.36	97.50	240.32	3052.8
20	nomore-localprop-SCore	64/95	67.37	2.45	550.43	103.23	265.34	3316.33
21	nomore-default-SCore	63/95	66.32	2.45	554.76	124.62	284.75	3415.78
22	nomore-D-SCore	62/95	65.26	2.77	559.88	161.15	313.59	3583.85
23	ASPeR-call-script	24/95	25.26	1.47	592.24	98.28	473.25	4906.79
24	ASPeRS30-call-script	21/95	22.11	1.51	561.20	88.99	487.04	4995.78
25	ASPeRS20-call-script	21/95	22.11	1.49	381.33	89.40	487.13	4980.24
26	pbmodels-wsatcc-SCore	6/95	6.32	25.57	529.80	208.15	575.25	5514.97

Table 6. Benchmarks used in the SCore competition

Benchmark Class	#	Solved	%	SAT	%	UNSAT	%	min	max	avg
15-Puzzle	10	236/260	90.77	121/130	93.08	115/130	88.46	0.74	480.13	25.49
Factoring	5	114/130	87.69	46/52	88.46	68/78	87.18	1.21	554.76	50.35
RLP-150	14	306/364	84.07	21/26	80.77	285/338	84.32	0.34	205.03	22.01
RLP-200	14	287/364	78.85	0/0		287/364	78.85	0.39	581.98	75.21
Schur Numbers	5	99/130	76.15	88/104	84.62	11/26	42.31	2.76	561.20	49.82
EqTest	5	93/130	71.54	0/0		93/130	71.54	0.66	592.24	75.02
Hamiltonian Path	14	219/364	60.16	201/338	59.47	18/26	69.23	0.24	559.88	64.74
Random Non-Tight	14	216/364	59.34	38/52	73.08	178/312	57.05	0.57	598.20	121.87
Blocked N-Queens	14	167/364	45.88	55/156	35.26	112/208	53.85	13.70	587.40	110.59

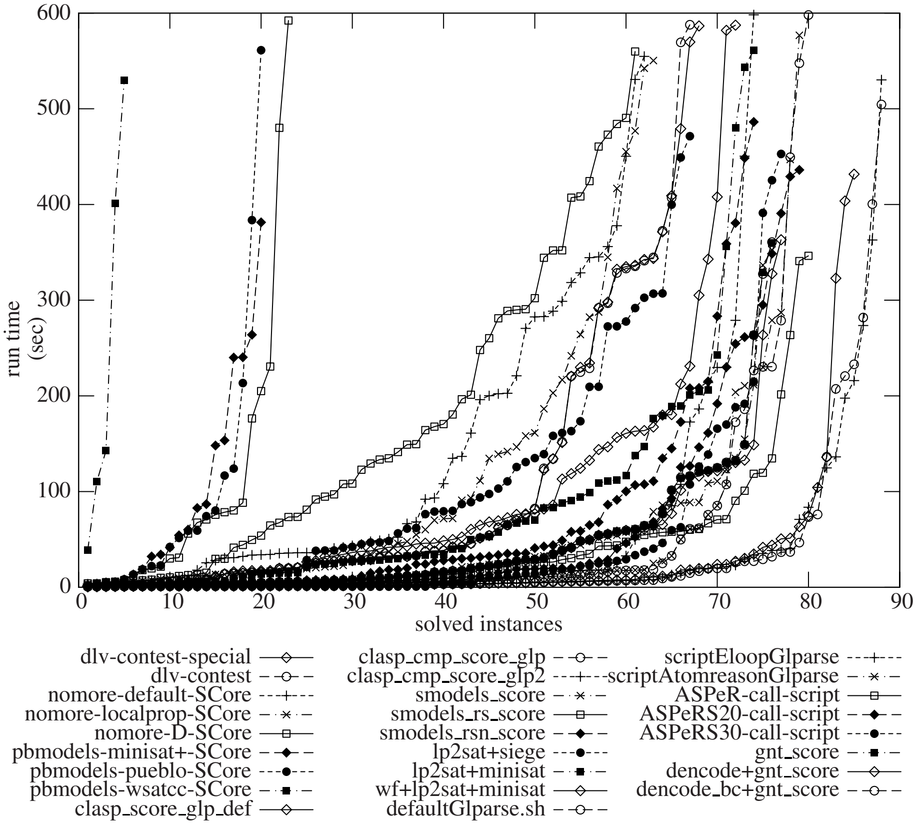


Fig. 2. Chart of the SCore competition

Table 7. Placing of call scripts in the SCore$^\vee$ competition

Place	Call Script	Solved	%	min	max	avg	avg^t	EuclDist
1	dlv-contest-special	54/55	98.18	0.03	258.73	23.59	34.07	279.35
2	dlv-contest	54/55	98.18	0.03	259.97	23.86	34.33	279.44
3	disjGlparseDefault	33/55	60.00	1.06	521.59	54.49	272.69	2631.4
4	dencode+gnt_score	29/55	52.73	2.23	521.51	56.34	313.34	2922.73
5	gnt_score	29/55	52.73	2.21	521.91	56.44	313.4	2922.87
6	dencode_bc+gnt_score	29/55	52.73	2.22	522.63	56.45	313.4	2923.17
7	disjGparseEloop	27/55	49.09	1.21	521.55	33.83	322.06	2978.46
8	disjGparseVerMin	27/55	49.09	1.22	523.40	33.98	322.14	2978.77

The winners of the SCore$^\vee$ competition are:

FIRST PLACE WINNER	*dlv*
SECOND PLACE WINNER	*cmodels*
THIRD PLACE WINNER	*gnt*

Table 8. Benchmarks used in the SCore$^{\vee}$ competition

Benchmark Class	#	Solved	%	SAT	%	UNSAT	%	min	max	avg
Grammar-Based Information Extraction	15	120/120	100.00	64/64	100.00	56/56	100.00	0.62	7.86	5.03
Disjunctive Loops	3	21/24	87.50	0/0		21/24	87.50	0.44	522.63	95.24
Strategic Companies	15	88/120	73.33	88/120	73.33	0/0		0.35	523.40	71.22
Mutex	7	18/56	32.14	0/0		18/56	32.14	0.03	259.97	37.41
Random Quantified Boolean Formulas	15	35/120	29.17	0/0		35/120	29.17	0.11	290.99	44.41

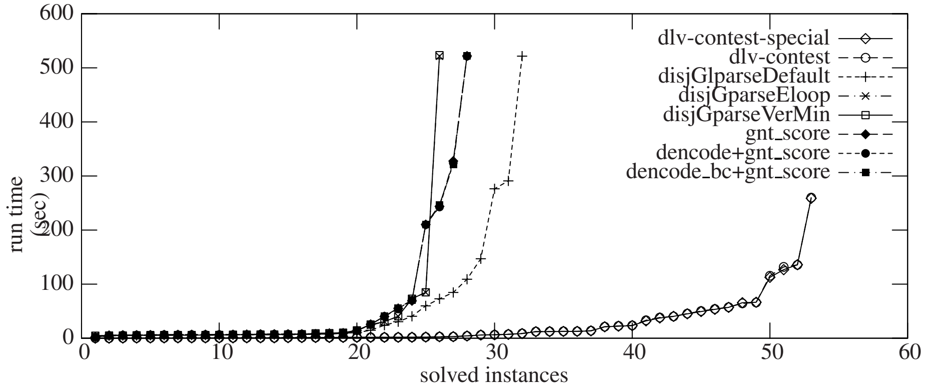


Fig. 3. Chart of the SCore$^{\vee}$ competition

Table 9. Placing of call scripts in the SLparse competition

Place	Call Script	Solved	%	min	max	avg	avg^t	EuclDist
1	clasp_cmp_slparse2	100/127	78.74	0.38	556.49	75.96	187.37	2791.89
2	clasp_cmp_slparse	94/127	74.02	0.41	502.53	61.37	201.33	2919.46
3	pbmodels-minisat+-SLparse	91/127	71.65	0.49	503.57	76.69	225.03	3241.06
4	clasp_slparse_def	89/127	70.08	0.37	546.50	55.62	218.5	3152.34
5	smodels_rs_slparse	87/127	68.50	0.23	576.28	95.92	254.69	3403.9
6	groundedDefault	81/127	63.78	0.25	407.49	46.20	246.79	3448.34
7	scriptAtomreasonGr	81/127	63.78	0.25	407.46	50.55	249.56	3465.67
8	scriptEloopGr	78/127	61.42	0.24	407.48	46.15	259.84	3598.7
9	smodels_slparse	75/127	59.06	0.26	518.08	102.76	306.35	3958.86
10	smodels_rsn_slparse	74/127	58.27	0.35	596.39	70.52	291.49	3815.31
11	pbmodels-pueblo-SLparse	69/127	54.33	0.25	593.05	87.25	321.42	4189.03
12	nomore-D-slparse	54/127	42.52	1.08	530.39	152.78	409.84	4765.06
13	nomore-localprop-slparse	50/127	39.37	1.08	517.63	120.80	411.34	4846.58
14	nomore-default-slparse	49/127	38.58	1.08	549.80	143.44	423.85	4920.23
15	gnt_slparse	35/127	27.56	2.10	482.13	81.40	457.08	5276.26
16	dencode+gnt_slparse	35/127	27.56	2.18	482.81	81.64	457.15	5276.38
17	dencode_bc+gnt_slparse	35/127	27.56	2.10	485.36	81.70	457.16	5276.53
18	script.assat.lparse-output	30/127	23.62	1.00	225.28	38.64	467.4	5379.18
19	pbmodels-wsatcc-SLparse	25/127	19.69	1.12	272.98	46.56	491.05	5585.82

The detailed placement of call scripts is given in Table 7. Table 8 gives statistics about the benchmark classes used in the SCore$^{\vee}$ competition. The performance of all participating call scripts is charted in Figure 3.

Table 10. Benchmarks used in the SLparse competition

Benchmark Class	#	Solved	%	SAT	%	UNSAT	%	min	max	avg
RLP-200	5	89/95	93.68	18/19	94.74	71/76	93.42	0.25	465.69	60.94
RLP-150	5	87/95	91.58	17/19	89.47	70/76	92.11	0.25	183.29	14.17
Factoring	4	69/76	90.79	36/38	94.74	33/38	86.84	0.63	549.80	64.09
verifyTest-variableSearchSpace (TOAST)	5	81/95	85.26	81/95	85.26	0/0		0.24	303.32	16.43
Random Non-Tight	5	75/95	78.95	41/57	71.93	34/38	89.47	0.41	518.08	123.56
Knight's Tour	5	71/95	74.74	71/95	74.74	0/0		1.04	248.97	28.29
Su-Doku	3	42/57	73.68	42/57	73.68	0/0		18.15	176.69	68.00
searchTest-plain (TOAST)	5	68/95	71.58	18/38	47.37	50/57	87.72	1.54	339.51	45.50
searchTest-verbose (TOAST)	5	63/95	66.32	63/95	66.32	0/0		25.84	485.36	136.07
Hamiltonian Path	5	60/95	63.16	60/95	63.16	0/0		0.37	530.39	58.02
Weighted Spanning Tree	5	58/95	61.05	58/95	61.05	0/0		3.24	596.39	122.04
Solitaire Forward	5	55/95	57.89	55/95	57.89	0/0		1.19	593.05	49.06
Bounded Spanning Tree	5	54/95	56.84	54/95	56.84	0/0		7.43	413.63	70.11
Hamiltonian Cycle	5	51/95	53.68	51/95	53.68	0/0		0.47	464.71	51.30
Solitaire Backward	5	47/95	49.47	33/76	43.42	14/19	73.68	0.30	552.11	71.72
Towers of Hanoi	5	43/95	45.26	43/95	45.26	0/0		6.28	478.30	169.96
Blocked N-Queens	5	40/95	42.11	36/76	47.37	4/19	21.05	1.57	590.04	193.59
Social Golfer	5	37/95	38.95	24/38	63.16	13/57	22.81	0.84	291.48	30.70
Schur Numbers	5	31/95	32.63	11/57	19.30	20/38	52.63	1.23	496.82	104.57
Hashiwokakero	5	26/95	27.37	0/0		26/95	27.37	6.71	377.69	72.36
Weighted Latin Square	5	23/95	24.21	6/19	31.58	17/76	22.37	0.23	576.28	144.13
15-Puzzle	5	17/95	17.89	17/95	17.89	0/0		106.66	502.53	327.56
Weight-Bounded Dominating Set	5	15/95	15.79	15/95	15.79	0/0		1.55	467.51	112.55
Traveling Salesperson	5	12/95	12.63	12/95	12.63	0/0		0.35	212.66	20.24
Solitaire Backward (2)	5	11/95	11.58	11/95	11.58	0/0		5.32	330.89	101.55
Car Sequencing	5	7/95	7.37	7/95	7.37	0/0		7.17	556.49	249.51

6.3 Results of the SLparse Competition

The winners of the SLparse competition are:

FIRST PLACE WINNER	*clasp*
SECOND PLACE WINNER	*pbmodels*
THIRD PLACE WINNER	*smodels*

The detailed placement of call scripts is given in Table 9. Table 10 gives statistics about the benchmark classes used in the SLparse competition. The performance of all participating call scripts is charted in Figure 4.

7 Discussion

This *First Answer Set Programming System Competition* offers many interesting lessons stemming from running diverse solvers on multifaceted benchmark instances. Some of the lessons may have general implications on the future developments in ASP.

First, the experiences gained from the effort to design the competition clearly point out that the lack of well-defined input, intermediate, and output languages is a major problem. In some cases, it forced the competition team to resort to "ad hoc" solutions. Further, there is no standard core ASP language covering programs with aggregates, which makes it difficult to design a single and fair field for all systems to compete. No standard way in which errors are signaled and no consensus on how to deal with incomplete solvers are somewhat less critical but also important issues. Benchmark selection is a major problem. The way benchmarks and their instances are chosen may

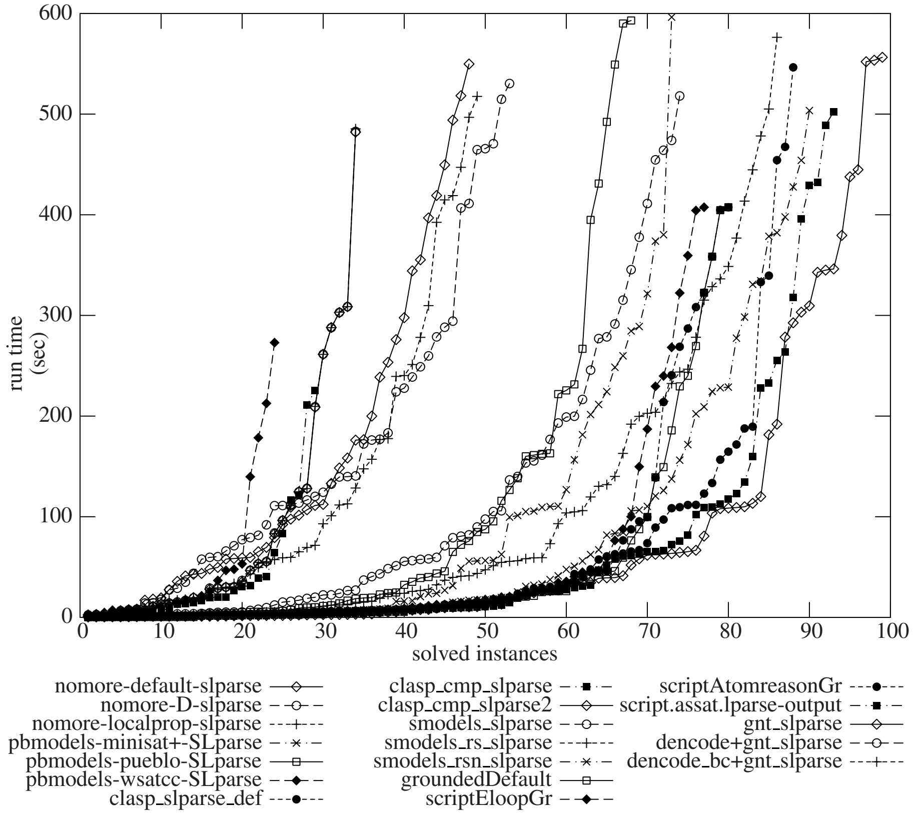


Fig. 4. Chart of the SLparse competition

have a significant impact on the results in view of diverging performances of solvers and different degrees of difficulty among the instances of benchmark classes. Sometimes even grounding problem encodings on problem instances to produce ground programs on which solvers were to compete was a major hurdle (see below).

This first edition of the competition focused on the performance of solvers on ground programs, which is certainly important. However, the roots of the ASP approach are in declarative programming and knowledge representation. For both areas, *modeling* knowledge domains and problems that arise in them is of major concern (this is especially the case for knowledge representation). By developing the MGS category, we tried to create a platform where ASP systems could be differentiated from the perspective of their modeling functionality. However, only one group chose to develop programs specialized to their system (hence, this group and their system are the well-deserved winner). All other groups relied on default encodings. It is critical that a better venue for testing modeling capabilities is provided for future competitions.

Further, not only modeling support and the performance of solvers determine the quality of an ASP system. Grounding is an essential part of the process too and, in some cases, it is precisely where the bottleneck lies. The MGS category was the only category that took both the grounding time and the solving time into account. It is important to stress more the role of grounding in future competitions.

There will be future competitions building on the experiences of this one. Their success and their impact on the field will depend on continued broad community participation in fine-tuning and expanding the present format. In this respect, the *First Answer Set Programming System Competition* should encourage the further progress in the development of ASP systems and applications, similar to competitions in related areas, such as SATisfiability, Quantified Boolean Formulas, and Pseudo-Boolean constraints.

Acknowledgments

This project would not have been possible without strong and broad support from the whole ASP community. We are grateful to all friends and colleagues who contributed ideas on the contest format, submitted benchmark problems, and provided continued encouragement. Most of all, we want to thank all the competitors. Without you, there would have been no competition.

Mirosław Truszczyński acknowledges the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
2. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In Apt, K., Marek, W., Truszczyński, M., Warren, D., eds.: The Logic Programming Paradigm: a 25-Year Perspective. Springer (1999) 375–398
3. Gelfond, M., Leone, N.: Logic programming and knowledge representation — the A-prolog perspective. Artificial Intelligence **138**(1-2) (2002) 3–38
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
5. Colmerauer, A., Kanoui, H., Pasero, R., Roussel, P.: Un systeme de communication homme-machine en Francais. Technical report, University of Marseille (1973)
6. Kowalski, R.: Predicate logic as a programming language. In Rosenfeld, J., ed.: Proceedings of the Congress of the International Federation for Information Processing, North Holland (1974) 569–574
7. McCarthy, J.: Circumscription — a form of nonmonotonic reasoning. Artificial Intelligence **13**(1-2) (1980) 27–39
8. Reiter, R.: A logic for default reasoning. Artificial Intelligence **13**(1-2) (1980) 81–132
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proceedings of the International Conference on Logic Programming, MIT Press (1988) 1070–1080
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3-4) (1991) 365–385
11. Marek, W., Truszczyński, M.: Stable semantics for logic programs and default theories. In Lusk, E., Overbeek, R., eds.: Proceedings of the North American Conference on Logic Programming, MIT Press (1989) 243–256

12. Bidoit, N., Froidevaux, C.: Negation by default and unstratifiable logic programs. Theoretical Computer Science **78**(1) (1991) 85–112
13. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: A deductive system for non-monotonic reasoning. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning, Springer (1997) 364–375
14. Niemelä, I., Simons, P.: Smodels — an implementation of the stable model and well-founded semantics for normal logic programs. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning, Springer (1997) 420–429
15. Borchert, P., Anger, C., Schaub, T., Truszczyński, M.: Towards systematic benchmarking in answer set programming: The Dagstuhl initiative. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning, Springer (2004) 3–7
16. Le Berre, D., Simon, L., eds.: Special Volume on the SAT 2005 Competitions and Evaluations. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4) (2006)
17. (`http://asparagus.cs.uni-potsdam.de/contest`)
18. (`http://asparagus.cs.uni-potsdam.de`)
19. 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) (2006) 499–562
20. Syrjänen, T.: Lparse 1.0 user's manual. (http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz)

CR-MODELS: An Inference Engine for CR-Prolog

Marcello Balduccini

Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
marcello.balduccini@ttu.edu

Abstract. CR-Prolog is an extension of the knowledge representation language A-Prolog. The extension is built around the introduction of *consistency-restoring rules* (cr-rules for short), and allows an elegant formalization of events or exceptions that are unlikely, unusual, or undesired. The flexibility of the language has been extensively demonstrated in the literature, with examples that include planning and diagnostic reasoning.

In this paper we present the design of an inference engine for CR-Prolog that is efficient enough to allow the practical use of the language for medium-size applications. The capabilities of the inference engine have been successfully demonstrated with experiments on an application independently developed for use by NASA.

1 Introduction

In recent years, A-Prolog – a knowledge representation language based on the answer set semantics [8] – was shown to be a useful tool for knowledge representation and reasoning (e.g. [5,7]). The language is expressive and has a well understood methodology of representing defaults, causal properties of actions and fluents, various types of incompleteness, etc. Over time, several extensions of A-Prolog have been proposed, aimed at improving event further the expressive power of the language.

One of these extensions, called CR-Prolog [3], is built around the introduction of *consistency-restoring rules* (cr-rules for short). The intuitive idea behind cr-rules is that they are normally not applied, even when their body is satisfied. They are only applied if the regular program (i.e. the program consisting only of conventional A-Prolog rules) is inconsistent. The language also allows the specification of a partial preference order on cr-rules, intuitively regulating the application of cr-rules.

One of the most immediate uses of cr-rules is an elegant encoding of events or exceptions that are unlikely, unusual, or undesired (and preferences can be used to formalize the relative likelihood of these events and exceptions).

The flexibility of CR-Prolog has been extensively demonstrated in the literature [1,3,4,6], with examples including planning and diagnostic reasoning. For example, in [3], cr-rules have been used to model exogenous actions that may occur, unobserved, and cause malfunctioning in a physical system. In [1,4], cr-rules have been applied to the task finding high quality plans. The technique consists in encoding requirements that high quality plans must satisfy, and using cr-rules to formalize exceptions to the requirements, that should be considered only as a last resort.

Most of the uses of CR-Prolog in the literature are not strongly concerned with computation time, and use relatively simple prototypes of CR-Prolog inference engines. However, to allow the use of CR-Prolog for practical applications, an efficient inference engine is needed. In this paper, we present the design of an inference engine for CR-Prolog that is efficient enough to allow the practical use of CR-Prolog for medium-size applications. The paper is organized as follows. In the next section, we introduce the syntax and semantics of CR-Prolog. Section 3 contains the description of the algorithm of the inference engine. Finally, in Section 4 we talk about related work and draw conclusions.

2 CR-Prolog

Like A-Prolog, CR-Prolog is a knowledge representation language that allows the formalization of commonsense knowledge and reasoning. The consistency-restoring rules introduced in CR-Prolog allow the encoding of statements that should be used "as rarely as possible, and only if strictly necessary to obtain a consistent set of conclusions," with preferences intuitively determining which statements should be given precedence. The language has been shown to allow the elegant formalization of various sophisticated reasoning tasks that are problematic to encode in A-Prolog.

The syntax of CR-Prolog is determined by a typed signature Σ consisting of types, typed object constants, and typed function and predicate symbols. We assume that the signature contains symbols for integers and for the standard functions and relations of arithmetic. Terms are built as in first-order languages.

By *simple arithmetic terms* of Σ we mean its integer constants. By *complex arithmetic terms* of Σ we mean terms built from legal combinations of arithmetic functions and simple arithmetic terms (e.g. $3+2\cdot 5$ is a complex arithmetic term, but $3+\cdot\ 2\ 5$ is not). Atoms are expressions of the form $p(t_1,\ldots,t_n)$, where p is a predicate symbol with arity n and t's are terms of suitable types. Atoms formed by arithmetic relations are called *arithmetic atoms*. Atoms formed by non-arithmetic relations are called *plain atoms*. We allow arithmetic terms and atoms to be written in notations other than prefix notation, according to the way they are traditionally written in arithmetic (e.g. we write $3 = 1+2$ instead of $=(3,+(1,2))$). Literals are atoms and negated atoms, i.e. expressions of the form $\neg p(t_1,\ldots,t_n)$. Literals $p(t_1,\ldots,t_n)$ and $\neg p(t_1,\ldots,t_n)$ are called *complementary*. By $\overline{l}$ we denote the literal complementary to l. The syntax of the statements of CR-Prolog is defined as follows.

Definition 1. *A* regular rule ρ *is a statement of the form:*

$$r: h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \leftarrow l_1, l_2, \ldots l_m, not\ l_{m+1}, not\ l_{m+2}, \ldots, not\ l_n. \tag{1}$$

where r is a term that uniquely denotes ρ (called name of the rule), $l_1,\ldots,l_m$ are literals, and h_i's and $l_{m+1},\ldots,l_n$ are plain literals. We call h_1 OR h_2 OR $\ldots$ OR h_k the head *of the rule ($head(r)$); $l_1,l_2,\ldots l_m, not\ l_{m+1}, not\ l_{m+2},\ldots, not\ l_n$ is its* body *($body(r)$), and $pos(r)$, $neg(r)$ denote, respectively, $\{l_1,\ldots,l_m\}$ and $\{l_{m+1},\ldots,l_n\}$.*

The informal reading of the rule (in terms of the reasoning of a rational agent about its own beliefs) is the same used in A-Prolog: "if you believe $l_1,\ldots,l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then believe one of $h_1, \ldots, h_k$." The connective "not" is called *default negation*. To simplify the presentation, we allow the rule name to be omitted whenever possible.

A rule such that $k = 0$ is called *constraint*, and is considered a shorthand of:

$$false \leftarrow \text{not } false, l_1, l_2, \ldots l_m, \text{not } l_{m+1}, \text{not } l_{m+2}, \ldots, \text{not } l_n.$$

Definition 2. *A* consistency-restoring rule *(or* cr-rule*) is a statement of the form:*

$$r : h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \overset{+}{\leftarrow} l_1, l_2, \ldots l_m, not\ l_{m+1}, not\ l_{m+2}, \ldots, not\ l_n. \qquad (2)$$

where r, h_i's and l_i's are as before.

The intuitive reading of a cr-rule is "if you believe $l_1, \ldots, l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then you *may possibly* believe one of $h_1, \ldots, h_k$." The implicit assumption is that this possibility is used as little as possible, and only to restore consistency of the agent's beliefs.

Definition 3. *A* CR-Prolog program *is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a typed signature and Π is a set of regular rules and cr-rules.*

In this paper we often denote programs of CR-Prolog by their second element. The corresponding signature is denoted by $\Sigma(\Pi)$. We also extend the basic operations on sets to programs in a natural way, so that, for example, $\Pi_1 \cup \Pi_2$ is the program whose signature and set of rules are the unions of the respective components of Π_1 and Π_2.

The terms, atoms and literals of a program Π are denoted respectively by $terms(\Pi)$, $atoms(\Pi)$ and $literals(\Pi)$. Given a set of relations $\{p_1,\ldots,p_m\}$, $atoms(\{p_1,\ldots,p_m\},\Pi)$ denotes the set of atoms from the signature of Π formed by every p_i. $literals(\{p_1,\ldots, p_m\},\Pi)$ is defined in a similar way. To simplify notation, we allow the use of $atoms(p,\Pi)$ as an abbreviation of $atoms(\{p\},\Pi)$ (and similarly for $literals$).

Given a CR-Prolog program, Π, the *regular part* of Π is the set of its regular rules, and is denoted by $reg(\Pi)$. The set of cr-rules of Π is denoted by $cr(\Pi)$.

Example 1.

$$\begin{cases} r_1 : p \overset{+}{\leftarrow} \text{not } r. & r_2 : q \overset{+}{\leftarrow} \text{not } r. \\ s. & \leftarrow \text{not } p, \text{not } q. \end{cases}$$

The regular part of the program (consisting of the last two rules) is inconsistent. Consistency can be restored by applying either r_1 or r_2, or both. Since cr-rules should be applied as little as possible, the last case is not considered. Hence, the agent is forced to believe either $\{s,p\}$ or $\{s,q\}$.[1]

When different cr-rules are applicable, it is possible to specify preferences on which one should be applied by means of atoms of the form $prefer(r_1,r_2)$, where r_1, r_2 are names of cr-rules. The atom informally says "do not consider solutions obtained using r_2 unless no solution can be found using r_1." The next example shows the effect of the introduction of preferences in the program from Example 1.

[1] The examples in this section are only aimed at illustrating the features of the language, and not its usefulness. Please refer to e.g. [3,1] for more comprehensive examples.

Example 2.

$$\begin{cases} r_1 : p \overset{+}{\leftarrow} \text{not } r. & r_2 : q \overset{+}{\leftarrow} \text{not } r. \\ s. & prefer(r_1, r_2). \\ \leftarrow \text{not } p, \text{not } q. & \end{cases}$$

The preference prevents the agent from applying r_2 unless no solution can be found using r_1. We have seen already that r_1 is sufficient to restore consistency. Hence, the agent has only one set of beliefs, $\{s, p, prefer(r_1, r_2)\}$.

Notice that our reading of the preference atom $prefer(r_1, r_2)$ rules out solutions in which r_1 and r_2 are applied simultaneously, as the use of r_2 is allowed only if no solution is obtained by applying r_1.

As usual, we assume that rules containing variables are shorthands for the sets of their ground instances.

Now we define the semantics of CR-Prolog. In the following discussion, Π denotes an arbitrary CR-Prolog program. Also, for every $R' \subseteq cr(\Pi)$, $\theta(R')$ denotes the set of regular rules obtained from R' by replacing every connective $\overset{+}{\leftarrow}$ with $\leftarrow$. Notice that the regular part of any CR-Prolog program is an A-Prolog program. We will begin by introducing some terminology.

An atom is in *normal form* if it is an arithmetic atom or if it is a plain atom and its arguments are either non-arithmetic terms or simple arithmetic terms. Notice that literals that are not in normal form can be mapped into literals in normal form by applying the standard rules of arithmetic. For example, $p(4+1)$ is mapped into $p(5)$. For this reason, in the following definition of the semantics of CR-Prolog, we assume that all literals are in normal form.

A literal l is *satisfied* by a consistent set of plain literals S (denoted by $S \models l$) if: (1) l is an arithmetic literal and is true according to the standard arithmetic interpretation; or (2) l is a plain literal and $l \in S$. If l is not satisfied by S, we write $S \not\models l$. An expression not l, where l is a plain literal, is satisfied by S if $S \not\models l$. A set of literals and literals under default negation (not l) is satisfied by S if each element of the set is satisfied by S. A rule is satisfied by S if either its head is satisfied or its body is not satisfied.

Next, we introduce the transitive closure of relation $prefer$. To simplify the presentation, we use, whenever possible, the same term to denote both a rule and its name. For example, given rules $r_1, r_2 \in cr(\Pi)$, the fact that r_1 is preferred to r_2 will be expressed by a statement $prefer(r_1, r_2)$. Notice that this is made possible by the fact that rules are uniquely identified by their names.

Definition 4. For every set of literals, S, from the signature of Π, and every r_1,r_2 from $cr(\Pi)$, $prefs_S(r_1, r_2)$ is true iff (1) $prefer(r_1, r_2) \in S$, or (2) there exists $r_3 \in cr(\Pi)$ such that $prefer(r_1, r_3) \in S$ and $prefs_S(r_3, r_2)$.

To see how the definition works, consider the following example.

Example 3. Given $S = \{prefer(r_1, r_2), prefer(r_2, r_3), a, q, p\}$ and $cr(\Pi)$ consisting of cr-rules r_1, r_2, r_3:

- $prefs_S(r_1, r_2)$ holds (because $prefer(r_1, r_2) \in S$).
- $prefs_S(r_2, r_3)$ holds (because $prefer(r_2, r_3) \in S$).
- $prefs_S(r_1, r_3)$ holds (because $prefer(r_1, r_2) \in S$ and $prefs_S(r_2, r_3)$ holds).

The semantics of CR-Prolog is given in three steps. Intuitively, in the first step we look for combinations of cr-rules that restore consistency. Preferences are not considered, with the exception that solutions deriving from the simultaneous use of two cr-rules between which a preference exists are discarded.

Definition 5. Let $S \subseteq literals(\Pi)$ and $R \subseteq cr(\Pi)$. $\mathscr{V} = \langle S, R \rangle$ is a *view* of Π if:

1. S is an answer set of $reg(\Pi) \cup \theta(R)$, and
2. for every r_1, r_2, if $prefs_S(r_1, r_2)$, then $\{r_1, r_2\} \not\subseteq R$, and
3. for every r in R, $body(r)$ is satisfied by S.

We denote the elements of $\mathscr{V}$ by $\mathscr{V}^S$ and $\mathscr{V}^R$ respectively. The cr-rules in $\mathscr{V}^R$ are said to be *applied*.

Example 4. Consider the program, P_1:

$$\begin{cases} r_1 : t \overset{+}{\leftarrow} . & r_2 : p \overset{+}{\leftarrow} q. \\ r_3 : s \overset{+}{\leftarrow} . & r_4 : q \overset{+}{\leftarrow} . \\ \leftarrow \text{not } t, \text{not } p, \text{not } s. & prefer(r_1, r_3). \end{cases}$$

The regular part of the program is inconsistent. According to Definition 5, $\mathscr{V}_1 = \langle\{t, prefer(r_1, r_3)\}, \{r_1\}\rangle$ is a view of P_1. In fact: (1) $\mathscr{V}_1^S$ is an answer set of $reg(P_1) \cup \theta(\mathscr{V}_1^R)$; (2) $\{r_1, r_3\} \not\subseteq \mathscr{V}_1^R$; and (3) the body of r_1 is trivially satisfied. On the other hand, $\mathscr{V}_x = \langle\{t, s, prefer(r_1, r_3)\}, \{r_1, r_3\}\rangle$ is not a view of P_1, because it does not satisfy condition (2) of the definition. In fact, $pref_{\mathscr{V}_1^S}(r_1, r_3)$ holds but $\{r_1, r_3\} \subseteq \mathscr{V}_1^R$. Similarly, $\mathscr{V}_y = \langle\{t, prefer(r_1, r_3)\}, \{r_1, r_2\}\rangle$ is not a view of P_1. In this case, condition (3) of the definition is not satisfied, as the body of r_2 does not hold in $\mathscr{V}_1^S$. It is not difficult to show that the views of P_1 are (from now on, we omit preference atoms, whenever possible, to save space):

$$\begin{array}{ll} \mathscr{V}_1 = \langle\{t\}, \{r_1\}\rangle & \mathscr{V}_2 = \langle\{t, q\}, \{r_1, r_4\}\rangle \\ \mathscr{V}_3 = \langle\{s\}, \{r_3\}\rangle & \mathscr{V}_4 = \langle\{s, q\}, \{r_3, r_4\}\rangle \\ \mathscr{V}_5 = \langle\{p, q\}, \{r_2, r_4\}\rangle & \mathscr{V}_6 = \langle\{s, p, q\}, \{r_2, r_3, r_4\}\rangle \\ \mathscr{V}_7 = \langle\{t, p, q\}, \{r1, r_2, r_4\}\rangle & \end{array}$$

The second step in the definition of the answer sets of Π consists in selecting the best views with respect to the preferences specified. Particular attention must be paid to the case when preferences are dynamic. The intuition is that we consider only preferences on which there is agreement in the views under consideration.

Definition 6. For every pair of views of Π, $\mathscr{V}_1$ and $\mathscr{V}_2$, $\mathscr{V}_1$ *dominates* $\mathscr{V}_2$ if there exist $r_1 \in \mathscr{V}_1^R$, $r_2 \in \mathscr{V}_2^R$ such that $pref_{(\mathscr{V}_1^S \cap \mathscr{V}_2^S)}(r_1, r_2)$.

Example 5. Let us consider the views of program P_1 from Example 4. View $\mathscr{V}_1$ dominates $\mathscr{V}_3$: in fact, $\mathscr{V}_1^S \cap \mathscr{V}_3^S = \{prefer(r_1, r_3)\}$ and $pref_{\{prefer(r_1,r_3)\}}(r_1, r_3)$ obviously holds. On the other hand, $\mathscr{V}_1$ does not dominate $\mathscr{V}_5$, as neither $pref_{\{prefer(r_1,r_3)\}}(r_1, r_2)$ nor $pref_{\{prefer(r_1,r_3)\}}(r_1, r_4)$ hold.

Definition 7. A view, $\mathscr{V}$, is a candidate answer set of Π if, for every view $\mathscr{V}'$ of Π, $\mathscr{V}'$ does not dominate $\mathscr{V}$.

Example 6. According to the conclusions from Example 4, $\mathscr{V}_3$ is not a candidate answer of P_1, as it is dominated by $\mathscr{V}_1$. Conversely, it is not difficult to see that $\mathscr{V}_1$ is not dominated by any other view, and is therefore a candidate answer set. Overall, the candidate answer sets of P_1 are: $\mathscr{V}_1 = \langle\{t\},\{r_1\}\rangle \mathscr{V}_2 = \langle\{t,q\},\{r_1,r_4\}\rangle \mathscr{V}_5 = \langle\{p,q\},\{r_2,r_4\}\rangle \mathscr{V}_7 = \langle\{t,p,q\},\{r1,r_2,r_4\}\rangle$.

Finally, we select the candidate answer sets that are obtained by applying a minimal set (w.r.t. set-theoretic inclusion) of cr-rules.

Definition 8. A set of literals, S, is an *answer set* of Π if:

1. there exists $R \subseteq cr(\Pi)$ such that $\langle S,R\rangle$ is a candidate answer set of Π, and
2. for every candidate answer set $\langle S',R'\rangle$ of Π, $R' \not\subset R$.

Example 7. Consider $\mathscr{V}_1$ and $\mathscr{V}_2$ from the list of the candidate answer sets of P_1 from Example 6. Since $\mathscr{V}_1^R \subseteq \mathscr{V}_2^R$, $\mathscr{V}_2$ is not an answer set of P_1. According to Definition 8, the answer sets of P_1 are: $\mathscr{V}_1 = \langle\{t\},\{r_1\}\rangle$ $\mathscr{V}_5 = \langle\{p,q\},\{r_2,r_4\}\rangle$.

It is worth pointing out how the above definitions deal with cyclic preferences. For simplicity, let us focus on static preferences. Let r be a cr-rule that occurs in the preference cycle. It is not difficult to see that, for any view $\mathscr{V}$, $pref_{(\mathscr{V}^S \cap \mathscr{V}^S)}(r,r)$ holds. This prevents any view where r is used from being a candidate answer set. Hence, the cr-rules involved in preference cycle cannot be used to restore consistency.

3 The CRMODELS Algorithm

The algorithm for computing the answer sets of CR-Prolog programs is based on a generate-and-test approach. We begin our description of CRMODELS by presenting the algorithm at a high level of abstraction. Next, we increase the level of detail in various steps, until we have a complete specification of CRMODELS.

At a high level of abstraction, one answer set of a CR-Prolog program Π can be computed as show below (Figure 1). Notice that, in the algorithm, $\perp$ is used to indicate the absence of a solution. The algorithm begins by looking for a view $\mathscr{V}$ such that $|\mathscr{V}^R| = 0$. If one is found, CRMODELS$_1$ checks that $\mathscr{V}$ is a candidate answer set of Π (line 5). Notice that, because $|\mathscr{V}^R| = 0$, the condition of Definition 6 is never satisfied (as there is no $r \in \mathscr{V}^R$). Hence, if a view if found for $i = 0$, that view is a candidate answer set, which causes the test at line 5 to succeed. Such a candidate answer set is also minimal w.r.t. set-theoretic inclusion on $\mathscr{V}^R$, which implies that $\mathscr{V}^S$ is an answer set of Π according to Definition 8. Hence, the algorithm returns $\mathscr{V}^S$ and terminates.

Now let us consider what happens if no view is found for $i = 0$. According to line 4, $\mathscr{V}$ is set to $\perp$, which causes the test on line 5 to fail. Because the termination condition of the inner loop (line 8) is true, the loop terminates, i is incremented and, assuming Π contains at least one cr-rule, execution goes back to line 4, where a view $\mathscr{V}$ with

```
Algorithm: CRMODELS$_1$
input: $\Pi$: CR-Prolog program
output: one answer set of $\Pi$
var $i$: number of cr-rules to be applied
1. $i := 0$ { first we look for an answer set of $reg(\Pi)$ }
2. while $(i \leq |cr(\Pi)|)$ do    { outer loop }
3.     repeat    { inner loop }
4.         generate new view $\mathcal{V}$ of $\Pi$ s.t. $|\mathcal{V}^R| = i$; if none is found, $\mathcal{V} := \bot$
5.         if $\mathcal{V}$ is candidate answer set of $\Pi$ then { test fails if $\mathcal{V} = \bot$ }
6.             return $\mathcal{V}^S$ { answer set found }
7.         end if
8.     until $\mathcal{V} = \bot$
9.     $i := i + 1$ { consider views obtained with a larger number of cr-rules }
10. done
11. return $\bot$ { signal that no answer set was found }
```

Fig. 1. Algorithm CRMODELS$_1$

$|\mathcal{V}^R| = 1$ is computed. It is important to notice[2] that, because of the iteration over increasing values of i in the outer loop (lines 2–10), the first candidate answer set found by the algorithm is always guaranteed to be set-theoretically minimal (with respect to the set of cr-rules used). Hence, according to Definition 8, $\mathcal{V}^S$ is an answer set of Π. That explains why the return statement at line 6 is executed without further testing. If no candidate answer set is found for $i = 1$, the iterations of the outer loop continue for increasing values of i until either a candidate answer set is found or the condition on line 2 becomes false (i.e. all possible combinations of cr-rules have been considered). In this case, the algorithm returns $\bot$.

In our approach, both the generation and the test steps (lines 4 and 5 in Figure 1) are reduced to the computation of answer sets of *A-Prolog programs*. To allow a compact representation of the A-Prolog programs involved in these steps, we introduce the following *macros*.

– A macro-rule of the form: $\{p(X)\}$. informally says that any X can have property p, and stands for the rules: $p(X) \leftarrow \text{not } \neg p(X).\ \ \neg p(X) \leftarrow \text{not } p(X)$.
– A macro-rule of the form: $\leftarrow \text{not } i\{p(X)\}j$. informally states that only between i and j X's can have property p and is expanded as follows. Let t denote the cardinality of the ground atoms of the form $p(X)$ and $\Delta(m)$ denote the collection of inequalities: $X_k \neq X_h$ for every k, h such that $1 \leq k \leq m, 1 \leq h \leq m, k \neq h$. The macro-rule stands for:

$$\leftarrow p(X_1), p(X_2), \ldots, p(X_j), p(X_{j+1}), \Delta(j+1).$$
$$\leftarrow \text{not } p(X_1), \text{not } p(X_2), \ldots, \text{not } p(X_{j-i}), \Delta(j-i).$$

We call the former a *choice macro* and the latter a *cardinality macro*. These macros allow for compact programs without committing to a particular extension of A-Prolog (and to its inference engine). Moreover, the structure of the macros is simple enough

[2] A refinement of this statement is proven in [2].

to allow their translation, at the time of the implementation of the algorithm, to more efficient expressions, specific of the inference engine used.

Central to the execution of steps 5 and 6 of the algorithm is the notion of *hard reduct*. The hard reduct of a CR-Prolog program Π, denoted by $hr(\Pi)$, maps Π into an A-Prolog program. The importance of $hr(\Pi)$ is in the fact that *there is a one-to-one correspondence between the views of Π and the answer sets of $hr(\Pi)$* [2].

The signature of $hr(\Pi)$ is obtained from the signature of Π by the addition of predicate symbols $appl$, $is\_preferred$, $bodytrue$, o_appl, $o\_is\_preferred$, $dominates$. For simplicity we assume that none of those predicate names occurs in the signature of Π. We also assume that the signature of Π already contains the predicate name $prefer$. In the description of the hard reduct that follows, variable R, possibly indexed, ranges over the names of cr-rules.

Definition 9 (Hard Reduct of Π). *Let Π be a CR-Prolog program. The hard reduct of Π, $hr(\Pi)$, consists of:*

1. *Every regular rule from Π.*
2. *For every cr-rule $r \in cr(\Pi)$ with head h_1 OR $\ldots$ OR h_k and body $l_1, \ldots l_m, not\ l_{m+1}, \ldots, not\ l_n$, two rules: h_1 OR $\ldots$ OR $h_k \leftarrow l_1, \ldots l_m, not\ l_{m+1}, \ldots, not\ l_n, appl(r)$. and $bodytrue(r) \leftarrow l_1, \ldots l_m, not\ l_{m+1}, \ldots, not\ l_n$.*
3. *The* generator rule*, (intuitively allowing the application of arbitrary sets of cr-rules): { appl(R) }.*
4. *A constraint prohibiting the application of a cr-rule when its the body is not satisfied (intuitively corresponding to condition (3) of Definition 5):*

$$\leftarrow not\ bodytrue(R), appl(R).$$

5. *Rules defining the transitive closure of relation $prefer$:*

$$is\_preferred(R_1,R_2) \leftarrow prefer(R_1,R_2).$$
$$is\_preferred(R_1,R_2) \leftarrow prefer(R_1,R_3), is\_preferred(R_3,R_2).$$

6. *A rule prohibiting the application of cr-rules r_1 and r_2 if r_1 is preferred to r_2 (intuitively corresponding to condition (2) of Definition 5):*

$$\leftarrow appl(R_1), appl(R_2), is\_preferred(R_1,R_2).$$

Example 8. Let us compute the hard reduct of the following program, P_2:

$$\begin{cases} r_1 : p \overset{+}{\leftarrow} \text{not } q. & r_2 : s \overset{+}{\leftarrow} . \\ r_3 : \leftarrow \text{not } p, \text{not } s. & r_4 : prefer(r_1, r_2). \end{cases}$$

According to item (1) above, $hr(P_2)$ contains the regular rules r_3 and r_4. For cr-rule r_1, $hr(P_2)$ contains $\{p \leftarrow \text{not } q, appl(r_1).\quad bodytrue(r_1) \leftarrow \text{not } q.\}$. For r_2, $hr(P_2)$ contains $\{s \leftarrow appl(r_2).\quad bodytrue(r_2).\}$. Items (3 – 6) result in the addition of the rules:

$$\begin{array}{ll} \{appl(R)\}. & \leftarrow \text{not } bodytrue(R), appl(R). \\ is\_preferred(R_1,R_2) \leftarrow prefer(R_1,R_2). & \leftarrow appl(R_1), appl(R_2), is\_preferred(R_1,R_2). \\ is\_preferred(R_1,R_2) \leftarrow prefer(R_1,R_3), is\_preferred(R_3,R_2). & \end{array}$$

The answer sets of $hr(P_2)$ are:

$$\{p, appl(r_1), bodytrue(r_1), bodytrue(r_2), prefer(r_1,r_2), is\_preferred(r_1,r_2)\}$$
$$\{s, appl(r_2), bodytrue(r_1), bodytrue(r_2), prefer(r_1,r_2), is\_preferred(r_1,r_2)\}$$

corresponding to the views $\mathcal{V}_1 = \langle\{p, prefer(r_1,r_2)\},\{r_1\}\rangle, \mathcal{V}_2 = \langle\{s, prefer(r_1,r_2)\},\{r_2\}\rangle$.

In the generation step of the algorithm (line 4 from Figure 1), we find a view $\mathcal{V}$ of Π such that $\mathcal{V}^R$ has a specified cardinality i (the task of finding a *new* view satisfying the condition will be addressed later). The task is reduced to that of computing an answer set of $hr(\Pi)$ containing exactly i occurrences of atoms of the form $appl(R)$. In turn, this is reduced to finding an answer set of the *i-generator of* Π, $\gamma_i(\Pi)$, defined below.

Definition 10 (i-Generator of Π). *Let Π be a CR-Prolog program, and i a non-negative integer such that $i \leq |cr(\Pi)|$. The* i-generator of Π *is the program:* $hr(\Pi) \cup \{\ \ \leftarrow not\ i\{appl(R)\}i.\ \ \}$.

It is not difficult to show that $\gamma_i(\Pi)$ has the following properties [2]: (1) M is an answer set of $\gamma_0(\Pi)$ iff $M \cap \Sigma(\Pi)$ is an answer set of $reg(\Pi)$; (2) Every answer set of $\gamma_i(\Pi)$ is an answer set of $hr(\Pi)$; (3) Every answer set M of $\gamma_i(\Pi)$ contains exactly i atoms of the form $appl(R)$.

Example 9. Consider program P_2 from Example 8. The i-generators for P_2 for various values of i and the corresponding answer sets are as follows:

– $\gamma_0(P_2) = hr(P_2) \cup \{\ \ \leftarrow \text{not } 0\{appl(R)\}0\ \ \}$.
  The program has no answer sets, since the constraint prevents any cr-rules from being applied and the regular part of P_2 is inconsistent.
– $\gamma_1(P_2) = hr(P_2) \cup \{\ \ \leftarrow \text{not } 1\{appl(R)\}1\ \ \}$.
  The program allows the application of 1 cr-rule at a time. Its answer sets are:

$$\{p, appl(r_1), bodytrue(r_1), bodytrue(r_2), prefer(r_1,r_2), is\_preferred(r_1,r_2)\}$$
$$\{s, appl(r_2), bodytrue(r_1), bodytrue(r_2), prefer(r_1,r_2), is\_preferred(r_1,r_2)\}$$

– $\gamma_2(P_2) = hr(P_2) \cup \{\ \ \leftarrow \text{not } 2\{appl(R)\}2\ \ \}$.
  The program is inconsistent. In fact, of the only two cr-rules in P_2, one is preferred to the other, and the constraint added to $hr(P_2)$ by item (6) of Definition 9 prevents the application of two cr-rules if one of them is preferred to the other.

Intuitively, the task of generating a *new* view at each execution of line 4 of the algorithm can be accomplished, with $\gamma_i(\Pi)$, by keeping track of the answer sets of $\gamma_i(\Pi)$ found so far and by adding suitable constraints to prevent them from being generated again. More precisely, for each answer set M that has already been found, we need a constraint $\{\leftarrow \lambda(M), \nu(M).\}$ where $\lambda(M)$ is the list of the literals that occur in M and $\nu(M)$ is a list not l_1, not $l_2, \ldots,$ not l_k containing all the literals from the signature of $hr(\Pi)$ that do not belong to M. Let U be the set of the constraints for all the answer sets that have already been found. It is not difficult to see that the answer sets of the program: $\gamma_i(\Pi) \cup U$ correspond exactly to the "new" answer sets of $\gamma_i(\Pi)$.

The test step of the algorithm (line 5 from Figure 1) checks whether a view $\mathcal{V}$ found during the generation step is a candidate answer set of Π. Let M be the answer set corresponding to $\mathscr{V}$. The test is reduced to checking whether a suitable A-Prolog program is consistent. The A-Prolog program is called the *tester* for M w.r.t Π, and is defined below.

Definition 11 (Tester for M w.r.t. Π, $\tau(M,\Pi)$). *Let Π be a CR-Prolog program and M be an answer set corresponding to a view $\mathscr{V}$ of Π. The* tester *for M w.r.t. Π, $\tau(M,\Pi)$, contains:*

1. *The hard reduct of Π.*
2. *For each atom $appl(r) \in M$, a rule: $o\_appl(r)$.*
3. *For each atom $is\_preferred(r_1,r_2) \in M$, a rule: $o\_is\_preferred(r_1,r_2)$.*
4. *The rules:*

$$\begin{array}{l} dominates \leftarrow appl(R_1), o\_appl(R_2), \\ \qquad\qquad is\_preferred(R_1,R_2), o\_is\_preferred(R_1,R_2). \\ \leftarrow not\ dominates. \end{array}$$

Intuitively, relations o_appl and $o\_is\_preferred$ are used to store information about which cr-rules have been applied to obtain M and which preferences hold in the model. The first rule of item (4) above embodies the conditions of Definition 6, while the constraint enforces Definition 7.

The following is a list of important properties of $\tau(M,\Pi)$ [2]: (1) If M does not contain any atom formed by $appl$, $\tau(M,\Pi)$ is inconsistent; (2) Every answer set of $\tau(M,\Pi)$ contains an answer set of $hr(\Pi)$ (they differ only by the atoms formed by relations o_appl, $o\_is\_preferred$, and $dominates$); (3) M' is an answer set of $\tau(M,\Pi)$ iff the view corresponding to M' dominates the view encoded by M; (4) $\tau(M,\Pi)$ is inconsistent iff there exists no view of Π that dominates the view, $\mathscr{V}$, encoded by M (i.e. $\mathscr{V}$ is a candidate answer set according to Definition 7).

Example 10. Consider program P_2 from Example 8 and the answer set, M, of $\gamma_i(\Pi)$:

$$\{s, appl(r_2), bodytrue(r_1), bodytrue(r_2), prefer(r_1,r_2), is\_preferred(r_1,r_2)\}.$$

The tester for M w.r.t. P_2, $\tau(M,P_2)$ consists of $hr(P_2)$ together with (the constraint from item (4) of Definition 11 has been grounded for sake of clarity):

$$\begin{cases} o\_appl(r_2). \qquad o\_is\_preferred(r_1,r_2). \\ dominates \leftarrow appl(r_1), appl(r_2), is\_preferred(r_1,r_2), o\_is\_preferred(r_1,r_2). \\ \leftarrow \text{not } dominates. \end{cases}$$

It is not difficult to show that $\tau(M,P_2)$ has a unique answer set: $\{p, appl(r_1), bodytrue(r_1), bodytrue(r_2), prefer(r_1,r_2), is\_preferred(r_1,r_2), \quad o\_appl(r_2), o\_is\_preferred(r_1,r_2), dominates\}$. In fact, view $\mathscr{V}_1 = \langle\{s\},\{r_2\}\rangle$ is no a candidate answer set, as it is dominated by $\mathscr{V}_2 = \langle\{p\},\{r_1\}\rangle$. On the other hand, $\tau(M',P_2)$, where M' is the answer set encoding $\mathscr{V}_2$ is inconsistent, implying that $\mathscr{V}_2$ is a candidate answer set.

We can now describe the complete CRMODELS algorithm. We need the following terminology. Given an A-Prolog program Π, the set of the answer sets of Π is denoted by $\alpha_*(\Pi)$. We also define an operator $\alpha_1(\Pi)$, which non-deterministically returns one

```
Algorithm: CRMODELS
input: $\Pi$: CR-Prolog program
output: the answer sets of $\Pi$
var  $i$: number of cr-rules to be applied
     $M$: a set of literals or $\bot$
     $\mathscr{A}$: a set of answer sets of $\Pi$
     $C, C'$: sets of constraints
1. $C := \emptyset$;  $\mathscr{A} := \emptyset$
2. $i := 0$ { first we look for an answer set of $reg(\Pi)$ }
3. while $(i \leq |cr(\Pi)|)$ do   { outer loop }
4.          $C' := \emptyset$
5.          repeat   { inner loop }
6.                   if $\gamma_i(\Pi) \cup C$ is inconsistent then
7.                            $M := \bot$
8.                   else
9.                            $M := \alpha_1(\gamma_i(\Pi) \cup C)$
10.                            if $\tau(M,\Pi)$ is inconsistent then   { answer set found }
11.                                 $\mathscr{A} := \mathscr{A} \cup \{M \cap \Sigma(\Pi)\}$
12.                                 $C' := C' \cup \{ \leftarrow \lambda(M \cap atoms(appl, hr(\Pi))). \}$
13.                            end if
14.                            $C := C \cup \{ \leftarrow \lambda(M), \nu(M). \}$
15.                    end if
16.           until $M = \bot$
17.           $C := C \cup C'$
18.           $i := i + 1$ { consider views obtained with a larger number of cr-rules }
19. done
20. return $\mathscr{A}$
```

Fig. 2. Algorithm CRMODELS

of the answer sets of Π, or $\bot$ is Π is inconsistent. Recall that, given a set of literals M from the signature of $hr(\Pi)$, $\lambda(M)$ denotes the list (as opposed to the set) of the literals that occur in M and $\nu(M)$ is the list not l_1, not $l_2, \ldots,$ not l_k containing all the literals from the signature of $hr(\Pi)$ that do not belong to M.

Algorithm CRMODELS is shown in Figure 2 below. Notice that, differently from CRMODELS$_1$, CRMODELS *computes all the answer sets of the program*. The answer sets of the program are stored in the set $\mathscr{A}$. The algorithm works as follows. At the time of the first execution of line 6, the consistency of $\gamma_0(\Pi)$ is checked (C is $\emptyset$). From the properties of the i-generator, it follows that $\gamma_0(\Pi)$ is consistent iff $reg(\Pi)$ is consistent. If the test succeeds, M is set to one of the answer sets of $\gamma_0(\Pi)$ and the consistency of $\tau(M,\Pi)$ is tested. Since no cr-rules were used to generate M (i is 0), $\tau(M,\Pi)$ must be inconsistent according to the properties of $\tau(M,\Pi)$. Hence, the restriction of M to $\Sigma(\Pi)$ is added to the set of answer sets of Π, $\mathscr{A}$. Notice that the set returned corresponds to an answer set of $reg(\Pi)$, as expected. If instead $\gamma_0(\Pi)$ is inconsistent, M is set to $\bot$, the inner loop terminates and a new iteration of the outer loop is performed. When line 6 is executed again, $\gamma_1(\Pi)$ is checked for consistency. If the program is inconsistent, the algorithm proceeds to check $\gamma_2(\Pi)$, etc. On the other hand, if $\gamma_1(\Pi)$ is consistent, one of its answer sets is assigned to M and consistency of $\tau(M,\Pi)$ is tested. If the program is inconsistent, it follows that M encodes a candidate answer set (as well as an answer set, as explained at the beginning of Section 3) and its restriction to $\Sigma(\Pi)$ is returned. Finally, if instead $\tau(M,\Pi)$ is found to be consistent, the algorithm needs to prevent future computations of the answer sets of $\gamma_1(\Pi) \cup C$ (lines 6 and 9) from considering M again. This is accomplished on line 14 by adding a suitable constraint to set C.

As the algorithm computes all the answer sets of the program, CRMODELS needs to ensure that the set of cr-rules applied at each generation step is minimal. Set C' has a key role in this. As can be seen from line 12, every time an answer set of Π is found, we add to C' a constraint whose body contains the atoms of the form $appl(R)$ that occur in the answer set. The idea is to use C' to prevent any strict superset of the corresponding cr-rules from being applied in the future generation steps (lines 6 and 9). However, particular attention must be paid to the way C' is used, because each constraint in C' can prevent the generation step from using *any* superset of the corresponding cr-rules — *not only the strict supersets*. This would affect the computation when multiple answer sets exist for a fixed choice of cr-rules. Therefore, the use of the constraints added to C' during one iteration of the outer loop is delayed until the beginning of the following iteration, when the cardinality of the sets of cr-rules considered is increased by 1. This ensures that only the strict supersets of the constraints in C' are considered at all times.

Let us stress that the implementation correctly deals with preference cycles, discussed at the end of Section 2: for any cr-rule r from a preference cycle and any M such that $appl(r) \in M$, there always exists an answer set of $\tau(M,\Pi)$ (it contains M itself, together with appropriate definitions of o_appl and $o\_is\_preferred$). Hence, cr-rules from preference cycles cannot be used by the implementation to restore consistency.

The following theorems guarantee termination, soundness, and completeness of CRMODELS. The proofs cannot be shown because of space restrictions, but can be found, together with the description of the implementation of the algorithm, in [2].

Theorem 1. CRMODELS(Π) *terminates for any CR-Prolog program* Π.

Theorem 2. *For every CR-Prolog program* Π*, if* $J \in$ CRMODELS(Π)*, then* J *is an answer set of* Π.

Theorem 3. *For every CR-Prolog program* Π*, if* J *is an answer set of* Π*, then* $J \in$ CRMODELS(Π).

4 Related Work and Conclusions

There are no previous published results on the design and implementation of an inference engine for CR-Prolog. However, this paper builds on years of research on the topic, which resulted in various prototypes. Here we extend previous work by L. Kolvekar [9], where the first description of the CRMODELS algorithm was given. The algorithm and theoretical results presented here are a substantial simplification of the ones from [9].

In this paper we have described our design of an inference engine for CR-Prolog. The inference engine is aimed at allowing practical applications of CR-Prolog that require the efficient computation of the answer sets of medium-size programs.

The efficiency of CRMODELS has been demonstrated experimentally on 2000 planning problems by using a modified version of the experiment from [10]. The modification consisted in replacing the A-Prolog planning module from [10] with a CR-Prolog based module capable of finding plans that satisfy (if at all possible) 3 sets of non-trivial requirements, aimed at improving plan quality. The planning module has been tested both with and without preferences on the sets of requirements. The experiments have been successful (refer to [4] for a more detailed discussion of experiments and results): the average time to find a plan was about 200 seconds, against an average time of 10 seconds for the original A-Prolog planner[3], with an increase of about one order of magnitude in spite of the substantially more complex reasoning task (the quality of plans increased, depending on the parameters used to measure it, between 19% and 96%). Moreover, the average time obtained with the CR-Prolog planner was substantially lower than the limit for practical use by NASA, which is 20 minutes.

The proofs of the theorems in this paper and a discussion on the implementation of CRMODELS can be found in [2]. An implementation of the algorithm is available for download from `http://www.krlab.cs.ttu.edu/Software/`.

The author would like to thank Michael Gelfond for his help. This research was partially supported, over time, by United Space Alliance contract NAS9-20000, NASA contract NASA-NNG05GP48G, and ATEE/DTO contract ASU-06-C-0143.

References

1. Marcello Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS), Jun 2004.
2. Marcello Balduccini. Computing Answer Sets of CR-Prolog Programs. Technical report, Texas Tech University, 2006. http://krlab.cs.ttu.edu/∼marcy/bib.php.
3. Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.
4. Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 2006.
5. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.
6. Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Journal of Theory and Practice of Logic Programming (TPLP)*, 2005. (submitted).
7. Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, pages 413–451. Springer Verlag, Berlin, 2002.
8. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.
9. Loveleen Kolvekar. Developing an Inference Engine for CR-Prolog with Preferences. Master's thesis, Texas Tech University, Dec 2004.
10. Monica Nogueira. *Building Knowledge Systems in A-Prolog*. PhD thesis, University of Texas at El Paso, May 2003.

[3] All the experiments were run on the same computer.

Debugging ASP Programs by Means of ASP[*]

Martin Brain[1], Martin Gebser[2], Jörg Pührer[3], Torsten Schaub[2,**], Hans Tompits[3], and Stefan Woltran[3]

[1] Department of Computer Science, University of Bath, Bath, BA2 7AY, United Kingdom
`mjb@cs.bath.ac.uk`

[2] Institut für Informatik, Universität Potsdam, August-Bebel-Straße 89, D-14482 Potsdam, Germany
`{gebser,torsten}@cs.uni-potsdam.de`

[3] Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9–11, A–1040 Vienna, Austria
`{puehrer,tompits,stefan}@kr.tuwien.ac.at`

Abstract. Answer-set programming (ASP) has become an important paradigm for declarative problem solving in recent years. However, to further improve the usability of answer-set programs, the development of software-engineering tools is vital. In particular, the area of debugging provides a challenge in both theoretical and practical terms. This is due to the purely declarative nature of ASP that, on the one hand, calls for solver-independent methodologies and, on the other hand, does not directly apply to tracing techniques. In this paper, we propose a novel methodology, which rests within ASP itself, to sort out errors on the conceptual level. Our method makes use of *tagging*, where the program to be analyzed is rewritten using dedicated control atoms. This provides a flexible way to specify different types of debugging requests and a first step towards a dedicated (meta level) debugging language.

1 Introduction

Answer-set programming (ASP) has become a popular approach to declarative problem solving. The highly declarative semantics of the language decouples the problem specification from the computation of a solution. As a consequence, there is no general handle on the solving process whenever the output is in question. This deprives us of applying standard, procedural debugging methodologies and has led to a significant lack of methods and tools for debugging logic programs in ASP. However, the semantics itself allows for debugging methodologies that explain *why*, rather than *how*, a program is wrong. Another challenge is that the specification of a problem and its solutions are expressed in different fragments of the underlying language. While an encoding is usually posed in terms of predicate and variable symbols, a solution is free of variables and consists of ground atomic formulas.

We address this gap by proposing a novel debugging methodology that allows for debugging logic programs by appeal to ASP itself. To this end, we exploit and further

[*] This work was partially supported by the Austrian Science Fund (FWF) under project P18019.

[**] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

extend the *tagging technique* introduced by Delgrande, Schaub, and Tompits [1] for compiling ordered logic programs into standard ones. The idea is to compile a program in focus (once) and to subsequently accomplish different types of debugging requests by appeal to special debugging modules using dedicated control atoms, called *tags*. Tags allow for controlling the formation of answer sets and reflect different properties (like the applicability status of a rule, for instance) and therefore can be used for manipulating the evaluation of the program (like the actual application of a rule).

The basic tagging technique is then used to conceive a (meta level) *debugging language* providing dedicated debugging statements. The idea here is to first translate a program into its tagged form and then to analyze it by means of debugging statements. More specifically, starting with a program Π over an alphabet $\mathcal{A}$, Π is translated into a tagged kernel program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over an extended alphabet $\mathcal{A}^+$, and a debugging request Δ, formulated in the debugging language, is then compiled into a tagged program $\mathcal{D}[\Delta]$ over $\mathcal{A}^+$. The debugging results are eventually read off the answer sets of the combined tagged program $\mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{D}[\Delta]$. In this paper, we focus on the basic constituents of such a debugging language, confining ourselves to a detailed account of the tagging method.

Our approach has several advantageous distinct features: Firstly, it is based on meta-programming techniques that keep it within the realm of ASP. The dedicated debugging language offers an easy and modular way of specifying debugging requests. Notably, it allows the users to pose their requests with variables, which provides a tight connection to an encoding at hand. Secondly, the different debugging techniques are derived from semantic principles and relate to different characterizations of answer set formation. As a consequence, we can ascribe meaning to different debugging outcomes as well as the underlying compilation techniques. This is nicely demonstrated by our extrapolation techniques that allow for debugging incoherent logic programs. Finally, our approach has been implemented within the tool `spock`, which is publicly available at [2].

Our approach is not meant to be universal. For one thing, it aims at exploring the limits of debugging within the realm of ASP. For another, it nicely complements the majority of existing approaches, all of which are external to ASP [3,4,5,6]. Most of them rely on graph-based characterizations, in the simplest case dependency graphs, and use specific algorithms for analyzing such graphs.

2 Background

Given an alphabet $\mathcal{A}$, a (*normal*) *logic program* is a finite set of rules of form

$$a \leftarrow b_1, \dots, b_m, \mathit{not}\ c_{m+1}, \dots, \mathit{not}\ c_n\ , \tag{1}$$

where $a, b_i, c_j \in \mathcal{A}$ are *atoms* for $0 \leq i \leq m \leq j \leq n$. A *literal* is an atom a or its negation $\mathit{not}\ a$. For a rule r of form (1), let $\mathit{head}(r) = a$ be the *head* of r and $\mathit{body}(r) = \{b_1, \dots, b_m, \mathit{not}\ c_{m+1}, \dots, \mathit{not}\ c_n\}$ be the *body* of r. Furthermore, we define $\mathit{body}^+(r) = \{b_1, \dots, b_m\}$ and $\mathit{body}^-(r) = \{c_{m+1}, \dots, c_n\}$. The set of atoms occurring in a program Π is given by $At(\Pi)$. For regrouping rules sharing the same head a, we use $\mathit{def}(a, \Pi) = \{r \in \Pi \mid \mathit{head}(r) = a\}$. For uniformity, we assume that any integrity constraint $\leftarrow \mathit{body}(r)$ is expressed as a rule $w \leftarrow \mathit{body}(r), \mathit{not}\ w$, where

w is a globally new atom. Moreover, we allow nested expressions of form $not\ not\ a$, where a is some atom, in the body of rules. Such rules are identified with normal rules in which $not\ not\ a$ is replaced by $not\ a^\star$, where $a^\star$ is a globally new atom, together with an additional rule $a^\star \leftarrow not\ a$. We also take advantage of (singular) *choice rules* of form $\{a\} \leftarrow body(r)$ [7], which are an abbreviation for $a \leftarrow body(r), not\ not\ a$.

We define answer sets following the approach of Lin and Zhao [8]. Given a program Π, let $PF(\Pi) \cup CF(\Pi)$ be the *completion* of Π [9], where

$$PF(\Pi) = \left\{ body(r) \rightarrow head(r) \mid r \in \Pi \right\} \text{ and}$$
$$CF(\Pi) = \left\{ a \rightarrow \textstyle\bigvee_{r \in def(a,\Pi)} body(r) \mid a \in \mathcal{A} \right\}.[1]$$

A *loop* is a (non-empty) set of atoms that circularly depend upon each other in a program's positive atom dependency graph [8]. Programs having an acyclic positive atom dependency graph are *tight* [10]. The *loop formula* associated with a loop L is

$$LF(\Pi, L) = \neg \left(\textstyle\bigvee_{r \in R(\Pi,L)} body(r) \right) \rightarrow \textstyle\bigwedge_{a \in L} \neg a \ ,$$

where $R(\Pi, L) = \{r \in \Pi \mid head(r) \in L,\ body^+(r) \cap L = \emptyset\}$. We denote the set of all loops in Π by $loop(\Pi)$. The set of all loop formulas of Π is $LF(\Pi) = \{LF(\Pi, L) \mid L \in loop(\Pi)\}$. A set X of atoms is an *answer set* of a logic program Π iff X is a model of $PF(\Pi) \cup CF(\Pi) \cup LF(\Pi)$. If Π is tight, then the answer sets of Π coincide with the models of $PF(\Pi) \cup CF(\Pi)$ (models of the latter are also referred to as the *supported models* of Π). The set Π_X of *generating rules* of a set X of atoms from program Π is defined as $\{r \in \Pi \mid body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}$.

As an example, consider $\Pi_1 = \{a \leftarrow;\ c \leftarrow not\ b, not\ d;\ d \leftarrow a, not\ c\}$ and its two answer sets $\{a, c\}$ and $\{a, d\}$. The completion of Π_1 is logically equivalent to $a \wedge \neg b \wedge (c \leftrightarrow \neg b \wedge \neg d) \wedge (d \leftrightarrow a \wedge \neg c)$; its models coincide with the answer sets of Π_1. Adding $\{b \leftarrow e;\ e \leftarrow b\}$ to Π_1 induces the loop $\{b, e\}$ but leaves the set of answer sets of Π_1 intact. Unlike this, the completion of Π_1 becomes $a \wedge (b \leftrightarrow e) \wedge (c \leftrightarrow \neg b \wedge \neg d) \wedge (d \leftrightarrow a \wedge \neg c)$ and admits an additional model $\{a, b, d, e\}$. This supported model violates the loop formula $LF(\Pi_1, \{b, e\}) = \top \rightarrow \neg b \wedge \neg e$, which denies it the status of an answer set.

3 Debugging Modules

Our approach relies on the *tagging technique* introduced by Delgrande, Schaub, and Tompits [1] for compiling ordered logic programs back into normal programs. The idea is to rewrite a program by introducing so-called *tags* that allow for controlling the formation of answer sets. More formally, given a logic program Π over $\mathcal{A}$ and a set $\mathcal{N}$ of names for all rules in Π, we consider an enriched alphabet $\mathcal{A}^+$ obtained from $\mathcal{A}$ by adding new pairwise distinct propositional atoms such as $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, $\mathsf{ok}(n_r)$, $\mathsf{ko}(n_r)$, etc., where $n_r \in \mathcal{N}$ for each $r \in \Pi$. Intuitively, $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ express whether a rule r is applicable or blocked, respectively, while $\mathsf{ok}(n_r)$ and $\mathsf{ko}(n_r)$ are used for manipulating the application of r. Further tags are introduced in the sequel.

[1] Strictly speaking, $CF(\Pi)$ should take $\mathcal{A}$ as additional argument; for simplicity, we leave this implicit. Moreover, $body(r)$ is understood as a conjunction of (classical) literals within $PF(\Pi)$, $CF(\Pi)$, and $LF(\Pi, L)$ in what follows.

3.1 Kernel Debugging Module

Our *kernel translation*, $\mathcal{T}_{\mathsf{K}}$, decomposes rules of a given program such that they can be accessed by tags for controlling purposes.

Definition 1. *Let Π be a logic program over $\mathcal{A}$. Then, the program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for $r \in \Pi$, $b \in body^+(r)$, and $c \in body^-(r)$:*

$$\begin{array}{rlrl} head(r) & \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r)\ , & \mathsf{bl}(n_r) & \leftarrow \mathsf{ok}(n_r), not\ b\ , \\ \mathsf{ap}(n_r) & \leftarrow \mathsf{ok}(n_r), body(r)\ , & \mathsf{bl}(n_r) & \leftarrow \mathsf{ok}(n_r), not\ not\ c\ , \\ & & \mathsf{ok}(n_r) & \leftarrow not\ \overline{\mathsf{ok}}(n_r)\ . \end{array}$$

An auxiliary atom $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, or $\mathsf{ok}(n_r)$, respectively, occurs in an answer set X of $\mathcal{T}_{\mathsf{K}}[\Pi]$ only if $r \in \Pi$. Also, for any $r \in \Pi$, there is a priori no atom $\mathsf{ko}(n_r)$ contained in an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$, whereas $\mathsf{ok}(n_r)$ is contained in any answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$ by default. The role of $\overline{\mathsf{ok}}(n_r)$ is to implement potential changes of this default behavior.

The following proposition collects more interesting relations.

Proposition 1. *Let Π be a logic program over $\mathcal{A}$ and X an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$. Then, for any $r \in \Pi$ and $a \in \mathcal{A}$:*

1. $\mathsf{ap}(n_r) \in X$ *iff* $r \in \Pi_X$ *iff* $\mathsf{bl}(n_r) \notin X$*;*
2. *if $a \in X$, then $\mathsf{ap}(n_r) \in X$ for some $r \in def(a, \Pi)$;*
3. *if $a \notin X$, then $\mathsf{bl}(n_r) \in X$ for all $r \in def(a, \Pi)$.*

The relation between auxiliary atoms and original atoms from $\mathcal{A}$ is described next.

Theorem 1. *Let Π be a logic program over $\mathcal{A}$. We have a one-to-one correspondence between the answer sets of Π and $\mathcal{T}_{\mathsf{K}}[\Pi]$ satisfying the following conditions:*

1. *If X is an answer set of Π, then*
$$X \cup \{\mathsf{ok}(n_r) \mid r \in \Pi\} \cup \{\mathsf{ap}(n_r) \mid r \in \Pi_X\} \cup \{\mathsf{bl}(n_r) \mid r \in \Pi \setminus \Pi_X\}$$
is an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$.
2. *If Y is an answer set of $\mathcal{T}_{\mathsf{K}}[\Pi]$, then $(Y \cap \mathcal{A})$ is an answer set of Π.*

3.2 Extrapolating Non-existing Answer Sets

Whenever a program Π has no answer set, this means in terms of the characterization by Lin and Zhao [8] that there is no interpretation jointly satisfying $PF(\Pi)$, $CF(\Pi)$, and $LF(\Pi)$. In other words, each interpretation X over $\mathcal{A}$ causes at least one of the following problems:

- If X falsifies $PF(\Pi)$, then there is some rule r in Π such that $body^+(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$, but $head(r) \notin X$.
- If X falsifies $CF(\Pi)$, then there is some atom a in X that lacks a supporting rule, that is, $body^+(r) \not\subseteq X$ or $body^-(r) \cap X \neq \emptyset$ for each $r \in def(a, \Pi)$.
- If X falsifies $LF(\Pi)$, then X contains some loop L in Π that is unfounded with respect to X, that is, $X \not\models LF(\Pi, L)$.

This intuition is captured in the debugging model described below. It aims at analyzing incoherent situations by figuring out which rules or atoms cause some of the aforementioned problems. The names of the translations, $\mathcal{T}_\mathsf{P}$, $\mathcal{T}_\mathsf{C}$, and $\mathcal{T}_\mathsf{L}$, reflect their respective purpose, indicating problems originating from the program, its completion, or its (non-trivial) loop formulas. We use abnormality atoms with a corresponding naming schema to indicate the respective problem: $\mathsf{ab}_p(n_r)$ signals that rule r is falsified under some interpretation, $\mathsf{ab}_c(a)$ points out that atom a is true but has no support, and $\mathsf{ab}_l(a)$ aims at indicating an unfounded atom a.

Definition 2. *Let Π be a logic program over $\mathcal{A}$ and A a set of atoms. Then:*

1. *The logic program $\mathcal{T}_\mathsf{P}[\Pi]$ over $\mathcal{A}^+$ consists of the following rules, for all $r \in \Pi$:*

$$\{\, head(r) \,\} \leftarrow \mathsf{ap}(n_r)\ , \qquad \mathsf{ab}_p(n_r) \leftarrow \mathsf{ap}(n_r), not\ head(r)\ ,$$
$$\mathsf{ko}(n_r) \leftarrow\ .$$

2. *The logic program $\mathcal{T}_\mathsf{C}[\Pi, A]$ over $\mathcal{A}^+$ consists of the following rules, for all $a \in A$, where $\{r_1, \ldots, r_k\} = def(a, \Pi)$:*

$$\{\, a \,\} \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k})\ , \qquad \mathsf{ab}_c(a) \leftarrow a, \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k})\ .$$

3. *The logic program $\mathcal{T}_\mathsf{L}[A]$ over $\mathcal{A}^+$ consists of the following rules, for all $a \in A$:*

$$\{\, \mathsf{ab}_l(a) \,\} \leftarrow not\ \mathsf{ab}_c(a)\ , \qquad a \leftarrow \mathsf{ab}_l(a)\ .$$

The purpose of adding facts $(\mathsf{ko}(n_r) \leftarrow)$ in $\mathcal{T}_\mathsf{P}[\Pi]$ is to avoid the application of the rule $(head(r) \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r))$ in $\mathcal{T}_\mathsf{K}[\Pi]$ (rather than to enforce a re-compilation in conjunction with $\mathcal{T}_\mathsf{K}[\Pi]$). Regarding $\mathcal{T}_\mathsf{C}[\Pi, A]$, note that $def(a, \Pi)$ might be empty, in which case we obtain for a the choice rule $(\{\, a \,\} \leftarrow)$ along with $(\mathsf{ab}_c(a) \leftarrow a)$. Observe that $\mathcal{T}_\mathsf{L}[A]$ allows us to add $\mathsf{ab}_l(a)$ to an answer set if a is supported. In contrast to $\mathsf{ab}_p(n_r)$ in $\mathcal{T}_\mathsf{P}[\Pi]$ and $\mathsf{ab}_c(a)$ in $\mathcal{T}_\mathsf{C}[\Pi, A]$, indicating violations of $PF(\Pi)$ or $CF(\Pi)$, respectively, the presence of $\mathsf{ab}_l(a)$ in an answer set does not necessarily indicate the violation of any loop formula in $LF(\Pi)$. In fact, as the number of loops for Π can be exponential, we cannot reasonably check loop formula violations within $\mathcal{T}_\mathsf{L}[A]$ (via an additional argument Π and tagged rules for analyzing loop formulas). Rather, as discussed below, we filter occurrences of $\mathsf{ab}_l(a)$ in answer sets by *minimization*.

Next, we put things together. The answer sets of the subsequent translation are thought of as extrapolations of putative yet non-existing answer sets of the original program Π. That is, an atom $\mathsf{ab}_p(n_r)$ signals that an answer set could be obtained if rule r was not contained in Π. Dually, $\mathsf{ab}_c(a)$ indicates that an answer set could be obtained if atom a would be supported, that is, if it would be derivable by some rule. Finally, $\mathsf{ab}_l(a)$ points to the violation of a loop formula that involves a. Moreover, we provide further possibilities to parametrize a debugging request by two additional arguments, Π' and A. Hereby, Π' restricts the set of rules (from program Π) whose violation is tolerated for debugging purposes, while A restricts the atoms that can be assumed true though being unsupported or belonging to a (non-trivial) unfounded set.

Definition 3. *Let Π be a logic program over $\mathcal{A}$, $\Pi' \subseteq \Pi$, and $A \subseteq At(\Pi)$. Then:*

$$\mathcal{T}_{\mathsf{E}}[\Pi, \Pi', A] = \mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{T}_{\mathsf{P}}[\Pi'] \cup \mathcal{T}_{\mathsf{C}}[\Pi, A] \cup \mathcal{T}_{\mathsf{L}}[A] \ .$$

Moreover, let $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi'] = \mathcal{T}_{\mathsf{E}}[\Pi, \Pi', At(\Pi')]$ and $\mathcal{T}_{\mathsf{E}}[\Pi] = \mathcal{T}_{\mathsf{E}}[\Pi, \Pi, At(\Pi)]$.

Note that $\mathcal{T}_{\mathsf{L}}[A]$ can be omitted in the definition of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi', A]$ if Π is tight.

We list some basic properties first.

Proposition 2. *Let Π be a logic program over $\mathcal{A}$ and X an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$. Then, for each $r \in \Pi$:*

1. $\mathsf{ab}_p(n_r) \in X$ *iff* $\mathsf{ap}(n_r) \in X$, $\mathsf{bl}(n_r) \notin X$, *and* $head(r) \notin X$;
2. $\mathsf{ab}_p(n_r) \notin X$ *if* $\mathsf{ab}_c(head(r)) \in X$ *or* $\mathsf{ab}_l(head(r)) \in X$.

Moreover, for every $a \in At(\Pi)$, it holds that:

1. $\mathsf{ab}_c(a) \in X$ *and* $\mathsf{ab}_l(a) \notin X$ *iff* $a \in X$ *and* $(X \cap \mathcal{A}) \not\models (\bigvee_{r \in def(a,\Pi)} body(r))$;
2. $\mathsf{ab}_c(a) \notin X$ *if* $a \in X$ *and* $(X \cap \mathcal{A}) \models (\bigvee_{r \in def(a,\Pi)} body(r))$;
3. $\mathsf{ab}_c(a) \notin X$ *and* $\mathsf{ab}_l(a) \notin X$ *if* $a \notin X$;
4. $\mathsf{ab}_c(a) \notin X$ *if* $\mathsf{ab}_l(a) \in X$.

The next result shows that abnormality-free answer sets of the translated program correspond to the answer sets of the original program.[2] To this end, we introduce, for a program Π, $\mathsf{AB}(\Pi) = (\{\mathsf{ab}_p(n_r) \mid r \in \Pi\} \cup \{\mathsf{ab}_c(a), \mathsf{ab}_l(a) \mid a \in At(\Pi)\})$.

Theorem 2. *Let Π be a logic program over $\mathcal{A}$. Then, it holds that:*

1. *If X is an answer set of Π, then*

$$X \cup \{\mathsf{ok}(n_r), \mathsf{ko}(n_r) \mid r \in \Pi\} \cup \{\mathsf{ap}(n_r) \mid r \in \Pi_X\} \cup \{\mathsf{bl}(n_r) \mid r \in \Pi \setminus \Pi_X\}$$

   is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$.
2. *If Y is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ such that $(Y \cap \mathsf{AB}(\Pi)) = \emptyset$, then $(Y \cap \mathcal{A})$ is an answer set of Π.*

The interesting case, however, is when the original program is incoherent. For illustrating this, let us consider three simple examples. To begin with, consider:

$$\Pi_2 = \{\ n_{r_1} : a \leftarrow,\ n_{i_1} : \leftarrow a\ \} \ .$$

The program $\mathcal{T}_{\mathsf{K}}[\Pi_2]$ consists of the following rules:

$$\begin{array}{rlrl}
a & \leftarrow \mathsf{ap}(n_{r_1}), not\ \mathsf{ko}(n_{r_1}) \ , & & \leftarrow \mathsf{ap}(n_{i_1}), not\ \mathsf{ko}(n_{i_1}) \ , \\
\mathsf{ap}(n_{r_1}) & \leftarrow \mathsf{ok}(n_{r_1}) \ , & \mathsf{ap}(n_{i_1}) & \leftarrow \mathsf{ok}(n_{i_1}), a \ , \\
 & & \mathsf{bl}(n_{i_1}) & \leftarrow \mathsf{ok}(n_{i_1}), not\ a \ , \\
\mathsf{ok}(n_{r_1}) & \leftarrow not\ \overline{\mathsf{ok}}(n_{r_1}) \ , & \mathsf{ok}(n_{i_1}) & \leftarrow not\ \overline{\mathsf{ok}}(n_{i_1}) \ .
\end{array}$$

[2] Due to relaxing Π by tolerating abnormalities, $\mathcal{T}_{\mathsf{E}}[\Pi]$ always admits (abnormal) answer sets.

We obtain $\mathcal{T}_{\mathsf{E}}[\Pi_2, \{r_1\}]$ by adding $(\mathcal{T}_{\mathsf{P}}[\{r_1\}] \cup \mathcal{T}_{\mathsf{C}}[\Pi_2, \{a\}] \cup \mathcal{T}_{\mathsf{L}}[\{a\}])$, given next:

$$\begin{array}{rlrl}
\mathsf{ko}(n_{r_1}) \leftarrow & , & \{\, a \,\} \leftarrow & \mathsf{bl}(n_{r_1}) \ , \\
 & & \mathsf{ab}_c(a) \leftarrow & a, \mathsf{bl}(n_{r_1}) \ , \\
\{\, a \,\} \leftarrow & \mathsf{ap}(n_{r_1}) \ , & \{\, \mathsf{ab}_l(a) \,\} \leftarrow & \mathit{not}\ \mathsf{ab}_c(a) \ , \\
\mathsf{ab}_p(n_{r_1}) \leftarrow & \mathsf{ap}(n_{r_1}), \mathit{not}\ a \ , & a \leftarrow & \mathsf{ab}_l(a) \ .
\end{array}$$

The unique answer set of $\mathcal{T}_{\mathsf{E}}[\Pi_2, \{r_1\}]$ is:[3]

$$\{\underline{\mathsf{ab}_p(n_{r_1})}, \mathsf{ap}(n_{r_1}), \mathsf{bl}(n_{i_1}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{i_1}), \mathsf{ko}(n_{r_1})\} \ . \tag{2}$$

Note that applying the modules from Definition 2 only to subprogram $\{r_1\}$ makes us focus on answer set candidates satisfying the residual program $\{i_1\}$. The abnormality tag $\mathsf{ab}_p(n_{r_1})$ signals that, in order to obtain an answer set, rule r_1 must not be applied.

As another example, consider:

$$\Pi_3 = \{\ n_{r_1} : a \leftarrow b,\ n_{i_1} : \ \leftarrow \mathit{not}\ a \ \} \ .$$

Program $\mathcal{T}_{\mathsf{E}}[\Pi_3, \{r_1\}, \{a\}]$ has the unique answer set:

$$\{a, \underline{\mathsf{ab}_c(a)}, \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{i_1}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{i_1}), \mathsf{ko}(n_{r_1})\} \ , \tag{3}$$

indicating that a lacks supporting rules.

Finally, consider the following program:

$$\Pi_4 = \{\ n_{r_1} : a \leftarrow b,\ n_{r_2} : b \leftarrow a,\ n_{i_1} : \ \leftarrow \mathit{not}\ a \ \} \ .$$

Program $\mathcal{T}_{\mathsf{E}}[\Pi_4, \{r_1, r_2\}]$ has four answer sets (omitting $\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{i_1})$):

$$\{a, \underline{\mathsf{ab}_p(n_{r_2})}, \underline{\mathsf{ab}_c(a)}, \mathsf{bl}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ , \tag{4}$$
$$\{a, b, \underline{\mathsf{ab}_l(a)}, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ , \tag{5}$$
$$\{a, b, \underline{\mathsf{ab}_l(b)}, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ , \tag{6}$$
$$\{a, b, \underline{\mathsf{ab}_l(a)}, \underline{\mathsf{ab}_l(b)}, \mathsf{ap}(n_{r_1}), \mathsf{ap}(n_{r_2}), \mathsf{bl}(n_{i_1}), \mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2})\} \ . \tag{7}$$

The last example illustrates that the relaxation mechanisms underlying translation $\mathcal{T}_{\mathsf{E}}$ can lead to overly involved explanations of the source of incoherence, as manifested by the first and last answer set (cf. (4) and (7)). Therefore, we suggest focusing on answer sets containing a *minimum number* of instances of ab predicates.[4] In the last case, this gives the second and third answer set (cf. (5) and (6)). Indeed, adding only one fact, either $(a \leftarrow)$ or $(b \leftarrow)$, to the (untagged) incoherent program Π_4 makes it coherent.

The next results shed some more light on the semantic links between the original and the transformed program, whenever the former admits no answer set.

[3] In what follows, we underline abnormality tags.

[4] This can be implemented via `minimize` statements as available in `Smodels`.

Theorem 3. *Let Π be a logic program over $\mathcal{A}$. Then, it holds that:*

1. *If Y is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ and $\mathsf{ab}_p(n_r) \in Y$, then $(Y \cap \mathcal{A}) \not\models (body(r) \rightarrow head(r))$, where $(body(r) \rightarrow head(r)) \in PF(\Pi)$;*
2. *If Y is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ and $\mathsf{ab}_c(a) \in Y$, then $(Y \cap \mathcal{A}) \not\models (a \rightarrow \bigvee_{r \in def(a,\Pi)} body(r))$, where $(a \rightarrow \bigvee_{r \in def(a,\Pi)} body(r)) \in CF(\Pi)$;*
3. *If Y is an answer set of $\mathcal{T}_{\mathsf{E}}[\Pi]$ such that, for some $L \in loop(\Pi)$, we have $L \subseteq (Y \cap \mathcal{A})$, $(Y \cap \mathcal{A}) \not\models LF(\Pi, L)$, and $(Y \cap \mathcal{A}) \models (\bigvee_{r \in def(a,\Pi)} body(r))$ for every $a \in L$, then $\mathsf{ab}_l(a') \in Y$ for some $a' \in L$.*

The same results hold for partial compilations, $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi', A]$, but are omitted for brevity.

For illustration, let us return to the last three examples. Intersecting the only answer set (2) of $\mathcal{T}_{\mathsf{E}}[\Pi_2, \{r_1\}]$ with the alphabet of Π_2 yields the empty set. We obtain $\emptyset \not\models (a \leftarrow)$, as indicated by $\mathsf{ab}_p(n_{r_1})$ in (2). Note that the empty set is the only subset of $At(\Pi_2)$ that satisfies integrity constraint $i_1 \in \Pi_2$. Proceeding analogously with the only answer set (3) of $\mathcal{T}_{\mathsf{E}}[\Pi_3, \{r_1\}, \{a\}]$ yields $\{a\}$, and we obtain $\{a\} \not\models (a \rightarrow b)$, as signaled by $\mathsf{ab}_c(a)$ in (3). In fact, $\{a\}$ is the only subset of the atoms subject to extrapolation that satisfies integrity constraint $i_1 \in \Pi_3$.

Finally, consider the two abnormality-minimum answer sets (5) and (6) of $\mathcal{T}_{\mathsf{E}}[\Pi_4, \{r_1, r_2\}]$. Both are actually symmetrical since they refer to the same loop $\{a, b\}$ through different elements, as indicated by $\mathsf{ab}_l(a)$ and $\mathsf{ab}_l(b)$, respectively. Hence, both answer sets (5) and (6) of $\mathcal{T}_{\mathsf{E}}[\Pi_4, \{r_1, r_2\}]$ induce candidate set $\{a, b\}$, which falsifies its own loop formula: $\{a, b\} \not\models (\top \rightarrow \neg a \wedge \neg b)$. Note that $\{a\}$ is actually another candidate subset of $At(\Pi_4)$. However, this candidate suffers from two abnormalities, as indicated by the non-minimum answer set (4) through $\mathsf{ab}_p(n_{r_2})$ and $\mathsf{ab}_c(a)$. In fact, we have $\{a\} \not\models (b \leftarrow a)$, violating r_2, and $\{a\} \not\models (a \rightarrow b)$, violating the completion.

The next result captures a more realistic scenario, in which only a subset of a program is subject to extrapolation and only abnormality-minimum answer sets of the translation are considered. From the perspective of an original program Π, the abnormality-minimum answer sets of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi']$ provide us with the candidate sets among $At(\Pi)$ that satisfy the requirement of being an answer set of Π under a minimum number of *repairs* on Π'. A repair is either the deletion of a rule r or an addition of a fact $(a \leftarrow)$ (which prevents a from being not supported or unfounded). The former repair refers to $\mathsf{ab}_p(n_r)$, which is clearly avoided when r is deleted, and the latter to $\mathsf{ab}_c(a)$ or $\mathsf{ab}_l(a)$, since $(a \rightarrow \bigvee_{r \in def(a, \Pi \cup \{a \leftarrow\})} body(r))$ and $LF(\Pi \cup \{a \leftarrow\}, L)$, for any loop L containing a, then amount to $(a \rightarrow \top)$ and $(\bot \rightarrow \bigwedge_{a \in L} \neg a)$.

Theorem 4. *Let Π be a logic program over $\mathcal{A}$ and (Π_1, Π_2) a partition of Π such that $(\{head(r_1) \mid r_1 \in \Pi_1\} \cap At(\Pi_2)) = \emptyset$. Furthermore, let $\mathcal{M}$ be the set of all answer sets Y of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi_2]$ such that the cardinality of $(Y \cap \mathsf{AB}(\Pi_2))$ is minimum among all answer sets of $\mathcal{T}_{\mathsf{E}}[\Pi, \Pi_2]$. Then, it holds that:*

1. *If $Y \in \mathcal{M}$, then $(Y \cap \mathcal{A})$ satisfies all formulas in $(PF(\Pi_1) \cup (CF(\Pi_1) \setminus \{a \rightarrow \bot \mid a \in At(\Pi_2)\}) \cup LF(\Pi_1))$ and all formulas in $(PF(\Pi_2) \cup CF(\Pi_2) \cup LF(\Pi_2))$ under a minimum number of repairs on Π_2;*
2. *If $X \subseteq \mathcal{A}$ satisfies all formulas in $(PF(\Pi_1) \cup (CF(\Pi_1) \setminus \{a \rightarrow \bot \mid a \in At(\Pi_2)\}) \cup LF(\Pi_1))$ and all formulas in $(PF(\Pi_2) \cup CF(\Pi_2) \cup LF(\Pi_2))$ under a minimum number of repairs on Π_2, then there is a $Y \in \mathcal{M}$ such that $X = (Y \cap \mathcal{A})$.*

3.3 Ordering Rule Applications

Our last debugging component allows for imposing an order on the rule applications. To this end, we make use of ordered logic programs following the framework of Delgrande, Schaub, and Tompits [1].

Definition 4. *Let Π be a logic program over $\mathcal{A}$ and let $<$ be a strict partial order over Π. Then, $\mathcal{T}_o[\Pi, <]$ consists of the following rules, for all $r, r' \in \Pi$ with $r < r'$ and $\{r_1, \ldots, r_k\} = \{r'' \mid r < r''\}$:*

$$\begin{array}{ll} \overline{\mathsf{ok}}(n_r) \leftarrow \ , & \mathsf{rdy}(n_r, n_{r'}) \leftarrow \mathsf{ap}(n_{r'}) \ , \\ \mathsf{ok}(n_r) \leftarrow \mathsf{rdy}(n_r, n_{r_1}), \ldots, \mathsf{rdy}(n_r, n_{r_k}) \ , & \mathsf{rdy}(n_r, n_{r'}) \leftarrow \mathsf{bl}(n_{r'}) \ . \end{array}$$

Furthermore, $\mathcal{T}_o[<]$ is a shortcut for $\mathcal{T}_o[\Pi_<, <]$, where $\Pi_< = \{r, r' \in \Pi \mid r < r'\}$.

The order $<$ on Π singles out the answer sets of Π whose rule application and/or blockage is compatible with $<$. That is, given the order $r_1 < r_2$, the higher-ranked rule r_2 must be applied or found to be blocked before r_1. In other words, $<$ selects those answer sets X of Π that can be generated by an appropriate sequence of elements of Π_X. We refer for a detailed formal elaboration to Delgrande, Schaub, and Tompits [1].[5]

3.4 Debugging Programs with Variables

With two exceptions, the translations discussed so far carry over to programs with variables simply by using parametrized names. To this end, replace every occurrence of a name n_r by $n_r(\boldsymbol{X})$, where $\boldsymbol{X}$ is the sequence of variables occurring in rule r. Accordingly, a rule $(r : p(X, Y) \leftarrow q(1, X), \mathit{not}\ s(Y))$ gets the name $n_r(X, Y)$, for instance, yielding

$$\begin{array}{ll} \mathsf{ap}(n_r(X, Y)) \leftarrow \mathsf{ok}(n_r(X, Y)), q(1, X), \mathit{not}\ s(Y) & \text{and} \\ \quad \{\, p(X, Y) \,\} \leftarrow \mathsf{ap}(n_r(X, Y)) \ . & \end{array}$$

Similarly, we get for an atomic formula $p(X, Y)$:

$$\{\, \mathsf{ab}_l(p(X, Y)) \,\} \leftarrow \mathit{not}\ \mathsf{ab}_c(p(X, Y)) \ .$$

The final representation (e.g., with or without function symbols) of an atomic formula like $\mathsf{ab}_l(p(X, Y))$ depends on the language capacities of the target ASP solver.

The first exception is resolved by replacing the rules in Item 2 of Definition 2 by

$$\{\, a(\boldsymbol{X}) \,\} \leftarrow \mathit{not}\ \mathsf{apx}(a(\boldsymbol{X})) \quad \text{and} \quad \mathsf{ab}_c(a(\boldsymbol{X})) \leftarrow a(\boldsymbol{X}), \mathit{not}\ \mathsf{apx}(a(\boldsymbol{X})) \ ,$$

and by adding, for each $r \in \mathit{def}(a, \Pi)$, the rule

$$\mathsf{apx}(a(\boldsymbol{X})) \leftarrow \mathsf{ap}(n_r(\boldsymbol{Y})) \ ,$$

[5] For obtaining the precise semantics of [1], predicate bl must actually be replaced by another predicate $\mathsf{bl}^\star$ using no nested expressions like $\mathit{not}\ \mathit{not}\ c$. The definition of $\mathsf{bl}^\star$ is omitted here.

where apx is an auxiliary predicate symbol and $\boldsymbol{X}$ is a subsequence of the variables in $\boldsymbol{Y}$. An atom $\mathsf{apx}(a(\boldsymbol{c}))$ indicates that $a(\boldsymbol{c})$ is not derivable given that all putative rules in $\mathit{def}(a, \Pi)$ are inapplicable.

The second exception concerns translation $\mathcal{T}_o$ which necessitates that the set of dominating rules $\{r'' \mid r < r''\}$ is ground in order to guarantee that all instances have been applied or blocked (cf. Definition 4). Unlike this, the name of r, n_r, can be parametrized.

Apart from firmer instantiations, further optimizations are possible by use of domain predicates for names and atoms, like name and atom. For instance, a rule $(\mathsf{ok}(n_r) \leftarrow \mathit{not}\ \overline{\mathsf{ok}}(n_r))$ could be represented as $(\mathsf{ok}(X) \leftarrow \mathit{name}(X), \mathit{not}\ \overline{\mathsf{ok}}(X))$.

3.5 Implementation

The tool `spock` implements a collection of methods for debugging ASP programs, following the ideas introduced above. The system can be used either with `DLV` [11] or with `Smodels` [7] (together with `Lparse`) and is obtainable at [2], where also some further information about the system is available.

The tool is written in Java 5.0 and published under the GNU General Public License. The input of `spock` is a logic program in the core language of either `DLV` or `Smodels`, read from the system standard input and/or from (multiple) files. The output varies according to the selected functionalities, determined by a set of options. The most important syntax extension of the input programs, however, is the labeling of rules, allowing debugging mechanisms to explicitly refer to certain rules.

4 Elements of a Debugging Language

In order to enhance the usability and convenience of our debugging technique, we provide in this section some basic elements for a higher-level debugging language. The idea is to compile a program once and to explore it subsequently by means of debugging statements. To this end, we assume that the program Π has been compiled into $\mathcal{T}_{\mathsf{K}}[\Pi]$. A debugging request, Δ, formulated in the debugging language, is compiled by means of a function $\mathcal{D}$ into a tagged program $\mathcal{D}[\Delta]$ such that the debugging results are read off the answer sets of the combined tagged program $\mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{D}[\Delta]$. We stress that the subsequent discussion is intended as a starting point only, given at a rather informal level; a detailed elaboration of the full language will be explored elsewhere.

The first type of expressions are referred to as *enforcement statements* since they may alter the original set of answer sets. The following expressions are enforcement statements (in what follows, let C be a set of literals over $At(\Pi)$ and n_r the name of some $r \in \Pi$):

– **block** n_r **if** C;
– **apply** n_r **if** C;
– **assign** $a_1 = v_1, \ldots, a_k = v_k$ **if** C, where $a_i \in At(\Pi)$ and $v_i \in \{\mathbf{t}, \mathbf{f}\}$ for $i = 1, \ldots, k$.

The semantics of enforcement statements is given via a compilation function $\mathcal{D}$:

- $\mathcal{D}[\textbf{block}\ n_r\ \textbf{if}\ C] = \{\mathsf{ko}(n_r) \leftarrow C,\ \underline{\mathsf{bl}}(n_r) \leftarrow C,\ \overline{\mathsf{ok}}(n_r) \leftarrow C\}$;
- $\mathcal{D}[\textbf{apply}\ n_r\ \textbf{if}\ C] = \{\mathsf{ap}(n_r) \leftarrow C,\ \overline{\mathsf{ok}}(n_r) \leftarrow C\}$;
- $\mathcal{D}[\textbf{assign}\ a_1 = v_1, \ldots, a_k = v_k\ \textbf{if}\ C] = \{a_i \leftarrow C \mid v_i = \mathbf{t}\} \cup \{\leftarrow a_i, C \mid v_i = \mathbf{f}\}$.

For example, $\textbf{block}\ n_r(X)\ \textbf{if}\ \textit{not}\ \ (X > 5)$ blocks rule r unless instantiated with objects larger than 5. Also, one may use tags within the precondition, as in $\textbf{block}\ n_r(X)\ \textbf{if}\ \mathsf{ap}(n_s(X))$.

Unlike the above, *projection statements* do not alter the original set of answer sets but return a specific subset of the original answer sets:

- $\textbf{blocked}\ n_1, \ldots, n_k\ \textbf{if}\ C$;
- $\textbf{applied}\ n_1, \ldots, n_k\ \textbf{if}\ C$;
- $\textbf{assigned}\ a_1 = v_1, \ldots, a_k = v_k\ \textbf{if}\ C$, where $a_i \in At(\Pi)$ and $v_i \in \{\mathbf{t}, \mathbf{f}\}$ for $i = 1, \ldots, k$.

The semantics of projection statements is again given in terms of a compilation:

- $\mathcal{D}[\textbf{blocked}\ n_1, \ldots, n_k\ \textbf{if}\ C] = \{\leftarrow \textit{not}\ \mathsf{bl}(n_1), \ldots, \textit{not}\ \mathsf{bl}(n_k), C\}$;
- $\mathcal{D}[\textbf{applied}\ n_1, \ldots, n_k\ \textbf{if}\ C] = \{\leftarrow \textit{not}\ \mathsf{ap}(n_1), \ldots, \textit{not}\ \mathsf{ap}(n_k), C\}$;
- $\mathcal{D}[\textbf{assigned}\ a_1 = v_1, \ldots, a_k = v_k\ \textbf{if}\ C] = \{\leftarrow \textit{not}\ a_i, C \mid v_i = \mathbf{t}\} \cup \{\leftarrow a_i, C \mid v_i = \mathbf{f}\}$.

For example, $\textbf{blocked}\ n_r(X), n_s(Y)\ \textbf{if}\ X \neq Y$ eliminates answer sets to which rules r and s contribute with different instantiations. Applying the compilation function $\mathcal{D}$, we get

$$\begin{aligned}\mathcal{D}[\textbf{blocked}\ n_r(X), n_s(Y)\ \textbf{if}\ X \neq Y] &= \\ &\{\leftarrow \textit{not}\ \mathsf{bl}(n_r(X)), \textit{not}\ \mathsf{bl}(n_s(Y)), X \neq Y\}.\end{aligned}$$

Observe that the translation of the corresponding enforcement statement would yield multiple rules whose instantiations cannot be controlled in a synchronous fashion.

Next, we introduce expressions for analyzing incoherent situations by extrapolation, correspondingly called *extrapolation statements*:

- $\textbf{extrapolate}\ n_r\ \textbf{if}\ C$;
- $\textbf{extrapolate\_}x\ s\ \textbf{if}\ C$, where $x \in \{\mathbf{p}, \mathbf{c}, \mathbf{l}\}$ and $s \in (\{n_r \mid r \in \Pi\} \cup At(\Pi))$;
- $\textbf{minimize}\ X$, where $X \subseteq \{p, c, l\}$.

Further constructs, for instance, statements applying to entire sets of rules and/or atoms, are definable in a straightforward way but omitted for space reasons.

For a program Π, the function $\mathcal{D}$ defines the semantics of extrapolation statements:

- $\mathcal{D}[\textbf{extrapolate}\ n_r\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{E}}[\Pi, \{r\}]$;
- $\mathcal{D}[\textbf{extrapolate\_p}\ s\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{P}}[\{r\}]$, provided that $s \in \{n_r \mid r \in \Pi\}$;
- $\mathcal{D}[\textbf{extrapolate\_c}\ s\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{C}}[\Pi, \{s\}]$, provided that $s \in At(\Pi)$;
- $\mathcal{D}[\textbf{extrapolate\_l}\ s\ \textbf{if}\ C] = \mathcal{T}_{\mathsf{L}}[\{s\}]$, provided that $s \in At(\Pi)$.

The semantics of the **minimize** command depends on the capacities of the underlying ASP solver. For instance, in `Smodels`, a global minimization of abnormalities, using a binary predicate `ab`/2, could be expressed as `minimize`$\{$`ab`$(X, Y)\}$.

A pragmatic variant of extrapolation is *variation*:

– $\mathbf{vary}\ a_1, \ldots, a_k\ \mathbf{if}\ C$, where $a_i \in At(\Pi)$ for $i = 1, \ldots, k$.

The semantics of variation statements is as follows:

– $\mathcal{D}[\mathbf{vary}\ a_1, \ldots, a_k\ \mathbf{if}\ C] = \{\, \{a_i\} \leftarrow C \mid i = 1, \ldots, k\,\} \cup \{\, \mathsf{ko}(n_r) \leftarrow C \mid r \in \Pi,$ $\mathit{head}(r) = a_i, i = 1, \ldots, k\,\}$.

Finally, we introduce some procedural flavor, which allows for imposing a certain order of rule and/or atom considerations. A basic *ordinal statement* is an expression of the following form:

– $s\ \mathbf{before}\ t$, where $s, t \in \{n_r \mid r \in \Pi\}$ or $s, t \in At(\Pi)$.

Its semantics is defined in the following way:

– $\mathcal{D}[s\ \mathbf{before}\ t] = \mathcal{T}_o[\{r_t < r_s\}]$, provided that $s = n_{r_s}$ and $t = n_{r_t}$;
– $\mathcal{D}[s\ \mathbf{before}\ t] = \mathcal{T}_o[\{r_t < r_s \mid \mathit{head}(r_s) = s, \mathit{head}(r_t) = t\}]$, provided that $s, t \in At(\Pi)$.

The (atom-based) generalization to sets is defined as $A\ \mathbf{before}\ B$ where $A, B \subseteq At(\Pi)$, and it could be used to debug a generate-and-test encoding by specifying the generate atoms' precedence over the test atoms.

5 Discussion and Related Work

Although debugging is mentioned as a possible application in many papers on program analysis (often with the implicit assertion that incoherent and erroneous programs are the same thing), there are relatively few papers that are primarily focused on debugging.

The `noMoRe` system [3] uses a graph-based algorithm for computing the answer sets of a program. The interface to the system allows the computation process to be visualized and animated, so that the user can observe certain parts of the computation process. Given some background knowledge on how answer-set computation algorithms work, this is an intuitive and appealing approach to debugging. However, as it works on ground programs, dealing with larger programs is difficult. Also, at the conceptual level, it blurs the distinction between what a program means and how its answer sets are computed.

Brain and De Vos [4] start by characterizing bugs as mismatches between what the programmer expects and the actual answer sets of a program. Two query algorithms, answering why some set S is in some answer set A and why some set S is not contained in any answer set, can then be used to explore these mismatches. The algorithms suggested are procedural and similar to the ones used in ASP solvers, suggesting that an approach using a program-level transformation would be more practical.

Syrjänen [5] discusses a pragmatic approach to debugging, focusing on incoherent ground programs, whose source of incoherence is an active constraint (or an odd negative cycle). Derived from the field of symbolic diagnosis [12], the corresponding system uses an approach similar to ours, using program transformations to find minimal sets of constraints that would be active and to find odd cycles.

Pontelli and Son [6] adopt the concept of *justifications* [13,14,15] to the context of ASP. Roughly, justifications formalize reasons why an atom is in an answer set.

One important area that is not considered by the existing approaches is that of interfaces to debugging systems. Most of the proposed methods produce very large amounts of structured information, and it is difficult to automatically identify which parts of it are of interest to the programmer [4]. Thus, the design of the debugging interface is critical to the utility of the finished system. It must allow the programmers to quickly and easily focus in on areas that they consider to be erroneous without overloading them with information. This is an open and important area of research if ASP is to achieve truly wide-spread use. Other directions of future work include extending the results given here to handle constructs such as cardinality rules, disjunction, aggregates, and functions. Also, there is a potential to use similar tagging systems to allow programs to reason about their own consistency. We mention *consistency restoring rules* [16] as one such example.

References

1. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3**(2) (2003) 129–187
2. (`http://www.kr.tuwien.ac.at/research/debug`)
3. Bösel, A., Linke, T., Schaub, T.: Profiling answer set programming: The visualization component of the `noMoRe` system. In Proc. JELIA'04. Springer (2004) 702–705
4. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In Proc. ASP'05. (2005) 141–152
5. Syrjänen, T.: Debugging inconsistent answer set programs. In Proc. NMR'06. (2006) 77–83
6. Pontelli, E., Son, T.: Justifications for logic programs under answer set semantics. In Proc. ICLP'06. Springer (2006) 196–210
7. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
8. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
9. Clark, K.: Negation as failure. In Logic and Data Bases. Plenum (1978) 293–322
10. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
12. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1) (1987) 57–95
13. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying proofs using memo tables. In Proc. PPDP '00. (2000) 178–189
14. Pemmasani, G., Guo, H., Dong, Y., Ramakrishnan, C., Ramakrishnan, I.: Online justification for tabled logic programs. In Proc. FLOPS'04. Springer (2004) 24–38
15. Specht, G.: Generating explanation trees even for negations in deductive database systems. In Proc. LPE'93. (1993) 8–13
16. Balduccini, M., Gelfond, M.: Logic programs with consistency-restoring rules. In Proc. Commonsense'03. (2003) 9–18

A Purely Model-Theoretic Semantics for Disjunctive Logic Programs with Negation*

Pedro Cabalar[1], David Pearce[2], Panos Rondogiannis[3], and William W. Wadge[4]

[1] Department of Computer Science
Corunna University, Spain
cabalar@udc.es

[2] Department of Informatics, Statistics and Telematics
Universidad Rey Juan Carlos, Madrid, Spain
davidandrew.pearce@urjc.es

[3] Department of Informatics & Telecommunications
University of Athens, Athens, Greece
prondo@di.uoa.gr

[4] Department of Computer Science
University of Victoria, Victoria, Canada
wwadge@cs.uvic.ca

Abstract. We present a purely model-theoretic semantics for disjunctive logic programs with negation, building on the *infinite-valued* approach recently introduced for normal logic programs [9]. In particular, we show that every disjunctive logic program with negation has a *non-empty* set of minimal infinite-valued models. Moreover, we show that the infinite-valued semantics can be equivalently defined using Kripke models, allowing us to prove some properties of the new semantics more concisely. In particular, for programs without negation, the new approach collapses to the usual minimal model semantics, and when restricted to normal logic programs, it collapses to the well-founded semantics. Lastly, we show that every (propositional) program has a finite set of minimal infinite-valued models which can be identified by restricting attention to a finite subset of the truth values of the underlying logic.

1 Introduction

The semantics of disjunctive logic programs with negation has been the subject of a number of recent research works [2,3,4,8,11,12]. A comparative study of all these approaches easily leads to the conclusion that at present there is no consensus on what is the "right approach" to the semantics of disjunctive logic programs with negation. In other words, the quest for an intuitive and broadly acceptable semantic approach appears at present to be unfulfilled. Motivated by this state of affairs, we introduce in this paper a novel, purely model-theoretic

* Partially supported by the the CICyT project TIC-2003-9001-C02, and by the Greek General Secretariat for Research and Technology and the Spanish MEC under a joint project of Scientific and Technological collaboration between Greece and Spain.

semantics for disjunctive programs, which generalizes both the well-founded semantics of normal logic programs as well as the minimal model semantics of (negationless) disjunctive programs. An important characteristic of the new approach is that it is purely logical: the meaning of a program is characterized solely by examining its models (actually, its *infinite-valued models*, see below). Having a purely logical semantics allows one to reason about programs using properties of the logic under consideration.

The present work builds on the *infinite-valued* approach that was recently introduced for normal logic programs [9]. In [9] a new infinite-valued logic is introduced and it is demonstrated that every normal logic program has a unique minimum model under this new logic; moreover, it is shown that when this model is collapsed to three-valued logic, it coincides with the well-founded semantics. It is therefore natural to ask: "can the infinite-valued approach be lifted to the class of disjunctive logic programs with negation giving in this way a respectable new semantics for this class of programs?".

In this paper we reply affirmatively to this question. In particular, we argue that the semantics of a disjunctive logic program with negation can be captured by the program's *set of minimal infinite-valued models* which, as we demonstrate, is always non-empty. Moreover, we show that the infinite-valued semantics can be equivalently defined using Kripke models. This alternative characterization allows us to prove properties of the new semantics more concisely. In particular, we prove that when restricted to programs without negation, the new approach collapses to the usual minimal model semantics, and when restricted to normal logic programs, it collapses to the well-founded semantics. Finally, we demonstrate that every program has a *finite* set of minimal infinite-valued models; actually, these models can be identified by restricting attention to a finite subset of the truth values of the underlying logic. We conclude with a comparison of the proposed approach with some other proposals for assigning semantics to disjunctive logic programs with negation that are related to well-founded semantics.

2 The Infinite-Valued Semantics

In this section we define the infinite-valued semantics for disjunctive logic programs with negation. Our presentation extends the one given in [9] for normal logic programs. We study the class of disjunctive logic programs:

Definition 1. *A* disjunctive logic program *is a finite set of clauses of the form:*

$$p_1 \vee \cdots \vee p_n \leftarrow q_1, \ldots, q_k, \sim r_1, \ldots, \sim r_m \qquad (1)$$

where $n \geq 1$ and $k, m \geq 0$.

Note that in this paper we consider only finite programs; the results can be lifted to the more general first-order case (with a corresponding notational overhead).

The basic idea behind the infinite-valued approach is that in order to have a purely model theoretic semantics for negation-as-failure, one should consider a richer logical framework than classical logic. Informally, we extend the domain of truth values and use these extra values to distinguish between ordinary negation and negation-as-failure. Consider for example the following (normal) program:

$$p \leftarrow \qquad r \leftarrow \sim p \qquad s \leftarrow \sim q$$

Under the negation-as-failure approach both p and s receive the value *True*. We would argue, however, that in some sense p is "truer" than s. Namely, p is true because there is a rule which says so, whereas s is true only because we are never obliged to make q true. In a sense, s is true only by default. Our truth domain adds a "default" truth value T_1 just below the "real" truth T_0, and a weaker false value F_1 just above ("not as false as") the real false F_0. We can then understand negation-as-failure as combining ordinary negation with a weakening. Thus $\sim F_0 = T_1$ and $\sim T_0 = F_1$. Since negations can effectively be iterated, our domain requires a whole sequence $\ldots, T_3, T_2, T_1$ of weaker and weaker truth values below T_0 but above a neutral value 0; and a mirror image sequence $F_1, F_2, F_3, \ldots$ above F_0 and below 0. In fact, in [9] a T_α and a F_α are introduced for all countable ordinals α; since in this paper we deal with finite propositional programs, we will not need this generality here.

In [9] it is demonstrated that, over this extended domain, every normal logic program with negation has a unique minimum model; and that in this model, if we collapse all the T_α and F_α to *True* and *False* respectively, we get the three-valued well-founded model [10]. Considering the above example program, its minimum infinite-valued model is $\{(p, T_0), (q, F_0), (r, F_1), (s, T_1)\}$, and therefore its well-founded model is $\{(p, T), (q, F), (r, F), (s, T)\}$. In this paper we extend the results of [9] by demonstrating that *every disjunctive logic program has a (non-empty) set of minimal infinite-valued models.*

The above discussion can now be formalized as follows. We first need to define an infinite-valued logic whose truth domain consists of the following values:

$$F_0 < F_1 < F_2 \cdots < 0 < \cdots < T_2 < T_1 < T_0$$

Intuitively, F_0 and T_0 are the classical *False* and *True* values and 0 is the *undefined* value. The values below 0 are ordered like the natural numbers. The values above 0 have exactly the reverse order. In the following we denote by V the set consisting of the above truth values. A notion that will prove useful in the sequel is that of the *order* of a given truth value:

Definition 2. *The* order *of a truth value is defined as follows:* $order(T_n) = n$, $order(F_n) = n$ *and* $order(0) = +\infty$.

Let Q be a set of propositional symbols out of which our programs are constructed. Interpretations are then defined as follows:

Definition 3. *An infinite-valued interpretation is a function from the set Q of propositional symbols to the set V of truth values.*

In the rest of the paper, the term "interpretation" will mean an infinite-valued one (unless otherwise stated). As a special case of interpretation, we will use $\emptyset$ to denote the interpretation that assigns the F_0 value to all members of Q.

Definition 4. *The meaning of a formula with respect to an interpretation I can be defined as follows:*

$$I(\sim A) = \begin{cases} T_{n+1} & \text{if } I(A) = F_n \\ F_{n+1} & \text{if } I(A) = T_n \\ 0 & \text{if } I(A) = 0 \end{cases} \qquad \begin{array}{r} I(A \wedge B) = min\{I(A), I(B)\} \\ I(A \vee B) = max\{I(A), I(B)\} \\ I(A \leftarrow B) = \begin{cases} T_0 & \text{if } I(A) \geq I(B) \\ I(A) & \text{if } I(A) < I(B) \end{cases} \end{array}$$

The notion of satisfiability of a clause can now be defined:

Definition 5. *Let Π be a program and I an interpretation. Then, I* satisfies *a clause $p_1 \vee \cdots \vee p_k \leftarrow l_1, \ldots, l_n$ of Π if $I(p_1 \vee \cdots \vee p_k) \geq I(l_1 \wedge \cdots \wedge l_n)$. Moreover, I is a* model *of Π if I satisfies all clauses of Π.*

We denote by L_∞ the *infinite-valued logic* induced by infinite-valued models. Given an interpretation of a program, we adopt specific notations for the set of atoms of the program that are assigned a specific truth value and for the subset of the interpretation that corresponds to a particular order:

Definition 6. *Let Π be a program, I an interpretation and $v \in V$. Then $I \parallel v = \{p \in Q \mid I(p) = v\}$. Moreover, if $n < \omega$, then $I \sharp n = \{(p, v) \in I \mid order(v) = n\}$.*

The following relations on interpretations will be used later in the paper:

Definition 7. *Let I and J be interpretations and $n < \omega$. We write $I =_n J$, if for all $k \leq n$, $I \parallel T_k = J \parallel T_k$ and $I \parallel F_k = J \parallel F_k$.*

Definition 8. *Let I and J be interpretations and $n < \omega$. We write $I \sqsubset_n J$, if for all $k < n$, $I =_k J$ and either $I \parallel T_n \subset J \parallel T_n$ and $I \parallel F_n \supseteq J \parallel F_n$, or $I \parallel T_n \subseteq J \parallel T_n$ and $I \parallel F_n \supset J \parallel F_n$. We write $I \sqsubseteq_n J$ if $I =_n J$ or $I \sqsubset_n J$.*

Definition 9. *Let I and J be interpretations. We write $I \sqsubset_\infty J$, if there exists $n < \omega$ (that depends on I and J) such that $I \sqsubset_n J$. We write $I \sqsubseteq_\infty J$ if either $I = J$ or $I \sqsubset_\infty J$.*

In comparing two interpretations I and J we consider first *only* those variables assigned "standard" truth values (T_0 or F_0) by at least one of the two interpretations. If I assigns T_0 to a particular variable and J does not, or J assigns F_0 to a particular variable and I does not, then we can rule out $I \sqsubseteq_\infty J$. Conversely, if J assigns T_0 to a particular variable and I does not, or I assigns F_0 to a particular variable and J does not, then we can rule out $J \sqsubseteq_\infty I$. If both these conditions apply, we can immediately conclude that I and J are incomparable. If exactly one of these conditions holds, we can conclude that $I \sqsubseteq_\infty J$ or $J \sqsubseteq_\infty I$, as appropriate. However, if neither apply, then I and J are equal in terms of standard truth values; they both assign T_0 to each of one group of variables and F_0 to each of another. In this case we must now examine the variables assigned F_1 or T_1. If this examination proves inconclusive, we move on to T_2 and F_2, and so on. Thus $\sqsubseteq_\infty$ gives the standard truth values the highest priority, T_1 and F_1 the next priority, T_2 and F_2 the next, and so on.

It is easy to see that the relation $\sqsubseteq_\infty$ on the set of interpretations is a partial order (i.e., it is reflexive, transitive and antisymmetric). On the other hand, for every $n < \omega$, the relation $\sqsubseteq_n$ is a preorder (i.e., reflexive and transitive).

From the above discussion it should be now clear that *the infinite-valued semantics of a disjunctive logic program with negation is captured by the set of $\sqsubseteq_\infty$-minimal infinite-valued models of the program. In Section 4 we'll see that this set is always non-empty. These ideas are illustrated by the following example:*

Example 1. Consider the program:

$$b \vee l \leftarrow \sim p \qquad\qquad l \vee p \leftarrow$$

We examine the minimal models of this program. Obviously, in every minimal model either l or p must have the value T_0 (this is due to the second clause). Assume first that l is T_0; then this immediately gives the minimal model $\{(l, T_0), (p, F_0), (b, F_0)\}$. Assume on the other hand that p is T_0; this implies that $\sim p$ is F_1, and therefore either b or l must be F_1 (or greater). Therefore, we also have the minimal models $\{(l, F_0), (p, T_0), (b, F_1)\}$ and $\{(l, F_1), (p, T_0), (b, F_0)\}$.

We denote by L_∞^{min} the non-monotonic logic induced by $\sqsubseteq_\infty$-minimal L_∞ models.

3 Kripke Semantics

We present an alternative but equivalent representation of the infinite-valued semantics in terms of Kripke models. This representation is useful in several respects. First, as a heuristic device it may help in visualising and proving properties about the semantics (Section 5). Second, it may help to relate the semantics to other approaches based on possible-worlds frames, such as equilibrium and partial equilibrium logic ([6,4]). Third, it may provide a basis in the future when searching for axiomatic systems capturing the underlying logic L_∞.

Definition 10 (Centered linear frame). *A* centered linear frame *is a Kripke frame $\langle W, \leq \rangle$ with a set of worlds W consisting of two distinguished elements w_∞, w'_∞ plus two ω-sequences $w_0, w_1, \ldots$ and $w'_0, w'_1, \ldots$ and a linear ordering '$\leq$' satisfying, $w_i \leq w_{i+1}$, $w_i \leq w_\infty$, $w'_{i+1} \leq w'_i$, $w'_\infty \leq w'_i$ and $w_\infty \leq w'_\infty$ for any $i < \omega$.*

From Definition 10 it follows that $w_i \leq w'_j$ for any i, j. Furthermore, we can depict both infinite chains $w_0, w_1, \ldots$ and $\ldots, w'_1, w'_0$ respectively bounded by w_∞ and w'_∞ in the middle as follows:

$$w_0 \leq w_1 \leq \cdots \leq w_\infty \leq w'_\infty \leq \cdots \leq w'_1 \leq w'_0.$$

Given any world $w \notin \{w'_\infty, w'_0\}$ we define $next(w)$ as the immediate successor of w in the chain, that is, $next(w_i) = w_{i+1}$, $next(w_\infty) = w'_\infty$ and $next(w'_{i+1}) = w'_i$.

Definition 11 (Centered linear model). *A* centered linear model *is a Kripke model $\langle W, \leq, \sigma \rangle$ where $\langle W, \leq \rangle$ is a centered linear frame and $\sigma : Q \times W \longrightarrow \{0, 1\}$ is a valuation such that $\sigma(p, w) = 1, w \leq u \Rightarrow \sigma(p, u) = 1$ and $\sigma(p, w_0) = 0$ and $\sigma(p, w'_0) = 1$ for all atoms p.*

Given a Kripke model, we let W_i, W'_i stand for the sets of atoms that are true at the respective worlds w_i, w'_i, for $i = 0, 1, \ldots, \infty$. From Definition 11 we conclude:

$$\emptyset = W_0 \subseteq W_1 \subseteq \cdots \subseteq W_\infty \subseteq W'_\infty \subseteq \cdots \subseteq W'_1 \subseteq W'_0 = Q \tag{2}$$

where in particular

$$\bigcup_i W_i \subseteq W_\infty \text{ and } W'_\infty \subseteq \bigcap_i W'_i.$$

An interesting way of describing a centered linear model M is using the sequence $M = (\mathbf{W}_0, \mathbf{W}_1, \ldots, \mathbf{W}_\infty)$ where each $\mathbf{W}_i$ is a three-valued interpretation $\mathbf{W}_i = (W_i, W'_i)$ so that atoms in W_i, $W'_i \setminus W_i$ and $Q \setminus W'_i$ are respectively seen as *true*, *undefined* and *false* up to order i. We may define the standard "less or equal truth" relation $\preceq$ between pairs so that $\mathbf{W}_i \preceq \mathbf{W}_j$ iff $W_i \subseteq W_j$ and $W'_i \subseteq W'_j$. This allows rephrasing (2) as a simple chain $\mathbf{W}_0 \preceq \mathbf{W}_1 \preceq \cdots \preceq \mathbf{W}_\infty$ with $\mathbf{W}_0 = \langle \emptyset, Q \rangle$ assigning false to all atoms. Interpretation $\mathbf{W}_\infty$ contains the maximum truth in the chain and is important for comparisons with well-founded semantics (Proposition 4 in Section 5). A three-valued interpretation like $\mathbf{W} = (W, W)$ is said to be *complete* (no undefined atoms). We say that $M = (\mathbf{W}_0, \mathbf{W}_1, \ldots, \mathbf{W}_\infty)$ is *i-complete* for some $i \in \{1, 2, \ldots, \infty\}$ when $\mathbf{W}_i$ is complete. Note that this means that $\forall j \ (j \geq i \Rightarrow \mathbf{W}_j = \mathbf{W}_i)$ and $\mathbf{W}_\infty = \mathbf{W}_i$.

Definition 12 (Routley frame). *A* Routley frame *is a triple $\langle W, \leq, * \rangle$ where $\langle W, \leq \rangle$ is a Kripke frame and $* : W \to W$ is such that $x \leq y$ if $y^* \leq x^*$.*

Definition 13 (Zigzag model). *A* zigzag model *is a tuple $\langle W, \leq, *, \sigma \rangle$ where $\langle W, \leq, \sigma \rangle$ is a centered linear Kripke model and $\langle W, \leq, * \rangle$ is a Routley frame with $*$ defined as: $(w_i)^* = w'_i$ and $(w'_i)^* = w_i$ for $i = 0$ and $i = \infty$; $(w'_{j+1})^* = w_j$ and $(w_{j+1})^* = w'_j$ for all $j < \omega$.*

The structure below shows the effect of $*$ in solid lines, and the linear accessibility relation $\leq$ in dotted lines:

$$\begin{tikzcd}
w_0 \arrow[r, dotted] \arrow[d, bend left] & w_1 \arrow[r, dotted] \arrow[dl] & w_2 \arrow[r, dotted] \arrow[dl] & \cdots \arrow[r, dotted] \arrow[dl] & w_\infty \arrow[d, dotted] \arrow[d, bend left] \\
w'_0 \arrow[u, bend left] & w'_1 \arrow[l, dotted] \arrow[ul] & w'_2 \arrow[l, dotted] \arrow[ul] & \cdots \arrow[l, dotted] \arrow[ul] & w'_\infty \arrow[l, dotted] \arrow[u, bend left]
\end{tikzcd}$$

The name of "zigzag" comes from the path followed by successive applications of the $*$-function. Given a zigzag model $M = \langle W, \leq, *, \sigma \rangle$, valuation σ is extended to an arbitrary formula φ by means of the usual conditions for positive connectives in intuitionistic logic, and for negation by the following condition: $\sigma(\sim\varphi, w) = 1$ iff $\sigma(\varphi, w^*) = 0$.

Proposition 1. *For any zigzag model $\langle W, \leq, *, \sigma \rangle$ and any formula φ, $\sigma(\varphi, w) = 1$ and $w \leq u \Rightarrow \sigma(\varphi, u) = 1$.*

We say that M is a *model* of a theory Γ, written $M \models \Gamma$, if $\sigma(\varphi, w_1) = 1$ for all $\varphi \in \Gamma$ (note that satisfaction is in world w_1 and not w_0).

Definition 14 (Induced valuation). *Given a zigzag model $M = \langle W, \leq, *, \sigma \rangle$ we define its* induced valuation *function $\hat{M}$ that assigns a value $\hat{M}(\varphi) \in V$ to any formula φ as follows:*

$$\hat{M}(\varphi) \overset{\text{def}}{=} \begin{cases} T_i \textit{ iff } w_i \not\models \varphi \textit{ and } w_{i+1} \models \varphi \\ F_i \textit{ iff } w'_{i+1} \not\models \varphi \textit{ and } w'_i \models \varphi \\ 0 \quad \textit{iff } w_\infty \not\models \varphi \textit{ and } w'_\infty \models \varphi \end{cases}$$

This definition applied to atoms implies $\hat{M} \parallel T_i = W_{i+1} \backslash W_i$, $\hat{M} \parallel F_i = W'_i \backslash W'_{i+1}$ and $\hat{M} \parallel 0 = W'_\infty \setminus W_\infty$. Notice that this assignment is well constructed due to condition (2). Truth constants T and F can be incorporated as special atoms satisfying $\hat{M}(T) = T_0$ and $\hat{M}(F) = F_0$, that is, $T \in W_1 \setminus W_0$ and $F \in W'_0 \setminus W'_1$.

Proposition 2. *Let $M = (\mathbf{W}_0, \mathbf{W}_1, \ldots, \mathbf{W}_\infty)$ be a zigzag model. Then the three-valued interpretation $\mathbf{W}_\infty$ corresponds to collapsing all $\hat{M}(p) = T_i$ to true, all $\hat{M}(p) = F_i$ to false and $\hat{M}(p) = 0$ to undefined.*

It is not difficult to see that for any infinite-valued interpretation I we can always build a zigzag model M whose induced valuation coincides with I on all atoms – the next theorem asserts that it also coincides for any arbitrary formula.

Theorem 1. *Let I be an infinite-valued interpretation and M a zigzag model such that $\hat{M}(p) = I(p)$ for all atom p. Then $I(\varphi) = \hat{M}(\varphi)$ for any formula φ.*

Proof. We begin defining the last world $last(\varphi)$ in the chain at which formula φ does not hold so that $last(\varphi) = w_i$ when $\hat{M}(\varphi) = T_i$, $last(\varphi) = w'_{i+1}$ when $\hat{M}(\varphi) = F_i$ and $last(\varphi) = w_\infty$ when $\hat{M}(\varphi) = 0$. Note that $last(\varphi) \notin \{w'_\infty, w'_0\}$ and so $next(last(\varphi))$ (the first world at which φ holds) is always defined.

Lemma 1. *$\hat{M}(\alpha) \geq \hat{M}(\beta)$ iff $last(\alpha) \leq last(\beta)$.*

Thus, $last(\alpha) = last(\beta)$ implies $\hat{M}(\alpha) = \hat{M}(\beta)$. We proceed now by structural induction.

1. For the base case, if φ is an atom p, $\hat{M}(p) = I(p)$ by construction.
2. If $\varphi = \alpha \wedge \beta$, the last world at which φ does not hold is $max(last(\alpha), last(\beta))$. By Lemma 1 we conclude $\hat{M}(\alpha \wedge \beta) = min(\hat{M}(\alpha), \hat{M}(\beta))$. If $\varphi = \alpha \vee \beta$ the proof is analogous.
3. If $\varphi = \alpha \rightarrow \beta$, we have two cases: first, if $\hat{M}(\alpha) \leq \hat{M}(\beta)$ then, by Lemma 1, $last(\alpha) \geq last(\beta)$ and this means that any world w_k satisfies $w_k \not\models \alpha$ or $w_k \models \beta$, excepting w_0. In other words, $last(\alpha \rightarrow \beta) = w_0$ and so $\hat{M}(\alpha \rightarrow \beta) = T_0$. Otherwise, when $\hat{M}(\alpha) > \hat{M}(\beta)$, by Lemma 1 we get $last(\alpha) < last(\beta)$. Then, the last world w_k where $w_k \models \alpha$ but $w_k \not\models \beta$ is $last(\beta)$, and thus $\alpha \rightarrow \beta$ is also false until $last(\beta)$. As a result, $\hat{M}(\alpha \rightarrow \beta) = \hat{M}(\beta)$.
4. If $\varphi = \sim \alpha$. Assume $\hat{M}(\alpha) = T_i$, that is, $w_i \not\models \alpha$ and $w_{i+1} \models \alpha$. As $w_i = (w'_{i+1})^*$ and $w_{i+1} = (w'_{i+2})^*$ we get $w'_{i+1} \models \sim \alpha$ and $w'_{i+2} \not\models \sim \alpha$. But then, from Definition 14, we get $\hat{M}(\alpha) = F_{i+1}$. Analogously, when $\hat{M}(\alpha) = F_i$ we have $w'_{i+1} \not\models \alpha$ and $w'_i \models \alpha$ that, since $w'_{i+1} = (w_{i+2})^*$ and $w'_i = (w_{i+1})^*$, we

get $w_{i+1} \not\models \sim\alpha$ and $w_{i+2} \models \sim\alpha$ and so $\hat{M}(\alpha) = T_{i+1}$. Finally, when $\hat{M}(\alpha) = 0$ we have $w_\infty \not\models \alpha$ and $w'_\infty \models \alpha$, but as $w_\infty = (w'_\infty)^*$ and $w'_\infty = (w_\infty)^*$, we obtain $w'_\infty \models \sim\alpha$ and $w_\infty \not\models \sim\alpha$ that means $\hat{M}(\sim\alpha) = 0$. □

We can now alternatively define L^{min}_∞ in terms of minimal Kripke models. Let $M_1 = (\mathbf{W}_0, \mathbf{W}_1, \ldots, \mathbf{W}_\infty)$ and $M_2 = (\mathbf{U}_0, \mathbf{U}_1, \ldots, \mathbf{U}_\infty)$ be a pair of zigzag models. We say that $M_1 \trianglelefteq M_2$ iff either $M_1 = M_2$ or $\exists i \big(\forall j\ (j \le i \Rightarrow \mathbf{W}_j = \mathbf{U}_j) \wedge \mathbf{W}_i \prec \mathbf{U}_i\big)$.

Proposition 3. *$M_1 \trianglelefteq M_2$ iff $\hat{M}_1 \sqsubseteq_\infty \hat{M}_2$.*

Therefore, we have two alternative but equivalent definitions of the semantics of disjunctive logic programs with negation. In the rest of the paper, the two approaches will be used interchangeably.

4 Existence of Minimal Models

In this section we demonstrate that every disjunctive program has at least one minimal infinite-valued model. The proof is based on the dual of Zorn's Lemma[1]:

Lemma 2 (The dual of Zorn's Lemma). *Every non-empty partially ordered set in which each downward chain has a lower bound, contains a minimal element.*

First, notice that the set of models of a disjunctive logic program is non-empty, because the interpretation which assigns to every propositional atom the value T_0 is always a model. Second, notice that the set of models of a program is partially ordered by the $\sqsubseteq_\infty$ relation. It suffices to show that every (possibly transfinite) downwards chain of models under $\sqsubseteq_\infty$, has a lower bound which is also a model of the program.

Therefore, consider a chain $\mathcal{M}$ of infinite-valued models of a disjunctive program Π: $M_0 \sqsupseteq_\infty M_1 \sqsupseteq_\infty M_2 \sqsupseteq_\infty \cdots \sqsupseteq_\infty M_\alpha \sqsupseteq_\infty \cdots$. We describe at an intuitive level the construction of a lower bound M of this chain. We first start with all models that belong to $\mathcal{M}$ and we "intersect" them at their zero'th level of truth values. This gives us a (partial) interpretation that will serve as the zero'th level of the lower bound M. We now consider only those elements of the chain $\mathcal{M}$ whose zero'th order part agrees with the partial interpretation we have just constructed. We repeat the above process with this new set of models and with the order one values. In the limit of this process, certain atoms may have not received a value; we assign to them the value 0. We now formalize the above construction:

Definition 15. *Let S be a set of infinite-valued interpretations of a given program and $n \in \omega$. Then, we define $\bigwedge^n S = \{(p, T_n) \mid \forall M \in S, M(p) = T_n\}$ and $\bigvee^n S = \{(p, F_n) \mid \exists M \in S, M(p) = F_n\}$. Moreover, we define $\bigodot^n S = (\bigwedge^n S) \bigcup (\bigvee^n S)$.*

[1] Notice that the proof given in this section can be extended to apply to infinite propositional programs (and therefore also to the case of first-order programs).

Let Π be a program and let $\mathcal{M}$ be a downward chain of models of Π. We can now define the following sequence of sets of models of Π:

$$\begin{aligned} S_0 &= \mathcal{M} \\ S_{n+1} &= \{M \in S_n \mid M\sharp n = \bigodot^n S_n\} \end{aligned}$$

We now have the following lemma:

Lemma 3. *For every $n < \omega$, $S_n \neq \emptyset$.*

Proof. We demonstrate by induction on n that S_n is a non-empty chain identical to $\mathcal{M}$ the only difference being that an initial part of $\mathcal{M}$ may be missing from S_n. The base case is obvious. Assume the statement holds for n ie., that S_n is a nonempty downward chain of the form $M_\alpha \sqsupseteq_\infty M_{\alpha+1} \sqsupseteq_\infty M_{\alpha+2} \sqsupseteq_\infty \cdots$. For the induction step observe that since $M_\alpha \parallel T_n \supseteq M_{\alpha+1} \parallel T_n \supseteq \cdots$ and $M_\alpha \parallel F_n \subseteq M_{\alpha+1} \parallel F_n \subseteq \cdots$, after an initial segment of this chain, all the members of the chain agree on their level n components. Therefore, S_{n+1} forms a non-empty chain. This establishes the desired result. □

Example 2. Consider the program just consisting of rule: $s \vee p \leftarrow \sim s$

Moreover, consider the following models of the above program:
$M_n = \{(s, T_n), (p, F_0)\}$. Clearly, $M_0 \sqsupseteq_\infty M_1 \sqsupseteq_\infty M_2 \sqsupseteq_\infty \cdots$. Then, it is not hard to see that $S_0 = \{M_0, M_1, M_2, M_3, \ldots\}$, $S_1 = \{M_1, M_2, M_3, \ldots\}$, and so on.

We can now demonstrate the main theorem of this section which actually states that for every downward chain of models of a given disjunctive program, there exists a lower bound:

Theorem 2. *Let Π be a program and let $\mathcal{M}$ be a downwards chain of models of Π. Then $\mathcal{M}$ has a lower bound which is a model of Π.*

Proof. Consider models $N_0 \in S_1, N_1 \in S_2, \ldots, N_k \in S_{k+1}, \ldots,\ k < \omega$. We construct an interpretation M as follows:

$$M(p) = \begin{cases} (\bigcup_{k<\omega}(N_k\sharp k))(p) & \text{if this is defined} \\ 0 & \text{otherwise} \end{cases}$$

We claim that M is a model of the program. Assume it is not. Consider then a clause $p_1 \vee \cdots \vee p_m \leftarrow B$ such that $M(p_1 \vee \cdots \vee p_m) < M(B)$. There are three cases:

- $M(p_1 \vee \cdots \vee p_m) = F_k$, $k < \omega$. But then, $N_k(p_1 \vee \cdots \vee p_m) = F_k$ and since N_k is a model of Π, we have $N_k(B) \leq F_k$. But this implies that $M(B) \leq F_k$, and therefore $M(p_1 \vee \cdots \vee p_m) \geq M(B)$ (contradiction).
- $M(p_1 \vee \cdots \vee p_m) = T_k$, $k < \omega$. Then, $N_k(p_1 \vee \cdots \vee p_m) = T_k$ and since N_k is a model of Π, we have $N_k(B) \leq T_k$. But this easily implies that $M(B) \leq T_k$, and therefore $M(p_1 \vee \cdots \vee p_m) \geq M(B)$ (contradiction).

- $M(p_1 \vee \cdots \vee p_m) = 0$. But then, for every $k < \omega$, we have $N_k(p_1 \vee \cdots \vee p_m) < T_k$. Now, since N_k is a model of Π for every k, it is $N_k(B) < T_k$. But this then implies that $M(B) \leq 0$, and therefore $M(p_1 \vee \cdots \vee p_m) \geq M(B)$ (contradiction).

Moreover, it is straightforward to see that, by construction, M is a lower bound for all the members of the chain. □

The above discussion leads to the following theorem:

Theorem 3. *Every disjunctive logic program with negation has a non-empty set of minimal infinite-valued models.*

Proof. Immediate from the dual of Zorn's Lemma and Theorem 2. □

Example 3. Consider again the program:

$$s \vee p \leftarrow \sim s$$

together with the models:

$$M_n = \{(s, T_n), (p, F_0)\}$$

Applying the above construction to the chain $M_0 \sqsupseteq_\infty M_1 \sqsupseteq_\infty M_2 \sqsupseteq_\infty \cdots$, we get the lower bound $M = \{(s, 0), (p, F_0)\}$ of the chain.

Actually, it is not hard to see that the above program has two minimal models, namely $\{(p, T_1), (s, F_0)\}$ and $\{(p, F_0), (s, 0)\}$.

Example 4. Consider the program:

$$\begin{array}{ll} p \vee q \vee r & \leftarrow \\ p & \leftarrow \sim q \\ q & \leftarrow \sim r \\ r & \leftarrow \sim p \end{array}$$

By inspection, this program has the three minimal models $\{(p, T_0), (q, T_2), (r, F_1)\}$, $\{(p, F_1), (q, T_0), (r, T_2)\}$ and $\{(p, T_2), (q, F_1), (r, T_0)\}$.

5 Properties of the Minimal Model Semantics

We turn to properties of the minimal infinite-valued semantics. First we see that new approach extends the well-founded semantics of normal logic programs:

Proposition 4. *A normal logic program Π has a $\trianglelefteq$-minimum model $M = (\mathbf{W}_0, \mathbf{W}_1, \ldots, \mathbf{W}_\infty)$ where $\mathbf{W}_\infty$ is the well-founded model of Π.*

Proof. [9] shows that every normal logic program has a minimum infinite-valued model (Theorem 7.4) which when collapsed to three-valued logic coincides with the well-founded model (Theorem 7.6). From these two results and Proposition 2, the result above follows immediately. □

In other words, when we restrict the syntax to that of normal logic programs, the infinite-valued approach provides the well-founded model of the program, apart from additional information. In the case of the $\sqsubseteq_\infty$-least model characterisation, the well-founded model is obtained by collapsing all the T_i and F_i values in the model into T and F respectively. In the case of the corresponding $\trianglelefteq$-least zigzag model, the well-founded model is *directly obtained* by just keeping the $\mathbf{W}_\infty$ pair and ignoring all the rest. In fact, when we later compare the infinite-valued approach to other disjunctive well-founded semantics, we will also restrict the study to pair $\mathbf{W}_\infty$ so that we just handle a three-valued interpretation. A different and interesting open topic is the possible utility of the rest of information contained in the infinite-valued minimal models, which captures somehow the ordering or level in which we make default assumptions to compute the final result.

As we are now going to see, the approach we propose is compatible with the minimal model semantics for (negation-less) disjunctive logic programs. A *positive disjunctive* logic program is a set of clauses like (1) where $m = 0$ (i.e., they have no negated literals). Given a classical model X we can define a corresponding 1-complete zigzag model M^X so that $\mathbf{W}_1 = (X, X)$. This implies all $W_j = W'_j = X$ except $W_0 = \emptyset$ and $W'_0 = Q$ that are fixed. An i-complete model assigns to any atom a value of order smaller than i – when $i = 1$ the value can just be T_0 or F_0. The following pair of lemmas can be easily proved.

Lemma 4. *For a positive disjunctive program Π, $X \models \Pi$ in classical logic iff $M^X \models \Pi$.*

Lemma 5. *Any $\trianglelefteq$-minimal model of a positive disjunctive program Π is 1-complete.*

Theorem 4. *Let Π be a positive disjunctive logic program. Then: (i) if X is a minimal classical model of Π then M^X is a $\trianglelefteq$-minimal model of Π; (ii) if $M = (\mathbf{W}_0, \mathbf{W}_1, \dots, \mathbf{W}_\infty)$ is a $\trianglelefteq$-minimal model of Π, then M is 1-complete, and W_1 is a minimal classical model of Π.*

Proof. (i) If X is a minimal classical model, by Lemma 4, $M^X \models \Pi$. Assume we have some $\trianglelefteq$-minimal model of Π strictly smaller than M^X – by Lemma 5, such a model is 1-complete, call it M^Y, and since it is strictly smaller than M^X, $Y \subset X$. By Lemma 4, $Y \models \Pi$, and this contradicts minimality of classical model X. For (ii), Lemma 5 directly implies that M is 1-complete, i.e., $\mathbf{W}_1 = (W_1, W_1)$. By Lemma 4, $W_1 \models \Pi$. Assume there exists a smaller classical model $Y \subset W_1$. By Lemma 4, $M^Y \models \Pi$. Since M^Y is $\trianglelefteq$-smaller than M^X we get a contradiction. □

6 Identifying the Minimal Models

In this section we demonstrate that every disjunctive logic program has a *finite* set of minimal infinite-valued models. This result is not immediately obvious since the underlying logic has an infinite number of truth values. The key idea behind the proof is that if $|S|$ is the number of propositional symbols of a program, then it suffices to consider as possible candidates for minimality the models of the program that use truth values with order at most $|S| - 1$. The following definition will be needed in the proof of the theorem:

Definition 16. *Let Π be a program and let M be an infinite-valued model of Π. We will say that M* contains a gap at order *$\delta \in \omega$ if:*

- *For every $n < \delta$, there exists an atom p in Π with $order(M(p)) = n$.*
- *There does not exist an atom p in Π with $order(M(p)) = \delta$.*
- *There exists an atom p of Π such that $\delta < order(M(p)) < \infty$.*

The proof of the following theorem demonstrates that minimal models cannot contain gaps. This easily implies that in our search for minimal models we need only inspect a finite number of models.

Theorem 5. *Let Π be a program, S be the set of propositional symbols that appear in P and let M be a minimal infinite-valued model of Π. Then, for every propositional symbol $p \in S$, $M(p) \in \{0, F_0, T_0, \ldots, F_{|S|-1}, T_{|S|-1}\}$.*

Proof. It suffices to demonstrate that if a model of Π contains a gap then it can not be minimal. The theorem then follows by the fact that if a program does not contain a gap then its propositional symbols will necessarily get values from the set $\{0, F_0, T_0, \ldots, F_{|S|-1}, T_{|S|-1}\}$.

Assume for the sake of contradiction that M is a minimal model of Π that contains a gap at order $\delta \in \omega$. We establish a contradiction by constructing a model M^* of Π such that $M^* \sqsubset M$. Let $m > \delta$ be the least natural number such $M \sharp m \neq \emptyset$. We distinguish the following two cases:

Case 1: There exists some p such that $(M \sharp m)(p) = T_m$. We define the following interpretation:

$$M^*(p) = \begin{cases} T_{m+1}, \text{ if } M(p) = T_m \\ M(p), \text{ otherwise} \end{cases}$$

Obviously, $M^* \sqsubset M$. Let $p_1 \vee \cdots \vee p_n \leftarrow B$ be a clause in Π. By a simple case analysis on the possible values of $M^*(p_1 \vee \cdots \vee p_n)$, we get that M^* satisfies the given clause and therefore the whole program. Consequently, M is not a minimal model of Π (contradiction).

Case 2: There does not exist any p such that $(M \sharp m)(p) = T_m$. We define the following interpretation:

$$M^*(p) = \begin{cases} T_{n-1}, \text{ if } M(p) = T_n \text{ and } n > \delta \\ F_{n-1}, \text{ if } M(p) = F_n \text{ and } n > \delta \\ M(p), \text{ otherwise} \end{cases}$$

Obviously, $M^* \sqsubset M$. Let $p_1 \vee \cdots \vee p_n \leftarrow B$ be a clause in Π. By a simple case analysis on the possible values of $M^*(p_1 \vee \cdots \vee p_n)$, we get that M^* satisfies the given clause and therefore the whole program. Consequently, M is not a minimal model of Π (contradiction). Therefore, if a model of Π contains a gap, it cannot be a minimal one. □

7 Related Approaches

In this section we mention some examples that are useful for comparing the infinite-valued approach L_∞^{min} with other existing approaches to the semantics of disjunctive logic programs with negation, in particular, STATIC (of [8]), D-WFS (of [2,3]), WFDS (of [11]), WFS$_\mathrm{d}$ of [1] and with PEL (of [4]). We observe in particular that L_∞^{min} differs from all these semantics.

Example 1 was presented in [11], where it was reasoned that b should be false, while STATIC and D-WFS just fail to derive any information. In fact, the three semantics WFDS, PEL and L_∞^{min} allow one to derive the falsity of b.

Consider now the following example borrowed from [1]: $\{(a \vee b \leftarrow \sim b), (b \leftarrow \sim b)\}$. For this example, L_∞^{min} yields the minimal model $\{(a, F_0), (b, 0)\}$. PEL agrees that a should be false and b undefined. However, WFDS makes both atoms undefined.

An interesting observation is that, as in PEL, the *unfolding* transformation rule (see eg [3]) does not preserve equivalence in L_∞^{min}. Unfolding atom b on program $\Pi_1 = \{(a \vee b), (a \leftarrow \sim a), (c \leftarrow a \wedge b)\}$ leads to program $\Pi_2 = \{(a \vee b), (a \leftarrow \sim a), (c \vee a \leftarrow a)\}$. Both programs have the minimal model $\{(a, T_0), (b, F_0), (c, F_0)\}$ but Π_1 has a second minimal model $\{(a, 0), (b, T_0), (c, 0)\}$ while the second minimal model of Π_2 is $\{(a, 0), (b, T_0), (c, F_0)\}$ (in fact, PEL agrees with these results too). It follows that L_∞^{min} differs from WFS$_\mathrm{d}$ ([1]).

Another similarity between PEL and the L_∞^{min} semantics is that applying the *S-Implication* (S-IMP) transformation rule from WFDS [12] does not generally yield a strongly equivalent program. For instance, the result of applying S-IMP on program $\Pi_3 = \{(b \vee c \leftarrow a), (b \leftarrow a \wedge \sim c)\}$ deletes the second rule $\Pi_4 = \{(b \vee c \leftarrow a)\}$. However, if we add $\Pi_5 = \{(c \leftarrow \sim a), (a \leftarrow \sim a)\}$ to both programs, we obtain that $\Pi_3 \cup \Pi_5$ and $\Pi_4 \cup \Pi_5$ have different L_∞^{min} models: both have unique minimal models, but the former makes all atoms undefined, while the latter makes b false and the rest undefined.

Partial equilibrium logic (PEL) ([4]) is a general nonmonotonic framework that extends the partial stable model semantics of [7]. In some respects it appears to be conceptually close to the semantics of L_∞^{min}. In particular, it also provides a purely declarative, model-theoretic semantics and is even based on Routley frames. However, the most important difference has to do with the existence of L_∞^{min} model for disjunctive programs, something not guaranteed by PEL. This is illustrated by Example 4, which has no PEL models whereas, as we saw before, it has three L_∞^{min} models. Another difference is that in the L_∞^{min} approach the intended models are reached via one minimization process. In PEL one first defines partial stable or partial equilibrium models through a minimization process, while a second minimality condition captures those models that are said to be well-founded.

8 Conclusions

We have introduced a new purely model-theoretic semantics for disjunctive logic programs with negation and showed that every such program has at least one minimal infinite-valued model. The new semantics generalizes both the minimal model semantics of positive disjunctive logic programs as well as the well-founded semantics of normal logic programs. Future work includes the study of efficient proof procedures for the new semantics and possible applications of the new approach. An interesting open question is the possible application of the additional information provided by the infinite-valued approach not present in other variants of well-founded semantics which return three-valued answers. This extra information is related to the level or ordering in which default assumptions are made to compute the final result, and can be of valuable help for debugging an unexpected outcome, pointing out unobserved dependences or even capturing priorities as different truth levels. We also plan to investigate the underlying logic L_∞ of this approach in more detail. Another interesting topic for future research is the generalization of the recently introduced *game semantics* of negation [5] to the case of disjunctive logic programs.

References

1. J. Alcantara, C.V. Damasio and L.M. Pereira. A Well-Founded Semantics with Disjunction. In *Proceedings of the International Conference on Logic Programming (ICLP'05)*, LNCS 3668, pages 341–355, 2005.
2. S. Brass and J. Dix. Characterizations of the (disjunctive) Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997.
3. S. Brass and J. Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning*, 20(1):143–165, 1998.
4. P. Cabalar, S. Odintsov, D. Pearce, and A. Valverde. Analysing and Extending Well-Founded and Partial Stable Semantics using Partial Equilibrium Logic. In *Proceedings of the International Conference on Logic Programming (ICLP'06)*, LNCS 4079, pages 346–360, Seattle, USA, August 2006.
5. Ch. Galanaki, P. Rondogiannis and W.W. Wadge. An Infinite-Game Semantics for Well-Founded Negation. *Annals of Pure and Applied Logic*, 2007 (to appear).
6. D. Pearce. Equilibrium Logic. *Ann. Math & Artificial Int.*, 47 (2006), 3-41.
7. Przymusinski, T. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence* 12:141–187, 1994.
8. T. Przymusinski. Static Semantics of Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):323–357, 1995.
9. P. Rondogiannis and W.W. Wadge. Minimum Model Semantics for Logic Programs with Negation-as-Failure. *ACM Transactions on Computational Logic*, 6(2):441–467, 2005.
10. A. van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
11. K. Wang. Argumentation-based Abduction in Disjunctive Logic Programming. Journal of Logic Programming, 45(1-3):105–141, 2000.
12. K. Wang and L. Zhou. Comparisons and Computation of Well-Founded Semantics for Disjunctive Logic Programs. ACM Transactions on Computational Logic, 6(2):295–327, 2005.

Complexity of Default Logic on Generalized Conjunctive Queries*

Philippe Chapdelaine[1], Miki Hermann[2], and Ilka Schnoor[3]

[1] GREYC (UMR 6072), Université de Caen, France
pchapdel@info.unicaen.fr
[2] LIX (UMR 7161), École Polytechnique, France
hermann@lix.polytechnique.fr
[3] Theoretische Informatik, Universität Hannover, Germany
ilka.schnoor@thi.uni-hannover.de

Abstract. Reiter's default logic formalizes nonmonotonic reasoning using default assumptions. The semantics of a given instance of default logic is based on a fixpoint equation defining an extension. Three different reasoning problems arise in the context of default logic, namely the existence of an extension, the presence of a given formula in an extension, and the occurrence of a formula in all extensions. Since the end of 1980s, several complexity results have been published concerning these default reasoning problems for different syntactic classes of formulas. We derive in this paper a complete classification of default logic reasoning problems by means of universal algebra tools using Post's clone lattice. In particular we prove a trichotomy theorem for the existence of an extension, classifying this problem to be either polynomial, NP-complete, or Σ_2P-complete, depending on the set of underlying Boolean connectives. We also prove similar trichotomy theorems for the two other algorithmic problems in connection with default logic reasoning.

1 Introduction

Nonmonotonic reasoning is one of the most important topics in computational logic and artificial intelligence. Different logics formalizing nonmonotonic reasoning have been developed and studied since the late 1970s. One of the most known is Reiter's *default logic* [21], which formalizes nonmonotonic reasoning using default assumptions. Default logic can express facts like "by default, a formula φ is true", in contrast with standard classical logic, which can only express that a formula φ is true or false.

Default logic is based on the principle of defining the semantics of a given set of formulas W (also called *premises* or *axioms*) through a fixpoint equation by means of a finite set of defaults D. The possible extensions of a given set W of axioms are the sets E, stable under a specific transformation, i.e., satisfying the identity $\Gamma(E) = E$. These fixpoint sets E represent the different possible sets of knowledge that can be adopted on the base of the premises W. Three important decision problems arise in the context of reasoning in default logic. The first is to decide whether for a given

* Supported by ÉGIDE 05835SH, DAAD D/0205776 and DFG VO 630/5-1.

set of axioms W and defaults D there exists a fixpoint. The second, called *credulous reasoning*, is the task to determine whether a formula φ occurs in at least one extension of the set W. The third one, called *skeptical reasoning* asks to determine whether a given formula φ belongs to *all* extensions of W.

At the end of 1980s and the beginning of 1990s, several complexity results were proved for default logic reasoning. Several authors have investigated the complexity of the three aforementioned problems for syntactically restricted versions of propositional default logic. Kautz and Selman [13] proved the NP-completeness of propositional default reasoning restricted to disjunction-free formulas, i.e., all propositional formulas occurring in the axioms W and the defaults D are conjunctions of literals. Furthermore, they show that for very particular restrictions default reasoning is feasible in polynomial time. Stillman [23,24] extends the work of Kautz and Selman by analyzing further subclasses of disjunction-free default theories, as well as some other classes that allow a limited use of disjunction. The work of Kautz and Selman [13], as well as of Stillman [23,24] provided a good understanding of the tractability frontier of propositional default reasoning. The complexity of the general case was finally settled by Gottlob in [11], where he proved that propositional default reasoning is complete for the second level of the polynomial hierarchy. All these complexity results indicate that default logic reasoning is more complicated than that of the standard propositional logic.

In the scope of the aforementioned results a natural question arises whether the previous analysis covers all possible cases. We embark on this challenge by making two generalizations. First, the usual clauses have been generalized to constraints based on Boolean relations. Second, we allow in the axioms W and the defaults D not only formulas built as conjunctions of constraints, but also conjunctive queries, i.e., existential positive conjunctive formulas built upon constraints. This approach using a restricted existential quantification can be seen as a half way between the usual propositional formulas and the default query language DQL defined in [6]. Moreover, this approach is natural in the scope of relation-based constraints, since it allows us to use the universal algebra tools to reason about complexity. We take advantage of the closed classes of Boolean functions and relations, called clones and co-clones, which allow us to prove a complexity result for a single representant of this class, that extends by means of closure properties to all Boolean functions or queries, respectively, in the same class. Using these algebraic tools we deduce a complete classification of the three default reasoning problems parametrized by sets of Boolean constraints. Similar classification, using universal algebra tools and Post lattice, had been already done for other nonmonotonic reasoning formalisms, namely circumscription [16] and abduction [8,17]. Finally, a complexity classification of propositional default logic along other lines, studying the structural aspects of the underlying formulas, had been done in [1]. Our approach to the complexity classification differs from Ben-Eliyahu's [1] in the following points: (1) the class of formulas in the the axioms, prerequisite, justification, and consequence of defaults is always the same; (2) the classification is performed on the set of underlying Boolean relations S, taking the role of a parameter, from which the formulas are built and not on the input formulas itself; (3) the studied classes of formulas are closed under conjunction and existential quantification. The aforementioned requirements for uniformity of the formulas in all three parts of defaults and in the axioms, plus the closure under conjunction exclude prerequisite-free, justification-free, normal, semi-normal, or any other syntactically restricted default theories from this classification.

2 Preliminaries

Throughout the paper we use the standard correspondence between predicates and relations. We use the same symbol for a predicate and its corresponding relation, since the meaning will always be clear from the context, and we say that the predicate *represents* the relation.

An n-ary *logical relation* R is a Boolean relation of arity n. Each element of a logical relation R is an n-ary Boolean vector $m = (m_1, \ldots, m_n) \in \{0,1\}^n$. Let V be a set of variables. A *constraint* is an application of R to an n-tuple of variables from V, i.e., $R(x_1, \ldots, x_n)$. An assignment $I \colon V \to \{0,1\}$ satisfies the constraint $R(x_1, \ldots, x_n)$ if $(I(x_1), \ldots, I(x_n)) \in R$ holds.

Example 1. Equivalence is the binary relation defined by $eq = \{00, 11\}$. Given the ternary relations $nae = \{0,1\}^3 \smallsetminus \{000, 111\}$ and 1-in-3 $= \{100, 010, 001\}$, the constraint $nae(x,y,z)$ is satisfied if not all variables are assigned the same value and 1-in-3(x,y,z) is satisfied if exactly one of the variables x, y, and z is assigned to 1.

Throughout the text we refer to different types of Boolean constraint relations following Schaefer's terminology [22]. We say that a Boolean relation R is *1-valid* if $1 \cdots 1 \in R$ and it is *0-valid* if $0 \cdots 0 \in R$; *Horn* (*dual Horn*) if R can be represented by a conjunctive normal form (CNF) formula having at most one unnegated (negated) variable in each clause; *bijunctive* if it can be represented by a CNF formula having at most two variables in each clause; *affine* if it can be represented by a conjunction of linear functions, i.e., a CNF formula with $\oplus$-clauses (XOR-CNF); *complementive* if for each $(\alpha_1, \ldots, \alpha_n) \in R$, also $(\neg\alpha_1, \ldots, \neg\alpha_n) \in R$. A set S of Boolean relations is called 0-valid (1-valid, Horn, dual Horn, affine, bijunctive, complementive) if every relation in S is 0-valid (1-valid, Horn, dual Horn, affine, bijunctive, complementive).

Let R be a Boolean relation of arity n. The *dual relation* to R is the set of vectors $\mathrm{dual}(R) = \{(\neg\alpha_1, \ldots, \neg\alpha_n) \mid (\alpha_1, \ldots, \alpha_n) \in R\}$. Note that $R^\neg = R \cup \mathrm{dual}(R)$ is a complementive relation called the complementive closure of R. The set $\mathrm{dual}(S) = \{\mathrm{dual}(R) \mid R \in S\}$ denotes the corresponding dual relations to the set of relations S.

Let S be a non-empty finite set of Boolean relations. An *S-formula* is a finite conjunction of *S-clauses*, $\varphi = c_1 \wedge \cdots \wedge c_k$, where each S-clause c_i is a constraint application of a logical relation $R \in S$. An assignment I satisfies the formula φ if it satisfies all clauses c_i. We denote by $\mathrm{sol}(\varphi)$ the set of satisfying assignments of a formula φ.

Schaefer in his seminal paper [22] developed a complexity classification of the satisfiability problem of S-formulas. *Conjunctive queries* turn out to be useful in order to obtain this result. Given a set S of Boolean relations, we denote by $\textsc{coq}(S)$ the set of all formulas of the form

$$F(x_1, \ldots, x_k) \quad = \quad \exists y_1 \exists y_2 \cdots \exists y_l \; \varphi(x_1, \ldots, x_k, y_1, \ldots, y_l),$$

where φ is an S-formula. We call these existentially quantified formulas *conjunctive queries over S*, with $\boldsymbol{x} = (x_1, \ldots, x_k)$ being the vector of *distinguished variables*.

$\mathrm{Pol}(R) \supseteq \mathrm{E}_2$	$\Leftrightarrow$	R is Horn	$\mathrm{Pol}(R) \supseteq \mathrm{V}_2$	$\Leftrightarrow$	R is dual Horn
$\mathrm{Pol}(R) \supseteq \mathrm{D}_2$	$\Leftrightarrow$	R is bijunctive	$\mathrm{Pol}(R) \supseteq \mathrm{L}_2$	$\Leftrightarrow$	R is affine
$\mathrm{Pol}(R) \supseteq \mathrm{N}_2$	$\Leftrightarrow$	R is complementive	$\mathrm{Pol}(R) \supseteq \mathrm{R}_2$	$\Leftrightarrow$	R is disjunction-free
$\mathrm{Pol}(R) \supseteq \mathrm{I}_0$	$\Leftrightarrow$	R is 0-valid	$\mathrm{Pol}(R) \supseteq \mathrm{I}_1$	$\Leftrightarrow$	R is 1-valid
$\mathrm{Pol}(R) \supseteq \mathrm{I}$	$\Leftrightarrow$	R is 0- and 1-valid	$\mathrm{Pol}(R) \supseteq \mathrm{I}_2$	$\Leftrightarrow$	R is Boolean

Fig. 1. Polymorphism correspondences

As usually in computational complexity, we denote by $A \leq_m B$ a polynomial-time many-one reduction from the problem A to problem B. If there exist reductions $A \leq_m B$ and $B \leq_m A$, we say that the problems A and B are *polynomially equivalent*, denoted by $A \equiv_m B$.

3 Closure Properties of Constraints

There exist easy criteria to determine if a given relation is Horn, dual Horn, bijunctive, or affine. We recall these properties here briefly for completeness. An interested reader can find a more detailed description with proofs in the paper [5] or in the monograph [7]. Given a logical relation R, the following *closure properties* fully determine the structure of R, where $\oplus$ is the exclusive or and maj is the majority operation:

– R is Horn if and only if $m, m' \in R$ implies $(m \wedge m') \in R$.
– R is dual Horn if and only if $m, m' \in R$ implies $(m \vee m') \in R$.
– R is affine if and only if $m, m', m'' \in R$ implies $(m \oplus m' \oplus m'') \in R$.
– R is bijunctive if and only if $m, m', m'' \in R$ implies $\mathrm{maj}(m, m', m'') \in R$.

The notion of closure property of a Boolean relation has been defined more generally, see for instance [12,18]. Let $f\colon \{0,1\}^k \to \{0,1\}$ be a Boolean function of arity k. We say that R is *closed under* f, or that f is a *polymorphism* of R, if for any choice of k vectors $m_1, \ldots, m_k \in R$, not necessarily distinct, we have that

$$\Big(f\big(m_1[1], \ldots, m_k[1]\big),\ \ldots,\ f\big(m_1[n], \ldots, m_k[n]\big)\Big) \in R, \tag{1}$$

i.e., that the new vector constructed coordinate-wise from $m_1, \ldots, m_k$ by means of f belongs to R. We denote by $\mathrm{Pol}(R)$ the set of all polymorphisms of R and by $\mathrm{Pol}(S)$ the set of Boolean functions that are polymorphisms of every relation in S. It turns out that $\mathrm{Pol}(S)$ is a *closed set of Boolean functions*, also called a *clone*, for every set of relations S. In fact, a clone is a set of functions containing all projections and closed under composition. A clone generated by a set of functions F, i.e., a set containing F, all projections, and closed under composition, is denoted by $[F]$. All closed classes of Boolean functions were identified by Post [20]. Post also detected the inclusion structure of these classes, which is now referred to as *Post's lattice*, presented in Fig. 2 with the notation from [2]. We did not use the previously accepted notation for the clones, as in [18,19], since we think that the new one used in [2] is better suited mnemotechnically and also scientifically than the old one. The correspondence of the most studied classes with respect to the polymorphisms of a relation R is presented in Fig. 1. The class I_2

is the closed class of Boolean functions generated by the identity function, thus for every Boolean relation R we have $\mathrm{Pol}(R) \supseteq \mathrm{I}_2$. If the condition $\mathrm{Pol}(S) \supseteq C$ holds for $C \in \{\mathrm{E}_2, \mathrm{V}_2, \mathrm{D}_2, \mathrm{L}_2\}$, i.e., S being Horn, dual Horn, bijunctive, or affine, respectively, then we say that the set of relations S belongs to the *Schaefer's class*.

A Galois correspondence has been exhibited between the sets of Boolean functions $\mathrm{Pol}(S)$ and the sets of Boolean relations S. A basic introduction to this correspondence can be found in [18] and a comprehensive study in [19]. See also [5]. This theory helps us to get elegant and short proofs for results concerning the complexity of conjunctive queries. Indeed, it shows that the smaller the set of polymorphisms is, the more expressive the corresponding conjunctive queries are, which is the cornerstone for applying the algebraic method to complexity (see [2] and [5] for surveys). The following proposition can be found, e.g., in [5,18,19].

Proposition 2. *Let S_1, S_2 be two sets of Boolean relations. The inclusion* $\mathrm{Pol}(S_1) \subseteq \mathrm{Pol}(S_2)$ *implies* $\mathrm{COQ}(S_1 \cup \{eq\}) \supseteq \mathrm{COQ}(S_2 \cup \{eq\})$.

Given a k-ary Boolean function $f\colon \{0,1\}^k \longrightarrow \{0,1\}$, the set of *invariants* $\mathrm{Inv}(f)$ of f is the set of Boolean relations closed under f. More precisely, a relation R belongs to $\mathrm{Inv}(f)$ if the membership condition (1) holds for any collection of not necessarily distinct vectors $m_i \in R$ for $i = 1, \ldots, k$. If F is a set of Boolean functions then $\mathrm{Inv}(F)$ is the set of invariants for each function $f \in F$. It turns out that $\mathrm{Inv}(F)$ is a *closed set of Boolean relations*, also called a *co-clone*, for every set of functions F. In fact, a co-clone is a set of relations (identified by their predicates) closed under conjunction, variable identification, and existential quantification. A co-clone generated by a set of relations S is denoted by $\langle S \rangle$. Polymorphisms and invariants relate clones and co-clones by a Galois correspondence. This means that $F_1 \subseteq F_2$ implies $\mathrm{Inv}(F_1) \supseteq \mathrm{Inv}(F_2)$ and $S_1 \subseteq S_2$ implies $\mathrm{Pol}(S_1) \supseteq \mathrm{Pol}(S_2)$. Geiger [10] proved the identities $\mathrm{Pol}(\mathrm{Inv}(F)) = [F]$ and $\mathrm{Inv}(\mathrm{Pol}(S)) = \langle S \rangle$ for all sets of Boolean functions F and relations S.

4 Default Logic

A *default* [21] is an expression of the form

$$\frac{\alpha : \mathsf{M}\beta_1, \ldots, \mathsf{M}\beta_m}{\gamma} \tag{2}$$

where $\alpha, \beta_1, \ldots \beta_m, \gamma$ are propositional formulas. The formula α is called the *prerequisite*, $\beta_1, \ldots, \beta_m$ the *justification* and γ the *consequence* of the default. The notation with M serves only to syntactically and optically distinguish the justification from the prerequisite. A *default theory* is a pair $T = (W, D)$, where D is a set of defaults and W a set of propositional formulas also called the *axioms*. For a default theory $T = (W, D)$ and a set E of propositional formulas let $\Gamma(E)$ be the minimal set such that the following properties are satisfied:

(D1) $W \subseteq \Gamma(E)$
(D2) $\Gamma(E)$ is deductively closed
(D3) If

$$\frac{\alpha : \mathsf{M}\beta_1, \dots, \mathsf{M}\beta_m}{\gamma} \in D, \quad \alpha \in \Gamma(E), \quad \text{and} \quad \neg\beta_1, \dots, \neg\beta_m \notin E$$

then $\gamma \in \Gamma(E)$

Any fixed point of Γ, i.e., a set E of formulas satisfying the identity $\Gamma(E) = E$, is an *extension* for T. Each extension E of a default theory $T = (W, D)$ is identified by a subset $gd(E, T)$ of D, called the *generating defaults* of E, defined as

$$gd(E,T) = \left\{ \frac{\alpha : \mathsf{M}\beta_1, \dots, \mathsf{M}\beta_m}{\gamma} \in D \;\middle|\; \alpha \in E, \neg\beta_1 \notin E, \dots, \neg\beta_m \notin E \right\}.$$

There exists an equivalent constructive definition of the extension. It has been proved equivalent to the previous definition by Reiter in [21], whereas some authors, like Kautz and Selman [13], take it for the initial definition of the extension. Define $E_0 = W$ and

$$E_{i+1} = \mathrm{Th}(E_i) \cup \left\{ \gamma \;\middle|\; \frac{\alpha : \mathsf{M}\beta_1, \dots, \mathsf{M}\beta_m}{\gamma} \in D,\ \alpha \in E_i,\ \text{and}\ \neg\beta_1, \dots, \neg\beta_m \notin E \right\},$$

where $\mathrm{Th}(E)$ is the deductive closure of the set of formulas E. Then the *extension* of the default theory $T = (W, D)$ is the union $E = \bigcup_{i=0}^{\infty} E_i$. Notice the presence of the final union E in the conditions $\neg\beta_i \notin E$.

We generalize the default theories in the same way as propositional formulas are generalized to S-formulas. For a non-empty finite set of Boolean relations S, an *S-default* is an expression of the form (2), where $\alpha, \beta_1, \dots \beta_m, \gamma$ are formulas from $\textsc{coq}(S)$. An *S-default theory* is a pair $T(S) = (D, W)$, where D is a set of S-defaults and W a set of formulas from $\textsc{coq}(S)$. An *S-extension* is a minimal set of $\textsc{coq}(S)$-formulas including W and closed under the fixpoint operator Γ.

Three algorithmic problems are investigated in connection with default logic, namely the existence of an extension for a given default theory T, the question whether a given formula φ belongs to some extension of a default theory (called credulous or brave reasoning), and the question whether φ belongs to every extension of a theory (called skeptical or cautious reasoning). We express them as constraint satisfaction problems.

Problem: $\textsc{extension}(S)$
Input: An S-default theory $T(S) = (W, D)$.
Question: Does $T(S)$ have an S-extension?

Problem: $\textsc{credulous}(S)$
Input: An S-default theory $T(S) = (W, D)$ and an S-formula φ.
Question: Does φ belong to *some* S-extension of $T(S)$?

Problem: $\textsc{skeptical}(S)$
Input: An S-default theory $T(S) = (W, D)$ and an S-formula φ.
Question: Does φ belong to *every* S-extension of T.

To be able to use the algebraic tools for exploration of complexity results by means of clones and co-clones, and to exploit Post's lattice, we need to establish a Galois connection for the aforementioned algorithmic problems.

Theorem 3. *Let S_1 and S_2 be two sets of relations such that the inclusion $\mathrm{Pol}(S_1) \subseteq \mathrm{Pol}(S_2)$ holds. Then we have the following reductions among problems:*

$$\textsc{extension}(S_2) \leq_m \textsc{extension}(S_1) \qquad \textsc{credulous}(S_2) \leq_m \textsc{credulous}(S_1)$$
$$\textsc{skeptical}(S_2) \leq_m \textsc{skeptical}(S_1)$$

Proof. Since $\mathrm{Pol}(S_1) \subseteq \mathrm{Pol}(S_2)$ holds, then any conjunctive query on S_2 can be expressed by a logically equivalent conjunctive query using only relations from S_1, according to Proposition 2. Let $T(S_2) = (W_2, D_2)$ be an S_2-default theory. Perform the aforementioned transformation for every conjunctive query in W_2 and D_2 to get corresponding sets of preliminaries W_1 and defaults D_1, equivalent to W_2 and D_2, respectively. Therefore the default theory $T(S_2)$ has an S-extension if and only if $T(S_1) = (W_1, D_1)$ has one. An analogous result holds for credulous and skeptical reasoning. □

Post's lattice is symmetric according to the main vertical line $\mathrm{BF} \longleftrightarrow \mathrm{I}_2$ (see Figure 2), expressing graphically the duality between various clones and implying the duality between the corresponding co-clones. This symmetry extends to all three algorithmic problems observed in connection with default logic, as we see in the following lemma. It will allow us to considerably shorten several proofs.

Lemma 4. *Let S be a set of relations. Then the following equivalences hold:*

$$\textsc{extension}(S) \equiv_m \textsc{extension}(\mathrm{dual}(S))$$
$$\textsc{credulous}(S) \equiv_m \textsc{credulous}(\mathrm{dual}(S))$$
$$\textsc{skeptical}(S) \equiv_m \textsc{skeptical}(\mathrm{dual}(S))$$

Proof. It is clear that $\varphi(\boldsymbol{x}) = R_1(\boldsymbol{x}) \wedge \cdots \wedge R_k(\boldsymbol{x})$ belongs to an S-extension E of the default theory $T(S)$ if and only if the $\mathrm{dual}(S)$-formula $\varphi'(\boldsymbol{x}) = \mathrm{dual}(R_1)(\boldsymbol{x}) \wedge \cdots \wedge \mathrm{dual}(R_k)(\boldsymbol{x})$ belongs to a $\mathrm{dual}(S)$-extension E' of the default theory $T(\mathrm{dual}(S))$. □

5 Complexity Results

Complexity results for reasoning in default logic started to be published in early 1990s. Gottlob [11] proved that deciding the existence of an extension for a propositional default theory is Σ_2P-complete. Kautz and Selman [13] investigated the complexity of propositional default logic reasoning with unit clauses. They proved that deciding the existence of an extension for this special case is NP-complete. Zhao and Ding [26] also investigated the complexity of several special cases of default logic, when the formulas are restricted to special cases of bijunctive formulas. We complete here the complexity classification for default logic by the algebraic method.

Proposition 5. *If S is 0-valid or 1-valid, i.e., if $\mathrm{Pol}(S) \supseteq \mathrm{I}_0$ or $\mathrm{Pol}(S) \supseteq \mathrm{I}_1$, then every S-default theory always has a unique S-extension.*

Proof. Consider Reiter's constructive definition of the extension of an S-default theory $T(S) = (W, D)$. Since every formula in W and D is 0-valid (respectively 1-valid), every justification β of any default is also 0-valid (1-valid). Then $\neg\beta$ is *not* 0-valid (1-valid) and therefore it cannot appear in any S-extension E. Therefore any default from D is satisfied if and only if its prerequisite α is in the set E_i for some i. Since every formula in D is 0-valid (1-valid), whatever consequence γ is added to E_i, there cannot be a contradiction with the formulas previously included into E_i. Hence we just have to add to E every consequence γ recursively derived from the prerequisites until we reach a fixpoint E. Since we start with a finite set of axioms W and there is only a finite set of defaults D, an S-extension E always exists and it is unique. $\square$

We need to distinguish the Σ_2P-complete cases from the cases included in NP. The following proposition identifies the largest classes of relations for which the existence of an extension is a member of NP. According to the Galois connection, we need to identify the smallest clones that contain the corresponding polymorphisms. The reader is invited to consult Figure 1 to identify the clones of polymorphisms corresponding to the mentioned relational classes.

Proposition 6. *If S is Horn, dual Horn, bijunctive, or affine, i.e., if the inclusions* $\mathrm{Pol}(S) \supseteq \mathrm{E}_2$, $\mathrm{Pol}(S) \supseteq \mathrm{V}_2$, $\mathrm{Pol}(S) \supseteq \mathrm{D}_2$, *or* $\mathrm{Pol}(S) \supseteq \mathrm{L}_2$ *hold, then the problem* EXTENSION(S) *is in* NP.

Proof. We present a nondeterministic polynomial algorithm which finds an extension for an S-default theory $T(S) = (W, D)$.

1. Guess a set $D' \subseteq D$ of generating defaults.
2. For every COQ(S)-formula $\varphi \in W \cup \{\gamma \mid \gamma$ consequence of $d \in D'\}$ verify that $\varphi \nvDash \neg\beta$ holds for every justification β in D', i.e., check that $\varphi \wedge \beta$ is satisfiable.
3. Check that D' is minimal, i.e., for every S-default $\frac{\alpha:\mathsf{M}\beta_1,\ldots,\mathsf{M}\beta_m}{\gamma} \in D \smallsetminus D'$ and every COQ(S)-formula $\varphi \in W \cup \{\gamma \mid \gamma$ consequence of $d \in D'\}$ verify that $\varphi \nvDash \alpha$ or $\varphi \nvDash \beta_i$ holds for an i.

Step 1 ensures $\Gamma(E) \subseteq E$. Instead of $\varphi \nvDash \alpha$ and $\varphi \nvDash \beta_i$ for an i we check whether $\varphi \Rightarrow \alpha$ and $\varphi \Rightarrow \beta_i$ hold, respectively. Note that $\theta \Rightarrow \rho$ holds if and only if $\theta \equiv \rho \wedge \theta$. Equivalence is decidable in polynomial time for S-formulas from Schaefer's class [3], which extends to conjunctive queries. Therefore we can decide if $\varphi \wedge \beta$, $\varphi \Rightarrow \alpha$, $\varphi \Rightarrow \beta_i$ hold, and also if $\varphi \nvDash \alpha$, $\varphi \nvDash \beta_i$ for an i, in polynomial time. Hence, Steps 2 and 3 can be performed in polynomial time. $\square$

Now we need to determine the simplest relational classes for which the extension problem is NP-hard. The first one has been implicitly identified by Kautz and Selman [13] as the class of formulas consisting only of literals.

Proposition 7. *If* $\mathrm{Pol}(S) \subseteq \mathrm{R}_2$ *holds then* EXTENSION(S) *is* NP*-hard.*

Proof. Kautz and Selman proved in [13] using a reduction from 3SAT, that the extension problem is NP-hard for default theories $T = (W, D)$, where all formulas in the axioms W and the defaults D are literals. Böhler *et al.* identified in [4] that the relational class generated by the sets of satisfying assignments to a literal is the co-clone $\mathrm{Inv}(\mathrm{R}_2)$. Therefore from the Galois connection and Theorem 3 follows that the inclusion $\mathrm{Pol}(S) \subseteq \mathrm{R}_2$ implies that the extension problem for $T(S)$ is NP-hard. $\square$

The second simplest class with an NP-hard extension problem contains all relations which are at the same time bijunctive, affine, and complementive.

Proposition 8. *If* $\mathrm{Pol}(S) \subseteq \mathrm{D}$ *holds, then* EXTENSION(S) *is* NP*-hard.*

Proof. Recall first that $\mathrm{Inv}(D)$ is generated by the relation $\{01, 10\}$ (see [4]), which is the set of satisfying assignments of the clause $x \oplus y$, or equivalently of the affine clause $x \oplus y = 1$. Note that the affine clause $x \oplus y = 0$ represents the equivalence relation $x \equiv y$ belonging to every co-clone. Hence, the co-clone $\mathrm{Inv}(D)$ contains both relations generated by $x \oplus y = 1$ and $x \oplus y = 0$.

We present a polynomial reduction from the NP-complete problem NAE-3SAT (Not-All-Equal 3SAT [9, page 259]) to EXTENSION(S). Consider the following instance of NAE-3SAT represented by the formula $\varphi(x_1, \ldots, x_n) = \bigwedge_{i=1}^{k} nae(u_i, v_i, t_i)$ built upon the variables $x_1, \ldots, x_n$, where $nae(x, y, z)$ ensures that the variables x, y, z do not take the same Boolean value. We first build the following $2(n-1)$ defaults

$$d_i^0 = \frac{\top : \mathsf{M}(x_i \oplus x_{i+1} = 0)}{x_i \oplus x_{i+1} = 0} \quad \text{and} \quad d_i^1 = \frac{\top : \mathsf{M}(x_i \oplus x_{i+1} = 1)}{x_i \oplus x_{i+1} = 1}$$

for each $i = 1, \ldots, n-1$. For each clause $nae(u, v, t)$ in the formula φ we build the corresponding default

$$d(u, v, t) = \frac{\top : \mathsf{M}(u \oplus z = 1), \mathsf{M}(v \oplus z = 1), \mathsf{M}(t \oplus z = 1)}{\bot}$$

where z is a new variable. From each pair (d_i^0, d_i^1) exactly one default will apply. It will assign two possible pairs of truth values (b_i, b_{i+1}) to the variables x_i and x_{i+1}. This way the first set of default pairs separates the variables $x_1, \ldots, x_n$ into two equivalence classes. All variables in one equivalence class take the same truth value.

Note that the formula $(u \oplus z = 1) \wedge (v \oplus z = 1) \wedge (t \oplus z = 1)$ is satisfied only if the identity $u = v = t$ holds. Therefore the default $d(u, v, t)$ applies if and only if the clause $nae(u, v, t)$ is not satisfied. Let D be the set of all constructed defaults d_i^0, d_i^1, and $d(u, v, t)$ for each clause $nae(u, v, t)$ from φ. This implies that the formula $\varphi(x_1, \ldots, x_n)$ has a solution if and only if the default theory $(\emptyset, D)$ has an extension. The proposition then follows from Theorem 3. □

Finally, we deal with the most complicated case of default theories. The following proposition presents a generalization of Gottlob's proof from [11] that the existence of an extension is Σ_2P-complete.

Proposition 9. *If* $\mathrm{Pol}(S) \subseteq \mathrm{N}_2$ *holds then* EXTENSION(S) *is* Σ_2P*-hard.*

Proof. Let $\psi = \exists \boldsymbol{x}\, \forall \boldsymbol{y}\, \varphi(\boldsymbol{x}, \boldsymbol{y})$ be a quantified Boolean formula, with the variable vectors $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_m)$, such that the relation $R = \mathrm{sol}(\varphi(\boldsymbol{x}, \boldsymbol{y}))$ satisfies the condition $\mathrm{Pol}(R) = \mathrm{I}_2$. Let $R^\neg$ be the dual closure of the relation R. It is clear that $R(\boldsymbol{x}, \boldsymbol{y})$ is satisfiable if and only if $R^\neg(\boldsymbol{x}, \boldsymbol{y})$ is. Suppose that $\mathrm{Pol}(S) = \mathrm{N}_2$ holds, meaning that S is a set of complementive relations. Since $R^\neg$ is complementive, the relation $\bar{R} = \{0,1\}^{n+m} \smallsetminus R^\neg$ must be complementive as well. Therefore both relations $R^\neg$ and $\bar{R}$ must be in the co-clone $\langle S \rangle = \mathrm{Inv}(\mathrm{Pol}(S)) = \mathrm{Inv}(\mathrm{N}_2)$ generated by the relations S. Moreover, we have that $\bar{R}(\boldsymbol{x}, \boldsymbol{y}) = \neg R^\neg(\boldsymbol{x}, \boldsymbol{y})$.

The identity relation is included in every co-clone, therefore we can use the identity predicate $(x = y)$. Since S is complementive, the co-clone $\langle S \rangle$ contains the relation

nae, according to [4]. By identification of variables we can construct the predicate $nae(x, y, y)$ which is identical to the inequality predicate $(x \neq y)$.

Construct the S-default theory $T(S) = (W, D)$ with the empty set of axioms $W = \emptyset$ and the defaults $D = D_1 \cup D_2$, where

$$D_1 = \left\{ \frac{\top : \mathsf{M}(x_i = x_{i+1})}{x_i = x_{i+1}}, \frac{\top : \mathsf{M}(x_i \neq x_{i+1})}{x_i \neq x_{i+1}} \;\middle|\; i = 1, \ldots, n-1 \right\},$$
$$D_2 = \left\{ \frac{\top : \mathsf{M}\bar{R}(\boldsymbol{x}, \boldsymbol{y})}{\bot} \right\}.$$

The satisfiability of ψ is the generic Σ_2P-complete problem [25]. To prove Σ_2P-hardness for S-EXTENSION where $\mathrm{Pol}(S) = \mathrm{N}_2$, it is sufficient to show that ψ is valid if and only if $T(S)$ has an extension by same reasoning as in the proof of Theorem 5.1 in [11]. Since $\mathrm{I}_2 \subseteq \mathrm{N}_2$ and $\mathrm{Pol}(S) = \mathrm{N}_2$ hold, the proof of our proposition follows. □

Gottlob [11] proved the Σ_2P membership of the extension problem using a constructive equivalence between default logic and autoepistemic logic, previously exhibited by Marek and Truszczyński [14], followed by a Σ_2P-membership proof of the latter, which itself follows from a previous result of Niemelä [15]. A straightforward generalization of these results to S-default theories and the aforementioned propositions allow us to prove the following trichotomy theorem.

Theorem 10. *Let S be a set of Boolean relations. If S is 0-valid or 1-valid then the problem* EXTENSION(S) *is decidable in polynomial time. Else if S is Horn, dual Horn, bijunctive, or affine, then* EXTENSION(S) *is* NP*-complete. Otherwise* EXTENSION(S) *is Σ_2P-complete.*

Gottlob exhibited in [11] an intriguing relationship between the EXTENSION problem and the two other algorithmic problems observed in connection with default logic reasoning. In fact, the constructions used in the proofs for the EXTENSION problem can be reused for the CREDULOUS and SKEPTICAL problems, provided we make some minor changes. These changes can be carried over to our approach as well, as we see in the following theorems.

Theorem 11. *Let S be a set of Boolean relations. If S is 0-valid or 1-valid then the problem* CREDULOUS(S) *is decidable in polynomial time. Else if S is Horn, dual Horn, bijunctive, or affine, then* CREDULOUS(S) *is* NP*-complete. Otherwise the problem* CREDULOUS(S) *is Σ_2P-complete.*

Proof. The extension E constructed in the proof of Proposition 5 is unique and testing whether a given S-formula φ belongs to E takes polynomial time. The nondeterministic polynomial-time algorithm from the proof of Proposition 6 can be extended by the additional polynomial-time step

4. Check whether $\varphi \in \mathrm{Th}(W \cup \{\gamma \mid \gamma$ consequence of $d \in D'\})$ holds.

to test whether a given S-formula φ belongs to E. If $\mathrm{Pol}(S) \subseteq \mathrm{R}_2$ holds, it is sufficient to take the default theory $T = (W, D)$ with the axiom $W = \{\varphi(x_1, \ldots, x_n)\}$ and the defaults

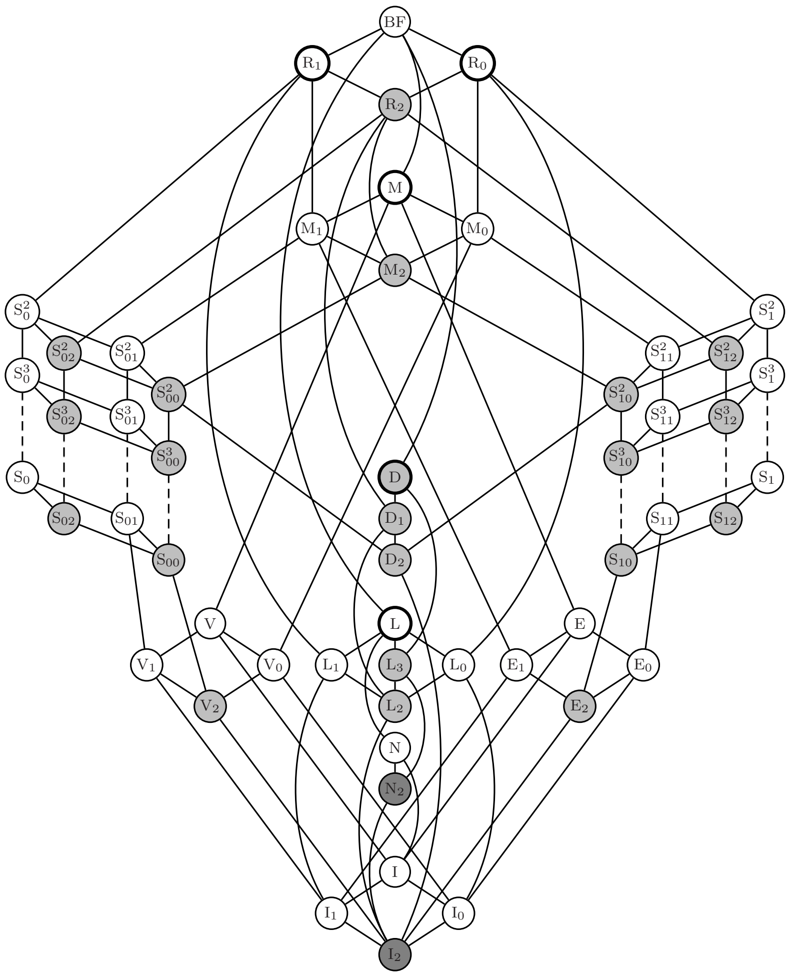


Fig. 2. Graph of all closed classes of Boolean functions

$$D = \left\{ \frac{\top : \mathsf{M}x_i}{x_i},\ \frac{\top : \mathsf{M}\neg x_i}{\neg x_i} \;\middle|\; i = 1, \ldots, n \right\}.$$

Note that the possible truth value assignments correspond to different extensions of the default theory T. Hence φ belongs to an extension of T if and only if there exists an extension of T. The same construction also works for $\mathrm{Pol}(S) \subseteq \mathrm{D}$ and $\mathrm{Pol}(S) \subseteq \mathrm{N}_2$, provided that we take the set of defaults $D = \{d_i^0, d_i^1 \mid i = 1, \ldots, n-1\}$ in the former and D_1 in the latter case. □

Theorem 12. *Let S be a set of Boolean relations. If S is 0-valid or 1-valid then the problem* SKEPTICAL(S) *is decidable in polynomial time. Else if S is Horn, dual Horn,* *bijunctive, or affine, then* SKEPTICAL(S) *is* coNP*-complete. Otherwise the problem* SKEPTICAL(S) *is* Π_2P*-complete.*

Proof. Skeptical reasoning is dual to the credulous one. For each credulous reasoning question whether an S-formula $\varphi(\boldsymbol{x}) = R_1(\boldsymbol{x}) \wedge \cdots \wedge R_k(\boldsymbol{x})$ belongs to an extension of a default theory $T(S) = (W, D)$, we associate the (dual) skeptical reasoning question whether the $\mathrm{dual}(S)$-formula $\varphi'(\boldsymbol{x}) = \mathrm{dual}(R_1)(\boldsymbol{x}) \wedge \cdots \wedge \mathrm{dual}(R_k)(\boldsymbol{x})$ belongs to *no* extension of the corresponding dual default theory $T(\mathrm{dual}(S)) = (W', D')$. Every S-formula in W and D is replaced by its corresponding $\mathrm{dual}(S)$-formula in W' and D'. Note that the co-clones $\mathrm{Inv}(\mathrm{N}_2)$, $\mathrm{Inv}(\mathrm{L}_2)$, $\mathrm{Inv}(\mathrm{D}_2)$, $\mathrm{Inv}(\mathrm{D})$, and $\mathrm{Inv}(\mathrm{R}_2)$ are closed under duality, i.e., for each $X \in \{\mathrm{Inv}(\mathrm{N}_2), \mathrm{Inv}(\mathrm{L}_2), \mathrm{Inv}(\mathrm{D}_2), \mathrm{Inv}(\mathrm{D}), \mathrm{Inv}(\mathrm{R}_2)\}$ we have $X = \mathrm{dual}(X)$. Moreover we have the identities $\mathrm{dual}(\mathrm{Inv}(\mathrm{E}_2)) = \mathrm{Inv}(\mathrm{V}_2)$ and $\mathrm{dual}(\mathrm{Inv}(\mathrm{V}_2)) = \mathrm{Inv}(\mathrm{E}_2)$, what relates the co-clones of Horn and dual Horn relations. Using now Lemma 4, the result follows from Theorem 11. □

6 Concluding Remarks

We found a complete classification for reasoning in propositional default logic, observed for the three corresponding algorithmic problems, namely of the existence of an extension, the presence of a given formula in an extension, and the membership of a given formula in all extensions. To be able to take advantage of the algebraic proof methods, we generalized the propositional default logic formulas to conjunctive queries. This generalization is in the same spirit and it is done along the same guidelines as the one going from the satisfiability problem SAT for Boolean formulas in conjunctive normal form to the constraint satisfaction problem CSP on the Boolean domain. Gottlob [11], Kautz and Selman [13], Stillman [23,24], and Zhao with Ding [26] explored a large part of the complexity results for default logic reasoning. We completed the aforementioned results and found that only a trivial subclass of default theories have the three algorithmic problems decidable in polynomial time. The corresponding polymorphism clones are colored white in Figure 2. Another part of default theories (composed of Horn, dual Horn, bijunctive, or affine relations) have NP-complete (resp. coNP-complete) algorithmic problems, with the corresponding polymorphism clones colored light gray in Figure 2. Finally, for the default theories, based on complementive or on all relations, the algorithmic problems are Σ_2P-complete (resp. Π_2P-complete), with the corresponding polymorphism clones colored dark gray in Figure 2. This implies the existence of a trichotomy theorem for each of the studied algorithmic problems.

References

1. R. Ben-Eliyahu-Zohary. Yet some more complexity results for default logic. *Artificial Intelligence*, 139(1):1–20, 2002.
2. E. Böhler, N. Creignou, S. Reith, and H. Vollmer. Playing with Boolean blocks, parts I and II. *SIGACT News*, 34(4):38–52, 2003 and 35(1):22–35, 2004.
3. E. Böhler, E. Hemaspaandra, S. Reith, and H. Vollmer. Equivalence and isomorphism for Boolean constraint satisfaction. In J. C. Bradfield (ed), *Proc. 11th CSL, Edinburgh (UK)*, LNCS 2471, pages 412–426. Springer-Verlag, September 2002.

4. E. Böhler, S. Reith, H. Schnoor, and H. Vollmer. Bases for Boolean co-clones. *Information Processing Letters*, 96(2):59–66, 2005.
5. A. Bulatov, P. Jeavons, and A. Krokhin. Classifying the complexity of constraints using finite algebras. *SIAM Journal on Computing*, 34(3):720–742, 2005.
6. M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, 1997.
7. N. Creignou, S. Khanna, and M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*, SIAM, Philadelphia (PA), 2001.
8. N. Creignou and B. Zanuttini. A complete classification of the complexity of propositional abduction. Submitted, October 2004.
9. M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Co, 1979.
10. D. Geiger. Closed systems of functions and predicates. *Pacific Journal of Mathematics*, 27(1):95–100, 1968.
11. G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 2(3):397–425, 1992.
12. P. Jeavons, D. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the Association for Computing Machinery*, 44(4):527–548, 1997.
13. H. A. Kautz and B. Selman. Hard problems for simple default logics. *Artificial Intelligence*, 49(1-3):243–279, 1991.
14. W. Marek and M. Truszczyński. Modal logic for default reasoning. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):275–302, 1990.
15. I. Niemelä. On the decidability and complexity of autoepistemic reasoning. *Fundamenta Informaticae*, 17(1-2):117–155, 1992.
16. G. Nordh. A trichotomy in the complexity of propositional circumscription. In F. Baader and A. Voronkov (eds), *Proc. 11th LPAR, Montevideo (Uruguay)*, LNCS 3452, pages 257–269, 2005.
17. G. Nordh and B. Zanuttini. Propositional abduction is almost always hard. In L. P. Kaelbling and A. Saffiotti (eds), *Proc. 19th IJCAI, Edinburgh (UK)*, pages 534–539, 2005.
18. N. Pippenger. *Theories of Computability*. Cambridge University Press, Cambridge, 1997.
19. R. Pöschel and L. A. Kalužnin. *Funktionen- und Relationenalgebren*. Deutscher Verlag der Wissenschaften, Berlin, 1979.
20. E. L. Post. The two-valued iterative systems of mathematical logic. *Annals of Mathematical Studies*, 5:1–122, 1941.
21. R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.
22. T. J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th Symposium on Theory of Computing (STOC'78), San Diego (California, USA)*, pages 216–226, 1978.
23. J. Stillman. It's not my default: the complexity of membership problems in restricted propositional default logics. In *Proc. 8th AAAI, Boston (MA, USA)*, pages 571–578, 1990.
24. J. Stillman. The complexity of propositional default logics. In *Proc. 10th AAAI, San Jose (California, USA)*, pages 794–799, July 1992.
25. C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):23–33, 1976.
26. X. Zhao and D. Ding. Complexity results for 2CNF default theories. *Fundamenta Informaticae*, 45(4):393–404, 2001.

A Preference-Based Framework for Updating Logic Programs

James P. Delgrande[1], Torsten Schaub[2,1], and Hans Tompits[3]

[1] School of Computing Science, Simon Fraser University,
Burnaby, B.C., Canada V5A 1S6
jim@cs.sfu.ca

[2] Institut für Informatik, Universität Potsdam,
August-Bebel-Straße 89, D-14482 Potsdam, Germany
torsten@cs.uni-potsdam.de

[3] Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9–11, A–1040 Wien, Austria
tompits@kr.tuwien.ac.at

Abstract. We present a framework for updating logic programs under the answer-set semantics that builds on existing work on preferences in logic programming. The approach is simple and general, making use of two distinct complementary techniques: defaultification and preference. While defaultification resolves potential conflicts by inducing more answer sets, preferences then select among these answer sets, yielding the answer sets generated by those rules that have been added more recently. We examine instances of the framework with respect to various desirable properties; for the most part, these properties are satisfied by instances of our framework. Finally, the proposed framework is also easily implementable by off-the-shelf systems.

1 Introduction

Over the last decade, *answer-set programming* (ASP) [1] has become a major approach for knowledge representation and reasoning. Given that knowledge is always subject to change, there has been a substantial effort in developing approaches to updating logic programs under the answer-set semantics [2,3,4,5,6,7,8,9,10]. Unfortunately, the problem of update appears to be intrinsically more difficult in a nonmonotonic setting (such as in ASP) than in a monotonic one, such as in propositional logic [11]. As a consequence, many approaches are quite involved and the set of approaches is rather heterogeneous.

In contrast to this, we propose a simple and general framework for updating logic programs that is based on two well-known parameterisable techniques in ASP: *defaultification* [12,13] and *preference handling* [14,15,16]. This is based on the following idea: The primary purpose of updating mechanisms is to resolve conflicts among more recent and less recent rules. To this end, we first need to detect potential conflicts between newer and older rules. Second, we need to prevent them from taking place. And finally, we need to resolve conflicts in favour of more recent updating rules. The two last steps are accomplished by defaultification and preferences. While defaultification resolves potential conflicts by inducing complementary answer sets, preferences then are used to select among these answer sets, producing those answer sets generated by rules that have been added more recently. As a result, our approach is easily implementable by appeal to existing off-the-shelf technology for preference handling, such as the front-end tool `plp`[1] (used in conjunction with standard ASP-solvers, like `smodels`[2] or `dlv`[3]), the genuine preference-handling ASP-solver `nomore`$^<$[4], or meta-interpretation methods [17].

Our techniques have further advantages: First, defaultification also allows for the elimination of incoherent situations, even in an updating program or in intermediate programs in an updating sequence. Second, preferences provide a modular way of capturing an update history, rather than an explicit program transformation, as done for instance in the approaches of Eiter *et al.* [9] or Zhang and Foo [4].

After giving some background, we introduce our framework in Section 3, along with an evaluation according to update principles proposed by Eiter *et al.* [9] in the context of ASP. Section 4 gives a more detailed comparison to the latter approach and shows how our approach deals with two well-known examples from the literature. The paper concludes with a discussion in Section 5.

2 Background

Given an alphabet $\mathcal{P}$, an *extended logic program*, or simply a *program*, is a finite set of rules of form

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

where $n \geq m \geq 0$ and each l_i $(0 \leq i \leq n)$ is a *literal*, that is, an *atom* $a \in \mathcal{P}$ or its negation $\neg a$. The set of all literals is given by $\mathcal{L} = \mathcal{P} \cup \{\neg a \mid a \in \mathcal{P}\}$. For a literal l, define $\overline{l} = a$ if $l = \neg a$, and $\overline{l} = \neg a$ if $l = a$. The set of atoms occurring in a program Π is denoted by $atom(\Pi)$. For a rule r of form (1), let $head(r) = l_0$ be the *head* of r and $body(r) = \{l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n\}$ the *body* of r. Furthermore, let $body^+(r) = \{l_1, \ldots, l_m\}$ and $body^-(r) = \{l_{m+1}, \ldots, l_n\}$. Rule r is *positive*, if $body^-(r) = \emptyset$.

A set of literals X is *consistent* if it does not contain a complementary pair a, $\neg a$ of literals. We say that X is *logically closed* iff it is either consistent or equals $\mathcal{L}$. The smallest set of literals being both logically closed and closed under a set Π of positive rules is denoted by $Cn(\Pi)$. The *reduct*, Π^X, of Π relative to a set X of literals is defined by $\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi,\ body^-(r) \cap X = \emptyset\}$ [18]. A set X of atoms is an *answer set* of an extended logic program Π if $Cn(\Pi^X) = X$. Two programs Π_1 and Π_2 are said to be *equivalent*, written $\Pi_1 \equiv \Pi_2$, if both programs have the same answer sets.

An *ordered logic program* is a pair $(\Pi, <)$, where Π is a logic program and $< \subseteq \Pi \times \Pi$ is a strict partial order. Given $r_1, r_2 \in \Pi$, the relation $r_1 < r_2$ expresses that r_2 has *higher priority* than r_1. This informal interpretation can be made precise in different ways. In what follows, we consider three such interpretations:

[1] `http://www.cs.uni-potsdam.de/~torsten/plp`
[2] `http://www.tcs.hut.fi/Software/smodels`
[3] `http://www.dlvsystem.com`
[4] `http://www.cs.uni-potsdam.de/wv/nomorepref`

B-preference [14], *D-preference* [15], and *W-preference* [16]. Given $(\Pi, <)$, all of these approaches use $<$ for selecting preferred answer sets among the standard answer sets of Π. The approaches are defined as follows. Let X be a consistent set of literals and define $\Pi_X = \{r \in \Pi \mid \mathit{body}^+(r) \subseteq X \text{ and } \mathit{body}^-(r) \cap X = \emptyset\}$. Then:

1. X is $<_D$*-preserving*, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of Π_X such that, for every $i, j \in I$, we have that:
   (a) if $r_i < r_j$, then $j < i$;
   (b) $\mathit{body}^+(r_i) \subseteq \{\mathit{head}(r_k) \mid k < i\}$; and
   (c) if $r_i < r'$ and $r' \in \Pi \setminus \Pi_X$, then
      i. $\mathit{body}^+(r') \not\subseteq X$ or
      ii. $\mathit{body}^-(r') \cap \{\mathit{head}(r_k) \mid k < i\} \neq \emptyset$.
2. X is $<_W$*-preserving*, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of Π_X such that, for every $i, j \in I$, we have that:
   (a) if $r_i < r_j$, then $j < i$;
   (b) i. $\mathit{body}^+(r_i) \subseteq \{\mathit{head}(r_k) \mid k < i\}$ or
      ii. $\mathit{head}(r_i) \in \{\mathit{head}(r_k) \mid k < i\}$; and
   (c) if $r_i < r'$ and $r' \in \Pi \setminus \Pi_X$, then
      i. $\mathit{body}^+(r') \not\subseteq X$ or
      ii. $\mathit{body}^-(r') \cap \{\mathit{head}(r_k) \mid k < i\} \neq \emptyset$ or
      iii. $\mathit{head}(r') \in \{\mathit{head}(r_k) \mid k < i\}$.
3. X is $<_B$*-preserving*, if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of Π_X such that, for every $i, j \in I$, we have that:
   (a) if $r_i < r_j$, then $j < i$; and
   (b) if $r_i < r'$ and $r' \in \Pi \setminus \Pi_X$, then
      i. $\mathit{body}^+(r') \not\subseteq X$ or
      ii. $\mathit{body}^-(r') \cap \{\mathit{head}(r_k) \mid k < i\} \neq \emptyset$ or
      iii. $\mathit{head}(r') \in X$.

As shown by Schaub and Wang [16], the three strategies yield an increasing number of preferred answer sets. That is, D–preference is stronger than W–preference, which is stronger than B–preference, which is stronger than the empty preference (i.e., having no preference).

Analogously to the unordered case, we call two ordered logic programs *equivalent* iff they have the same order-preserving answer sets, and we use again "$\equiv$" as a symbol for program equivalence. Note that an unordered program Π is trivially equivalent to the program $(\Pi, \emptyset)$ having an empty order relation, as every answer set of Π is a $<_\sigma$-preserving answer set of $(\Pi, <)$, for $<= \emptyset$ and $\sigma \in \{D, B, W\}$. Hence, allowing a slight abuse of notation, we sometimes identify an unordered program Π with $(\Pi, \emptyset)$.

3 The Basic Framework

3.1 Update Programs

The primary purpose of updating mechanisms is to resolve conflicts among newer and older rules. As mentioned, our approach is to first detect potential conflicts, second, to stop them from taking place, and third, to resolve these conflicts in favour of the updating rules.

A potential conflict manifests itself by complementary head literals. Two rules r_1, r_2 are said to be *conflicting* if $head(r_1) = \overline{head(r_2)}$. We represent potential conflicts among rules within two programs Π_1 and Π_2 in terms of the set

$$C(\Pi_1, \Pi_2) = \{(r_1, r_2) \mid r_1 \in \Pi_1, r_2 \in \Pi_2, head(r_1) = \overline{head(r_2)}\}.$$

For avoiding conflicts, we weaken rules by turning them into *defaultised* rules, as originally used by Sadri and Kowalski [12]. For rule r, we define

$$r^d = head(r) \leftarrow body(r), not\ \overline{head(r)}.$$

Similarly, for a program Π, we define $\Pi^d = \{r^d \mid r \in \Pi\}$ and call it the *defaultification* of Π. For example, program $\{a \leftarrow, \neg a \leftarrow\}$ has the answer set $\mathcal{L}$, while $\{a \leftarrow, \neg a \leftarrow\}^d = \{a \leftarrow not\ \neg a, \neg a \leftarrow not\ a\}$ has two answer sets, $\{a\}$ and $\{\neg a\}$. Note that, given that bodies of rules are *sets*, we have $r^d = (r^d)^d$, for every rule r.

The next result shows how the aforementioned "weakening" is to be understood.

Theorem 1. *Let Π be a logic program.*
Every consistent answer set of Π is an answer set of Π^d.

We propose to use preferences among rules for resolving inconsistencies. This provides us with several degrees of freedom: First, one can choose among different preference-handling strategies; and second, these strategies can be imposed in different ways on the rules. As well, defaultification can be applied universally or selectively to rules. We next detail three specific ways of applying the framework.

To begin with, we give the following very basic definition of an update operator on logic programs:

Definition 1. *The update program obtained for updating program Π_1 by the program Π_2 via update operator $*_0$ is given by the ordered logic program*

$$\Pi_1 *_0 \Pi_2 = (\Pi_1^d \cup \Pi_2^d, \Pi_1^d \times \Pi_2^d).$$

Thus, the ordered logic program is over $\Pi_1^d \cup \Pi_2^d$, and $<$ is defined so that every rule in Π_1^d has less priority than every rule in Π_2^d. We do not suggest that this is a good update operator; in fact this operator is usually too strict, since it establishes a preference between *all* rules in programs Π_1 and Π_2 even though they may not conflict. However, it provides a simple, basic instance of our approach.

A conflict-oriented approach is the following.

Definition 2. *The update program obtained for updating program Π_1 by the program Π_2 via update operator $*_1$ is given by the ordered logic program*

$$\Pi_1 *_1 \Pi_2 = (\Pi_1^d \cup \Pi_2^d, C(\Pi_1^d, \Pi_2^d)).$$

Operator $*_1$ globally weakens the rules in the program and imposes preferences along potential conflicts.

A refinement of the above approach is the following:

Definition 3. *The update program obtained for updating program Π_1 by the program Π_2 via update operator $*_2$ is given by the ordered logic program*

$$\Pi_1 *_2 \Pi_2 = (\Pi_c^d \cup ((\Pi_1 \cup \Pi_2) \setminus \Pi_c)), C(\Pi_1^d, \Pi_2^d)),$$

where $\Pi_c = \{r_1, r_2 \mid (r_1, r_2) \in C(\Pi_1, \Pi_2)\}$.

Operator $*_2$ restricts defaultification to conflicting rules. Unlike the previous update operations, this necessitates program transformations whenever conflicting rules are encountered upon iterated updates (see below). Further definitions are possible.

As well, one must also specify a preference-handling strategy, which we consider next.

Definition 4. *Let $\Pi_1 * \Pi_2$ be an update program for some update operator $*$, $\sigma \in \{B, D, W\}$ a preference-handling strategy, and X a set of literals.*

Then, X is an answer set of $\Pi_1 * \Pi_2$ with respect to σ *iff X is a $<_\sigma$-preserving answer set of $\Pi_1 * \Pi_2$.*

Depending on the chosen preference-handling strategy, update programs may admit several, one, or no answer sets. The latter is worth illustrating because it motivates the increasing restriction of preferences among rules when defining our update operators. For instance, the update program $\{b \leftarrow not\ a\} *_0 \{c \leftarrow not\ b\}$ has no answer set with respect to either the B-, D-, or W-strategy. In contrast, we obtain answer set $\{b\}$ when applying $*_1$ and $*_2$. For another example, consider $\{a \leftarrow\} *_0 \{b \leftarrow a\}$. We obtain answer set $\{a, b\}$ with respect to the B strategy, but no answer set with the D or W strategy. In contrast to this, we get answer set $\{a, b\}$ when applying $*_1$ and $*_2$ no matter which preference-handling strategy we chose.

For illustrating the different behaviour of $*_0$, $*_1$, and $*_2$ with respect to inconsistent programs, consider $\Pi_1 = \{r_1 : a \leftarrow,\ r_2 : \neg a \leftarrow\}$ and $\Pi_2 = \{r_3 : b \leftarrow a\}$. Here, we have

$$r_1^d = a \leftarrow not\ \neg a, \qquad r_2^d = \neg a \leftarrow not\ a,$$
$$r_3^d = b \leftarrow a, not\ \neg b,$$

and $C(\Pi_1^d, \Pi_2^d) = \emptyset$. Hence, we get the following update programs:

$$\Pi_1 *_0 \Pi_2 = (\{r_1^d,\ r_2^d,\ r_3^d\}, \{r_1^d < r_3^d, r_2^d < r_3^d\}),$$
$$\Pi_1 *_1 \Pi_2 = \Pi_1^d \cup \Pi_2^d = \{r_1^d,\ r_2^d,\ r_3^d\}, \quad \text{and}$$
$$\Pi_1 *_2 \Pi_2 = \Pi_1 \cup \Pi_2 = \{r_1,\ r_2,\ r_3\}.$$

Clearly, Π_1 is inconsistent, i.e., it has the single answer set $\mathcal{L}$. Under B-preference, $\Pi_1 *_0 \Pi_2$ has $\{\neg a\}$ and $\{a, b\}$ as its answer sets, whereas under D- and W-preference, only $\{\neg a\}$ is an answer set of $\Pi_1 *_0 \Pi_2$. Roughly speaking, $\{a, b\}$ is not an answer set under D- and W-preference because of the "prescriptive" nature of these preference strategies. For $\Pi_1 *_1 \Pi_2$, we also get the two answer sets $\{\neg a\}$ and $\{a, b\}$, while $\Pi_1 *_2 \Pi_2$ again yields the inconsistent answer set $\mathcal{L}$.

The common factor, however, is that each selection criterion chooses its preferred answer sets among those of the defaultification of the union of the original programs.

Theorem 2. *Let $\Pi_1 *_i \Pi_2$ be an update program for some $i = 0, 1, 2$ and let $\sigma \in \{B, D, W\}$ be a preference-handling strategy.*

Then, every answer set of $\Pi_1 *_i \Pi_2$ with respect to σ is an answer set of $(\Pi_1 \cup \Pi_2)^d$.

Iterated updates are easily defined.

Definition 5. *Let $(\Pi_1, \ldots, \Pi_n)$ be a sequence of logic programs, for $n \geq 2$, and let $*$ be a binary update operator.*

Then, $*(\Pi_1, \ldots, \Pi_n)$, the update program obtained from $(\Pi_1, \ldots, \Pi_n)$*, is the ordered logic program given as follows:*

$$*(\Pi_1, \ldots, \Pi_n) = \begin{cases} \Pi_1 * \Pi_2, & \text{if } n = 2; \\ ([*(\Pi_1, \ldots, \Pi_{n-1})] * \Pi_n), & \text{if } n > 2. \end{cases}$$

Definition 6. *Let $*(\Pi_1, \ldots, \Pi_n)$ be an update program for some update operator $*$, let $\sigma \in \{B, D, W\}$ be a preference-handling strategy, and let X be a set of literals.*

Then, X is an answer set of $*(\Pi_1, \ldots, \Pi_n)$ with respect to σ *iff X is a $<_\sigma$-preserving answer set of $*(\Pi_1, \ldots, \Pi_n)$.*

Whenever convenient, we write $(\Pi_1 * \ldots * \Pi_n)$ instead of $*(\Pi_1, \ldots, \Pi_n)$. As in Theorem 2, every answer set of $(\Pi_1 * \ldots * \Pi_n)$ is selected among the ones of $(\Pi_1 * \ldots * \Pi_n)^d$.

3.2 Properties of Updates

Different instantiations of our framework yield different properties. To this end, we examine some properties proposed by Eiter *et al.* [9]. We focus below on the slightly more elaborate operators $*_1$ and $*_2$. For comparison, we also mention whether a property at hand is satisfied by the update operation defined by Eiter *et al.* [9], which we denote by $\circ_e$ (the operator $\circ_e$ is formally defined in Section 4).

The first property is the following:[5]

Initialisation: $\emptyset * \Pi \equiv \Pi$. (Fulfilled by $\circ_e$.)

While this property holds for $*_2$ over all preference strategies, it is not satisfied by $*_1$, no matter which preference-handling strategy is used. To see this, consider the program $\{a \leftarrow, \neg a \leftarrow not\ a\}$. While $\emptyset *_2 \Pi = \Pi$ and hence $\emptyset *_2 \Pi \equiv \Pi$, we get

$$\emptyset *_1 \Pi = \Pi^d = \{a \leftarrow not\ \neg a, \neg a \leftarrow not\ a\} \neq \Pi.$$

Π has the single answer set $\{a\}$, but Π^d admits two answer sets, $\{a\}$ and $\{\neg a\}$.

A similar situation is encountered when regarding the following property:

Idempotency: $\Pi * \Pi \equiv \Pi$. (Fulfilled by $\circ_e$.)

For analogous reasons as above, $*_1$ fails to satisfy this property, while it is satisfied by $*_2$, whenever Π has consistent answer sets (see below).

[5] Henceforth we understand a property to hold for all strategies unless otherwise mentioned.

In fact, despite the lack of the previous two properties, $*_1$ yields only consistent answer sets, even if an update is inconsistent. For instance, in a variation of one of the above examples, updating program $\Pi_1 = \{a \leftarrow\}$ by $\Pi_2 = \{b \leftarrow, \neg b \leftarrow\}$ yields

$$\Pi_1 *_1 \Pi_2 = \Pi_1^d \cup \Pi_2^d = \{a \leftarrow not\ \neg a\} \cup \{b \leftarrow not\ \neg b, \neg b \leftarrow not\ b\},$$

from which we obtain two answer sets, $\{a, b\}$ and $\{a, \neg b\}$. Unlike this, $\Pi_1 *_2 \Pi_2 = \Pi_1 \cup \Pi_2$ has the inconsistent answer set $\mathcal{L}$ (as in the above example).

Another property deals with the addition of tautologies (or more generally, the influence of redundant information).

Tautology: If $head(r) \in body^+(r)$, for all $r \in \Pi_2$, then $\Pi_1 * \Pi_2 \equiv \Pi_1$.
(Violated by $\circ_e$.)

This property is violated by most update approaches in the literature. For example, let us update $\Pi_1 = \{a \leftarrow not\ \neg a, \neg a \leftarrow\}$ by $\Pi_2 = \{a \leftarrow a\}$. No matter which of the above update operators we take, we obtain a single answer set $\{a\}$ from the update program, which is generated by rule $a \leftarrow not\ \neg a$ in Π_1. The conclusion of $\neg a$ is prohibited by the single rule in Π_2, taking precedence over the second one in Π_1.

For another example, consider $((\Pi_1 * \Pi_2) * \Pi_3)$ where

$$\Pi_1 = \{a \leftarrow\},\ \Pi_2 = \{\neg a \leftarrow\},\ \Pi_3 = \{a \leftarrow a\}\ .$$

Clearly, $(\Pi_1 * \Pi_2)$ induces a single answer set $\{\neg a\}$ in all of the above approaches, including $\circ_e$. Unlike this, update operation $\circ_e$ results in two answer sets, $\{\neg a\}$ and $\{a\}$, once Π_3 has been added and so does each update operation in our framework when using the preference-handling strategy B. This is different, however, when using strategy D, in which case we obtain only a single answer set $\{\neg a\}$ from $((\Pi_1 * \Pi_2) * \Pi_3)$. It remains for future work to see how the addition of tautologies can be counterbalanced by stronger preference-handling strategies. A general approach to overcome this deficiency is proposed by Alferes *et al.* [19]; it also remains future work whether that technique applies in our framework as well.

The next property deals with iterated updates.

Associativity: $(\Pi_1 * (\Pi_2 * \Pi_3)) \equiv ((\Pi_1 * \Pi_2) * \Pi_3)$. (Violated by $\circ_e$.)[6]

This property holds for all instances of our framework. In fact, one can show that both updates yield the same update programs.

Absorption: If $\Pi_2 = \Pi_3$, then $((\Pi_1 * \Pi_2) * \Pi_3) \equiv (\Pi_1 * \Pi_2)$. (Fulfilled by $\circ_e$.)

This property is also satisfied by all instances of our framework. This is also the case with the following generalisation of absorption:

Augmentation: If $\Pi_2 \subseteq \Pi_3$, then $((\Pi_1 * \Pi_2) * \Pi_3) \equiv (\Pi_1 * \Pi_3)$. (Violated by $\circ_e$.)

[6] Strictly speaking, in the approach of Eiter *et al.* [9], the associativity principle is formulated not in terms of the update operation itself, but in terms of the associated update program $P_{\triangleleft}$ (see Section 4); the same applies for the disjointness and parallelism properties below.

Table 1. Summary of update properties

	$*_1$	$*_2$	$\circ_e$
Initialisation		$\surd$	$\surd$
Idempotency		$\surd$	$\surd$
Tautology			
Associativity	$\surd$	$\surd$	
Absorption	$\surd$	$\surd$	$\surd$
Augmentation	$\surd$	$\surd$	
Disjointness	$\surd$	$\surd$	$\surd$
Non-Interference	$\surd$	$\surd$	$\surd$

The next property captures update with disjoint programs.

Disjointness: If $atom(\Pi_1) \cap atom(\Pi_2) = \emptyset$, then $(\Pi_1 \cup \Pi_2) * \Pi_3 \equiv (\Pi_1 * \Pi_3) \cup (\Pi_2 * \Pi_3)$. (Fulfilled by $\circ_e$.)

This principle is satisfied by all instances of our framework.

The next property is a variant of the previous.

Parallelism: If $atom(\Pi_2) \cap atom(\Pi_3) = \emptyset$, then $\Pi_1 * (\Pi_2 \cup \Pi_3) \equiv (\Pi_1 * \Pi_2) \cup (\Pi_1 * \Pi_3)$. (Violated by $\circ_e$.)

This property does not hold in our approach. To see this, let $\Pi_3 = \emptyset$. Clearly, we obtain different results from $\Pi_1 * \Pi_2$ and $(\Pi_1 * \Pi_2) \cup \Pi_1$. Arguably, given this example, unrestricted parallelism is not a desirable property.

The last property deals with commutativity when dealing with non-interacting update programs.

Non-Interference: If $atom(\Pi_2) \cap atom(\Pi_3) = \emptyset$, then $(\Pi_1 * \Pi_2) * \Pi_3 \equiv (\Pi_1 * \Pi_3) * \Pi_2$. (Fulfilled by $\circ_e$.)

This property is satisfied by all instances of our framework.

These properties are summarised in Table 1 (with the exception of parallelism, which we feel is undesirable). We note that $*_2$ satisfies the most properties, followed by $\circ_e$, and then $*_1$.

Up to now, we have ignored the treatment of integrity constraints (cf. Baral [1]) in updating logic program. In this respect, we simply follow the approach taken by Eiter *et al.* [9] by handling them as global constraints that are discarded in the defaultification and preference-handling process. Updating a program Π_1 by the program Π_2 in the presence of integrity constraints Π_c then amounts to computing the order-preserving answer sets of $(\Pi_1 * \Pi_2) \cup \Pi_c$. Although we do not detail it here, we mention that our approach allows for accommodating the update of integrity constraints just as well by making them subject to an appropriately adapted defaultification and preference-handling mechanism.

4 Examples and Properties vis-à-vis $\circ_e$

In what follows, we discuss some examples comparing the present update approach with the update approach due to Eiter *et al.* [9]. For simplicity, we focus on our operator $*_1$ under the weakest (reasonable) preference handling strategy, B, given that it is closest to $\circ_e$ (cf. Theorem 3 below).

In the approach of Eiter *et al.* [9], the semantics of an n-fold update $\Pi_1 \circ_e \cdots \circ_e \Pi_n$ is given by the semantics of an (ordinary) program $P_\lhd$, for $P = (\Pi_1, \ldots, \Pi_n)$, containing the following elements:

1. all integrity constraints in Π_i, $1 \leq i \leq n$;
2. for each $r \in \Pi_i$, $1 \leq i \leq n$:
$$l_i \leftarrow \mathit{body}(r), \mathit{not}\ \mathit{rej}(r), \quad \text{where } \mathit{head}(r) = l;$$
3. for each $r \in \Pi_i$, $1 \leq i < n$:
$$\mathit{rej}(r) \leftarrow \mathit{body}(r), \neg l_{i+1}, \quad \text{where } \mathit{head}(r) = l;$$
4. for each literal l occurring in P $(1 \leq i < n)$:
$$l_i \leftarrow l_{i+1}; \qquad\qquad l \leftarrow l_1.$$

Here, for each rule r, $\mathit{rej}(r)$ is a new atom not occurring in $\Pi_1, \ldots, \Pi_n$. Intuitively, $\mathit{rej}(r)$ expresses that r is "rejected." Likewise, each l_i, $1 \leq i \leq n$, is a new atom not occurring in $\Pi_1, \ldots, \Pi_n$. Then, answer sets of $\Pi_1 \circ_e \cdots \circ_e \Pi_n$ are given by the answer sets of $P_\lhd$, modulo the original language. This is similar to compiling ordered logic programs to standard ones as done in our previous work [15].

Example 1. Consider the following programs:

$$\begin{aligned}
\Pi_1 &= \{r_1 : \neg a \leftarrow\},\\
\Pi_2 &= \{r_2 : a \leftarrow b, \mathit{not}\ \neg a\},\\
\Pi_3 &= \{r_3 : b \leftarrow\}.
\end{aligned}$$

The program Π_1 has a single answer set, namely $\{\neg a\}$. In updating Π_1 by Π_2, nothing changes because $r_2^d = r_2$ is not applicable (b is not derivable). A further update by Π_3 changes this situation: b becomes derivable and r_2^d can be applied. In fact, since $r_1^d < r_2^d$, rule r_2^d must be applied before r_1^d and so r_1^d is defeated. Thus, $\{a, b\}$ is the single answer set of $*_1 P$, for $P = (\Pi_1, \Pi_2, \Pi_3)$. Observe that $\{a, b\}$ is of course also an answer set of the unordered program $\Pi_1^d \cup \Pi_2^d \cup \Pi_3^d$, together with $\{\neg a, b\}$. In fact, the latter set is the unique answer set of $\Pi_1 \cup \Pi_2 \cup \Pi_3$, which shows that answer sets of update programs $*P$ are not selected among answer sets of the union of the constituents of P, but rather of the union of the defaultification of its constituents (cf. Theorem 2). Note, however, that $\Pi_1 \circ_e \Pi_2 \circ_e \Pi_3$ has both $\{a, b\}$ and $\{\neg a, b\}$ as answer sets. ◊

The operations $*_1$ and $\circ_e$ also yield different results on inconsistent programs. Consider:

$$\begin{aligned} \Pi_1 &= \{b \leftarrow,\ \neg b \leftarrow\}, \\ \Pi_2 &= \{a \leftarrow\}, \\ \Pi_3 &= \{b \leftarrow\}, \end{aligned}$$

and let $P = (\Pi_1, \Pi_2)$. $P_\triangleleft$ has the set of all literals as its unique (inconsistent) answer set, but $*_1 P$ has $\{a, b\}$ and $\{a, \neg b\}$ as answer sets. On the other hand, both $\Pi_1 \circ_e \Pi_2 \circ_e \Pi_3$ and $\Pi_1 *_1 \Pi_2 *_1 \Pi_3$ have $\{a, b\}$ as unique answer set.

The same holds for programs which may become inconsistent due to new information:

$$\begin{aligned} \Pi_1 &= \{b \leftarrow a,\ \neg b \leftarrow a\}, \\ \Pi_2 &= \{a \leftarrow\}, \\ \Pi_3 &= \{\neg a \leftarrow\}. \end{aligned}$$

Π_1 has a consistent answer set, but $\Pi_1 *_1 \Pi_2$ has $\{a, b\}$ and $\{a, \neg b\}$ as answer sets, whereas $\Pi_1 \circ_e \Pi_2$ has the set of all literals as unique answer set. For the additional update with Π_3, both approaches yield $\{\neg a\}$ as unique answer set.

In general, we can formulate the following relation between the answer sets of $*_1$ and $\circ_e$:

Theorem 3. *Let $P = (\Pi_1, \dots, \Pi_n)$ be a sequence of programs such that $P_\triangleleft$ has only consistent answer sets.*

Then, any answer set of $*_1 P$ is also an answer set of $\Pi_1 \circ_e \cdots \circ_e \Pi_n$.

The converse does not hold, as Example 1 illustrates. Actually, there is an even simpler counterexample: consider

$$\Pi_1 \ = \ \{a \leftarrow\} \quad \text{and} \quad \Pi_2 \ = \ \{\neg a \leftarrow \mathit{not}\ a\}.$$

Then, $\Pi \circ_e \Pi_2$ has two answer sets, viz. $\{a\}$ and $\{\neg a\}$, while $\Pi_1 *_1 \Pi_2$ has only $\{\neg a\}$ as answer set. Actually, $\{\neg a\}$ is the only answer set of $\Pi_1 *_1 \Pi_2$ under any of the three preference strategies B, D, and W.

Finally, let us consider two examples on updating logic programs that have been discussed in the literature, showing that $*_1$ and $\circ_e$ behave the same in these cases.

Example 2 (Adapted from [5]). Consider the update of Π_1 by Π_2, where

$$\begin{aligned} \Pi_1 = \{\, & r_1 : \mathit{sleep} \leftarrow \mathit{not}\ \mathit{tv\_on}, \quad r_2 : \mathit{night} \leftarrow, \\ & r_3 : \mathit{tv\_on} \leftarrow, \quad r_4 : \mathit{watch\_tv} \leftarrow \mathit{tv\_on}\,\}, \\ \Pi_2 = \{\, & r_5 : \neg \mathit{tv\_on} \leftarrow \mathit{power\_failure},\ r_6 : \mathit{power\_failure} \leftarrow \,\}. \end{aligned}$$

The single answer set of both $\Pi_1 *_1 \Pi_2$ and $\Pi_1 \circ_e \Pi_2$ is

$$S = \{\mathit{power\_failure}, \neg \mathit{tv\_on}, \mathit{sleep}, \mathit{night}\}.$$

If new information arrives as program Π_3, given by

$$\Pi_3 = \{\, r_7 : \neg \mathit{power\_failure} \leftarrow \,\},$$

then again $\Pi_1 *_1 \Pi_2 *_1 \Pi_3$ and $\Pi_1 \circ_e \Pi_2 \circ_e \Pi_3$ have the unique answer set

$$T = \{\, \neg power\_failure, tv\_on, watch\_tv, night \,\}. \qquad \Diamond$$

Example 3 ([9]). An agent consulting different sources in search of a performance or a final rehearsal of a concert on a given weekend may be faced with the following situation. First, the agent is notified that there is no concert on Friday:

$$\Pi_1 = \{\, r_1 : \neg concert\_friday \leftarrow \ \}.$$

Later on, a second source reports that there is neither a final rehearsal on Friday nor a concert on Saturday:

$$\Pi_2 = \{\, r_2 : \neg final\_rehearsal\_friday \leftarrow, \quad r_3 : \neg concert\_saturday \leftarrow \ \}.$$

Finally, the agent is assured that there is a final rehearsal or a concert on Friday and that whenever there is a final rehearsal on Fridays, a concert on Saturday or Sunday follows:

$$\begin{aligned}\Pi_3 = \{ & r_4 : concert\_friday \leftarrow not\ final\_rehearsal\_friday,\\ & r_5 : final\_rehearsal\_friday \leftarrow not\ concert\_friday,\\ & r_6 : concert\_saturday \leftarrow final\_rehearsal\_friday, not\ concert\_sunday,\\ & r_7 : concert\_sunday \leftarrow final\_rehearsal\_friday, not\ concert\_saturday \ \}.\end{aligned}$$

The update program $\Pi_1 *_1 \Pi_2 *_1 \Pi_3$ has three answer sets:

$$\begin{aligned} S_1 &= \{final\_rehearsal\_friday, \neg concert\_friday, concert\_saturday\},\\ S_2 &= \{final\_rehearsal\_friday, \neg concert\_friday, \neg concert\_saturday,\\ &\qquad concert\_sunday\},\\ S_3 &= \{\neg final\_rehearsal\_friday, concert\_friday, \neg concert\_saturday\},\end{aligned}$$

where $\Pi_1 *_1 \Pi_2 *_1 \Pi_3$ is given as follows:

$$\begin{aligned} & \Pi_1^d \cup \Pi_2^d \cup \Pi_3^d =\\ & \{\, r_1^d : \neg concert\_friday \leftarrow not\ concert\_friday,\\ & \ \ r_2^d : \neg final\_rehearsal\_friday \leftarrow not\ final\_rehearsal\_friday,\\ & \ \ r_3^d : \neg concert\_saturday \leftarrow not\ concert\_saturday,\\ & \ \ r_4^d : concert\_friday \leftarrow not\ final\_rehearsal\_friday, not\ \neg concert\_friday,\\ & \ \ r_5^d : final\_rehearsal\_friday \leftarrow not\ concert\_friday,\\ & \qquad\qquad not\ \neg final\_rehearsal\_friday,\\ & \ \ r_6^d : concert\_saturday \leftarrow final\_rehearsal\_friday, not\ concert\_sunday,\\ & \qquad\qquad not\ \neg concert\_saturday,\\ & \ \ r_7^d : concert\_sunday \leftarrow final\_rehearsal\_friday, not\ concert\_saturday,\\ & \qquad\qquad not\ \neg concert\_sunday \,\},\end{aligned}$$

together with the following order:

$$r_1^d < r_4^d, \quad r_2^d < r_5^d, \quad \text{and} \quad r_3^d < r_6^d.$$

The same answer sets are obtained in case of $\Pi_1 \circ_e \Pi_2 \circ_e \Pi_3$. $\Diamond$

5 Discussion

We have presented a simple and general framework to updating logic programs under the answer-set semantics. Our approach is based on two well-known techniques in ASP: defaultification and preference handling. Depending on how we fix the interplay among both techniques, we obtain distinct update operations. An interesting feature is that inconsistencies are removed by defaultification and can subsequently be resolved through preferences. Also, the approach is iterative and the history of the continued updates is (mainly[7]) captured by preferences rather than explicit program transformations. Another advantage of this approach is that it is easily implementable by off-the-shelf systems developed for ASP.

More elaboration of the different instances of our framework is desirable. In particular, it will be interesting to see how the properties of the framework change with different ordering mechanisms, in particular, ones that are especially designed for update purposes. Another interesting question is in how far our framework is suitable as a uniform approach in which other approaches can be simulated. As well, although our focus has been on the development of a general framework, it appears that the approaches under strategy $*_2$ have good properties (as evidenced by our survey of properties, as well as the examples presented) and warrant fuller investigation in their own right.

Given the numerous approaches to updating logic programs, among them [2,3,4,5,6,7,8,9,10], a detailed comparison to the literature is beyond the scope of this paper. An excellent survey is given by Eiter *et al.* [9]. We have already compared our approach in some detail to the one of Eiter *et al.* [9]. We summarise here differences with some particularly related approaches. Alferes *et al.* [8] use similar techniques for combining update operations with preferences. However, in contrast to our approach, preferences are not used to implement updates but rather as an additional means for guiding the update. Zhang and Foo [4] were first in mapping update operations onto preference handling in answer-set programming. Unlike us, however, their approach is based on the elimination of conflicting rules and thus on rewriting logic programs. Furthermore, the resulting update program is given in terms of a set of programs, which prohibits iterated update operations as well as a comparison in view of the postulates discussed in Section 3. Interesting recent work [10] gives a framework for characterising update approaches in terms of the notion of *forgetting*.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Foo, N., Zhang, Y.: Towards generalized rule-based updates. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97). Volume 1. Morgan Kaufmann (1997) 82–88
3. Przymusinski, T., Turner, H.: Update by means of inference rules. Journal of Logic Programming **30** (1997) 125–143
4. Zhang, Y., Foo, N.Y.: Updating logic programs. In: Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98). (1998) 403–407

[7] Operator $*_2$ necessitates modifications to the program whenever new conflicts arise.

5. Alferes, J., Leite, J., Pereira, L., Przymusinska, H., Przymusinski, T.: Dynamic updates of non-monotonic knowledge bases. Journal of Logic Programming **45** (2000) 43–70
6. Alferes, J., Pereira, L., Przymusinska, H., Przymusinski, T.: LUPS - A language for updating logic programs. In: Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99). Volume 1730 of Lecture Notes in Artificial Intelligence, Springer (1999) 162–176
7. Inoue, K., Sakama, C.: Updating extended logic programs through abduction. In: Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99). Volume 1730 of Lecture Notes in Artificial Intelligence, Springer (1999) 147–161
8. Alferes, J.J., Dell'Acqua, P., Pereira, L.M.: A compilation of updates plus preferences. In: Proceedings of the Eighth International Conference on Logics in Artificial Intelligence (JELIA 2002). Volume 2424 of Lecture Notes in Computer Science, Springer (2002) 62–73
9. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming **2** (2002) 711–767
10. Zhang, Y., Foo, N.: A unified framework for representing logic program updates. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005), AAAI Press/The MIT Press (2005) 707–713
11. Winslett, M.: Reasoning about action using a possible models approach. In: Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88), AAAI Press/The MIT Press (1988) 89–93
12. Sadri, F., Kowalski, R.: A theorem-proving approach to database integrity. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann Publishers (1987) 313–362
13. Nieuwenborgh, D.V., Vermeir, D.: Preferred answer sets for ordered logic programs. Theory and Practice of Logic Programming **6** (2006) 107–167
14. Brewka, G., Eiter, T.: Preferred answer sets for extended logic programs. Artificial Intelligence **109** (1999) 297–356
15. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3** (2003) 129–187
16. Schaub, T., Wang, K.: A semantic framework for preference handling in answer set programming. Theory and Practice of Logic Programming **3** (2003) 569–607
17. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Computing preferred answer sets by meta-interpretation in answer set programming. Theory and Practice of Logic Programming **3** (2003) 463–498
18. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Proceedings of the Seventh International Conference on Logic Programming (ICLP'90), MIT Press (1990) 579–597
19. Alferes, J.J., Banti, F., Brogi, A., Leite, J.A.: Semantics for dynamic logic programming: A principle-based approach. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004). Volume 2923 of Lecture Notes in Computer Science, Springer (2004) 8–20

Well-Founded Semantics and the Algebraic Theory of Non-monotone Inductive Definitions[⋆]

Marc Denecker and Joost Vennekens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{marcd,joost}@cs.kuleuven.be

Abstract. Approximation theory is a fixpoint theory of general (monotone and non-monotone) operators which generalizes all main semantics of logic programming, default logic and autoepistemic logic. In this paper, we study inductive constructions using operators and show their confluence to the well-founded fixpoint of the operator. This result is one argument for the thesis that Approximation theory is the fixpoint theory of certain generalised forms of (non-monotone) induction. We also use the result to derive a new, more intuitive definition of the well-founded semantics of logic programs and the semantics of ID-logic, which moreover is easier to implement in model generators.

1 Introduction

This paper studies inductive constructions in relation to the well-founded semantics. The study of induction can be defined as the investigation of a class of effective construction techniques in mathematics. There, sets are frequently defined through a constructive process of iterating some recursive *recipe* that adds new elements to the set given that one has established the presence or absence of other elements in the set. In an inductive definition, this recipe is often represented as a collection of informal rules representing base cases and inductive cases. Inductive rules may be *monotone* or *non-monotone*. Consider for example the well-known definition of satisfiability, denoted $I \models \varphi$, by induction on the structure of (propositional) formulas:

– $I \models P$ if $P \in I$ and P is a propositional variable;
– $I \models \psi \wedge \phi$ if $I \models \psi$ and $I \models \phi$;
– $I \models \neg\psi$ if $I \not\models \psi$.

The third rule states that I satisfies $\neg\psi$ if I does *not* satisfy ψ. This is a non-monotone rule, in the sense that it adds a pair $(I, \neg\psi)$ in absence of the pair (I, ψ), and therefore, applying the "recipe" to sets of formulas does not preserve the order $\subseteq$.

Different forms of inductive constructions have been studied extensively in mathematical logic. Monotone induction was studied starting with [19], and

⋆ Works supported by IWT VLaanderen, FWO Vlaanderen.

later in landmark studies such as [1,16,20]. Also non-monotone forms of induction have been studied, such as *inflationary induction* [17] and iterated inductive definitions [10]. In computational logic, the results of these studies were used in extensions of logic with fixpoints constructs, such as FO(LFP) (see, e.g., [9]), μ-calculus and description logics. Inductive definitions are also related to logic programming. It was argued in [3,8] that the well-founded semantics of logic programming [23] correctly formalizes the semantics of different types of definitions that can be found in mathematics, e.g., recursion-free definitions, monotone inductive definitions, and non-monotone inductive definitions such as inductive definitions over well-founded orders (e.g., the definition of $\models$) and iterated inductive definitions. The fact that the inability to express inductive definitions is a well-known weakness of first order logic, has subsequently motivated an extension of FO with a new construct for representing definitions, whose semantics is based on the well-founded semantics [2,7,8]. This logic FO(ID), also called ID-logic, is in some sense an integration of classical logic and logic programming, and can be viewed equally well as a new member of the family of fixpoint logics and even as new (very general sort of) description logic. FO(ID) has recently been proposed as (one of) the underlying language for a constraint programming framework [15].

The study of inductive definitions is strongly related to fixpoint theory. An inductive definition corresponds to an algebraic lattice operator, and, in the monotone case, the object "defined" by such an operator is the least fixpoint. In this sense, Tarski's fixpoint theory of monotone operators [21] can be considered as an entirely abstract algebraic theory of monotone induction. This naturally raises the question whether this theory can be extended to general (monotone and non-monotone) operators in a way that matches with different forms of non-monotone induction. Such extensions are well-known for inflationary induction, but not for induction over a well-founded order or its generalization, iterated induction. But there is a promising candidate. Building on Fitting's work [12] on semantics of logic programming in bilattices, Denecker, Marek and Truszczynski showed that all main types of semantics of a logic program can be characterized algebraically in terms of the three-valued immediate consequence operator of logic programs[4,6]. The underlying theory, in [4] called *Approximation theory*, is an algebraic fixpoint theory for (bilattice extensions of) general lattice operators which defines the so-called *Kripke-Kleene, stable* and *well-founded* fixpoints of an operator. In case of the immediate consequence operator, these fixpoints are the models of the logic program in the corresponding semantics. This suggests that the well-founded fixpoint construction in this theory is the missing fixpoint theory of iterated induction.

The main goal of this paper is to explore the link between inductive definitions and Approximation theory. As a point of departure, we observe that an inductive definition defines a set by describing how to construct it, and as such, it is essentially a description of a construction process. Informally, such a process starts from the empty set, and proceeds by iteratively applying rules of the definition with a satisfied antecedent, until saturation follows. Of course, as a specification of such a process, an inductive definition is highly non-deterministic. Indeed, in the case of monotone induction, at any intermediate stage, many rules may be applicable. It is a key property of this sort of induction that the order in which the rules are applied does not matter: all such construction processes produce the same outcome. For non-monotone induction, on the other hand, the provided well-founded order must be respected (e.g., applying the third rule to derive $I \models \neg\phi$ can only be done *after* it has been established whether $I \models \phi$), but still, there remain infinitely many ways in which $\models$ can be constructed. The first goal of this paper is to formalize these construction processes and to study how they relate to the well-founded fixpoint construction of Approximation theory. We will define the notion of a *well-founded induction* of a (bilattice extension of an) operator, and demonstrate that all such inductions converge to the well-founded fixpoint.

Secondly, we will concretize this notion of a well-founded induction in the context of ID-logic. This leads to a new, intuitive and much simpler characterization of the well-founded model of a definition (and of a logic program) which does not involve fixpoint operators anymore. Our results thus allow to simplify the definition of the semantics of FO(ID), and of the well-founded semantics of logic programming.

Thirdly, our study of induction sequences has also computational relevance. First, the non-deterministic inference processes that we describe here generalize various methods of well-founded model computation as presented in e.g. [13]. Second, in ASP systems such as SModels [18], and in the FO(ID) model generator MIDL [14], the well-founded model is computed through a kind of constraint propagation mechanism. These systems do not iterate the immediate consequence operator, for this is too expensive. Instead, they iteratively perform inference steps, inferring truth or falsity of an atom with a true, respectively false body, and inferring falsity of unfounded sets. These computation steps are exactly the atomic inference steps that make up a well-founded induction. Thus, our study can give insight in the properties of intermediate objects constructed during such a constraint propagation proces, and may lead to easier correctness proofs for such systems.

2 Approximation Theory

Our presentation of Approximation theory is based on [4,6].

A structure $\langle L, \leq \rangle$ is a poset if $\leq$ is a partial order on the set L, i.e., a reflexive, asymmetric, transitive relation. The relation $\leq$ is a total order if, in addition, for each $x, y \in L, x \leq y$ or $y \leq x$. A subset S of a poset L is a chain if $\leq$ is a total order in S. The structure $\langle L, \leq \rangle$ is chain-complete if each of its chains C has a least upperbound $lub_{\leq}(C)$, and is a complete lattice if each subset $S \subseteq L$ has a least upperbound $lub_{\leq}(S)$ and a greatest lowerbound $glb_{\leq}(S)$. A chain complete poset has a least element $\bot$ and a complete lattice has both a least element $\bot$ and a greatest element $\top$.

Given a poset $\langle L, \leq \rangle$, an operator $O : L \to L$ is called $\leq$-monotone if O preserves $\leq$, i.e., $x \leq y$ implies $O(x) \leq O(y)$. An element $x \in L$ is a pre-fixpoint of O if $O(x) \leq x$, a fixpoint if $O(x) = x$ and a post-fixpoint if $x \leq O(x)$. The sets of all such $x \in L$ are denoted $Pre(O)$, respectively $Fix(O)$ and $Post(O)$. A monotone operator in a chain complete poset or a complete lattice has a least fixpoint which is also its least pre-fixpoint and the limit of the increasing sequence $\langle x_\xi \rangle_{\xi \geq 0}$, defined by transfinite induction:

– $x_0 = \bot$;
– $x_{\xi+1} = O(x_\xi)$;
– $x_\lambda = lub(\{x_\xi | \xi < \lambda\})$, for limit ordinals λ.

In Approximation theory, pairs $(x, y) \in L^2$ are used to approximate certain elements of L, namely those in the (possibly empty) interval $[x, y] = \{z \in L \mid x \leq z \leq y\}$. Abusing this correspondence between pairs and intervals, we sometimes write $z \in (x, y)$ instead of $z \in [x, y]$, to denote that (x, y) approximates z. On L^2, two natural orders can be defined:

$$\begin{array}{lll} \text{the product order:} & (x,y) \leq (u,v) & \text{if } x \leq u \text{ and } y \leq v \\ \text{the precision order:} & (x,y) \leq_{\text{p}} (u,v) & \text{if } x \leq u \text{ and } v \leq y \end{array}$$

The precision order is the most important. Indeed, if $(x, y) \leq_{\text{p}} (u, v)$, then $[x, y] \supseteq [u, v]$, i.e., (u, v) approximates fewer elements than (x, y). If L is a complete lattice, then both these orders are complete lattice orders in L^2.

In this paper, the relevant pairs of L^2 are the *consistent* pairs. A pair $(x, y) \in L^2$ is consistent if $x \leq y$ (or $[x, y] \neq \emptyset$), and the subset of L^2 consisting of such pairs is denoted L^c. The order $\leq$ is a complete lattice order in L^c, but $\leq_{\text{p}}$ is not, because L^c has no most precise element. However, $\langle L^c, \leq_{\text{p}} \rangle$ is chain-complete. Elements (x, x) of L^2 are called *exact*. The set of exact elements forms a natural embedding of L in L^2. They are also the maximally precise elements of L^c.

Approximation theory studies fixpoints of lattice operators $O : L \rightarrow L$ through the use of approximations of O. We define that an operator $A : L^2 \rightarrow L^2$ is a *approximator* if it is $\leq_{\text{p}}$-monotone. An approximator is *consistent* if it maps consistent pairs to consistent pairs. An approximator A *approximates* an operator $O\colon L \rightarrow L$ (*is an approximation of* O) if for each $x \in L$, $O(x) \in A(x, x)$. Such an operator A provides approximate information on O. Indeed, when $z \in (x, y)$, then $\leq_{\text{p}}$-monotonicity gives us $O(z) \in A(z, z) \subseteq A(x, y)$. Or, $O(z)$ is approximated by $A(x, y)$, and, abusing the duality of pairs and intervals, $O([x, y]) \subseteq A(x, y)$. It is easy to see that when A approximates an operator O, then A is consistent. For this reason, below we only consider consistent approximators. An approximator A is exact if it preserves exactness, i.e., if for all $x \in L$, $A(x, x)$ is exact. In general, an approximator A approximates a collection of lattice operators O, but when A is exact, then the only approximated operator is the operator which maps each $x \in L$ to $A(x, x)_1$ $(= A(x, x)_2)$. An approximator A is *symmetric* if for all $(x, y) \in L^2$, if $A(x, y) = (x', y')$ then $A(y, x) = (y', x')$. A symmetric approximator is exact.

For an approximator A on L^2 and lattice elements $x, y \in L$, the operators $\lambda z.A(z, y)_1$ and $\lambda z.A(x, z)_2$ on L will be denoted $A(\cdot, y)_1$, respectively $A(x, \cdot)_2$. These operators are monotone. We define an operator $(\cdot)^{A\downarrow}$ on L, called the *downward revision operator* of A, as $y^{A\downarrow} = \mathit{lfp}(A(\cdot, y))_1$ for each $y \in L$. We also define the *upward revision operator* $(\cdot)^{A\uparrow}$ of A as $x^{A\uparrow} = \mathit{lfp}(A(x,\cdot)_2)$ for every $x \in L$. Note that if A is symmetric, both operators are identical. We define the *stable operator* $\mathcal{S}t_A : L^2 \to L^2$ of A by $\mathcal{S}t_A(x,y) = (y^{A\downarrow}, x^{A\uparrow})$. It can easily be seen that both $(\cdot)^{A\downarrow}$ and $(\cdot)^{A\uparrow}$ are anti-monotone. It follows that $\mathcal{S}t_A$ is $\leq_{\mathrm{p}}$-monotone.

An approximator A defines a number of different fixpoints: the $\leq_{\mathrm{p}}$-least fixpoint of A, denoted $k(A)$, is called its *Kripke-Kleene fixpoint*, fixpoints of its stable operator $\mathcal{S}t_A$ are *stable fixpoints* and the $\leq_{\mathrm{p}}$-least fixpoint of $\mathcal{S}t_A$, denoted $w(A)$, is called the *well-founded fixpoint* of A. In [4,5], it was shown that all main semantics of logic programming, autoepistemic logic and default logic can be characterized in terms of the different types of fixpoints of approximation operators associated to theories in these logics. For example, in the context of logic programming, the four-valued van Emden-Kowalski operator $\mathcal{T}_P$ of a logic program P is a symmetric approximation of the two-valued van Emden-Kowalski operator. The downward revision operator of $\mathcal{T}_P$ (which is equal to the upward one, since $\mathcal{T}_P$ is symmetric) coincides with the Gelfond-Lifschitz stable operator P. The Kripke-Kleene, well-founded, stable and exact stable fixpoints of $\mathcal{T}_P$ coincide with, respectively, the Kripke-Kleene model, the well-founded model, the four-valued stable models and the stable models of the logic program P.

Given an approximator A on L^2, we denote by A^c its restriction to L^c. Conversely, any approximator A on L^c, i.e., a $\leq_{\mathrm{p}}$-monotone L^c-operator, can be extended to an approximator on L^2, in many ways. When A is exact then A can be extended to a symmetric approximator on L^2, in many ways. It was shown in [6], that all symmetric extensions of A have the same consistent stable fixpoints, the same well-founded fixpoint and the same exact stable fixpoints. This suggests that consistent stable fixpoints can also be algebraically characterized in terms of A^c. As shown in [6], this is indeed the case but the alternative characterization is slightly more more tedious, mainly because the revision operators $(\cdot)^{A\uparrow}$ and $(\cdot)^{A\downarrow}$ are only partial functions, since $A(\cdot,y)_1$ and $A(x,\cdot)_2$ are not operators on L, but only functions from $[\bot, y]$, respectively $[x, \top]$, to L. Consequently, they may have no least fixpoint.

A lattice operator O can have multiple approximations. This raises the question of how the different types of fixpoints of these approximators relate to each other. By point-wise extension of the precision order $\leq_{\mathrm{p}}$ on L^c, we obtain a precision order between L^c-approximators. When $A \leq_{\mathrm{p}} B$, then any operator O approximated by B is also approximated by A and $k(A) \leq_{\mathrm{p}} k(B)$, $w(A) \leq_{\mathrm{p}} w(B)$, and the set of exact stable fixpoints of A is a subset of that of B. Also, a lattice operator O has a most precise L^c-approximator, called the ultimate approximation. This operator, denoted $\mathcal{U}_O$, maps any tuple (x,y) to $(\mathit{glb}(O([x,y])), \mathit{lub}(O([x,y])))$. Because it is the most precise, its Kripke-Kleene and well-founded fixpoints are the most precise of all approximations of O, and the set of its exact stable fixpoints comprises the exact stable fixpoints of all approximations of O.

The precision order can be further extended to L^2-approximators, by defining $A \leq_{\mathrm{p}} B$ if $A^c \leq_{\mathrm{p}} B^c$ (or, equivalently, $A(x,y) \leq_{\mathrm{p}} B(x,y)$, for each $(x,y) \in L^c$).

3 Monotone and Well-Founded Inductions

Let $\langle L, \leq \rangle$ be a complete lattice and O a monotone operator on L.

Definition 1. *A monotone induction of O is a (possibly transfinite) sequence $\langle x_\xi \rangle_{\xi \leq \alpha}$ such that*

- $x_0 = \bot$;
- $x_\xi < x_{\xi+1} \leq O(x_\xi)$, *for every* $\xi < \alpha$;
- $x_\lambda = lub(\{x_\xi | \xi < \lambda\})$, *for every limit ordinal* $\lambda \leq \alpha$.

A monotone induction $\langle x_\xi \rangle_{\xi \leq \alpha}$ is terminal if it cannot be extended, i.e., there is no $x_{\alpha+1}$ such that $\langle x_\xi \rangle_{\xi \leq \alpha+1}$ is a monotone induction.

Clearly, a monotone induction is an increasing sequence and x_α is its limit. Note that the standard construction of the least fixpoint $lfp(O)$ is a terminal monotone induction. All terminal monotone inductions are confluent, i.e., have the same limit, namely $lfp(O)$.

Proposition 1. *The limit of each terminal monotone induction of O is $lfp(O)$.*

There are many ways in which a set, defined by monotone induction, can be constructed. E.g., the transitive closure T of a graph R can be constructed by an arbitrary process of (non-deterministically) selecting an edge (a, b) from R and adding it to T, or finding a pair $(a, b), (b, c)$ of edges in the current set T and extending this set with (a, c). All these processes lead to the same outcome, namely the transitive closure of R. Proposition 1 formalizes this property.

Let us now investigate the case of arbitrary lattice-operators O. Assume that we have an approximation A of O on L^2. First, note that A is a $\leq_{\mathrm{p}}$-monotone operator, so we can construct monotone inductions with A. Each terminal monotone induction of A constructs the Kripke-Kleene fixpoint $k(A)$.

Observe that a monotone induction of a consistent approximator A consists only of consistent pairs. Therefore, a monotone induction of such an A is also a monotone induction of any more precise operator B, because for any $x_{\xi+1}$ in such a sequence, $x_{\xi+1} \leq_p A(x_\xi)$ then implies that also $x_{\xi+1} \leq_p B(x_\xi)$. It follows from this that $k(A) \leq_{\mathrm{p}} k(B)$, as claimed earlier.

The weakness of the Kripke-Kleene fixpoint construction surfaces when we consider the case that O is monotone. Since $k(A)$ approximates all fixpoints of O, we have $k(A) \leq_{\mathrm{p}} (lfp(O), gfp(O))$. We therefore need to consider more precise constructions.

We call a pair $(x', y') \in L^2$ an A-refinement of $(x, y) \in L^2$ if:

- $(x, y) <_p (x', y') \leq_{\mathrm{p}} A(x, y)$; or
- $x' = x$ and $y' < y$ and $A(x, y')_2 \leq y'$.

Note that the second case is equivalent to saying that y' must be a pre-fixpoint of $A(x, \cdot)_2$. It follows that if $x^{A\uparrow} < y$, then taking $y' = x^{A\uparrow}$ gives us the least value for which (x, y') is an A-refinement by the second rule.

Definition 2. *A* well-founded induction *of A in (x, y) is a sequence $\langle(x_\xi, y_\xi)\rangle_{\xi\leq\alpha}$ such that*

- *$(x_0, y_0) = (\bot, \top)$;*
- *$(x_{\xi+1}, y_{\xi+1})$ is an A-refinement of (x_ξ, y_ξ), for each $\xi < \alpha$;*
- *$(x_\lambda, y_\lambda) = lub(\{(x_\xi, y_\xi) : \xi < \lambda\})$, for limit ordinal $\lambda \leq \alpha$.*

A well-founded induction is terminal *if its limit (x_α, y_α) has no A-refinement.*

A well-founded induction is a $\leq_\mathrm{p}$-increasing sequence of pairs with limit (x_α, y_α). The main task now is to prove that well-founded inductions are confluent and produce the well-founded fixpoint. This is the main technical contribution of this paper.

The proof of the convergence of all well-founded inductions is based on an invariance analysis. We will show that all pairs constructed during a well-founded induction satisfy certain invariants and that there is exactly one pair that satisfies these invariants and has no A-refinement. Hence, all well-founded inductions must converge to this pair.

The first invariant is A-contractingness. Recall that all elements in a monotone induction are post-fixpoints. A post-fixpoint (a, b) of A has the interesting property that $O([a, b]) \subseteq A(a, b) \subseteq (a, b)$. Therefore, the operator O is internal in $[a, b]$. In fact, it is *contracting* in $[a, b]$ since $(a, b) \supseteq A(a, b) \supseteq A^2(a, b) \supseteq \dots$. This property is our motivation for calling a post-fixpoint of A an A-*contracting pair*[1].

Proposition 2. *Each pair in a well-founded induction of A is A-contracting.*

The second invariant aims to express that the lower bound of a pair in an well-founded induction cannot grow too large. For example, if O is monotone, then the pair $(gfp(O), \top)$ could be contracting w.r.t. some approximation A. Unless $lfp(O) = gfp(O)$, this pair would never occur during a well-founded induction because $gfp(O)$ is too large.

Definition 3. *A pair (a, b) is A-prudent if $a \leq x$ for every $x \in L$ such that $A(x, b)_1 \leq x$.*

Equivalently, (a, b) is A-prudent if a is less than each pre-fixpoint x of $A(\cdot, b)_1$, or, more compactly, if $a \leq b^{A\downarrow}$. This definition extends the notion of A-prudence of L^c-approximators in [6] to the case of L^2-approximators.

When O is a monotone operator, then for each symmetric ultimate approximation $\mathcal{U}_O$ of O on L^2, for every pair (x, y), $\mathcal{U}_O(x, y)_1 = O(x)$. Consequently, a pair (a, b) is $\mathcal{U}_O$-prudent if a is less than each pre-fixpoint of O or equivalently, if $a \leq lfp(O)$.

Clearly, the least precise pair $(\bot, \top)$ is A-prudent. Since taking A-refinements and taking limits of A-prudent sequences both preserve A-prudence, we obtain a second invariant.

[1] In [6], A-contracting pairs were called A-reliable.

Proposition 3. *Each pair in a well-founded induction of A is A-prudent.*

The third invariant is consistency. To obtain this, however, we need to impose an additional condition on A.

Definition 4. *We say that an approximator A* gracefully degrades *if for all $(x, y) \in L^2$, $A(y, x)_1 \leq A(x, y)_2$.*

The intuition behind this definition is that the behaviour of such an operator on inconsistent pairs is constrained by its behaviour on consistent pairs. It cannot, for example, map all inconsistent pairs to the most precise pair $(\top, \bot)$. Clearly, a symmetric approximator gracefully degrades.

Lemma 1. *Assume that A degrades gracefully. If (a, b) is A-prudent and consistent, then $a \leq a^{A\uparrow}$.*

Proposition 4. *Each pair in a well-founded induction of a gracefully degrading approximator A is consistent.*

As mentioned in Section 2, all symmetric approximators extending an exact L^c-approximator A have the same consistent stable fixpoints. A more general condition that guarantees this is graceful degradation.

Corollary 1. *Two gracefully degrading L^2-approximators A, B for which $A^c = B^c$, have the same consistent stable fixpoints (and hence, $w(A) = w(B)$).*

A fourth invariant is that each element in a well-founded induction is less than each stable fixpoint. Recall that a stable fixpoint (c, d) satisfies $c = d^{A\downarrow}$ and $d = c^{A\uparrow}$.

Proposition 5. *Let (c, d) be a stable fixpoint of A. If $(a, b) \leq_{\mathrm{p}} (c, d)$, then for each (u, v) such that $(a, b) <_p (u, v) \leq_{\mathrm{p}} A(a, b)$, $(u, v) \leq_{\mathrm{p}} (c, d)$. If $(a, b) \leq_{\mathrm{p}} (c, d)$ then for each $y < b$ such that $A(a, y)_2 \leq y$, $(a, y) \leq_{\mathrm{p}} (c, d)$.*

Clearly, $(\bot, \top)$ approximates all stable fixpoints of A. This property is preserved by taking A-refinements and by taking limits of sequences of increasing precision. From this, we obtain the fourth invariant of well-founded inductions.

Proposition 6. *For each pair (x, y) in a well-founded induction of A and each stable fixpoint of (c, d) of A, $(x, y) \leq_p (x, d)$.*

We have now identified four main invariants. It follows that the limit (x, y) of a well-founded induction is contracting, prudent, less precise than each stable fixpoint of A and, if A gracefully degrades, consistent. In addition, we know that (x, y) has no A-refinement. What can be concluded from this?

Proposition 7. *Let (a, b) be an A-contracting, A-prudent pair such that (a, b) has no A-refinement. Then (a, b) is a stable fixpoint of A.*

Theorem 1. *There exists a least precise stable fixpoint of A, and it is the limit of each terminal well-founded induction of A. If A is gracefully degrading, then this least precise stable fixpoint is consistent.*

This theorem shows that all terminal well-founded inductions indeed reach the same limit and, moreover, that this limit is precisely the well-founded model.

Proposition 8. *Let A, B be gracefully degrading approximators on L^2 such that $A \leq_{\mathrm{p}} B$. A well-founded induction of A is a well-founded induction of B.*

In [6], it was proven that $w(A) \leq_{\mathrm{p}} w(B)$, which is also a corollary of the above proposition.

Another theorem links monotone inductions with well-founded inductions. One of the symmetric ultimate approximations of a monotone lattice operator $O : L \to L$ is the operator $\mathcal{U}_O : L^2 \to L^2$ which maps (x, y) to $(O(x), O(y))$ [6].

Theorem 2. *For any terminal monotone induction $\langle x_\xi \rangle_{\xi \leq \alpha}$ of O, the sequence $\langle (x_\xi, y_\xi) \rangle_{\xi \leq \alpha+1}$ with $y_\xi = \top$ for every $\xi \leq \alpha$ and $x_{\alpha+1} = y_{\alpha+1} = lfp(O)$, is a terminal well-founded induction of $\mathcal{U}_O$.*

4 Well-Founded Semantics of ID-Logic Definitions

We assume familiarity with classical logic. A vocabulary Σ consists of a set of predicate and function symbols. Propositional symbols and constants are 0-ary predicate symbols, respectively function symbols. Terms and FO formulas are defined as usual, and are built inductively from variables, constant and function symbols and logical connectives and quantifiers.

A *definition* is a set of rules of the form

$$\forall \bar{x} \quad (P(\bar{t}) \leftarrow \phi)$$

where ϕ is a FO formula over Σ and $\bar{t}$ is a tuple of terms over Σ such that the free variables of ϕ and the variables of $\bar{t}$ all occur in $\bar{x}$. We call $P(\bar{t})$ the head of the rule, and ϕ the body. The connective $\leftarrow$ is called *definitional implication* and is to be distinguished from material implication $\supset$. A predicate appearing in the head of a rule of a definition Δ is called a *defined predicate* of Δ , any other symbol is called an *open symbol* of Δ. The sets of defined predicates, respectively open symbols of Δ are denoted $Def(\Delta)$, respectively $Open(\Delta) = \Sigma \setminus Def(\Delta)$. For simplicity, we assume that every rule is of the form $\forall \bar{x}\ (P(\bar{x}) \leftarrow \phi)$. Every rule $\forall \bar{x} \quad (P(\bar{t}) \leftarrow \phi)$ can be transformed in an equivalent rule of that form. An FO(ID) (or ID-logic) formula is a boolean combination of FO formulas and definitions. An FO(ID) theory is a set of FO-ID formulas without free variables.

The semantics of the FO(ID) is an integration of standard two-valued FO semantics with the well-founded semantics of definitions. For technical reasons, we need to introduce some concepts from three-valued logic. Consider the set $\mathcal{THREE} = \{\mathbf{f}, \mathbf{u}, \mathbf{t}\}$. The *truth order* $\leq$ on this set is induced by $\mathbf{f} < \mathbf{u} < \mathbf{t}$; the *precision order* $\leq_{\mathrm{p}}$ is induced by $\mathbf{u} <_p \mathbf{f}$ and $\mathbf{u} <_p \mathbf{t}$. Define $\mathbf{f}^{-1} = \mathbf{t}, \mathbf{u}^{-1} = \mathbf{u}, \mathbf{t}^{-1} = \mathbf{f}$.

Given a domain D, a *value* for a n-ary function symbol is a function from D^n to D. A value for an n-ary predicate symbol is a function from D^n to $\mathcal{THREE}$. A Σ-interpretation I consists of a domain D^I, and a value σ^I for each symbol $\sigma \in \Sigma$. A two-valued interpretation is one in which predicates have range $\{\mathbf{f}, \mathbf{t}\}$. For each interpretation F for the function symbols of Σ, both truth and precision order have a pointwize extension to an order on all Σ-interpretations extending F.

A domain atom of I is a tuple of a predicate $P \in \Sigma$ and a tuple $(a_1, \dots, a_n) \in D^n$; it will be denoted $P(a_1, \dots, a_n)$, or more compactly, $P(\bar{a})$.

For a given Σ-interpretation I, symbol σ and a value v for σ, we denote by $I[\sigma/v]$ the $\Sigma \cup \{\sigma\}$-interpretation, that assigns to all symbols the same value as I, except that $\sigma^{I[\sigma/v]} = v$. Likewise, for a domain atom $P(\bar{a})$ and a truth value $v \in \mathcal{THREE}$, we define $I[P(\bar{a})/v]$ as the interpretation I' identical to I except that $P(\bar{a})^{I'} = P^{I'}(\bar{a}) = v$. Similarly, for any set U of domain atoms, $I[U/v]$ is identical to I except that all atoms in U have value v.

When all symbols of term t are interpreted in I, we define its value t^I using the standard induction. The truth value φ^I of an FO sentence φ in I is defined by induction on the subformula order:

- $P(t_1, \dots, t_n)^I := P^I(t_1^I, \dots, t_n^I)$;
- $(\psi \wedge \phi)^I := Min_{\leq}(\psi^I, \phi^I)$;
- $(\neg\psi)^I := (\psi^I)^{-1}$;
- $(\exists x\ \psi)^I = Max_{\leq}(\{\psi^{I[x/d]} \mid d \in D^I\})$.

We now define the semantics of definitions. Let Δ be a definition over Σ and O a two-valued $Open(\Delta)$-interpretation. Consider the collection $\mathcal{V}_O^\Sigma$ of three-valued Σ-structures extending O. On this set, we define the three-valued immediate consequence operator Ψ_Δ^O, also called the Fitting operator, which maps any $I \in \mathcal{V}_O^\Sigma$ to the O-extension J such that for each defined domain atom $P(\bar{a})$,

$$P(\bar{a})^J = Max_{\leq}(\{\varphi(\bar{a})^J | \forall \mathbf{x}(P(\mathbf{x}) \leftarrow \varphi) \in \Delta\}).$$

The Fitting operator [11] is the extension of the van Emden-Kowalski operator to three-valued structures.

Let L be the lattice of two-valued Σ-structures extending O. As shown in [6], $\mathcal{V}_O^\Sigma$ is isomorphic with L^c and the correspondence is between three-valued interpretations K and tuples of two-valued interpretations (I, J) such that for each domain atom $P(\bar{a})$,

$$\begin{cases} P(\bar{a})^K = \mathbf{t} \text{ and } P(\bar{a})^I = \mathbf{t} = P(\bar{a})^J; \\ P(\bar{a})^K = \mathbf{u} \text{ and } P(\bar{a})^I = \mathbf{f}, P(\bar{a})^J = \mathbf{t}; \\ P(\bar{a})^K = \mathbf{f} \text{ and } P(\bar{a})^I = \mathbf{f} = P(\bar{a})^J. \end{cases}$$

We denote the two components of a three-valued K by K_1 and K_2. In this view, the Fitting operator is an exact L^c-approximation of the van Emden-Kowalski operator, and has a well-founded fixpoint, denoted I_o^Δ. This is, in general, a three-valued structure. We extend the truth valuation function φ^I to all FO(ID) formulas by extending the above recursive rules with a new base case for definitions. For a given three-valued structure I and definition Δ, we define $\Delta^I = \mathbf{t}$ if $I = (I|_{Open(\Delta)})^{\Delta}$, and $\Delta^I = \mathbf{f}$ otherwise.

We are now ready to define the semantics of FO(ID). A structure I satisfies a FO(ID) sentence φ (is a model of φ) if I is two-valued and $\varphi^I = \mathbf{t}$. As usual, this is denoted $I \models \varphi$. I satisfies a FO(ID) theory T if I satisfies every $\varphi \in T$. Note that the semantics is two-valued and extends the semantics of classical logic. The restriction to consider only two-valued well-founded models boils down to the requirement that a definition Δ should be *total*, i.e., should define the truth of all defined domain atoms (see [8]).

We now apply the results of the previous section to derive an alternative definition of the well-founded model, which is simpler, more intuitive and more flexible than the one above. We first generalize the well-known concept of an unfounded set [23].

Definition 5. *Given a definition Δ and a three-valued Σ-structure I, an unfounded set of Δ in I is a non-empty set U of defined domain atoms such that each $P(\bar{a}) \in U$ is unknown in I and for each rule $\forall \bar{x}\ (P(\bar{x}) \leftarrow \phi(\bar{x})) \in \Delta$, $\phi(\bar{a})^{I[U/\mathbf{f}]} = \mathbf{f}$.*

When U is an unfounded set in an interpretation I which corresponds to a pair (J, K), then $I[U/\mathbf{f}]$ corresponds to $(J, K[U/\mathbf{f}])$. If, in addition, I is Ψ_Δ^O-contracting, then it is easy to see that each domain atom $P(\bar{a})$, false in $I[U/\mathbf{f}]$, is false in $\Psi_\Delta^O(I[U/\mathbf{f}])$, or, equivalently, $\Psi_\Delta^O(I[U/\mathbf{f}])_2 \leq I[U/\mathbf{f}]_2 = K[U/\mathbf{f}]$. Hence, $I[U/\mathbf{f}]$ is a Ψ_Δ^O-refinement of I.

Definition 6. *We define a well-founded induction of a definition Δ in an Open (Δ)-interpretation O as a sequence $\langle I^\xi \rangle_{\xi \leq \alpha}$ of three-valued Σ-structures extending O such that:*

- *for every defined predicate symbol P, P^{I^0} is the constant function* $\mathbf{u}$*,*
- *for each limit ordinal $\lambda \leq \alpha$, $I^\lambda = lub_{\leq_p}(\{I^\xi \mid \xi < \lambda\})$, and*
- *for every ordinal ξ, $I^{\xi+1}$ relates to I^ξ in one of the following ways.*
  - *$I^{\xi+1} := I^\xi[P(\bar{a})/\mathbf{t}]$, for some domain atom $P(\bar{a})$, unknown in I^ξ, such that for some rule $\forall \bar{x}\ (P(\bar{x}) \leftarrow \phi(\bar{x})) \in \Delta$, $\phi(\bar{a})^{I^\xi} = \mathbf{t}$;*
  - *$I^{\xi+1} := I^\xi[U/\mathbf{f}]$, where U is an unfounded set of Δ in I^ξ.*

A well-founded induction is terminal *if it cannot be extended anymore.*

We will call an interpretation $I[P(\bar{a})/\mathbf{t}]$ or $I[U/\mathbf{f}]$ satisfying the conditions in the above definition a Δ-refinement of I in O.

In such a sequence, for each $\xi < \alpha$, it either holds that $I^\xi <_p I^{\xi+1} = I^\xi[P(\bar{a})/\mathbf{t}] \leq_p \Psi_\Delta^O(I^\xi)$, or $I^{\xi+1} = I^\xi[U/\mathbf{f}]$ with U an unfounded set. It follows that $I^{\xi+1}$ is a Ψ_Δ^O-refinement. Hence, each well-founded induction of Δ in O is a well-founded induction of Ψ_Δ^O. The inverse is clearly not the case (in an induction of Ψ_Δ^O, many atoms can be made true at the same time). Still, a terminal well-founded induction of Δ with limit I^α is a terminal induction of Ψ_Δ^O. Indeed, suppose I^α has a Ψ_Δ^O-refinement J. Then either it must be that I^α has an unfounded set U, or $I^\alpha <_p \Psi_\Delta^O(I^\alpha)$ which implies that for at least one domain atom $P(\bar{a})$, $P(\bar{a})^I = \mathbf{u}$ while $P(\bar{a})^{\Psi^O_\Delta(I^\alpha)} = \mathbf{t}$ or $P(\bar{a})^{\Psi^O_\Delta(I^\alpha)} = \mathbf{f}$. In the latter case, $\{P(\bar{a})\}$ is an unfounded set. In all cases, I^α has a Δ-refinement.

Proposition 9. *A (terminal) well-founded induction of definition Δ in O is a (terminal) well-founded induction of the approximator Ψ^O_Δ.*

Therefore, the results of the previous section now directly yield following theorem, which gives us a characterization of the well-founded model as the limit of *any* well-founded induction.

Theorem 3. *There exist terminal well-founded inductions of Δ in O. Each well-founded induction of Δ in O is strictly increasing in precision. The limit of every terminal well-founded induction of Δ in O is the well-founded model O^Δ.*

5 Conclusion

Approximation theory is an extension of Tarski's least-fixpoint theorem of monotone lattice operators [21] to the case of arbitrary ones. The claim has been made that this theory is the (missing) fixpoint theory of generalized non-monotone forms of induction such as induction over a well-founded order and iterated induction. In this paper, we gave an argument for this, by investigating a natural class of constructive processes and showing that these are confluent, all having the well-founded model as their limit. This result allowed us to derive a new, simpler and more elegant definition of the well-founded semantics of rule sets, that does not rely on the immediate consequence operator. It would also allow to derive new, simpler constructive characterisations of the well-founded semantics of default logic and auto-epistemic logic. As we have argued in the introduction, this definition also provides a better model of what happens in current implementations of the well-founded semantics.

References

1. P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.
2. Marc Denecker. Extending classical logic with inductive definitions. In *First International Conference on Computational Logic (CL'2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 703–717. Springer, 2000.
3. Marc Denecker, Maurice Bruynooghe, and Victor Marek. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654, October 2001.
4. Marc Denecker, Victor Marek, and Mirosław Truszczyński. Approximating operators, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In J. Minker, editor, *Logic-based Artificial Intelligence*, chapter 6, pages 127–144. Kluwer Academic Publishers, 2000.
5. Marc Denecker, Victor Marek, and Mirosław Truszczyński. Uniform semantic treatment of default and autoepistemic logics. *Artificial Intelligence*, 143(1):79–122, 2003.

6. Marc Denecker, Victor Marek, and Mirosław Truszczyński. Ultimate approximation and its application in nonmonotonic knowledge representation systems. *Information and Computation*, 192(1):84–121, 2004.
7. Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'7)*, 2004.
8. Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions. *Transactions On Computational Logic (TOCL)*, 2007.
9. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer-Verlag, 1999.
10. S. Feferman. Formal theories for transfinite iterations of generalised inductive definitions and some subsystems of analysis. In A. Kino, J. Myhill, and R.E. Vesley, editors, *Intuitionism and Proof theory*, pages 303–326. North Holland, 1970.
11. M. Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
12. M. Fitting. The family of stable models. *Journal of Logic Programming*, 17(2,3&4):197–225, 1993.
13. Zbigniew Lonc and Mirosław Truszczyński. On the problem of computing the well-founded semantics. *Theory and practice of Logic Programming*, 1(5):591–609, 2001.
14. Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR'05*, volume 3835 of *Lecture Notes in Computer Science*, pages 565–579. Springer, 2005.
15. David Mitchell and Eugenia Ternovska. A framework for representing and solving np search problems. In *AAAI'05*, pages 430–435. AAAI Press/MIT Press, 2005.
16. Y. N. Moschovakis. *Elementary Induction on Abstract Structures.* North-Holland Publishing Company, Amsterdam- New York, 1974.
17. Y. N. Moschovakis. On non-monotone inductive definability. *Fundamenta Mathematica*, 82(39-83), 1974.
18. Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: a system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Breckenridge, Colorado, USA, April 2000. cs.AI/0003033.
19. E. Post. Formal reduction of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215, 1943.
20. C. Spector. Inductively defined sets of natural numbers. In *Infinitistic Methods (Proc. 1959 Symposium on Foundation of Mathematis in Warsaw)*, pages 97–102. Pergamon Press, Oxford, 1961.
21. A. Tarski. Lattice-theoretic fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5:285–309, 1955.
22. Bas C. van Fraassen. Singular Terms, truth-Value Gaps, and Free Logic. *The journal of Philosophy*, 63(17):481–495, 1966.
23. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

On the Complexity of Answer Set Programming with Aggregates*

Wolfgang Faber and Nicola Leone

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{faber,leone}@mat.unical.it

Abstract. The addition of aggregates has been one of the most relevant enhancements to the language of answer set programming (ASP). They strengthen the modelling power of ASP in terms of natural and concise problem representations.

In this paper, we carry out an in-depth study of the computational complexity of the language. The analysis pays particular attention to the impact of syntactical restrictions on programs in the form of limited use of aggregates, disjunction, and negation. While the addition of aggregates does not affect the complexity of the full language with negation and disjunction, it turns out that their presence does increase the complexity of non-disjunctive ASP programs up to the second level of the polynomial hierarchy. Interestingly, under cautious reasoning nonmonotone aggregates are even harder than disjunction (Π_2^P-complete vs co-NP-complete on positive programs). However, we show that there are large classes of aggregates the addition of which does not cause any complexity gap even for normal programs, including the fragment allowing for arbitrary monotone, arbitrary antimonotone, and stratified (i.e., non-recursive) nonmonotone aggregates. Moreover, we also prove that for positive programs with arbitrary monotone, stratified antimonotone, and stratified nonmonotone aggregates the complexity remains polynomial. This analysis provides some useful indications on the possibility to implement aggregates in existing reasoning engines.

1 Introduction

Answer Set Programs [1] are logic programs where (nonmonotone) negation may occur in the bodies, and disjunction may occur in the heads of rules. This language is very expressive in a precise mathematical sense: under brave and cautious reasoning, it allows to express every property of finite structures that is decidable in the complexity classes Σ_2^P and Π_2^P, respectively, [2]. The high expressive power of the language, along with its simplicity, and the availability of a number of efficient ASP systems has encouraged the usage of ASP and the investigation of new constructs enhancing its capabilities.

One of the most relevant improvements to the language of answer set programming has been the addition of aggregates [3,4,5,6,7,8,9,10]. Aggregates significantly enhance the language of answer set programming (ASP), allowing for natural and concise modelling of many problems. In this paper, we focus on the semantics for ASP with aggregates as defined in [4], which has been received favorably [10,11,12].

* Supported by M.I.U.R. within projects "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

Several properties of ASP with aggregates under this semantics have been analyzed and are well-understood, and some interesting complexity results have been derived [4,10,13]. However, a systematic complexity study, such as identifying precisely which language elements cause a complexity increase is missing.

In this work, we conduct a comprehensive analysis of the computational complexity of ASP with aggregates and fragments thereof, deriving a full picture of the complexity of the ASP languages where negation and/or disjunction are combined with the different kinds of aggregates (monotone, antimonotone, nonmonotone, stratified).[1] The analysis focuses on cautious reasoning, providing a complete view of the computational complexity of ASP with aggregates:

– The addition of aggregates does not increase the complexity of the full ASP language. Cautious reasoning on full ASP programs (with disjunction and negation) including all considered types of aggregates (monotone, antimonotone, and nonmonotone) even unstratified, remains Π_2^P-complete, as for standard DLP.
– The "cheapest" aggregates, from the complexity viewpoint, are the monotone ones, the addition of which does never cause any complexity increase, even for negation-free programs, and even for unstratified monotone aggregates.
– The "hardest" aggregates, from the complexity viewpoint, are the nonmonotone ones: even on non-disjunctive positive programs (definite horn clauses), their addition causes a big complexity jump from P up to Π_2^P. This means that nonmonotone aggregates are harder than disjunction (cautious reasoning is "only" co-NP on positive disjunctive programs), and provide the modelling power of negation and disjunction combined into one construct.
– Instead, antimonotone aggregates behave similar to negated literals: On non-disjunctive positive programs their presence increases the complexity from P to co-NP.
– The largest set of aggregates which can be added to non-disjunctive ASP without inducing a complexity overhead consists of arbitrary monotone, arbitrary antimonotone, and stratified nonmonotone aggregates. When adding these kinds of aggregates to non-disjunctive ASP, the complexity of reasoning remains in co-NP.

Our work complements and strengthens previous results on the complexity of ASP with aggregates reported in [4,10,13], adding new insight on the complexity impact of aggregates. An interesting novelty concerns, for instance, the complexity of nonmonotone aggregates: in [10,13], the authors show that they can simulate disjunction; here, we prove that they are even harder than disjunction (which cannot, instead, simulate nonmonotone aggregates by a polynomial rewriting).

Importantly, our complexity analysis gives us valuable information about intertranslatability of different languages, having relevant implications also on the possibility to implement aggregates in existing reasoning engines, or using rewriting-based techniques (like those employed in ASSAT [14] or Cmodels [15]) for their implementation (see Section 3).

[1] Note that the results mentioned here refer to the complexity of propositional programs. In Section 3, however, we discuss also the complexity of non-ground programs.

2 The DLP$^{\mathcal{A}}$ Language

In this section, we provide a formal definition of the syntax and semantics of the language DLP$^{\mathcal{A}}$– an extension of Disjunctive Logic Programming (DLP) by set-oriented functions (also called aggregate functions). For further background on DLP, we refer to [16,17].

2.1 Syntax

We assume that the reader is familiar with standard DLP; we refer to the respective constructs as *standard atoms, standard literals, standard rules*, and *standard programs*. A structure (e.g. standard atom, standard literal, conjunction) is ground, if neither the structure itself nor any substructures contain any variables.

Set Terms. A (DLP$^{\mathcal{A}}$) *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\,Vars : Conj\,\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard atoms.[2] A *ground set* is a set of pairs of the form $\langle \overline{t} : Conj\rangle$, where $\overline{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard atoms.

Aggregate Functions. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets[3] of constants to a constant.

Aggregate Literals. An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq, \neq\}$ is a comparison operator, and T is a term (variable or constant).

We note that our choice for the notation of aggregate atoms is primarily motivated by readability. One could define aggregate atoms as an arbitrary relation over a sequence of aggregate functions and terms. In fact, aggregates in DLV and cardinality and weight constraints for Smodels can be of the form $T \prec f(S) \prec U$, but semantically this is a shorthand for the conjunction of $T \prec f(S)$ and $f(S) \prec U$.

An *atom* is either a standard (DLP) atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, L is an *aggregate literal*.

DLP$^{\mathcal{A}}$ *Programs.* A *(*DLP$^{\mathcal{A}}$*) rule* r is a construct

$$a_1 \,\vee\, \cdots \,\vee\, a_n \texttt{ :- } b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_m.$$

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, and $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is referred to as the *head* of r, while the conjunction $b_1, ..., b_k,$ `not` $b_{k+1}, ...,$ `not` b_m is the *body* of r. Let $H(r) = \{a_1, \ldots, a_n\}$, $B^+(r) = \{b_1, \ldots, b_k\}$, $B^-(r) = \{\texttt{not } b_{k+1}, \ldots, \texttt{not } b_m\}$, and $B(r) = B^+(r) \cup B^-(r)$. Furthermore let $Pred(\sigma)$ denote the set of predicates that occur in σ, where σ may be a program, a rule, a set of atoms or literals, an atom or a literal. Whenever it is clear that this set has one element (for standard atoms and literals), $Pred(\sigma)$ may also denote a single predicate. A *(*DLP$^{\mathcal{A}}$*) program* is a set of DLP$^{\mathcal{A}}$ rules.

[2] Intuitively, a symbolic set $\{X : a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X \mid \exists Y s.t.\ a(X,Y), p(Y) \text{ is true}\}$.

[3] Note that aggregate functions are evaluated on the valuation of a (ground) set w.r.t. an interpretation, which is a multiset, cf. Section 2.2.

A *local* variable of r is a variable appearing solely in an aggregate function in r; a variable of r which is not local is called *global*. A *nested* atom of r is an atom appearing in an aggregate atom of r; an atom of r which is not nested is called *unnested*.

Safety. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard unnested literal of the body of r; (ii) each local variable of r that appears in a symbolic set $\{\mathit{Vars} : \mathit{Conj}\}$ also appears in Conj. Finally, a program is safe if all of its rules are safe.

Condition (i) is the standard safety condition adopted in datalog, to guarantee that the variables are range restricted [18], while Condition (ii) is specific for aggregates.

Aggregate-Stratification. A DLP$^{\mathcal{A}}$ program $\mathcal{P}$ is *stratified on an aggregate atom* A if there exists a level mapping $|| \; ||$ from $\mathit{Pred}(\mathcal{P})$ to ordinals, such that for each rule $r \in \mathcal{P}$ and for each $a \in \mathit{Pred}(H(r))$ the following holds: 1. For each $b \in \mathit{Pred}(B(r))$: $||b|| \leq ||a||$, 2. if $A \in B(r)$, then for each $b \in \mathit{Pred}(A)$: $||b|| < ||a||$, and 3. for each $b \in \mathit{Pred}(H(r))$: $||b|| = ||a||$. A DLP$^{\mathcal{A}}$ program $\mathcal{P}$ is *aggregate-stratified* if it is stratified on all aggregate atoms in $\mathcal{P}$.

2.2 Semantics

Universe and Base. Given a DLP$^{\mathcal{A}}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$ the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Given a set X, let $\overline{2}^X$ denote the set of all multisets over elements from X. Without loss of generality, we assume that aggregate functions map to $\mathbb{I}$ (the set of integers).

Instantiation. A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{\mathit{Vars} : \mathit{Conj}\}$, the *instantiation of* S is the following ground set of pairs $inst(S)$: $\{\langle\gamma(\mathit{Vars}) : \gamma(\mathit{Conj})\rangle \mid \gamma$ *is a local substitution for* $S\}$.[4] A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $\mathit{Ground}(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Interpretation. An *interpretation* for a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_{\mathcal{P}}$. A standard ground atom a is true w.r.t. an interpretation I, denoted $I \models a$, if $a \in I$; otherwise it is false w.r.t. I. A standard ground literal $\texttt{not}\ a$ is true w.r.t. an interpretation I, denoted $I \models \texttt{not}\ a$, if $I \not\models a$, otherwise it is false w.r.t. I.

An interpretation also provides a meaning to (ground) sets, aggregate functions and aggregate literals, namely a multiset, a value, and a truth value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset $I(S)$ defined as follows: Let $S_I = \{\langle t_1, ..., t_n\rangle \mid \langle t_1, ..., t_n : \mathit{Conj}\rangle \in S \ \wedge \ \mathit{Conj}$ is true w.r.t. $I\}$, then $I(S)$ is the multiset obtained as the projection of the tuples of S_I on their first constant, that is $I(S) = [t_1 \mid \langle t_1, ..., t_n\rangle \in S_I]$.

[4] Given a substitution σ and a DLP$^{\mathcal{A}}$ object Obj (rule, set, etc.), we denote by $\sigma(\mathit{Obj})$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathcal{P}$).

An instantiated aggregate atom $A = f(S) \prec k$ is true w.r.t. an interpretation I, denoted $I \models A$ if: (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal $\texttt{not}\ A = \texttt{not}\ f(S) \prec k$ is true w.r.t. an interpretation I, denoted $I \models \texttt{not}\ A$, if (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, A is false.

A rule r is *satisfied w.r.t.* I, denoted $I \models r$ if some head atom is true w.r.t. I ($\exists h \in H(r) : I \models h$) whenever all body literals are true w.r.t. I ($\forall b \in B(r) : I \models b$).

Definition 1. *A ground literal ℓ is* monotone*, if for all interpretations I, J, such that $I \subseteq J$, $I \models \ell$ implies $J \models \ell$;* antimonotone*, if for all interpretations I, J, such that $I \subseteq J$, $J \models \ell$ implies $I \models \ell$;* nonmonotone*, if it is neither monotone nor antimonotone.*

Note that positive standard literals are monotone, whereas negative standard literals are antimonotone. Aggregate literals may be monotone, antimonotone or nonmonotone, regardless whether they are positive or negative.

Example 1. Of $\#\texttt{count}\{Z : r(Z)\} > 1$ and $\texttt{not}\ \#\texttt{count}\{Z : r(Z)\} < 1$ all ground instances are monotone, while of $\#\texttt{count}\{Z : r(Z)\} < 1$, $\texttt{not}\ \#\texttt{count}\{Z : r(Z)\} > 1$ all ground instances are antimonotone. Nonmonotone literals include the sum over (possibly negative) integers and the average.

We will next recall the notion of answer sets for DLP$^{\mathcal{A}}$ programs as defined in [4]. An interpretation M is a model of a DLP$^{\mathcal{A}}$ program $\mathcal{P}$, denoted $M \models \mathcal{P}$, if $M \models r$ for all rules $r \in Ground(\mathcal{P})$. An interpretation M is a subset-minimal model of $\mathcal{P}$ if no $I \subset M$ is a model of $Ground(\mathcal{P})$.

Definition 2. *Given a ground* DLP$^{\mathcal{A}}$ *program $\mathcal{P}$ and an interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting rules in which a body literal is false w.r.t. I: $\mathcal{P}^I = \{r \mid r \in \mathcal{P}, \forall b \in B(r) : I \models b\}$*

Definition 3 (Answer Sets for DLP$^{\mathcal{A}}$ **Programs).** *Given a* DLP$^{\mathcal{A}}$ *program $\mathcal{P}$, an interpretation A of $\mathcal{P}$ is an answer set if it is a subset-minimal model of $Ground(\mathcal{P})^A$.*

3 Overview of Complexity Results

We analyze the complexity of DLP$^{\mathcal{A}}$ on cautious reasoning, a main reasoning task in nonmonotone formalisms, amounting to the following decision problem: Given a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ and a standard ground atom A, is A true in all answer sets of $\mathcal{P}$?

For identifying fragments of DLP$^{\mathcal{A}}$, we use the notation LP$^{\mathcal{L}}_{\mathcal{A}}$, where $\mathcal{L} \subseteq \{\texttt{not}, \vee\}$ and $\mathcal{A} \subseteq \{M_s, M, A_s, A, N_s, N\}$. Let $\mathcal{P} \in LP^{\mathcal{L}}_{\mathcal{A}}$. If $\texttt{not} \in \mathcal{L}$, then rules in $\mathcal{P}$ may contain negative literals. Likewise, if $\vee \in \mathcal{L}$, then rules in $\mathcal{P}$ may have disjunctive heads. If $M_s \in \mathcal{A}$ (resp., $A_s \in \mathcal{A}$, $N_s \in \mathcal{A}$), then $\mathcal{P}$ may contain monotone (resp. antimonotone, nonmonotone) aggregates, on which $\mathcal{P}$ is stratified. If $M \in \mathcal{A}$ (resp., $A \in \mathcal{A}$, $N \in \mathcal{A}$), then $\mathcal{P}$ may contain monotone (resp. antimonotone, nonmonotone) aggregates (on which $\mathcal{P}$ is not necessarily stratified). If a symbol is absent in a set, then the respective feature cannot occur in $\mathcal{P}$, unless another symbol is included which specifies a more general feature. For example, if $\mathcal{P} \in LP^{\{\}}_{\{A\}}$, then antimonotone aggregates on which $\mathcal{P}$ is stratified may occur in $\mathcal{P}$ even if A_s is not specified.

For the technical results, we consider ground (i.e., variable-free) DLP$^{\mathcal{A}}$ programs, and polynomial-time computable aggregate functions (note that all sample aggregate functions appearing in this paper fall into this class). However, we also provide a discussion on how results change when considering non-ground programs or aggregates which are harder to compute.

Table 1 summarizes the complexity results derived in the next sections for various fragments LP$^{\mathcal{L}}_{\mathcal{A}}$, where $\mathcal{L}$ is specified in columns and $\mathcal{A}$ in rows. Results for LP$^{\mathcal{L}}_{\mathcal{A}}$, where $M_s \in \mathcal{A}$ have been omitted from Table 1 for readability, as they are equal to those of the respective fragment containing M instead of M_s.

As already noticed in [4], the addition of aggregates does not increase the complexity of disjunctive logic programming, remaining Π^P_2-complete.

Monotone aggregates are the "cheapest," from the viewpoint of complexity. Their presence never causes any complexity increase, even for negation-free programs, and even for unstratified monotone aggregates. The largest polynomial-time computable fragment is $LP^{\{\}}_{\{M,A_s,N_s\}}$ (positive $\vee$-free programs), suggesting that also the stratified aggregates A_s and N_s are rather "cheap". Indeed, they behave similarly to stratified negation from the complexity viewpoint, and increase the complexity only in the case of positive disjunctive programs (from co-NP to Π^P_2).

Antimonotone aggregates act similarly to negation: In the positive $\vee$-free case their presence alone increases the complexity from P to co-NP. The complexity remains the same if monotone and stratified nonmonotone aggregates are added. The maximal co-NP-computable fragments are $LP^{\{\mathtt{not}\}}_{\{M,A,N_s\}}$ and $LP^{\{\vee\}}_{\{M\}}$.

The most "expensive" aggregates, from the viewpoint of complexity, are the nonmonotone ones: In the positive $\vee$-free case (definite Horn programs) they cause a big complexity jump from P to Π^P_2. For each language fragment containing nonmonotone aggregates we obtain Π^P_2-completeness. Intuitively, the reason is that nonmonotone aggregates can express properties which can be written using negation and disjunction in standard DLP.

Note that implemented ASP systems allow for expressing nonmonotone aggregates such as $1 < \#\mathtt{count}\{X : p(X)\} < 3$, which however, can be treated like a conjunction of a monotone and an antimonotone aggregate atom ($\#\mathtt{count}\{X : p(X)\} > 1$, $\#\mathtt{count}\{X : p(X)\} < 3$). The complexity of nondisjunctive programs with these constructs is therefore the same as for $LP^{\{\mathtt{not}\}}_{\{M,A\}}$ (lower than $LP^{\{\mathtt{not}\}}_{\{N\}}$). In [19], a broad class of nonmonotone aggregates, that can be rewritten as monotone and antimonotone aggregates in this style, is identified.

The above complexity results give us valuable information about intertranslatability of different languages, having important implications on the possibility to implement aggregates in existing reasoning engines. For instance, we know now that cautious reasoning on $LP^{\{\mathtt{not}\}}_{\{M,A,N_s\}}$ can be efficiently translated to UNSAT (the complement of propositional satisfiability) or to cautious reasoning on non-disjunctive ASP; thus, arbitrary monotone, arbitrary antimonotone, and stratified nonmonotone aggregates can

Table 1. The Complexity of Cautious Reasoning in ASP with Aggregates (Completeness Results under Logspace Reductions)

	$\{\}$	$\{\texttt{not}\}$	$\{\vee\}$	$\{\texttt{not}, \vee\}$	
$\{\}$	P	co-NP	co-NP	Π_2^P	1
$\{\mathbf{M}\}$	P	co-NP	co-NP	Π_2^P	2
$\{\mathbf{A_s}\}$	P	co-NP	Π_2^P	Π_2^P	3
$\{\mathbf{N_s}\}$	P	co-NP	Π_2^P	Π_2^P	4
$\{\mathbf{M}, \mathbf{A_s}\}$	P	co-NP	Π_2^P	Π_2^P	5
$\{\mathbf{M}, \mathbf{N_s}\}$	P	co-NP	Π_2^P	Π_2^P	6
$\{\mathbf{A_s}, \mathbf{N_s}\}$	P	co-NP	Π_2^P	Π_2^P	7
$\{\mathbf{M}, \mathbf{A_s}, \mathbf{N_s}\}$	P	co-NP	Π_2^P	Π_2^P	8
$\{\mathbf{A}\}$	co-NP	co-NP	Π_2^P	Π_2^P	9
$\{\mathbf{M}, \mathbf{A}\}$	co-NP	co-NP	Π_2^P	Π_2^P	10
$\{\mathbf{A}, \mathbf{N_s}\}$	co-NP	co-NP	Π_2^P	Π_2^P	11
$\{\mathbf{M}, \mathbf{A}, \mathbf{N_s}\}$	co-NP	co-NP	Π_2^P	Π_2^P	12
$\{\mathbf{N}\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P	13
$\{\mathbf{M}, \mathbf{N}\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P	14
$\{\mathbf{A_s}, \mathbf{N}\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P	15
$\{\mathbf{M}, \mathbf{A_s}, \mathbf{N}\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P	16
$\{\mathbf{A}, \mathbf{N}\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P	17
$\{\mathbf{M}, \mathbf{A}, \mathbf{N}\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P	18
	1	2	3	4	

be implemented efficiently on top of SAT solvers or non-disjunctive ASP systems. On the other hand, since nonmonotone aggregates (even without negation and disjunction) bring the complexity to Π_2^P, the existence of a polynomial translation from cautious reasoning with nonmonotone aggregates to UNSAT cannot exist (unless the polynomial hierarchy collapses).

As mentioned above, our results rely on the assumption that aggregate functions are computable in polynomial time. If one were to allow computationally more expensive aggregates, complexity would rise according to the complexity of additional oracles, which are needed to compute the truth value of an aggregate.

We end this overview by briefly addressing the complexity of non-ground programs. When considering data-complexity (i.e. a program $\mathcal{P}$ is fixed, while the input consists only of facts), the results are as for propositional programs. If, however, one considers program complexity (i.e. a program $\mathcal{P}$ is given as input), complexity rises in a similar manner as for aggregate-free programs. A non-ground program $\mathcal{P}$ can be reduced, by naive instantiation, to a ground instance of the problem. In the general case, where $\mathcal{P}$ is given in the input, the size of the grounding $Ground(\mathcal{P})$ is single exponential in the size of $\mathcal{P}$. Informally, the complexity of Cautious Reasoning increases accordingly by one exponential, from P to EXPTIME, co-NP to co-NEXPTIME, Π_2^P to co-NEXPTIME$^{\mathrm{NP}}$. For aggregate-free programs complexity results in the non-ground case are reported in [20]. For the other fragments, the results can be derived using complexity upgrading techniques as presented in [21].

4 Proofs of Hardness Results

All P-hardness results in the table (rows 1–8 in column 1) follow directly from the well-known result that (positive) propositional logic programming is P-hard [20].

We recall that in Theorems 4 and 5 of [4] it has been shown that each program $\mathcal{P} \in LP_{\{\}}^{\{\mathtt{not},\vee\}}$ can be transformed into an equivalent program $\Gamma(\mathcal{P}) \in LP_{\{A\}}^{\{\vee\}}$ in LOGSPACE, and if $\mathcal{P}$ is negation-stratified, then $\Gamma(\mathcal{P}) \in LP_{\{A_s\}}^{\{\vee\}}$. As a consequence of these results, we obtain hardness for positive non-disjunctive programs containing antimonotone aggregates.

Theorem 1. *Cautious reasoning over* $\mathrm{LP}_{\{A\}}^{\{\}}$ *programs is co-NP-hard.*

Proof. Follows from co-NP-hardness of cautious reasoning for positive disjunctive aggregate-free programs (programs in $LP_{\{\}}^{\{\vee\}}$), see Theorem 6.1 in [20], together with Theorems 4 and 5 of [4].

Whenever one allows for nonmonotone aggregates in positive, non-disjunctive programs, cautious reasoning becomes harder by one level in the polynomial hierarchy.

Theorem 2. *Cautious reasoning over* $\mathrm{LP}_{\{N\}}^{\{\}}$ *programs is* Π_2^P*-hard.*

Proof. We provide a reduction from 2QBF. Let $\Psi = \forall x_1, \ldots, x_m \exists y_1, \ldots, y_n : \Phi$, where w.l.o.g. Φ is a propositional formula in 3DNF format, over precisely the variables $x_1, \ldots, x_m, y_1, \ldots, y_n$. Let the $\mathrm{LP}_{\{N\}}^{\{\}}$ program Π^Ψ be:

$$
\begin{array}{l}
t(x_i, 1) \text{ :- } \#\mathtt{sum}\{\langle 1 : t(x_i, 1)\rangle, \langle -1 : t(x_i, -1)\rangle\} \geq 0.\\
t(x_i, -1) \text{ :- } \#\mathtt{sum}\{\langle 1 : t(x_i, 1)\rangle, \langle -1 : t(x_i, -1)\rangle\} \leq 0.\\
t(y_i, 1) \text{ :- } \#\mathtt{sum}\{\langle 1 : t(y_i, 1)\rangle, \langle -1 : t(y_i, -1)\rangle\} \geq 0.\\
t(y_i, -1) \text{ :- } \#\mathtt{sum}\{\langle 1 : t(y_i, 1)\rangle, \langle -1 : t(y_i, -1)\rangle\} \leq 0.\\
t(y_i, 1) \text{ :- } sat(-1). \quad t(y_i, -1) \text{ :- } sat(-1).\\
sat(-1) \text{ :- } \#\mathtt{sum}\{\langle -1 : sat(-1)\rangle, \langle -1 : unsat(-1)\rangle\} \leq 0.
\end{array}
$$

For each disjunct $c_i = l_{i,1} \wedge l_{i,2} \wedge l_{i,3}$ of Φ, we add: $sat(1) \text{ :- } \mu(l_{i,1}), \mu(l_{i,2}), \mu(l_{i,3})$, where $\mu(l)$ is $t(l, -1)$ if l is positive, and $t(l, 1)$ otherwise. The query $sat(-1)$? holds for Π^Ψ, iff Ψ is true.

Intuitively, $t(v, 1)$ and $t(v, -1)$ encode truth and falsity of the propositional variable v, respectively, while $sat(-1)$ encodes unsatisfiability of $\neg\Phi$. The first four rules encode all possible truth assignments for $x_1, \ldots, x_m, y_1, \ldots, y_n$. If $sat(-1)$ holds, the subsequent two rules entail that both truth and falsity of $y_1, \ldots, y_n$ must hold (the first four rules are then still satisfied). If an assignment for $x_1, \ldots, x_m, y_1, \ldots, y_n$ does not satisfy $\neg\Phi$, then $sat(-1)$ must hold in any answer set encoding the assignment. But then both truth and falsity will be derived for any y_i (saturation). Now, by minimality of answer sets, if for some assignment for $x_1, \ldots, x_m, y_1, \ldots, y_n$, $sat(-1)$ is not derived (hence the assignment does satisfy $\neg\Phi$), no answer set including $sat(-1)$ and the same assignment for $x_1, \ldots, x_m$ can exist, as it would be a superset. Moreover, whenever $sat(-1)$ is not derived for some $x_1, \ldots, x_m, y_1, \ldots, y_n$, the resulting model is not an answer set. In total, if $\neg\Psi \equiv \exists x_1, \ldots, x_m \forall y_1, \ldots, y_n : \neg\Phi$ is false (if and only if Ψ is true), then at least one answer set of Π^{Ψ} exists, and all answer sets of Π^{Ψ} contain $sat(-1)$. Otherwise, Π^{Ψ} has no answer sets.

We note that a related result — deciding whether an answer set exists for a positive, non-disjunctive program with weight constraints over possibly negative integers is Σ_2^P-complete — has been shown in [10]. Weight constraints can be monotone, antimonotone, or nonmonotone aggregate atoms. In particular, Ferraris elegantly shows that disjunctive programs can be efficiently rewritten to $\vee$-free programs with nonmonotonic aggregates (under brave reasoning). We strengthen Ferraris' result, showing that, under cautious reasoning and positive programs, the opposite direction does not hold: positive $\vee$-free programs with nonmonotonic aggregates cannot be efficiently translated into positive disjunctive programs (unless $\mathrm{P} = \mathrm{NP}$). Theorem 2 strengthens also Theorem 21 in [13], where Π_2^P-hardness is stated for $\mathrm{LP}^{\{\mathtt{not}\}}_{\{M,A,N\}}$ (and the reduction uses negation).

The following result has implicitly been given in Theorem 7 in [4].

Theorem 3 **([4]).** *Cautious reasoning over* $\mathrm{LP}^{\{\vee\}}_{\{A_s\}}$ *and* $\mathrm{LP}^{\{\vee\}}_{\{N_s\}}$ *programs is* Π_2^P*-hard.*

Leveraging results in the literature, we have shown hardness for all fields in Table 1.

Theorem 4. *Each field of Table 1 states the respective hardness of cautious reasoning for the corresponding fragment of* $\mathrm{DLP}^{\mathcal{A}}$.

5 Proofs of Membership Results

For the membership proofs, we will advance in the reverse order, and first prove results for richer languages, which cover also several results for sublanguages.

For the complete language, we recall Theorem 9 of [4]:

Theorem 5 **([4]).** *Cautious reasoning over* $\mathrm{LP}^{\{\mathtt{not},\vee\}}_{\{M,A,N\}}$ *programs is in* Π_2^P.

Concerning disjunctive programs, for most fragments cautious reasoning is in Π_2^P, with two exceptions which are in co-NP. The reason is that for the respective classes it is sufficient to look at an arbitrary model, rather than an answer set or a minimal model, as shown in Theorem 8 of [4]:

Theorem 6 **([4]).** *Cautious reasoning over* $\mathrm{LP}^{\{\vee\}}_{\{M\}}$ *programs is in co-NP.*

Π_2^P-memberships for non-disjunctive programs already follow from the respective result for disjunctive programs, and it remains to show co-NP- and P-memberships. Let us now consider the less complex language $\mathrm{LP}^{\{\}}_{\{M,A_s,N_s\}}$.

Lemma 1. *An* $\mathrm{LP}^{\{\}}_{\{M,A_s,N_s\}}$ *program has at most one answer set and the answer sets of a* $\mathrm{LP}^{\{\}}_{\{M,A_s,N_s\}}$ *program can be computed in polynomial time.*

Proof. For a $\mathrm{LP}^{\{\}}_{\{M,A_s,N_s\}}$ program $\mathcal{P}$, let us define an operator $\mathbb{T}_{\mathcal{P}}$ on interpretations of $\mathcal{P}$ as follows: $\mathbb{T}_{\mathcal{P}}(I) = \{h \mid r \in \mathcal{P}, I \models B(r), h \in H(r)\}$. Furthermore, given an interpretation I, let the sequence $\{\mathbb{T}^n_{\mathcal{P}}(I)\}_{n\in\mathcal{N}}$ be defined as $\mathbb{T}^0_{\mathcal{P}}(I) = I$ and $\mathbb{T}^i_{\mathcal{P}} = \mathbb{T}_{\mathcal{P}}(\mathbb{T}^{i-1}_{\mathcal{P}}(I))$ for $i > 0$. Since $\mathbb{T}_{\mathcal{P}}$ is monotone and the number of interpretations for $\mathcal{P}$ is finite, the sequence reaches a fixpoint $\mathbb{T}^{\infty}_{\mathcal{P}}(I)$.

Consider a level mapping $||\ ||$ such that for each rule $r \in \mathcal{P}$, for which $H(r) = \{h\}$ and an antimonotone or nonmonotone aggregate literal $A \in B(r)$, it holds for each predicate p nested in A that $||p|| < ||p'||$, where p' is the predicate of h. Moreover, $||p|| \le ||p'||$ holds for any p and p' such that p' occurs in the head and p in the body of a rule. W.l.o.g., we assume the co-domain of $||||$ to be $0,\ldots,n$. Based on $||\ ||$, we define a partition $\mathcal{P}_0,\ldots,\mathcal{P}_n,\mathcal{P}_{constr}$ of $\mathcal{P}$ as follows: $\mathcal{P}_i = \{r \mid r \in \mathcal{P}, H(r) = \{h\}, ||Pred(h)|| = i\}$, $\mathcal{P}_{constr} = \{r \mid r \in \mathcal{P}, H(r) = \emptyset\}$. Furthermore, we define $FP^0_{\mathcal{P}} = \mathbb{T}^{\infty}_{\mathcal{P}_0}(\emptyset)$ and $FP^i_{\mathcal{P}} = \mathbb{T}^{\infty}_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}})$ for $0 < i \le n$, and let $FP_{\mathcal{P}} = FP^n_{\mathcal{P}}$. If $FP_{\mathcal{P}}$ is a model of $\mathcal{P}_{constr}$, let $FM_{\mathcal{P}} = \{FP_{\mathcal{P}}\}$, otherwise $FM_{\mathcal{P}} = \emptyset$. In the sequel $H(\mathcal{P}) = \{h \mid \exists r \in \mathcal{P} : h \in H(r)\}$ will denote the set of head atoms of a program. We now show by induction that $FP_{\mathcal{P}} = A$ for each answer set A of $\mathcal{P}$. The base is $FP^0_{\mathcal{P}} \cap H(\mathcal{P}_0) = A \cap H(\mathcal{P}_0)$ for each answer set A of $\mathcal{P}$.

To prove $FP^0_{\mathcal{P}} \cap H(\mathcal{P}_0) \subseteq A \cap H(\mathcal{P}_0)$, we use another induction over $\mathbb{T}^i_{\mathcal{P}_0}(\emptyset)$. The base here is $\mathbb{T}^0_{\mathcal{P}_i}(\emptyset) = \emptyset \subseteq A$ for each answer set A of $\mathcal{P}$. Then, assuming that $S \subseteq A$ for each answer set A of $\mathcal{P}$, we can show that $\mathbb{T}_{\mathcal{P}_0}(S) \subseteq A$ for each answer set A of $\mathcal{P}$: Each rule $r \in \mathcal{P}_0$ is also in $\mathcal{P}$ and since A is a model of $\mathcal{P}$, whenever $S \models b$ for all $b \in B(r)$, then also for any answer set A, $A \models b$, as $B(r)$ may not contain antimonotone or nonmonotone aggregate literals, otherwise $||p|| < 0$ for some predicate in such an aggregate would hold. Since $H(r) = \{h\}$, h must be contained in each answer set. It follows that $FP^0_{\mathcal{P}} = \mathbb{T}^{\infty}_{\mathcal{P}_0} \subseteq A$. It is easy to see that $FP^0_{\mathcal{P}} \subseteq H(\mathcal{P}_0)$, so $FP^0_{\mathcal{P}} \cap H(\mathcal{P}_0) \subseteq A \cap H(\mathcal{P}_0)$.

Now assume that $X = (A \cap H(\mathcal{P}_0)) \setminus (FP^0_{\mathcal{P}} \cap H(\mathcal{P}_0)) \neq \emptyset$. We show that then $A \setminus X$ is a model of $\mathcal{P}^A$, contradicting the assumption that A is an answer set. Each rule in $\mathcal{P}^A \cap \mathcal{P}_0$ is clearly satisfied by $A \setminus X$, because it is satisfied by $FP^0_{\mathcal{P}}$. Now recall that each rule r in $\mathcal{P}^A \setminus \mathcal{P}_0$ has a true body w.r.t. A, which is either true or false w.r.t. $A \setminus X$. Since $H(r) \cap X = \emptyset$, r is also satisfied by $A \setminus X$. Therefore A is not an answer set of $\mathcal{P}$ if $X \neq \emptyset$, and so $FP^0_{\mathcal{P}} \cap H(\mathcal{P}_0) \supseteq A \cap H(\mathcal{P}_0)$. We have shown the base of the induction, $FP^0_{\mathcal{P}} \cap H(\mathcal{P}_0) = A \cap H(\mathcal{P}_0)$.

For the inductive step, we assume $FP^k_{\mathcal{P}} \cap H(\mathcal{P}_k) = A \cap H(\mathcal{P}_k)$ holds for all $k < i, i > 0$ and each answer set A. In order to show $FP^i_{\mathcal{P}} \cap H(\mathcal{P}_i) = A \cap H(\mathcal{P}_i)$, we use yet another induction over $\mathbb{T}^j_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}})$. The base is $\mathbb{T}^0_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}}) = FP^{i-1}_{\mathcal{P}} \subseteq A$ for each answer set A, which holds by the inductive hypothesis of the "larger" induction. Now, we assume that $\mathbb{T}^j_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}}) \subseteq A$ holds for each answer set, and show that $\mathbb{T}_{p_i}(\mathbb{T}^j_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}})) \subseteq A$ holds for each answer set. We observe that each rule $r \in \mathcal{P}_i$ is also in $\mathcal{P}$ and since A is a model of $\mathcal{P}$, whenever $\mathbb{T}^j_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}}) \models b$ for all $b \in B(r)$, then also for any answer set A, $A \models b$, because the only antimonotone or nonmonotone literals are aggregates which, however, contain only atoms formed by predicates p, for which $||p|| < i$. Any of these atoms are however in $H(\mathcal{P}_k)$ for $k < i$ and so by the inductive hypothesis (of the "larger" induction), $\mathbb{T}^j_{\mathcal{P}_i}(FP^{i-1}_{\mathcal{P}}) \cap H(\mathcal{P}_k) = A \cap H(\mathcal{P}_k)$. In total, we get $FP^i_{\mathcal{P}} = \mathbb{T}^{\infty}_{\mathcal{P}_i} \subseteq A$.

It remains to show that $FP^i_{\mathcal{P}} \cap H(\mathcal{P}_i) \supseteq A \cap H(\mathcal{P}_i)$. Similar to the base case of the "larger" induction, we assume $X = (A \cap H(\mathcal{P}_i)) \setminus (FP^i_{\mathcal{P}} \cap H(\mathcal{P}_i)) \neq \emptyset$. We show that then $A \setminus X$ is a model of $\mathcal{P}^A$, contradicting the assumption that A is an answer set. Each rule in $\mathcal{P}^A \cap \mathcal{P}_i$ is clearly satisfied by $A \setminus X$, because it is satisfied by $FP^i_{\mathcal{P}}$. Now recall that each rule r in $\mathcal{P}^A \setminus \mathcal{P}_i$ has a true body w.r.t. A, which is either true or false w.r.t. $A \setminus X$. Since $H(r) \cap X = \emptyset$, r is also satisfied by $A \setminus X$. Therefore A is not an answer set of $\mathcal{P}$ if $X \neq \emptyset$, and so $FP^i_{\mathcal{P}} \cap H(\mathcal{P}_i) \supseteq A \cap H(\mathcal{P}_i)$. We have shown the step of the induction, $FP^i_{\mathcal{P}} \cap H(\mathcal{P}_i) = A \cap H(\mathcal{P}_i)$ for each answer set A.

In total, for $FP_{\mathcal{P}}$ we have $FP_{\mathcal{P}} \cap (\bigcup_{i=1}^n H(\mathcal{P}_i)) = A \cap (\bigcup_{i=1}^n H(\mathcal{P}_i))$ for each answer set A of $\mathcal{P}$. Clearly each answer set of $\mathcal{P}$ is also an answer set of $(\bigcup_{i=1}^n H(\mathcal{P}_i)) = \mathcal{P} \setminus \mathcal{P}_{constr}$. Therefore, for each answer set A of $\mathcal{P}$, we know that $A = FP_{\mathcal{P}}$. It follows that $\mathcal{P}$ has at most one answer set. Moreover, since $FP_{\mathcal{P}}$ is an answer set of $\mathcal{P} \setminus \mathcal{P}_{constr}$, it is a minimal model of $(\mathcal{P} \setminus \mathcal{P}_{constr})^{FP_{\mathcal{P}}}$. If $FP_{\mathcal{P}}$ satisfies all rules in $\mathcal{P}_{constr}$, then $(\mathcal{P} \setminus \mathcal{P}_{constr})^{FP_{\mathcal{P}}} = \mathcal{P}^{FP_{\mathcal{P}}}$, and $FP_{\mathcal{P}}$ is an answer set of $\mathcal{P}$. If any rule of $\mathcal{P}_{constr}$ exists which is not satisfied by $FP_{\mathcal{P}}$, it cannot be an answer set of $\mathcal{P}$, so $FM_{\mathcal{P}}$ is the set of answer sets for $\mathcal{P}$. Computing $FP_{\mathcal{P}}$ and $FM_{\mathcal{P}}$ using $\mathbb{T}_{\mathcal{P}}$ is clearly polynomial.

Since there is at most one answer set, cautious reasoning is easy.

Theorem 7. *Cautious reasoning over* $\mathrm{LP}^{\{\}}_{\{M,A_s,N_s\}}$ *is in* P.

Proof. This is a simple consequence of Lemma 1. We compute the set of answer sets in polynomial time. If it is empty, all atoms are a cautious consequence. If there is one answer set, check in polynomial time whether it contains the query atom.

Let us now focus on the co-NP-memberships. For doing so, we will re-use the fact that answer sets of $\mathrm{LP}^{\{\}}_{\{M,A_s,N_s\}}$ programs are computable in polynomial time, as for answer set checking of $\mathrm{LP}^{\{\mathtt{not}\}}_{\{M,A,N_s\}}$ programs, one can eliminate antimonotone literals.

Lemma 2. *Given a* $\mathrm{LP}^{\{\mathtt{not}\}}_{\{M,A,N_s\}}$ *program* $\mathcal{P}$ *and an interpretation* $I \subseteq B_{\mathcal{P}}$*,* I *is a subset-minimal model of* $\mathcal{P}^I$ *iff it is a subset-minimal model of* $\Psi(\mathcal{P}^I)$*, which is derived from* $\mathcal{P}^I$ *by deleting all antimonotone literals.*

Proof. $(\Rightarrow)$ If I is a minimal model of $\mathcal{P}^I$, it is obviously also a model of $\Psi(\mathcal{P}^I)$. Moreover, each interpretation $N \subset I$ is not a model of $\mathcal{P}^I$, so there is at least one rule $r \in \mathcal{P}^I$, for which $N \not\models r$, that is all body atoms are true w.r.t. N but all head atoms are false w.r.t. N. Now there is a rule $r' \in \Psi(\mathcal{P}^I)$ with $H(r) = H(r')$ and $B(r) \supseteq B(r')$. So also the body of r' is true w.r.t. N, and hence r' is not satisfied by N. As a consequence, N is not a model of $\Psi(\mathcal{P}^I)$, and so I is a minimal model of $\Psi(\mathcal{P}^I)$.

$(\Leftarrow)$ Let I be a minimal model of $\Psi(\mathcal{P}^I)$. We first note that no rule in $\mathcal{P}^I$ (and $\Psi(\mathcal{P}^I)$) has a body literal which is false w.r.t. I by construction of $\mathcal{P}^I$. So for any rule in $\Psi(\mathcal{P}^I)$, all body literals are true w.r.t. I, and hence one of its head atoms is true w.r.t. I, since I is a model. Since each rule in $\Psi(\mathcal{P}^I)$ has a corresponding rule in $\mathcal{P}^I$ with equal head, I is also a model of $\mathcal{P}^I$.

Now, consider an arbitrary interpretation $N \subset I$. N is not a model of $\Psi(\mathcal{P}^I)$, that is, there is a rule $r \in \Psi(\mathcal{P}^I)$ with true body and false head w.r.t. N. By construction, there is a rule $r' \in p^I$, for which $H(r) = H(r')$ and $B(r) \subseteq B(r')$. By construction of p^I, all literals of r' are true w.r.t. I, and since each deleted body literal $\ell \in B(r') \setminus B(r)$ is an antimonotone literal, ℓ is also true w.r.t. N. Hence, the body of r' is true and the head of r' is false w.r.t. N. Hence N is not a model of $\mathcal{P}^I$, and we obtain that I is a minimal model of $\mathcal{P}^I$.

Theorem 8. *Cautious reasoning over* $\mathrm{LP}^{\{\mathtt{not}\}}_{\{M,A,N_s\}}$ *is in co-NP.*

Proof. We guess an interpretation I, and check whether it is an answer set and does not contain the queried atom. The latter check is clearly polynomial. Answer set checking amounts to checking whether I is a subset-minimal model of $\mathcal{P}^I$. Because of Lemma 2, I is a subset-minimal model of $\mathcal{P}^I$ iff I is a subset-minimal model of $\Psi(\mathcal{P}^I)$, in which all negative standard and antimonotone aggregate literals have been deleted (this transformation is obviously polynomial). Note that I is a subset-minimal model of $\mathcal{P}^I$ if I is an answer set of $\mathcal{P}^I$, hence if I is an answer set of $\Psi(\mathcal{P}^I)$. Now since $\Psi(\mathcal{P}^I) \in LP^{\{\}}_{\{M,N_s\}} \subseteq LP^{\{\}}_{\{M,A_s,N_s\}}$ we know by Lemma 1 that its answer sets (at most one) are computable in polynomial time, and hence also the set of minimal models of $\Psi(\mathcal{P}^I)$. If it is empty, I is not an answer set; otherwise there is exactly one minimal model, and we check whether it is equal to I. If it is, I is an answer set, otherwise it is not. Checking whether I is an answer set is therefore feasible in polynomial time.

All membership results of Table 1 follow from these results.

Theorem 9. *Each field of Table 1 states the respective membership of cautious reasoning for the corresponding fragment of* $\mathrm{DLP}^{\mathcal{A}}$.

6 Conclusions

Concluding, we have given a fine-grained analysis of the computational complexity of ASP with aggregates, drawing a full picture of the complexity of the ASP fragments where negation and/or disjunction are combined with different kinds of aggregates (monotone, antimonotone, nonmonotone, stratified or not). Importantly, we have shown that the presence of aggregate literals (of arbitrary kind) will not increase the computational complexity of full ASP programs (with disjunction and negation). However, the presence of unstratified nonmonotone aggregates does increase the complexity of normal, non-disjunctive programs up to Π^P_2.

Additionally we have singled out relevant classes of aggregates which do not cause any complexity overhead for normal programs and can be efficiently implemented in normal ASP systems. Likewise, we have identified classes of aggregates which do not increase the complexity even for positive non-disjunctive programs, for which the problem stays polynomial. The latter result is particularly interesting in a database setting.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
3. Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic. Logic Programming and Beyond. LNCS 2408 (2002) 413–451
4. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. LNCS 3229
5. Pelov, N., Denecker, M., Bruynooghe, M.: Partial stable models for logic programs with aggregates. In: LPNMR-7. LNCS 2923
6. Pelov, N., Truszczyński, M.: Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In: NMR 2004. (2004) 327–334
7. Marek, V.W., Niemelä, I., Truszczyński, M.: Logic Programming with Monotone Cardinality Atom. In: LPNMR-7. LNCS 2923
8. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. Theory and Practice of Logic Programming (2007) Accepted for publication, available in CoRR as cs.LO/0509024.
9. Tran Cao Son, Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. Theory and Practice of Logic Programming (2007) Accepted for publication, available in CoRR as cs.AI/0601051.
10. Ferraris, P.: Answer Sets for Propositional Theories. In: LPNMR'05. LNCS 3662
11. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: IJCAI 2005, Edinburgh, UK (2005) 90–96
12. Eiter, T., Ianni, G., Tompits, H., Schindlauer, R.: Effective Integration of Declarative Rules with External Evaluations for Semantic Web Reasoning. In: Proceedings of the 3rd European Semantic Web Conference (ESWC 2006). (2006) 273–287
13. Faber, W., Leone, N., Ricca, F.: Heuristics for Hard ASP Programs. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 1562–1563
14. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)
15. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS 2923
16. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
17. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
18. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
19. Faber, W.: Decomposition of Nonmonotone Aggregates in Logic Programming. In: Proceedings of the 20th Workshop on Logic Programming (WLP 2006), Vienna, Austria (2006) 164–171
20. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
21. Gottlob, G., Leone, N., Veith, H.: Succinctness as a Source of Expression Complexity. Annals of Pure and Applied Logic **97**(1–3) (1999) 231–260

Experimenting with Look-Back Heuristics for Hard ASP Programs*

Wolfgang Faber, Nicola Leone, Marco Maratea, and Francesco Ricca

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{faber,leone,maratea,ricca}@mat.unical.it

Abstract. Competitive native solvers for Answer Set Programming (ASP) perform a backtracking search by assuming the truth of literals. The choice of literals (the heuristic) is fundamental for the performance of these systems.

Most of the efficient ASP systems employ a heuristic based on look-ahead, that is, a literal is tentatively assumed and its heuristic value is based on its deterministic consequences. However, looking ahead is a costly operation, and indeed look-ahead often accounts for the majority of time taken by ASP solvers. For Satisfiability (SAT), a radically different approach, called look-back heuristic, proved to be quite successful: Instead of looking ahead, one uses information gathered during the computation performed so far, thus looking back. In this approach, atoms which have been frequently involved in inconsistencies are preferred.

In this paper, we carry over this approach to the framework of *disjunctive* ASP. We design a number of look-back heuristics exploiting peculiarities of ASP and implement them in the ASP system DLV. We compare their performance on a collection of hard ASP programs both structured and randomly generated. These experiments indicate that a very basic approach works well, outperforming all of the prominent disjunctive ASP systems — DLV (with its traditional heuristic), GnT, and CModels3 — on many of the instances considered.

1 Introduction

Answer set programming (ASP) is a comparatively novel programming paradigm, which has been proposed in the area of nonmonotonic reasoning and logic programming. The idea of answer set programming is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver to find such solutions [1]. The knowledge representation language of ASP is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, ASP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [2]. Thus, ASP is strictly more powerful than SAT-based programming, as it allows for solving problems which cannot be translated to SAT in polynomial time (unless $P = NP$). For instance, several problems in diagnosis and planning under incomplete knowledge are complete for the complexity class Σ_2^P or Π_2^P [3,4], and can be naturally encoded in ASP [5,6].

* Supported by M.I.U.R. within projects "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

Since the model generators of native ASP systems are similar to the DPLL procedure, employed in many SAT solvers, the heuristic (branching rule) for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) is fundamentally important for the efficiency of an ASP system. Many of the efficient ASP systems, and especially the disjunctive ASP systems, employ a heuristic based on look-ahead. This means that the available choices are hypothetically assumed, their deterministically entailed consequences are computed, and a heuristic function is evaluated on the result. The look-ahead approach has been shown to be effective [7,8] and it bears the additional benefit of detecting choices that deterministically cause an inconsistency. However, the sheer number of potential choices and the costly computations done for each of these makes the look-ahead a rather costly operation. Indeed, look-ahead often accounts for the majority of time taken by ASP solvers.

In SAT, a radically different approach, called look-back heuristics, proved to be quite successful [9]: Instead of making tentative assumptions and thus trying to look into the future of the computation, one uses information already collected during the computation so far, thus looking back; atoms which have been most frequently involved in inconsistencies are heuristically preferred (following the intuition that "most constrained" atoms are to be decided first).

In this paper, we take this approach from SAT to the framework of disjunctive ASP, trying to maximally exploit peculiarities of ASP, and experiment with alternative ways of addressing the key issues arising in this framework. The main contributions of the paper are as follows.

- We define a framework for look-back heuristics in disjunctive ASP. We build upon the work in [10], which describes a calculus identifying reasons for encountered inconsistencies in order to allow backjumping (i.e., avoiding backtracking to choices which do not contribute to an encountered inconsistency). For obtaining a "most constrained choices first" strategy, we prefer those choices that were the reasons for earlier inconsistencies. Our framework exploits the peculiarities of disjunctive ASP, a relevant feature concerns the full exploitation of "hidden" inconsistencies which are due to the failure of stable-model checks.
- We design a number of look-back heuristics for disjunctive ASP. In particular, we study different ways of making choices when information on inconsistencies is poor (e.g., at the beginning of the computation, when there is still nothing to look back to).
  We consider also different ways of choosing the "polarity" (positive or negative) of the atoms to be taken (intuitively negative choices keep the interpretation closer to minimality, which is mandatory in ASP).
- We implement all proposed heuristics in the ASP system DLV [11].
- We carry out an experimental evaluation of all proposed heuristics on programs encoding random and structured 2QBF formulas, the prototypical problem for Π_2^P (the class characterizing hard disjunctive ASP programs).

The results are very encouraging, the new heuristics perform very well compared to the traditional disjunctive ASP systems DLV, GnT [12] and CModels3 [13]. In particular, a very basic heuristic outperforms all other systems on a large part of the considered instances.

To our knowledge, while look-back heuristics have been widely studied for SAT (see, e.g., [9] [14], [15]), so far only few works have studied look-back heuristics for $\vee$-free ASP [16,17], and this is the first paper on look-back heuristics for disjunctive ASP.[1]

2 Answer Set Programming Language

A *(disjunctive) rule* r is a formula

$$a_1 \ \vee \ \cdots \ \vee \ a_n \text{ :- } b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \ \texttt{not} \ b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are function-free atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while $b_1, \cdots, b_k, \texttt{not} \ b_{k+1}, \cdots, \texttt{not} \ b_m$ is the *body*, of which $b_1, \cdots, b_k$ is the *positive body*, and $\texttt{not} \ b_{k+1}, \cdots, \texttt{not} \ b_m$ is the *negative body* of r.

An *(ASP) program* $\mathcal{P}$ is a finite set of rules. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_\mathcal{P}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_\mathcal{P}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_\mathcal{P}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $Ground(\mathcal{P})$ of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, its answer sets are defined using its ground instantiation $Ground(\mathcal{P})$ in two steps: First answer sets of positive programs are defined, then a reduction of general programs to positive ones is given, which is used to define answer sets of general programs.

A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\texttt{not} \ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_\mathcal{P}$.[2] A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. Interpretation I is *total* if, for each atom A in $B_\mathcal{P}$, either A or $\texttt{not} \ A$ is in I (i.e., no atom in $B_\mathcal{P}$ is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is an *answer set* for a positive program $\mathcal{P}$ if it is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

Example 1. For the positive program $\mathcal{P}_1 = \{a \vee b \vee c. , \ \text{:-} a.\}$, $\{b, \texttt{not} \ a, \texttt{not} \ c\}$ and $\{c, \texttt{not} \ a, \texttt{not} \ b\}$ are the only answer sets.

For the positive program $\mathcal{P}_2 = \{a \vee b \vee c. , \ \text{:-} a. , \ b \text{:-} c. , \ c \text{:-} b.\}$, $\{b, c, \texttt{not} \ a\}$ is the only answer set. ∎

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting all

[1] The disjunctive ASP system CModels3 [13] "indirectly" uses look-back heuristics, since it works on top SAT solvers which may employ this technique.

[2] We represent interpretations as set of literals, since we have to deal with partial interpretations in the next sections.

rules $r \in \mathcal{P}$ the negative body of which is false w.r.t. X and (ii) deleting the negative body from the remaining rules. An answer set of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

Example 2. Given the (general) program $\mathcal{P}_3 = \{a \vee b \,{:}{-}\, c.\,,\; b \,{:}{-}\, \texttt{not}\ a, \texttt{not}\ c.\,,\; a \vee c \,{:}{-}\, \texttt{not}\ b.\}$ and $I = \{b, \texttt{not}\ a, \texttt{not}\ c\}$, the reduct $\mathcal{P}_3^I$ is $\{a \vee b \,{:}{-}\, c., b.\}$. It is easy to see that I is an answer set of $\mathcal{P}_3^I$, and for this reason it is also an answer set of $\mathcal{P}_3$. ∎

3 Answer Set Computation

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine of the DLV system, which will be used for the experiments, but also other ASP systems, employ a similar procedure.

An answer set program $\mathcal{P}$ in general contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[3] The subsequent computations are then performed on $ground(\mathcal{P})$.

```
Function ModelGenerator(I: Interpretation): Boolean;
var   inconsistency: Boolean;
begin
    I := DetCons(I);
    if I = $\mathcal{L}$ then return False; (* inconsistency *)
    if no atom is undefined in I then return IsAnswerSet(I);
    Select an undefined ground atom $A$ according to a heuristic;
    if ModelGenerator($I \cup \{A\}$) then return True;
    else return ModelGenerator($I \cup \{\texttt{not}\ A\}$);
end;
```

Fig. 1. Computation of Answer Sets

The heart of the computation is performed by the Model Generator, which is sketched in Figure 1. The ModelGenerator function is initially called with parameter I set to the empty interpretation.[4] If the program $\mathcal{P}$ has an answer set, then the function returns True, setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the DPLL procedure employed by SAT solvers. It first calls a function DetCons(), which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals $\mathcal{L}$ upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, an atom A is

[3] Note that $ground(\mathcal{P})$ is usually not the full $Ground(\mathcal{P})$; rather, it is a subset (often much smaller) of it having precisely the same answer sets as $\mathcal{P}$.

[4] Observe that the interpretations built during the computation are 3-valued, that is, a literal can be True, False or Undefined w.r.t. I.

selected according to a heuristic criterion and ModelGenerator is called on $I \cup \{A\}$ and on $I \cup \{\texttt{not}\ A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a "good" atom A is crucial for the performance of an ASP system. In the next section, we describe some heuristic criteria for the selection of such branching atoms.

If no atom is left for branching, the Model Generator has produced a "candidate" answer set, the stability of which is subsequently verified by *IsAnswerSet(I)*. This function checks whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$ obtained by applying the GL-transformation w.r.t. I, and outputs the model, if so. *IsAnswerSet(I)* returns True if the computation should be stopped and False otherwise.

4 Reasons for Literals

Once a literal has been assigned a truth value during the computation, we can associate a reason for that fact with the literal. For instance, given a rule $a \mathrel{:-} b, c, \texttt{not}\ \ d.$, if b and c are true and d is false in the current partial interpretation, then a will be derived as true (by Forward Propagation). In this case, we can say that a is true "because" b and c are true and d is false. A special case are *chosen* literals, as their only reason is the fact that they have been chosen. The chosen literals can therefore be seen as being their own reason, and we may refer to them as elementary reasons. All other reasons are consequences of elementary reasons, and hence aggregations of elementary reasons.

Each literal l derived during the propagation (i.e., DetCons) will have an associated set of positive integers $R(l)$ representing the reason of l, which are essentially the recursion levels of the chosen literals which entail l. Therefore, for any chosen literal c, $|R(c)| = 1$ holds. For instance, if $R(l) = \{1, 3, 4\}$, then the literals chosen at recursion levels 1, 3 and 4 entail l. If $R(l) = \emptyset$, then l is true in all answer sets.

The process of defining reasons for derived (non-chosen) literals is called *reason calculus*. The reason calculus we employ defines the auxiliary concepts of satisfying literals and orderings among satisfying literals for a given rule. It also has special definitions for literals derived by the well-founded operator. Here, for lack of space, we do not report details of this calculus, and refer to [10] for a detailed definition.

When an inconsistency is determined, we use reason information in order to understand which chosen literals have to be undone in order to avoid the found inconsistency. Implicitly this also means that all choices which are not in the reason do not have any influence on the inconsistency. We can isolate two main types of inconsistencies: (i) Deriving conflicting literals, and (ii) failing stability checks. Of these two, the second one is a peculiarity of disjunctive ASP.

Deriving conflicting literals means, in our setting, that DetCons determines that an atom a and its negation $\texttt{not}\ \ a$ should both hold. In this case, the reason of the inconsistency is – rather straightforward – the combination of the reasons for a and $\texttt{not}\ \ a$: $R(a) \cup R(\texttt{not}\ .a)$.

Inconsistencies from failing stability checks are different and a peculiarity of disjunctive ASP, as non-disjunctive ASP systems usually do not employ a stability check. This situation occurs if the function IsAnswerSet(I) of Section 3 returns false, hence if the checked interpretation (which is guaranteed to be a model) is not stable. The reason for such an inconsistency is always based on an unfounded set, which has been determined inside IsAnswerSet(I) as a side-effect. Using this unfounded set, the reason for the inconsistency is composed of the reasons of literals which satisfy rules which contain unfounded atoms in their head (the cancelling assignments of these rules). Note that unsatisfied rules with unfounded atoms in their heads are not relevant for stability and hence do not contribute to the reason. The information on reasons for inconsistencies can be exploited for backjumping, as described in [10], by going back to the closest choice which is a reason for the inconsistency, rather than always to the immediately preceding choice. In the remainder of this paper, we will describe extensions of a backjumping-based solver by further exploiting the information provided by reasons. In particular, in the following section we describe how reasons for inconsistencies can be exploited for defining a look-back heuristic.

5 Heuristics

In this section we will first describe the two main heuristics for DLV (based on look-ahead), and subsequently define several new heuristics based on reasons, which are computed as side-effects of the backjumping technique. Throughout this section, we assume that a ground ASP program $\mathcal{P}$ and an interpretation I have been fixed. We first recall the "standard" DLV heuristic h_{UT} [7], which has recently been refined to yield the heuristic h_{DS} [18], which is more "specialized" for hard disjunctive programs (like 2QBF). These are look-ahead heuristics, that is, the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q, $ext(Q)$ will denote the interpretation resulting from the application of DetCons on $I \cup \{Q\}$; w.l.o.g., we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

Standard Heuristic of DLV (**h_{UT}**). This heuristic, which is the default in the DLV distribution, has been proposed in [7], where it was shown to be very effective on many relevant problems. It exploits a peculiar property of ASP, namely *supportedness*: For each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r. Since an ASP system must eventually converge to a supported interpretation, h_{DS} is geared towards choosing those literals which minimize the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule. The heuristic h_{UT} is "balanced", that is, the heuristic values of an atom Q depends on both the effect of taking Q and `not` Q, the decision between Q and `not` Q is based on the same criteria involving UT atoms.

Enhanced Heuristic of DLV (**h_{DS}**). The heuristic h_{DS}, proposed in [19] is based on h_{UT}, and is different from h_{UT} only for pairs of literals which are not ordered by h_{UT}. The idea of the additional criterion is that interpretations having a "higher degree of supportedness" are preferred, where the degree of supportedness is the average number of supporting rules for the true atoms. Intuitively, if all true atoms have many supporting rules in a model M, then the elimination of a true atom from the interpretation would violate many rules, and it becomes less likely finding a subset of M which is a model of $\mathcal{P}^M$ (which would disprove that M is an answer set). Interpretations with a higher degree of supportedness are therefore more likely to be answer sets. Just like h_{UT}, h_{DS} is "balanced".

The Look-back Heuristics (h_{LB}). We next describe a family of new look-back heuristics h_{LB}. Different to h_{UT} and h_{DS}, which provide a partial order on potential choices, h_{LB} assigns a number ($V(L)$) to each literal L (thereby inducing an implicit order). This number is periodically updated using the inconsistencies that occurred after the most recent update. Whenever a literal is to be selected, the literal with the largest $V(L)$ will be chosen. If several literals have the same $V(L)$, then negative literals are preferred over positive ones, but among negative and positive literals having the same $V(L)$, the ordering will be random.

In more detail, for each literal L, two values are stored: $V(L)$, the current heuristic value, and $I(L)$, the number of inconsistencies L has been a reason for (as discussed in Section 4) since the most recent heuristic value update. After having chosen k literals, $V(L)$ is updated for each L as follows: $V(L) := V(L)/2 + I(L)$. The motivation for the division (which is assumed to be defined on integers by rounding the result) is to give more impact to more recent values. Note that $I(L) \neq 0$ can hold only for literals that have been chosen earlier during the computation.

A crucial point left unspecified by the definition so far are the initial values of $V(L)$. Given that initially no information about inconsistencies is available, it is not obvious how to define this initialization. On the other hand, initializing these values seems to be crucial, as making poor choices in the beginning of the computation can be fatal for efficiency. Here, we present two alternative initializations: The first, denoted by h_{LB}^{MF}, is done by initializing $V(L)$ by the number of occurrences of L in the program rules. The other, denoted by h_{LB}^{LF}, involves ordering the atoms with respect to h_{DS}, and initializing $V(L)$ by the rank in this ordering. The motivation for h_{LB}^{MF} is that it is fast to compute and stays with the "no look-ahead" paradigm of h_{LB}. The motivation for h_{LB}^{LF} is to try to use a lot of information initially, as the first choices are often critical for the size of the subsequent computation tree.

We introduce yet another option for h_{LB}, motivated by the fact that answer sets for disjunctive programs must be minimal with respect to atoms interpreted as true, and the fact that the checks for minimality are costly: If we preferably choose false literals, then the computed answer set candidates may have a better chance to be already minimal. Thus even if the literal, which is optimal according to the heuristic, is positive, we will choose the corresponding negative literal first. If we employ this option in the heuristic, we denote it by adding AF to the superscript, arriving at $h_{LB}^{MF,AF}$ and $h_{LB}^{LF,AF}$ respectively.

Note also that the complexity of look-ahead heuristics is in general quadratic (in the number of atoms), and becomes linear is a bound on the number of atoms to be analyzed is a-priori known. On the other hand, h_{LB} heuristics are constant time, but need the values $V(L)$ to be re-ordered after having chosen k literals.

6 Experiments

We have implemented all the proposed heuristics in DLV; in this section, we report on their experimental evaluation.

6.1 Compared Methods

For our experiments, we have compared several versions of DLV [11], which differ on the employed heuristics and the use of backjumping. For having a broader picture, we have also compared our implementations to the competing systems GnT and CModels3. The considered systems are:
• **dlv.ut**: the standard DLV system employing h_{UT} (based on look-ahead).
• **dlv.ds**: DLV with h_{DS}, the look-ahead based heuristic specialized for Σ_2^P/Π_2^P hard disjunctive programs.
• **dlv.ds.bj**: DLV with h_{DS} and backjumping.
• **dlv.mf**: DLV with h_{LB}^{MF}.[5]
• **dlv.mf.af**: DLV with $h_{LB}^{MF,AF}$.
• **dlv.lf**: DLV with h_{LB}^{LF}.
• **dlv.lf.af**: DLV with $h_{LB}^{LF,AF}$.
• **gnt** [12]: The solver GnT, based on the Smodels system, can deal with disjunctive ASP. One instance of Smodels generates candidate models, while another instance tests if a candidate model is stable.
• **cm3** [13]: CModels3, a solver based on the definition of completion for disjunctive programs and the extension of loop formulas to the disjunctive case. CModels3 uses two SAT solvers in an interleaved way, the first for finding answer set candidates using the completion of the input program and loop formulas obtained during the computation, the second for verifying if the candidate model is indeed an answer set.

Note that we have not taken into account other solvers like Smodels$_{cc}$ [16] or Clasp [17] because our focus is on disjunctive ASP.

6.2 Benchmark Programs and Data

The proposed heuristic aims at improving the performance of DLV on disjunctive ASP programs. Therefore we focus on hard programs in this class, which is known to be able to express each problem of the complexity class Σ_2^P. All of the instances that we have considered in our benchmark analysis have been derived from instances for 2QBF, the canonical Σ_2^P-complete problem. This choice is motivated by the fact that many real-world, structured instances for problems in Σ_2^P are available for 2QBF on QBFLIB [20], and moreover, studies on the location of hard instances for randomly generated 2QBFs have been reported in [21,22,23].

The problem 2QBF is to decide whether a quantified Boolean formula (QBF) $\Phi = \forall X \exists Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = D_1 \wedge \ldots \wedge D_k$ is a CNF formula over $X \cup Y$, is valid.

[5] Note that all systems with h_{LB} heuristics exploit backjumping.

The transformation from 2QBF to disjunctive logic programming is a slightly altered form of a reduction used in [24]. The propositional disjunctive logic program $\mathcal{P}_\phi$ produced by the transformation requires $2 * (|X| + |Y|) + 1$ propositional predicates (with one dedicated predicate w), and consists of the following rules. Rules of the form $v \vee \bar{v}$. for each variable $v \in X \cup Y$.
Rules of the form $y \leftarrow w$. $\bar{y} \leftarrow w$. for each $y \in Y$. Rules of the form $w \leftarrow \bar{v}_1, \ldots, \bar{v}_m, v_{m+1}, \ldots, v_n$. for each disjunction $v_1 \vee ... \vee v_m \vee \neg v_{m+1} \vee ... \vee \neg v_n$ in ϕ. The rule $\leftarrow$ `not` w. The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ has no answer set [24].

We have selected both random and structured QBF instances. The random 2QBF instances have been generated following recent phase transition results for QBFs [21,22,23]. In particular, the generation method described in [23] has been employed and the generation parameters have been chosen according to the experimental results reported in the same paper. We have generated 13 different sets of instances, each of which is labelled with an indication of the employed generation parameters. In particular, the label "A-E-C-ρ" indicates the set of instances in which each clause has A universally-quantified variables and E existentially-quantified variables randomly chosen from a set containing C variables, such that the ratio between universal and existential variables is ρ. For example, the instances in the set "3-3-50-0.8" are 6CNF formulas (each clause having exactly 3 universally-quantified variables and 3 existentially-quantified variables) whose variables are randomly chosen from a set of 50 containing 22 universal and 28 existential variables, respectively. In order to compare the performance of the systems in the vicinity of the phase transition, each set of generated formulas has an increasing ratio of clauses over existential variables (from 1 to max$r$). Following the results presented in [23], max$r$ has been set to 21 for each of the sets 3-3-50-* and 3-3-70-*, and 12 for each of the 2-3-80-*. We have generated 10 instances for each ratio, thus obtaining, in total, 210 and 120 instances per set, respectively.

The structured instances we have analyzed are:

– **Narizzano-Robot** - These are real-word instances encoding the robot navigation problems presented in [25].
– **Ayari-MutexP** - These QBFs encode instances to problems related to the formal equivalence checking of partial implementations of circuits, as presented in [26].
– **Letz-Tree** - These instances consist of simple variable-independent subprograms generated according to the pattern: $\forall x_1 x_3 ... x_{n-1} \exists x_2 x_4 ... x_n (c_1 \wedge \ldots \wedge c_{n-2})$ where $c_i = x_i \vee x_{i+2} \vee x_{i+3}$, $c_{i+1} = \neg x_i \vee \neg x_{i+2} \vee \neg x_{i+3}$, $i = 1, 3, \ldots, n-3$.

The benchmark instances belonging to Letz-tree, Narizzano-robot, Ayari-MutexP have been obtained from QBFLIB [20], including the 32 Narizzano-robot instances used in the QBF Evaluation 2004, and all the $\forall\exists$ instances from Letz-tree and Ayari-MutexP.

6.3 Results

All the experiments were performed on a 3GHz PentiumIV equipped with 1GB of RAM, 2MB of level 2 cache running Debian GNU/Linux. Time measurements have been done using the `time` command shipped with the system, counting total CPU time for the respective process.

Table 1. Number of solved instances within timeout for Random 2QBF

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3
2-3-80-0.4	106	114	114	107	100	109	103	3	47
2-3-80-0.6	83	88	89	92	71	90	83	4	58
2-3-80-0.8	78	92	95	93	70	89	86	3	65
2-3-80-1.0	78	90	91	98	66	88	85	8	77
2-3-80-1.2	72	89	94	105	74	93	95	4	87
3-3-50-0.8	210	210	210	210	210	210	210	21	166
3-3-50-1.0	191	205	202	201	199	203	202	30	163
3-3-50-1.2	196	207	206	208	203	207	206	41	191
3-3-70-0.6	126	136	135	140	127	131	131	1	61
3-3-70-0.8	112	115	115	128	103	113	119	0	68
3-3-70-1.0	91	108	109	137	94	110	108	3	82
3-3-70-1.2	104	121	122	139	90	117	121	5	108
3-3-70-1.4	106	123	124	151	98	131	126	3	118
#Total	1552	1698	1706	1809	1505	1691	1675	126	1291

We start with the results of the experiments with random 2QBF formulas. For every instance, we have allowed a maximum running time of 6 minutes. In Table 1 we report, for each system, the number of instances solved in each set within the time limit. Looking at the table, it is clear that the new look-back heuristic combined with the "mf" initialization (corresponding to the system dlv.mf) performed very well on these domains, being the version which was able to solve most instances in most settings, particularly on the 3-3-70-* sets. Also dlv.lf performed quite well, while the other variants do no seem to be very effective. Considering the look-ahead versions of DLV, dlv.ds performed reasonably well. Considering GnT and CModels3, we can note that they could solve comparatively few instances.

Comparing between the 3-3-50-* and 3-3-70-* settings, we can see that dlv.mf is the system that scales best: It is on the average when considering 50 variables, while it is considerably better when considering 70 variables.

We do not report details on the execution times due to lack of space, as aggregated results such as average or median are problematic because of the many timeouts. However, for 3-3-50-0.8 all DLV-based systems terminated, and here the average times do not differ dramatically, the best being dlv.ds (23.62s), dlv.mf (25.26s) and dlv.ds.bj (26.02s). In other settings, such as 2-3-80-0.6, we observe that dlv.mf is the best on average time over the solved instances (18.31s), while all others solve fewer instances with a higher average time. Similar considerations hold for 3-3-70-1.2 where dlv.mf solves 17 instances more than the second best, dlv.ds.bj, yet its average time is about 30% lower (22.93s vs. 34.89s).

In Tables 2, 3 and 4, we report the results, in terms of execution time for finding one answer set, and number of instances solved within 11 minutes, about the groups: Letz-Tree, Narizzano-Robot, and Ayari-MutexP, respectively. The last columns (AS?) indicate if the instance has an answer set (Y), or not (N). A "–" in these tables indicates a timeout. For h_{LB} heuristics, we experimented a few different values for "k", and we obtained the best results for k=100. However, it would be interesting to analyze

Table 2. Execution time (seconds) and number of solved instances on Narizzano-Robot instances

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	AS?
2-38.1	0.31	0.31	0.31	0.36	0.39	0.37	0.33	44.4	1.31	N
2-64.1	0.30	0.30	0.32	0.36	0.39	0.37	0.33	43.77	1.3	N
2-93.1	0.31	0.30	0.32	0.36	0.39	0.37	0.33	44.35	1.3	N
2-69.4	–	–	–	536.97	–	–	–	–	263.05	N
2-3.5	–	–	–	14.39	352.94	387.11	364.89	–	431.55	N
2-61.6	–	–	–	629.35	–	–	–	–	–	Y
2-72.7	–	–	–	14.21	–	–	–	–	390.62	N
3-17.2	14.26	7.45	5.67	4.59	5.85	8.22	7.31	–	8.1	N
3-62.4	–	–	–	362.57	–	–	–	–	211.68	N
3-80.4	–	–	–	404.93	–	–	–	–	239.96	N
4-78.1	0.30	0.30	0.31	0.43	0.51	0.37	0.33	37.3	1.31	N
4-21.2	13.40	6.84	5.27	3.60	4.25	5.68	7.31	–	8.14	N
4-73.2	13.36	6.80	4.07	2.49	3.18	4.72	6.65	–	6.68	N
4-91.4	–	–	–	236.41	–	–	–	–	212.76	N
4-85.5	–	–	504.61	3.59	156.60	109.04	372.78	–	103.04	N
4-87.8	–	–	–	244.47	–	600.36	–	–	–	Y
5-29.1	0.30	0.30	0.31	0.43	0.51	0.36	0.32	37.1	1.3	N
5-5.2	13.39	6.83	4.09	2.50	3.18	4.73	6.68	–	6.66	N
5-75.3	655.78	188.80	71.56	14.70	31.44	62.93	47.85	–	34.74	N
5-18.5	–	–	–	357.04	–	–	–	–	–	Y
5-59.5	–	–	–	357.15	–	–	–	–	–	Y
5-55.6	–	–	–	5.51	–	233.23	–	–	219.39	N
5-4.9	–	–	–	89.16	–	–	–	–	–	Y
#Solved	10	10	11	23	12	15	12	5	18	

more thoroughly the effect of the factor k. In Table 2 we report only the instances which were solved within the time limit by at least one of the compared methods. On these instances, dlv.mf was able to solve all the shown 23 instances, followed by CModels3 (18) and dlv.lf (15). Moreover, dlv.mf was also always the fastest system on each instance (sometimes dramatically), if we consider the instances on which it took more than 1 sec.

In Table 3, we then report the results for Ayari-MutexP. In that domain all the versions of DLV were able to solve all 7 instances, outperforming both CModels3 and GnT which solved only one instance. Comparing the execution times required by all the variants of dlv we note that, also in this case, dlv.mf is the best-performing version.

About the Letz-Tree domain, the DLV versions equipped with look-back heuristics solved a higher number of instances and required less CPU time (up to two orders of magnitude less) than all competitors. In particular, the look-ahead based versions of DLV, GnT and CModels3 could solve only 3 instances, while dlv.mf and dlv.lf solved 4 and 5 instances, respectively. Interestingly, here the ”lf” variant is very effective in particular when combined with the “af” option.

Summarizing, DLV equipped with look-back heuristics showed very positive performance in all of the test cases presented, both random and structured, obtaining good results both in terms of number of solved instances and execution time compared to

Table 3. Execution time (seconds) and number of solved instances on Ayari-MutexP instances

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	AS?
mutex-2-s	0.01	0.01	0.01	0.01	0.01	0.01	0.01	1.89	0.65	N
mutex-4-s	0.05	0.05	0.05	0.06	0.05	0.06	0.05	–	–	N
mutex-8-s	0.21	0.2	0.23	0.21	0.21	0.23	0.21	–	–	N
mutex-16-s	0.89	0.89	0.98	0.89	0.89	1.01	0.9	–	–	N
mutex-32-s	3.67	3.72	4.06	3.63	3.64	4.16	3.79	–	–	N
mutex-64-s	15.38	16.08	17.64	14.97	15.04	18.08	16.97	–	–	N
mutex-128-s	69.07	79.39	90.92	62.97	62.97	92.92	93.05	–	–	N
#Solved	7	7	7	7	7	7	7	1	1	

Table 4. Execution time (seconds) and number of solved instances on Letz-Tree instances

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	AS?
exa10-10	0.18	0.17	0.17	0.04	0.1	0.06	0.06	0.12	0.03	N
exa10-15	7.49	7.09	7.31	0.34	0.71	0.48	0.38	6.46	0.73	N
exa10-20	278.01	264.53	275.1	12.31	17.24	5.43	2.86	325.26	67.56	N
exa10-25	–	–	–	303.67	432.32	44.13	19.15	–	–	N
exa10-30	–	–	–	–	–	166.93	129.54	–	–	N
#Solved	3	3	3	4	4	5	5	3	3	

traditionals DLV, GnT and CModels3. dlv.mf, the "classic" look-back heuristic, performed best in most cases, but good performance was obtained also by dlv.lf. The results of dlv.lf.af on the Letz-Tree instances show that this option can be fruitfully exploited in some particular domains.

7 Conclusions

We have defined a general framework for employing look-back heuristics in disjunctive ASP, exploiting the peculiar features of this setting. We have designed a number of look-back based heuristics, addressing some key issues arising in this framework. We have implemented all proposed heuristics in the DLV system, and carried out experiments on hard instances encoding 2QBFs, comprising randomly generated instances, generated according to the method proposed in [23], and structured instances from the QBFLIB archive (Letz-Tree, Narizzano-Robot, Ayari-MutexP). It turned out that the proposed heuristics outperform the traditional (disjunctive) ASP systems DLV, GnT and CModels3 in most cases, and a rather simple approach ("dlv.mf") works particularly well.

References

1. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418

3. Rintanen, J.: Improvements to the Evaluation of Quantified Boolean Formulae. In Dean, T., ed.: IJCAI 1999, Sweden,(1999) 1192–1197
4. Eiter, T., Gottlob, G.: The Complexity of Logic-Based Abduction. JACM **42**(1) (1995) 3–42
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
6. Leone, N., Rosati, R., Scarcello, F.: Enhancing Answer Set Planning. In: IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information. (2001) 33–42
7. Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: IJCAI 2001, Seattle, WA, USA,(2001) 635–640
8. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
9. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001, ACM (2001) 530–535
10. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications **19**(2) (2006) 155–172
11. 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) (2006) 499–562
12. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923
13. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662
14. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: ICCAD 2001. (2001) 279–285
15. Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust Sat-Solver. In: DATE 2002, IEEE Computer Society (2002) 142–149
16. Ward, J., Schlipf, J.S.: Answer Set Programming with Clause Learning. In: LPNMR-7. LNCS 2923
17. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07). (2007) To appear.
18. Faber, W., Ricca, F.: Solving Hard ASP Programs Efficiently. In: LPNMR'05. LNCS 3662
19. Faber, W., Leone, N., Ricca, F.: Solving Hard Problems for the Second Level of the Polynomial Hierarchy: Heuristics and Benchmarks. Intelligenza Artificiale **2**(3) (2005) 21–28
20. Narizzano, M., Tacchella, A.: QBF Solvers Evaluation page (2002) `http://www.qbflib.org/qbfeval/index.html/`.
21. Cadoli, M., Giovanardi, A., Schaerf, M.: Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In: AI*IA 97. Italy, (1997) 207–218
22. Gent, I., Walsh, T.: The QSAT Phase Transition. In: AAAI. (1999)
23. Chen, H., Interian, Y.: A model for generating random quantified boolean formulas. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05), Professional Book Center (2005) 66–71
24. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. AMAI **15**(3/4) (1995) 289–323
25. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. Artificial Intelligence **147**(1/2) (2003) 85–117
26. Ayari, A., Basin, D.A.: Bounded Model Construction for Monadic Second-Order Logics. In: Prooceedings of Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, 15-19, 2000, Chicago, IL, USA (2000)

Complexity of Rule Redundancy in Non-ground Answer-Set Programming over Finite Domains[⋆]

Michael Fink, Reinhard Pichler, Hans Tompits, and Stefan Woltran

Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{fink,tompits,stefan}@kr.tuwien.ac.at
pichler@dbai.tuwien.ac.at

Abstract. Recent research in answer-set programming (ASP) is concerned with the problem of finding faithful transformations of logic programs under the stable semantics. This is in particular relevant in practice when programs with variables are considered, where such transformations play a basic role in (offline) simplifications of logic programs. So far, such transformations of non-ground programs have been considered under the implicit assumption that the domain (i.e., the set of constants of the underlying language) is always suitably extensible. However, this may not be a desired scenario, e.g., if one needs to deal with a fixed number of objects. In this paper, we investigate how an explicit restriction of the domain influences the applicability of program transformations and we study in detail computational aspects for the concepts of tautological rules and rule subsumption. More precisely, we provide a full picture of the complexity to decide whether a non-ground rule is tautological or subsumed by another rule under several restrictions.

1 Introduction

Answer-set programming (ASP) has emerged as an important paradigm for declarative problem solving, and provides a host for many different application domains on the basis of nonmonotonic logic programs. The increasing popularity of ASP has also raised interest in the question of equivalence between programs [1,2], which is relevant concerning formal underpinnings for program optimization, where equivalence-preserving modifications are of primary interest; in particular, rewriting rules which allow to perform a local change in a program are important. Many such rules have been considered in the propositional setting [3,4,5,6], but just recently have they been extended to the practicably important case of non-ground programs [7].

In the latter work, a countable domain of constants is assumed, and although this is a reasonable assumption for many scenarios and also common in database theory, it is sometimes more desirable to consider the underlying language in a more restricted way, assuming only a finite, possibly fixed set of constants. While such a finite set comes for free in computing answer-sets of a (complete) program via its active domain, i.e., the set of constants occurring in the program under consideration, this is not the case

[⋆] This work was partially supported by the Austrian Science Fund (FWF) under project P18019.

for program replacements, which should be applicable in a local sense, for instance to program parts. To this end, such replacements have to take the underlying global domain into account, rather than the active domain of a given program.

In this paper, we consider two important replacements in non-ground answer-set programming under this point of view and analyze their complexity. Intuitively, one would expect that the complexity of a problem decreases as the domain under consideration decreases. In particular, one might hope to get more favorable complexity results when a finite domain is considered rather than a countable domain. On the one hand, Eiter *et al.* [1] show that the restriction to finite domains turns the, in general, undecidable problem of uniform equivalence between logic programs into a decidable one. On the other hand, there is a related problem in the literature where the complexity increases when the domain is restricted: Lassez and Marriot [8] identify so-called *implicit generalizations* as a formal basis of machine learning from counter examples. One of the main problems studied there is the following: Given an atom A and a set $\{B_1, \ldots, B_n\}$ of atoms over some domain $\mathcal{C}$, is every ground instance of A also a ground instance of some B_i? Lassez and Marriot [8] show that this problem is tractable in case of a countably infinite set of constants. This is in contrast to the case of a finite domain, where this problem becomes coNP-complete [9,10]. Now the question naturally arises whether this somewhat counter-intuitive effect of an increased complexity in case of a decreased size of domain also holds for replacements in non-ground answer-set programming. We show that this is indeed the case. In particular, our contributions are as follows, assuming the restriction to a finite domain:

– We show that the detection of tautological rules is NP-complete; and that hardness remains even for some restrictions on the syntax of rules.
– We show that the problem of deciding rule subsumption becomes Π_2^P-complete, and again hardness holds also under several restrictions.

These two main results reveal that complexity increases when we restrict our attention to finite domains, since the detection of tautological rules is tractable under countably infinite domains and rule subsumption is only NP-complete in this setting [7]. However, we also provide results where the problems under consideration are tractable under finite domains as well. In particular, we show that

– detecting tautological Horn rules remains tractable if the maximal arity of predicates is fixed by some constant, and
– detecting tautological rules, as well as rule subsumption, remains tractable in case the number of variables occurring in the involved rules is fixed by some constant.

2 Preliminaries

Our objects of interest are disjunctive logic programs formulated in a language $\mathcal{L}$ over a finite set $\mathcal{A}$ of *predicate symbols*, a finite set $\mathcal{V}$ of *variables*, and a set $\mathcal{C}$ of *constants* (also called the *domain*), which may be either finite or countably infinite.

An *atom* (over $\mathcal{L}$) is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol from $\mathcal{A}$ of arity $ar(p) = n$ and $t_i \in \mathcal{C} \cup \mathcal{V}$, for $1 \leq i \leq n$. A (*disjunctive*) *rule* (over $\mathcal{L}$), r, is an ordered pair of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m,$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms (with $n \geq 0$, $m \geq k \geq 0$, and $n + m > 0$), and "not" denotes *default negation*. The *head* of r is $H(r) = \{a_1, \ldots, a_n\}$, and the *body* of r is $B(r) = \{b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m\}$. We also define $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. We call r *positive* if $k = m$, and *Horn* if r is positive and $n = 1$. Furthermore, r is a *fact* if $m = 0$ and $n = 1$ (in which case "$\leftarrow$" is usually omitted). As well, r is *safe* if each variable occurring in $H(r) \cup B^-(r)$ also occurs in $B^+(r)$. By a *program* (over $\mathcal{L}$) we understand a finite set of safe rules (over $\mathcal{L}$).

Let ε be an atom, a rule, or a program. The set of variables occurring in ε is denoted by $\mathcal{V}_\varepsilon$, and ε is called *ground* if $\mathcal{V}_\varepsilon = \emptyset$. Similarly, we write $\mathcal{C}_\varepsilon$ to refer to the set of constants occurring in ε and $\mathcal{A}_\varepsilon$ to refer to the set of predicates occurring in ε. Furthermore, for a set $C \subseteq \mathcal{C}$ of constants, we write $B_{\mathcal{A},C}$ to denote the set of all ground atoms constructible from the predicate symbols from $\mathcal{A}$ and the constants from C. Moreover, for a set A of predicates, $ar_{\max}(A) = \max\{ar(p) \mid p \in A\}$.

Given a rule r and some $C \subseteq \mathcal{C}$, we define $grd(r, C)$ as the set of all rules $r\vartheta$ obtained from r by all possible substitutions $\vartheta : \mathcal{V}_r \rightarrow C$. Moreover, for any program P, the *grounding of P with respect to C* is given by $grd(P, C) = \bigcup_{r \in P} grd(r, C)$. The program $grd(P)$ is $grd(P, \mathcal{C}_P)$ for $\mathcal{C}_P \neq \emptyset$, and $grd(P, \{c\})$ otherwise, where c is an arbitrary element from $\mathcal{C}$.

By an *interpretation* (over $\mathcal{L}$) we understand a set of ground atoms (over $\mathcal{L}$). A ground rule r is *satisfied* by an interpretation I, symbolically $I \models r$, iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies a ground program P, symbolically $I \models P$, iff $I \models r$, for each $r \in P$. The *Gelfond-Lifschitz reduct* [11] of a ground program P with respect to an interpretation I is given by $P^I = \{H(r) \leftarrow B^+(r) \mid r \in P,\ I \cap B^-(r) = \emptyset\}$. An interpretation I is an *answer set* of P iff I is a minimal model of $grd(P)^I$. $\mathcal{AS}(P)$ denotes the set of all answer sets of a program P.

Programs P_1 and P_2 are called (*ordinarily*) *equivalent* iff $\mathcal{AS}(P_1) = \mathcal{AS}(P_2)$. Furthermore, P_1 and P_2 are *strongly equivalent* (*in $\mathcal{L}$*), in symbols $P_1 \equiv_s P_2$, iff, for each set S of rules, $\mathcal{AS}(P_1 \cup S) = \mathcal{AS}(P_2 \cup S)$. $(J, I)_C$ is an *SE-model* [1] of a program P iff (i) $J \subseteq I$, (ii) $I \models grd(P, C)$, and (iii) $J \models grd(P, C)^I$. We define $SE_C(P) = \{(J, I)_C \mid (J, I)_C$ is an SE-model of $P\}$, for a given $C \subseteq \mathcal{C}$, and $SE(P) = \bigcup_{C \subseteq \mathcal{C}} SE_C(P)$. For all programs P_1 and P_2, the following three conditions are equivalent [1]: (i) $P_1 \equiv_s P_2$, (ii) $SE(P_1) = SE(P_2)$, and (iii) $SE_C(P_1) = SE_C(P_2)$, for every finite $C \subseteq \mathcal{C}$.

Deciding strong equivalence is co-NEXPTIME-complete both for languages over a finite domain as well as for languages over an infinite domain [1]. In what follows, we assume familiarity with the basic complexity classes P, NP, coNP, Δ_2^P, Σ_2^P, and Π_2^P from the literature (cf., e.g., Garey and Johnson [12] for an overview).

3 Tautological Rules

In this section, we syntactically characterize rules which can be deleted in any program (over a given language $\mathcal{L}$), i.e., which are *tautological*. Following Eiter *et al.* [7], let us define

$$\Theta = \{r \mid B^+(r) \cap (H(r) \cup B^-(r)) \neq \emptyset\} \text{ and}$$
$$\Xi^{\mathcal{C}} = \{r \mid \text{ for each } \vartheta : \mathcal{V}_r \to \mathcal{C}, B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) \neq \emptyset\}.$$

Note that the former set does not explicitly refer to the domain of the underlying language. The following proposition rephrases results by Eiter *et al.* [7].

Proposition 1. *Let $\mathcal{L}$ be a language over an infinite domain $\mathcal{C}$ and r a rule. Then, the following conditions are equivalent:* (*i*) $P \equiv_{\mathrm{s}} P \setminus \{r\}$*, for each program P over $\mathcal{L}$;* (*ii*) $r \in \Theta$*; and* (*iii*) $r \in \Xi^{\mathcal{C}}$.

In other words, both sets, Θ and $\Xi^{\mathcal{C}}$, contain exactly the same rules in case the domain $\mathcal{C}$ of the underlying language $\mathcal{L}$ is infinite. Moreover, they capture *all* rules which can be faithfully removed in programs over $\mathcal{L}$. Also observe that both sets are equal if only ground rules are taken into account.

Deciding $r \in \Theta$ is an easy syntactic check. Thus, in case the underlying language is given over an infinite domain, the problem of recognizing exactly those rules which can be deleted in any program is an easy task. In particular this test is not harder than in the ground case (which was fully established by Inoue and Sakama [5] and by Lin and Chen [6]), and can be done in linear time.

The situation differs, however, if we restrict our attention to a finite domain. Consider $r : h(1) \vee h(0) \leftarrow h(X)$, which is obviously not contained in Θ. However, under the binary domain $\mathcal{C} = \{0, 1\}$, we have $r \in \Xi^{\mathcal{C}}$. Observe that each grounding $r\vartheta$ with $\vartheta : \mathcal{V}_r \to \mathcal{C}$ yields a tautological (ground) rule contained in Θ. As shown below, each $r \in \Xi^{\mathcal{C}}$ can be faithfully removed from a program. Thus, in the setting of a finite domain $\mathcal{C}$, we have, for each rule r, that $r \in \Theta$ implies $r \in \Xi^{\mathcal{C}}$, but not vice versa.

In the remainder of this section, we assume $\mathcal{L}$ to be given over a finite domain $\mathcal{C}$. Our first result shows that $\Xi^{\mathcal{C}}$ contains all rules which can be faithfully removed from programs in this scenario. Hence, we subsequently call rules $r \in \Xi^{\mathcal{C}}$ *tautological in* (*domain*) $\mathcal{C}$.

Theorem 1. *For a language $\mathcal{L}$ over finite domain $\mathcal{C}$, we have that for any program P and any rule r, P is strongly equivalent in $\mathcal{L}$ to $P \setminus \{r\}$ iff $r \in \Xi^{\mathcal{C}}$.*

The proof of this result is along the lines of proofs given by Eiter *et al.* [7].

3.1 Complexity of Detecting Tautological Rules in a Finite Domain

In what follows, we establish results about the computational cost for deciding whether a rule is tautological in $\mathcal{C}$, i.e., whether it is contained in $\Xi^{\mathcal{C}}$. Thus, assuming a finite domain, in view of Theorem 1, the considered problem amounts to checking which rules can be faithfully deleted from any program. Recall that in the infinite case, this problem is decidable in linear time by just checking $r \in \Theta$, according to Proposition 1. As we show below, in the finite case we observe an increase of the difficulty to recognize tautological rules, even in very restricted settings.

Theorem 2. *Given a rule r, checking whether r is tautological in a fixed finite domain $\mathcal{C}$ of size ≥ 2 is* coNP*-complete. Hardness holds even for positive rules with bounded arities.*

Proof. Membership is easy. In order to test that a rule is *not* tautological in $\mathcal{C}$, we guess a ground substitution $\vartheta : \mathcal{V}_r \to \mathcal{C}$ and check in polynomial time that $B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) = \emptyset$ holds.

For hardness, suppose that the domain $\mathcal{C}$ is of size $\ell + 1$ with $\ell \geq 1$. Without loss of generality, let $\mathcal{C}$ be of the form $\mathcal{C} = \{0, 1, , \dots, \ell\}$. We prove coNP-hardness via a reduction from UNSAT. Let $\phi = \bigwedge_{i=1}^n l_{i,1} \vee l_{i,2} \vee l_{i,3}$ be a formula in CNF over propositional atoms $X_1, \dots, X_m$, and consider a positive rule r_ϕ with

$$H(r_\phi) = \{c(0,0,0)\} \cup \{v(\alpha, \beta) \mid (\alpha, \beta) \in \mathcal{C}^2 \setminus \{(0,1), (1,0)\}\} \text{ and}$$
$$B(r_\phi) = \{c(l^*_{i,1}, l^*_{i,2}, l^*_{i,3}) \mid 1 \leq i \leq n\} \ \cup \ \{v(X_j, \bar{X}_j) \mid 1 \leq j \leq m\},$$

where $l^* = X$ if $l = X$, and $l^* = \bar{X}$ if $l = \neg X$, with $\bar{X}_1, \dots, \bar{X}_m$ being new variables. By a slight abuse of notation, we use X_i to denote either a propositional atom (in ϕ) or a first-order variable (in r_ϕ). We claim that ϕ is unsatisfiable iff r_ϕ is tautological in $\mathcal{C}$.

For the "only if"-direction, suppose that r_ϕ is not tautological in $\mathcal{C}$. Hence, since $B^-(r_\phi) = \emptyset$, there exists a substitution $\vartheta : \mathcal{V} \to \mathcal{C}$ such that $B^+(r_\phi\vartheta) \cap H(r_\phi\vartheta) = \emptyset$. Thus, for each propositional atom X_i in ϕ, the pair of first-order variables $(X_i, \bar{X}_i)$ in r_ϕ is either instantiated to $(0,1)$ or $(1,0)$, since otherwise $v(X_i, \bar{X}_i)\vartheta$ would match with some atom $v(\alpha, \beta)$ from $H(r_\phi)$. Thus, we can view ϑ as an assignment to the propositional variables X_i from ϕ. Now, since no atom in $B^+(r_\phi\vartheta)$ matches with $c(0,0,0)$ from $H(r_\phi)$, each clause in ϕ is satisfied "under ϑ", i.e., in each clause at least one propositional literal is assigned to true by the truth assignment ϑ. Hence, ϕ is satisfiable. The "if"-direction is by essentially the same arguments. $\square$

Note that according to Theorem 2, checking whether a disjunctive rule is tautological is coNP-hard even if the domain is fixed and, moreover, the number of predicate symbols and their arities are bounded. If we drop the restriction of bounded arities, then coNP-hardness can be shown even for Horn rules.

Theorem 3. *Given a Horn rule r, checking whether r is tautological in a fixed finite domain $\mathcal{C}$ of size ≥ 2 is* coNP*-complete.*

Proof. Membership is shown as above. Hardness is along the lines of Kunen [9] and Kapur *et al.* [10]. Suppose that the domain $\mathcal{C}$ is of size $\ell + 1$ with $\ell \geq 1$. Without loss of generality, let $\mathcal{C}$ be of the form $\mathcal{C} = \{0, 1, , \dots, \ell\}$. Again, we prove coNP-hardness via a reduction from UNSAT. Let $\phi = \bigwedge_{i=1}^n l_{i,1} \vee l_{i,2} \vee l_{i,3}$ be a formula in CNF over propositional atoms $X_1, \dots, X_m$. Without loss of generality, we may assume that every propositional variable X_j occurs at most once in each clause, i.e., for every $i \in \{1, \dots, n\}$, there cannot be two literals such that one is either identical to the other or one is the dual of the other. Note that clauses containing some propositional variable plus its dual can be faithfully deleted from ϕ.

Now consider a Horn rule r_ϕ such that

$$\begin{aligned} H(r_\phi) &= \{p(X_1, \dots, X_m)\} \text{ and} \\ B(r_\phi) &= \{p(s_{i1}, \dots, s_{im}) \mid 1 \leq i \leq n\} \ \cup \\ &\quad \{p(X_1, , \dots, X_{j-1}, \alpha, X_{j+1}, \dots, X_m) \mid \alpha \in \mathcal{C} \setminus \{0,1\} \text{ and } 1 \leq j \leq m\}, \end{aligned}$$

where the arguments s_{ij} with $1 \leq i \leq n$ and $1 \leq j \leq m$ are defined as follows:

$$s_{ij} = \begin{cases} 1 & \text{if the negative literal } \neg X_j \text{ occurs in the } i\text{-th clause of } \phi; \\ 0 & \text{if the positive literal } X_j \text{ occurs in the } i\text{-th clause of } \phi; \\ X_i & \text{otherwise.} \end{cases}$$

We claim that ϕ is unsatisfiable iff r_ϕ is tautological in $\mathcal{C}$.

For the "only if"-direction, suppose that ϕ is unsatisfiable. Moreover, let $\vartheta : \mathcal{V} \to \mathcal{C}$ be an arbitrary ground substitution over the variables in r_ϕ. Since $B^-(r_\phi) = \emptyset$, we have to show that $B^+(r_\phi\vartheta) \cap H(r_\phi\vartheta) \neq \emptyset$. First suppose that ϑ instantiates at least one variable X_j to $\alpha \in \mathcal{C} \setminus \{0,1\}$. Then, $p(X_1, \ldots, X_{j-1}, \alpha, X_{j+1}, \ldots, X_m)\vartheta$ and $p(X_1, \ldots, X_m)\vartheta$ are identical. Thus, in this case, $p(X_1, \ldots, X_m)\vartheta \in B^+(r_\phi\vartheta) \cap H(r_\phi\vartheta)$, and therefore $B^+(r_\phi\vartheta) \cap H(r_\phi\vartheta) \neq \emptyset$.

It remains to consider the case that ϑ instantiates all variables X_j to either 0 or 1. Hence, ϑ defines a truth assignment of $\{X_1, \ldots, X_m\}$. Since ϕ is unsatisfiable, there must exist some clause $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ with truth value false in ϑ. We claim that then $p(X_1, \ldots, X_m)\vartheta = p(s_{i1}, \ldots, s_{im})\vartheta$ holds, i.e., for every j, $X_j\vartheta = s_{ij}\vartheta$. We prove this claim by distinguishing the three cases of the definition of s_{ij}:

- If the negative literal $\neg X_j$ occurs in the i-th clause, then $s_{ij} = 1$. On the other hand, the i-th clause, and therefore also the literal $\neg X_j$, is false under the assignment ϑ. Thus, X_j has the value true in this assignment, i.e., $X_j\vartheta = 1 = s_{ij}\vartheta$.
- If the positive literal X_j occurs in the i-th clause, then $s_{ij} = 0$. On the other hand, the i-th clause, and therefore also the literal X_j, is false under the assignment ϑ. Thus, $X_j\vartheta = 0 = s_{ij}\vartheta$.
- If X_j does not occur in the i-th clause, then $s_{ij} = X_j$, and therefore $X_j\vartheta = s_{ij}\vartheta$ trivially holds.

The "if"-direction is shown analogously and is therefore omitted. □

Note that the coNP-completeness results in Theorems 2 and 3 were shown for an arbitrary but *fixed* finite domain $\mathcal{C}$ of size $|\mathcal{C}| \geq 2$. As far as coNP-hardness is concerned, we thus get slightly stronger results than if we considered the domain $\mathcal{C}$ as part of the problem input. On the other hand, it can be easily verified that the membership proofs clearly also work if the finite domain $\mathcal{C}$ is not fixed (i.e., if it is part of the problem input). In other words, detecting tautological rules has the same complexity no matter whether we have to deal with a specific finite domain or with all finite domains.

3.2 Tractable Cases

We conclude our discussion on the detection of tautological rules by identifying two tractable cases. The first one combines the restrictions considered in Theorems 2 and 3; the second one is obtained by a restriction on the variables occurring in a rule.

Theorem 4. *Given a Horn rule r, where the arity of all predicate symbols is bounded by some fixed constant, checking whether r is tautological in a finite domain is in* P.

Proof. Since r is Horn, the head $H(r)$ of r consists of a single atom A and $B^-(r)$ is empty. Hence, r is tautological in $\mathcal{C}$ iff $A\vartheta \in B^+(r\vartheta)$ holds for every ground substitution ϑ. In order to check this condition, we loop over all possible ground instantiations $A\sigma$ of A. Since the arity of the predicate symbols is bounded and the domain is finite, there are only polynomially many such ground instantiations. For each $A\sigma$, we have to check whether σ can be extended to a substitution ϑ such that $A\sigma = A\vartheta \in B^+(r\vartheta)$. This is a matching problem, which can be clearly solved in polynomial time. □

Theorem 5. *Given a rule r such that $|\mathcal{V}_r| \leq k$ for some fixed constant k, checking whether r is tautological in a finite domain is in* P.

Proof. Since the number of variables in r is bounded by a constant and the domain is finite, there are only polynomially many ground instances $r\vartheta$ of r. In order to test whether r is tautological in $\mathcal{C}$, we just have to test for each ground instance $r\vartheta$ of r whether $B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) \neq \emptyset$ holds. □

4 Rule Subsumption

Rule subsumption is a syntactic criterion to identify a rule r which can be faithfully deleted from any program containing another ("more general") rule s. For the ground case, Lin and Chen [6] generalized replacements from the literature [3,13] and showed that their syntactic criterion captures *all* such pairs of rules. The non-ground case was first studied by Eiter *et al.* [7] and Traxler [14]. As shown by the latter author, also rule subsumption can be characterized in two alternative ways (similarly as before for tautological rules), which turn out to be equivalent for languages over an infinite domain. To formulate these characterizations, let us define the following relations (for any pair of rules r, s):

$$
\begin{aligned}
s \leq r \text{ iff } & \text{there exists a substitution } \vartheta : \mathcal{V}_s \rightarrow \mathcal{V}_r \cup \mathcal{C}_r \text{ such that} \\
& H(s\vartheta) \subseteq H(r) \cup B^-(r) \text{ and } B(s\vartheta) \subseteq B(r); \\
s \preceq^{\mathcal{C}} r \text{ iff, } & \text{for each } \vartheta_r : \mathcal{V}_r \rightarrow \mathcal{C}, \text{ there exists a } \vartheta_s : \mathcal{V}_s \rightarrow \mathcal{C} \text{ such that} \\
& H(s\vartheta_s) \subseteq H(r\vartheta_r) \cup B^-(r\vartheta_r) \text{ and } B(s\vartheta_s) \subseteq B(r\vartheta_r).
\end{aligned}
$$

Observe that $\leq$ does not take the underlying domain into account, but only variables and constants involved in the two rules, r and s, while $\preceq^{\mathcal{C}}$ explicitly refers to the domain $\mathcal{C}$ of the underlying language. The following proposition collects results from Eiter *et al.* [7] and Traxler [14].

Proposition 2. *Let $\mathcal{L}$ be given over an infinite domain $\mathcal{C}$ and let r, s be rules. Then, the following relations hold:* (*i*) *$s \leq r$ iff $s \preceq^{\mathcal{C}} r$; and* (*ii*) *if $s \leq r$* (*or, equivalently, $s \preceq^{\mathcal{C}} r$*)*, then $P \equiv_{\mathrm{s}} P \setminus \{r\}$, for each P with $s \in P$.*

Note that not only in case $\mathcal{C}$ is infinite, but also if r is ground, the equivalence between $s \leq r$ and $s \preceq^{\mathcal{C}} r$ holds, for arbitrary s. As shown by Eiter *et al.* [7], given rules r, s, deciding whether $s \leq r$ holds is NP-complete. The proof was carried out by a reduction of the 3-coloring problem to checking containment in $\leq$. Inspecting the proof reveals that NP-hardness already holds if r is restricted to be ground. Again, we can show by a simple example that the equivalence between $\leq$ and $\preceq^{\mathcal{C}}$ does not hold in case $\mathcal{C}$ is finite. Consider, e.g., $s: \ q(X) \leftarrow p(X,X)$, $r: \ q(0) \leftarrow p(1,Y), p(Y,0), \mathrm{not}\ q(1)$, and $\mathcal{C} = \{0,1\}$. Then, $s \not\leq r$, while $s \preceq^{\mathcal{C}} r$. Again, we have the effect that, for each pair s, r of rules, $s \leq r$ implies $s \preceq^{\mathcal{C}} r$, but not vice versa.

Whenever $s \preceq^{\mathcal{C}} r$ holds, we say that r *is subsumed by* s (*in* $\mathcal{C}$). We next show that $s \preceq^{\mathcal{C}} r$ implies that r can be removed from each program containing s also in case $\mathcal{C}$ is finite.

Theorem 6. *If $s \preceq^{\mathcal{C}} r$, for a finite domain $\mathcal{C}$, then $P \equiv_{\mathrm{s}} P \setminus \{r\}$, for each program P with $s \in P$.*

Proof. We first show that $\{r,s\} \equiv_{\mathrm{s}} \{s\}$. To this end, we show that $grd(\{r,s\}, C) \equiv_{\mathrm{s}} grd(\{s\}, C)$, for any $C \subseteq \mathcal{C}$.

Consider any $C \subseteq \mathcal{C}$. By assumption, for every $\vartheta : \mathcal{V}_r \to C$, there exists a substitution $\vartheta_s : \mathcal{V}_s \to \mathcal{C}$ such that $H(s\vartheta_s) \subseteq H(r\vartheta_r) \cup B^-(r\vartheta_r)$ and $B(s\vartheta_s) \subseteq B(r\vartheta_r)$. Note that this implies $\vartheta_s(x) \in C$, for each $x \in \mathcal{V}_s$. We thus have $grd(\{r,s\}, C) = grd(\{s\}, C) \cup \{r\vartheta, s\vartheta_s \mid \vartheta : \mathcal{V}_r \to C\}$, with ϑ_s as above. To every subset $\{r\vartheta, s\vartheta_s\} \subseteq grd(\{r,s\}, C)$, i.e., for every $\vartheta : \mathcal{V}_r \to C$, we can apply Theorem 6 of Lin and Chen [6] and replace it by $\{s\vartheta_s\}$. By construction, the resulting program is strongly equivalent to $grd(\{r,s\}, C)$, and exactly matches $grd(\{s\}, C)$. Thus, $grd(\{r,s\}, C) \equiv_{\mathrm{s}} grd(\{s\}, C)$, for any $C \subseteq \mathcal{C}$.

Assume now that there exists a program P such that $s \in P$ but $P \not\equiv_{\mathrm{s}} P \setminus \{r\}$, i.e., $P \cup Q \not\equiv (P \setminus \{r\}) \cup Q$, for some program Q. For $P' = (P \setminus \{r,s\}) \cup Q$, we thus have $\{r,s\} \cup P' \not\equiv \{s\} \cup P'$, which implies $\{r,s\} \not\equiv_{\mathrm{s}} \{s\}$, a contradiction. Hence, $P \equiv_{\mathrm{s}} P \setminus \{r\}$ must hold for any program P with $s \in P$. $\square$

4.1 Complexity of Rule Subsumption

Concerning complexity, we already mentioned the NP-completeness of checking rule subsumption given an infinite domain. As in the previous section, we observe an increase of complexity when a finite domain is considered. Note that there is a subtle difference between the Π_2^P-hardness result in Theorem 7 below and the hardness results in Section 3: In Theorem 7, the domain $\mathcal{C}$ is considered to be part of the input and therefore $|\mathcal{C}|$ is not bounded by a fixed constant. In contrast, Section 3 provided hardness results even for a fixed domain.

Theorem 7. *Given rules r and s, checking whether r is subsumed by s in a finite domain $\mathcal{C}$ is Π_2^P-complete. Hardness holds even for Horn rules over a bounded number of predicate symbols with bounded arities.*

Proof. For membership, we show that the complementary problem is in Σ_2^P: Guess ϑ_r and check that for each $\vartheta_s : \mathcal{V}_s \to \mathcal{C}$, either $H(s\vartheta_s) \not\subseteq H(r\vartheta_r) \cup B^-(r\vartheta_r)$ or $B(s\vartheta_s) \not\subseteq B(r\vartheta_r)$ holds. The latter check is in coNP, since checking whether there exists some $\vartheta_s : \mathcal{V}_s \to \mathcal{C}$ with $H(s\vartheta_s) \subseteq H(r\vartheta_r) \cup B^-(r\vartheta_r)$ and $B(s\vartheta_s) \subseteq B(r\vartheta_r)$ is clearly in NP.

For showing hardness, we proceed along the lines of Pichler [15]. We reduce the Π_2^P-complete decision problem of ∀∃-QSAT to testing whether $s \preceq^{\mathcal{C}} r$ holds. To this end, let $\Phi = \forall X_1 \ldots \forall X_k \exists X_{k+1} \ldots \exists X_m \phi$ be a QBF with $\phi = \bigwedge_{i=1}^{n} l_{i,1} \vee l_{i,2} \vee l_{i,3}$, and let the domain $\mathcal{C}$ be of size $k+1$, i.e., without loss of generality, assume $\mathcal{C} = \{0, 1, , \ldots, k\}$. We use two rules, r_Φ and s_Φ, which have empty heads and purely positive bodies:

$$\begin{aligned} B^+(r_\Phi) = {} & \{p(1, X_1), \ldots, p(k, X_k)\} \cup \{v(\alpha, 0), v(0, \alpha) \mid \alpha \in \mathcal{C} \setminus \{0\}\} \cup \\ & \{c(\alpha, \beta, \gamma) \mid (\alpha, \beta, \gamma) \in \mathcal{C}^3 \setminus \{(0, 0, 0)\}\}, \\ B^+(s_\Phi) = {} & \{p(1, X_1), \ldots, p(k, X_k)\} \cup \{v(X_j, \bar{X}_j) \mid 1 \le j \le m\} \cup \\ & \{c(l_{i,1}^*, l_{i,2}^*, l_{i,3}^*) \mid 1 \le i \le n\}, \end{aligned}$$

where $l^* = X$ if $l = X$, and $l^* = \bar{X}$ if $l = \neg X$, with $\bar{X}_1, \ldots, \bar{X}_m$ being new atoms. We show that Φ is true iff $s_\Phi \preceq^{\mathcal{C}} r_\Phi$.

For the "only if"-direction, suppose that Φ is true and let ϑ_r be an arbitrary ground substitution of the variables $\{X_1, \ldots, X_k\}$ in r_Φ. We define a truth assignment I for $\{X_1, \ldots, X_k\}$ with $I(X_i)$ = false if $X_i\vartheta_r = 0$ and $I(X_i)$ = true otherwise. By hypothesis, Φ is true. Hence, there exists an extension J of I for $\{X_1, \ldots, X_m\}$ such that ϕ is true in J. From J, we define the ground substitution ϑ_s as follows:

$$X_i\vartheta_s = \begin{cases} 0 & \text{if } X_i \text{ is false in } J; \\ X_i\vartheta_r & \text{if } X_i \text{ is true in } J \text{ and } i \le k; \\ 1 & \text{if } X_i \text{ is true in } J \text{ and } i > k; \end{cases} \qquad \bar{X}\vartheta_s = \begin{cases} 0 \text{ if } X_i \text{ is true in } J\text{ ;} \\ 1 \text{ if } X_i \text{ is false in } J. \end{cases}$$

It remains to show that $B^+(s_\Phi\vartheta_s) \subseteq B^+(r_\Phi\vartheta_r)$ holds for ϑ_s. For every $i \le k$, we have $X_i\vartheta_s = X_i\vartheta_r$ by construction. Hence, every atom $p(i, X_i)\vartheta_s$ in $s_\Phi\vartheta_s$ is contained in $B^+(r_\Phi\vartheta_r)$. Moreover, by construction, for every $j \in \{1, \ldots, m\}$, exactly one of the variables X_j and $\bar{X}_j$ is instantiated to 0 by ϑ_s. Hence, every atom $v(X_j, \bar{X}_j)\vartheta_s$ is either of the form $v(\alpha, 0)$ or $v(0, \alpha)$, for some $\alpha \ne 0$. Thus, every atom $v(X_j, \bar{X}_j)\vartheta_s$ is contained in $B^+(r_\Phi\vartheta_r)$. Finally, ϕ is true in J, i.e., in all clauses of ϕ, at least one literal is true in J. Hence, by construction, for each i, at least one of the first-order variables $l_{i,1}^*, l_{i,2}^*, l_{i,3}^*$ is instantiated to a constant different from 0 by ϑ_s. Thus, all atoms $c(l_{i,1}^*, l_{i,2}^*, l_{i,3}^*)\vartheta_s$ are different from $c(0, 0, 0)$ and are therefore contained in $B^+(r_\Phi)$.

For the "if"-direction, suppose that $s_\Phi \preceq^{\mathcal{C}} r_\Phi$, and let I be an arbitrary truth assignment for $\{X_1, \ldots, X_k\}$. Then, we define the ground substitution ϑ_r over $\{X_1, \ldots, X_k\}$ as $X_i\vartheta_r = 0$ if $I(X_i)$ = false and $X_i\vartheta_r = 1$ if $I(X_i)$ = true. By hypothesis, $s_\Phi \preceq^{\mathcal{C}} r_\Phi$. Thus, there is a substitution ϑ_s over $\{X_1, \ldots, X_m\}$ where $B^+(s_\Phi\vartheta_s) \subseteq B^+(r_\Phi\vartheta_r)$. From ϑ_s we define the extension J of I for $\{X_1, \ldots, X_m\}$ as follows: $J(X_i)$ = false if $X_i\vartheta_s = 0$ and $J(X_i)$ = true if $X_i\vartheta_s \ne 0$.

For every $i \le k$, $X_i\vartheta_s = X_i\vartheta_r$ holds. Hence, J and I coincide on $\{X_1, \ldots, X_k\}$, and therefore J is indeed an extension of I. By assumption, every atom $v(X_j, \bar{X}_j)\vartheta_s$ is either of the form $v(\alpha, 0)$ or $v(0, \alpha)$, for some $\alpha \ne 0$. Thus, by the definition of J, we also have that $J(\bar{X}_i)$ = false if $\bar{X}_i\vartheta_s = 0$ and $J(\bar{X}_i)$ = true if $\bar{X}_i\vartheta_s \ne 0$. Finally, all atoms $c(l_{i,1}^*, l_{i,2}^*, l_{i,3}^*)\vartheta_s$ are contained in $B^+(r_\Phi)$ and are therefore different from $c(0, 0, 0)$, i.e., for each i, at least one of the first-order variables $l_{i,1}^*, l_{i,2}^*, l_{i,3}^*$ is instantiated to a constant different from 0 by ϑ_s. But then, in all clauses $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ of ϕ, at least one literal is true in J. Thus, ϕ is true in J. This holds for arbitrary I, consequently Φ is true. □

Note that in the proof of Theorem 7, the domain $\mathcal{C}$ is part of the input. This is in stark contrast to Theorems 2 and 3, were the domain $\mathcal{C}$ is arbitrary but fixed—thus leading to slightly stronger hardness results. However, in the Π_2^P-hardness proof of Theorem 7, it is crucial that there is no fixed bound on the size of the domain. In particular, we indeed need k pairwise distinct domain elements (where k corresponds to the number of universally quantified propositional variables in Φ) in order to make sure that any instantiation of a first-order variable X_i in r_Φ forces the same instantiation of the variable X_i (with precisely the same index i) in s_Φ.

Alternatively, we could have considered the domain $\mathcal{C}$ to be fixed (with $|\mathcal{C}| \geq 2$) and either let the number of predicate symbols or the arity of the predicate symbols be unbounded. In case of an unbounded number of predicate symbols, we can simply replace the atoms $p(1, X_1), \ldots, p(k, X_k)$ in both r_Φ and s_Φ by atoms of the form $p_1(X_1), \ldots, p_k(X_k)$. Likewise, if the domain is fixed and the arities of predicate symbols are unbounded, then we replace the atoms $p(1, X_1), \ldots, p(k, X_k)$ in both r_Φ and s_Φ by a single atom $p(X_1, \ldots, X_k)$.

However, if the domain $\mathcal{C}$ is fixed (or at least its cardinality is bounded) and, moreover, both the number of predicate symbols and their arity is bounded, then Π_2^P-completeness no longer holds unless the polynomial hierarchy collapses.

Theorem 8. *Given rules r and s, with $ar_{\max}(\mathcal{A}_{\{r,s\}}) \leq k$ and $|\mathcal{A}_{\{r,s\}}| \leq k'$ for fixed constants k, k', checking whether r is subsumed by s in a domain of fixed size is in Δ_2^P.*

Proof. Since the cardinality of $\mathcal{C}$, the number of predicate symbols, and the arity of predicate symbols are all bounded, there is only a constant number, K, of different ground rules in this language. Note that $s \not\preceq^{\mathcal{C}} r$ iff there exists a ground instance $r\vartheta_r$ of r such that, for every ground substitution $\vartheta_s : \mathcal{V}_s \rightarrow \mathcal{C}$, either $H(s\vartheta_s) \not\subseteq H(r\vartheta_r) \cup B^-(r\vartheta_r)$ or $B(s\vartheta_s) \not\subseteq B(r\vartheta_r)$.

In order to check whether such a ground instance $r\vartheta_r$ of r exists, we loop over all K ground rules t and check by one NP-oracle call that t is a ground instance of r and by another NP-oracle call that $s \not\preceq^{\mathcal{C}} t$. Note that both checks are indeed feasible by NP-oracles: On the one hand, checking whether t is a ground instance of r amounts to guessing a ground substitution ϑ_r and checking that $r\vartheta_r = t$ holds. On the other hand, checking whether $s \preceq^{\mathcal{C}} t$ amounts to guessing a ground substitution ϑ_s and checking that both $H(s\vartheta_s) \subseteq H(t) \cup B^-(t)$ and $B(s\vartheta_s) \subseteq B(t)$ hold. □

4.2 Restricting Variable Occurrences and Tractability

As in Section 3, we conclude our discussion by considering restrictions on the variables occurring in the rules. Since, for subsumption, we are dealing with two rules, we distinguish those cases where variable occurrences are restricted in either one of the rules, or in both. It turns out that just a restriction of variable occurrences in both rules guarantees tractability of subsumption detection.

Theorem 9. *Given rules r, s, checking whether r is subsumed by s in a domain of fixed size ≥ 2 is (a) NP-hard, if $|\mathcal{V}_r|$ is bounded by a fixed constant, and even if r is ground and purely positive, and (b) coNP-hard, if $|\mathcal{V}_s|$ is bounded by a fixed constant, and even if s consists of a single body atom.*

Proof. (a) Even when r is ground and purely positive, the problem of subsumption detection in answer-set programming corresponds to "normal" first-order subsumption of clauses, which is a well known NP-hard problem (cf. Problem [LO18] in Garey and Johnson [12]).

(b) Let $|\mathcal{C}| = k$. Without loss of generality, assume $\mathcal{C} = \{1, \ldots, k\}$. We first suppose that $k \geq 3$. In this case, we reduce the k-colorability problem to the complementary problem of rule subsumption. Let an instance of the k-colorability problem be given by the graph $G = (V, E)$, where V denotes the vertices and E the edges. We construct two rules, r_G and s_G, as follows:

$$B^+(r_G) = \{e(X_i, X_j) \mid \{v_i, v_j\} \in E\};$$
$$B^+(s_G) = \{e(X, X)\}.$$

We claim that G is k-colorable iff $s_G \not\preceq^{\mathcal{C}} r_G$.

For the 'if"-direction, suppose that G is k-colorable, i.e., there exists a coloring $f : V \to \{1, \ldots, k\}$ such that no two adjacent vertices are assigned with the same color. Now define the ground substitution $\vartheta_r : \mathcal{V} \to \mathcal{C}$ as $X_i\vartheta_r = f(v_i)$. Then, by construction, $B^+(r_G\vartheta_r)$ does not contain an atom $e(X_i, X_j)\vartheta$ with $X_i\vartheta = X_j\vartheta$, since otherwise also $f(v_i) = f(v_j)$ for some edge $\{v_i, v_j\}$ of the graph G. But this is impossible for a valid k-coloring f.

For the "only if"-direction, suppose that $s_G \not\preceq^{\mathcal{C}} r_G$. Hence, there exists a ground substitution $\vartheta : \mathcal{V} \to \mathcal{C}$ such that $B^+(r_G\vartheta_r)$ does not contain an atom $e(X_i, X_j)\vartheta$ with $X_i\vartheta = X_j\vartheta$. But then we can clearly define a valid k-coloring of the graph G as $f : V \to \{1, \ldots, k\}$ such that $f(v_i) = X_i\vartheta_r$.

It remains to consider the case where $|\mathcal{C}| = 2$. Without loss of generality, assume $\mathcal{C} = \{0, 1\}$. In this case, we establish coNP-hardness by a reduction of the 4-colorability problem to the complementary problem of rule subsumption. Let an instance of the 4-colorability problem be given by the graph $G = (V, E)$. We construct the rules r_G and s_G as follows:

$$B^+(r_G) = \{e(X_i, Y_i, X_j, Y_j) \mid \{v_i, v_j\} \in E\};$$
$$B^+(s_G) = \{e(X, Y, X, Y)\}.$$

We claim that G is 4-colorable iff $s_G \not\preceq^{\mathcal{C}} r_G$. The proof is essentially as in the case $k \geq 3$ above. However, now pairs of variables (X, Y) are considered as a binary encoding of the four colors in the graph. □

Theorem 10. *Given rules r, s such that $|\mathcal{V}_r \cup \mathcal{V}_s|$ is bounded by a fixed constant, checking whether r is subsumed by s in a finite domain is in* P.

Proof. Since the number of variables in r and s is bounded by a constant and the number of domain elements is finite, there are only polynomially many ground instances $r\vartheta_r$ and $s\vartheta_s$, respectively. Hence, we may test in a loop over all ground instances $r\vartheta_r$ if there exists an instance $s\vartheta_s$ such that $H(s\vartheta_s) \subseteq H(r\vartheta_r) \cup B^-(r\vartheta_r)$ and $B(s\vartheta_s) \subseteq B(r\vartheta_r)$ hold. The latter test requires simply a nested loop over polynomially many ground instances $s\vartheta_s$. □

Table 1. Complexity of detecting tautological rules in finite (possibly fixed) domains

	general case	$ar_{\max}(\mathcal{A}_r) \leq k$	$\lvert\mathcal{V}_r\rvert \leq k$
r disjunctive	coNP-complete	coNP-complete	in P
r positive	coNP-complete	coNP-complete	in P
r Horn	coNP-complete	in P	in P

Table 2. Complexity of detecting rule subsumption $s \preceq^{\mathcal{C}} r$ in fixed finite domains

	general case	$ar_{\max}(\mathcal{A}_{\{r,s\}}) \leq k$	$\lvert\mathcal{V}_r\rvert \leq k$
general case	Π_2^P-complete	Π_2^P-complete	NP-hard
$\lvert\mathcal{A}_{\{r,s\}}\rvert \leq k$	Π_2^P-complete	in Δ_2^P	NP-hard
$\lvert\mathcal{V}_s\rvert \leq k$	coNP-hard	coNP-hard	in P

5 Discussion and Conclusion

We investigated the complexity of applying rule eliminations in the setting of finite domains, and provided a full complexity picture with respect to several restrictions, in particular restricting the syntax to Horn rules, imposing a bound on predicate arities and/or on the number of variables (a summary of our results is given in Tables 1 and 2). Note that the concept of bounded predicate arities was suggested by Eiter *et al.* [16] in order to reduce the complexity of basic reasoning tasks in answer-set programming from nondeterministic exponential time classes to classes from the polynomial hierarchy. Similarly, Vardi [17] used bounded variables in order to narrow the gap between expression and data complexity of database queries (i.e., Horn programs).

The main observation of our results is that if we consider finite domains then the detection of tautological or subsumed rules becomes, in general, harder. More specifically, we observed an increase from P to coNP as well as from NP to Π_2^P. However, we also identified restrictions such that complexity does not increase. To wit, a restriction to Horn clauses makes the detection of tautological rules tractable, but only if we additionally impose a bound on the arities of predicate symbols (cf. the first two columns of Table 1).

As for the detection of subsumed rules, restricting to Horn clauses is irrelevant since all hardness results in Section 4 were shown for Horn clauses. On the other hand, Table 2 reflects the effects of other restrictions: In the second row, the case of fixing the number of predicate symbols by some constant is considered. This restriction leads to a decrease of complexity if it is combined with a bound on the arities of predicate symbols. However, in order to obtain tractability, more severe restrictions are required. For instance, a restriction on the number of variable occurrences in both r and s is a sufficient condition for the tractability of detecting subsumed rules (cf. the third row, last column, of Table 2).

We finally remark, however, that local checks for rule redundancy, as presented here, may pay off in program simplification since the complexity of checking rule redundancy (which amounts to testing strong equivalence) is in general much harder, viz. complete for co-NEXPTIME.

References

1. 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 Proc. AAAI'05, AAAI Press (2005) 695–700
2. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic **2(4)** (2001) 526–541
3. Brass, S., Dix, J.: Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. Journal of Logic Programming **38(3)** (1999) 167–213
4. Osorio, M., Navarro, J.A., Arrazola, J.: Equivalence in Answer Set Programming. In Proc. LOPSTR'01. Volume 2372 of LNCS, Springer-Verlag (2001) 57–75
5. Inoue, K., Sakama, C.: Equivalence of Logic Programs Under Updates. In: Proc. JELIA'04. Volume 3229 of LNCS, Springer-Verlag (2004) 174–186
6. Lin, F., Chen, Y.: Discovering Classes of Strongly Equivalent Logic Programs. In Proc. IJCAI'05, (2005) 516–521
7. Eiter, T., Fink, M., Tompits, H., Traxler, P., Woltran, S.: Replacements in Non-Ground Answer-Set Programming. In Proc. KR'06, AAAI Press (2006) 340–351
8. Lassez, J.L., Marriott, K.: Explicit Representation of Terms Defined by Counter Examples. Journal of Automated Reasoning **3(3)** (1987) 301–317
9. Kunen, K.: Answer Sets and Negation as Failure. In Proc. ICLP'87, MIT Press (1987) 219–228
10. Kapur, D., Narendran, P., Rosenkrantz, D., Zhang, H.: Sufficient-Completeness, Ground-Reducibility and their Complexity. Acta Informatica **28(4)** (1991) 311–350
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
12. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman (1979)
13. Wang, K., Zhou, L.: Comparisons and Computation of Well-founded Semantics for Disjunctive Logic Programs. ACM Transactions on Computational Logic **6(2)** (2005) 295–327
14. Traxler, P.: Techniques for Simplifying Disjunctive Datalog Programs with Negation. Master's thesis, Technische Universität Wien, Institut für Informationssysteme (2006)
15. Pichler, R.: On the Complexity of H-Subsumption. In Proc. CSL'98. Volume 1584 of LNCS, Springer-Verlag (1998) 355–371
16. Eiter, T., Faber, W., Fink, M., Pfeifer, G., Woltran, S.: Complexity of Answer Set Checking and Bounded Predicate Arities for Non-ground Answer Set Programming. In Proc. KR'04, AAAI Press (2004) 377–387
17. Vardi, M.: On the Complexity of Bounded-Variable Queries. In Proc. PODS'95, (1995) 266–276

Conflict-Driven Answer Set Enumeration

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub*

Institut für Informatik,
Universität Potsdam,
August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We elaborate upon a recently proposed approach to finding an answer set of a logic program based on concepts from constraint processing and satisfiability checking. We extend this approach and propose a new algorithm for enumerating answer sets. The algorithm, which to our knowledge is novel even in the context of satisfiability checking, is implemented in the *clasp* answer set solver. We contrast our new approach to alternative systems and different options of *clasp*, and provide an empirical evaluation.

1 Introduction

Answer set programming (ASP; [1]) has become a primary tool for declarative problem solving. Although the corresponding solvers are highly optimized (cf. [2,3]), their performance does not match the one of state-of-the-art solvers for satisfiability checking (SAT; [4]). While SAT-based ASP solvers like *assat* [5] and *cmodels* [6] exploit SAT solvers, the underlying techniques are not yet established in genuine ASP solvers. We addressed this deficiency in [7] by introducing a new computational approach to ASP solving, centered around the constraint processing (CSP; [8]) concept of a *nogood*. Apart from the fact that this allows us to easily integrate solving technology from the areas of CSP and SAT, it also provided us with a uniform representation of inferences from logic program rules, unfounded sets, as well as nogoods learned from conflicts.

While we have detailed in [7] how a single answer set is obtained, we introduce in what follows an algorithm for enumerating answer sets. In contrast to systematic backtracking approaches, the passage from computing a single to multiple solutions is non-trivial in the context of backjumping and clause learning. A popular approach consists in recording a found solution as a nogood and exempting it from nogood deletion. However, such an approach is prone to blow up in space in view of the exponential number of solutions in the worst case. Unlike this, our algorithm runs in polynomial space and is (to the best of our knowledge) even a novelty in the context of SAT.

After establishing the formal background, we describe in Section 3 the constraint-based specification of ASP solving introduced in [7]. Based on this uniform representation, we develop in Section 4 algorithms for answer set enumeration, relying on conflict-driven learning and backjumping. In Section 5, we provide a systematic empirical evaluation of different approaches to answer set enumeration, examining different systems as well as different options within our conflict-driven answer set solver *clasp*.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 Background

Given an alphabet $\mathcal{P}$, a (normal) *logic program* is a finite set of rules of the form $p_0 \leftarrow p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n$ where $0 \leq m \leq n$ and $p_i \in \mathcal{P}$ is an *atom* for $0 \leq i \leq n$. A *body literal* is an atom p or its negation $\mathit{not}\ p$. For a rule r, let $\mathit{head}(r) = p_0$ be the *head* of r and $\mathit{body}(r) = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}$ be the *body* of r. The set of atoms occurring in a logic program Π is denoted by $\mathit{atom}(\Pi)$. The set of bodies in Π is $\mathit{body}(\Pi) = \{\mathit{body}(r) \mid r \in \Pi\}$. For regrouping rule bodies sharing the same head p, define $\mathit{body}(p) = \{\mathit{body}(r) \mid r \in \Pi, \mathit{head}(r) = p\}$. In ASP, the semantics of a program Π is given by its *answer sets*. For a formal introduction to ASP, we refer the reader to [1].

We consider Boolean *assignments*, A, over the *domain* $\mathit{dom}(A) = \mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$. Formally, an assignment A is a sequence $(\sigma_1, \dots, \sigma_n)$ of *signed literals* σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in \mathit{dom}(A)$ and $1 \leq i \leq n$; $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*. (We omit the attribute *signed* for literals whenever clear from the context.) We denote the complement of a literal σ by $\overline{\sigma}$, that is, $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$. We let $A \circ B$ denote the sequence obtained by concatenating assignments A and B. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access true and false members of A via $A^{\mathbf{T}} = \{p \in \mathit{dom}(A) \mid \mathbf{T}p \in A\}$ and $A^{\mathbf{F}} = \{p \in \mathit{dom}(A) \mid \mathbf{F}p \in A\}$.

A *nogood* is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment that contains $\sigma_1, \dots, \sigma_n$. An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \mathit{dom}(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a *solution* for a set Δ of nogoods if $\delta \not\subseteq A$ for all $\delta \in \Delta$. For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is *unit-resulting* for δ wrt A if (1) $\delta \setminus A = \{\sigma\}$ and (2) $\overline{\sigma} \notin A$. By (1), σ is the single literal from δ that is not contained in A. This implies that a violated constraint does not have a unit-resulting literal. Condition (2) makes sure that no duplicates are introduced: If A already contains $\overline{\sigma}$, then it is no longer unit-resulting. For instance, literal $\mathbf{F}q$ is unit-resulting for nogood $\{\mathbf{F}p, \mathbf{T}q\}$ wrt assignment $(\mathbf{F}p)$, but neither wrt $(\mathbf{F}p, \mathbf{F}q)$ nor wrt $(\mathbf{F}p, \mathbf{T}q)$. Note that our notion of a unit-resulting literal is closely related to the *unit clause rule* of DPLL (cf. [4]). For a set Δ of nogoods and an assignment A, we call *unit propagation* the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

3 Nogoods of Logic Programs

Our approach is guided by the idea of Lin and Zhao [5] and decomposes ASP solving into (local) inferences obtainable from the *Clark completion* of a program [9] and those obtainable from loop formulas.

We begin with nogoods capturing inferences from the Clark completion of a program Π. The latter can be defined as follows:

$$\begin{aligned}
&\{p_\beta \equiv p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n \mid \\
&\qquad\qquad \beta \in \mathit{body}(\Pi), \beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}\} && (1)\\
&\cup \{p \equiv p_{\beta_1} \vee \dots \vee p_{\beta_k} \mid p \in \mathit{atom}(\Pi), \mathit{body}(p) = \{\beta_1, \dots, \beta_k\}\}\ . && (2)
\end{aligned}$$

This formulation relies on auxiliary atoms representing bodies; this avoids an exponential blow-up of the corresponding set of clauses. The first type of equivalences in (1) takes care of bodies, while the second one in (2) deals with atoms.

For obtaining the underlying set of constraints, we begin with the body-oriented equivalence in (1). Consider a body $\beta \in \mathit{body}(\Pi)$. The equivalence in (1) can be decomposed into two implications. First, we get $p_\beta \rightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$, which is equivalent to the conjunction of $\neg p_\beta \vee p_1, \ldots, \neg p_\beta \vee p_m, \neg p_\beta \vee \neg p_{m+1}, \ldots,$ $\neg p_\beta \vee \neg p_n$. These clauses express the following set of nogoods:

$$\Delta(\beta) = \{ \{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\} \} .$$

As an example, consider the body $\{x, \mathit{not}\ y\}$. We obtain the nogoods $\Delta(\{x, \mathit{not}\ y\}) = \{ \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T}y\} \}$. Similarly, the converse of the previous implication, viz. $p_\beta \leftarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$, gives rise to the nogood

$$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} .$$

Intuitively, $\delta(\beta)$ forces the truth of β or the falsity of a body literal in β. For instance, for body $\{x, \mathit{not}\ y\}$, we get the nogood $\delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\}$.

Proceeding analogously with the atom-based equivalences in (2), we obtain for an atom $p \in \mathit{atom}(\Pi)$ along with its bodies $\mathit{body}(p) = \{\beta_1, \ldots, \beta_k\}$ the nogoods

$$\Delta(p) = \{ \{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\} \} \quad \text{and} \quad \delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} .$$

For example, for an atom x with $\mathit{body}(x) = \{\{y\}, \{\mathit{not}\ z\}\}$, we get the nogoods $\Delta(x) = \{ \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\mathit{not}\ z\}\} \}$ and $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\mathit{not}\ z\}\}$.

Combining the four types of nogoods leads us to the following set of nogoods:

$$\begin{aligned} \Delta_\Pi = \quad & \{\delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \mathit{body}(\Pi)\} \\ & \cup \{\delta(p) \mid p \in \mathit{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \mathit{atom}(\Pi)\} . \end{aligned} \tag{3}$$

The nogoods in Δ_Π capture the *supported models* of a program [10]. Any answer set is a supported model, but the converse only holds for *tight* programs [11]. The mismatch on non-tight programs is caused by *loops* [5], responsible for cyclic support among true atoms. Such cyclic support can be prohibited by loop formulas. As shown in [12], the answer sets of a program Π are precisely the models of Π that satisfy the loop formulas of all non-empty subsets of $\mathit{atom}(\Pi)$.

For a program Π and some $U \subseteq \mathit{atom}(\Pi)$, we define the *external bodies* of U for Π as $\mathit{EB}_\Pi(U) = \{\mathit{body}(r) \mid r \in \Pi, \mathit{head}(r) \in U, \mathit{body}(r) \cap U = \emptyset\}$. The (disjunctive) *loop formula* of U for Π is

$$\neg\big(\bigvee\nolimits_{\beta \in \mathit{EB}_\Pi(U)} \big(\bigwedge\nolimits_{p \in \beta^+} p \wedge \bigwedge\nolimits_{p \in \beta^-} \neg p\big)\big) \rightarrow \neg\big(\bigvee\nolimits_{p \in U} p\big)$$

where $\beta^+ = \beta \cap \mathit{atom}(\Pi)$ and $\beta^- = \{p \mid \mathit{not}\ p \in \beta\}$. The loop formula of a set U of atoms forces all elements of U to be false if U is not *externally supported* [12]. To capture the effect of a loop formula induced by a set $U \subseteq \mathit{atom}(\Pi)$ such that $\mathit{EB}_\Pi(U) = \{\beta_1, \ldots, \beta_k\}$, we define the *loop nogood* of an atom $p \in U$ as

$$\lambda(p, U) = \{\mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k, \mathbf{T}p\} .$$

Overall, we get the following set of loop nogoods for a program Π:

$$\Lambda_\Pi = \bigcup_{U \subseteq atom(\Pi), U \neq \emptyset} \{\lambda(p, U) \mid p \in U\} \ . \tag{4}$$

As shown in [7], completion and loop nogoods allow for characterizing answer sets.

Theorem 1 ([7]). *Let Π be a logic program, let Δ_Π and Λ_Π as given in (3) and (4). Then, a set X of atoms is an answer set of Π iff $X = A^{\mathbf{T}} \cap atom(\Pi)$ for a (unique) solution A for $\Delta_\Pi \cup \Lambda_\Pi$.*

The nogoods in $\Delta_\Pi \cup \Lambda_\Pi$ describe a set of constraints that must principally be checked for computing answer sets. While the size of Δ_Π is linear in $atom(\Pi) \times body(\Pi)$, the one of Λ_Π is exponential. Thus, answer set solvers use dedicated algorithms that explicate loop nogoods in Λ_Π only on demand, either for propagation or model verification.

4 Answer Set Enumeration

We presented in [7] an algorithm for computing one answer set that is based upon *Conflict-Driven Clause Learning* (CDCL; [4]). In what follows, we combine ideas from the *First-UIP scheme* of CDCL and *Conflict-directed BackJumping* (CBJ; [13]) with particular propagation mechanisms for ASP in order to obtain an algorithm for enumerating a desired number of answer sets (if they exist). Our major objective is to use First-UIP learning and backjumping in the enumeration of solutions, while avoiding repeated solutions and the addition of (non-removable) nogoods to the original problem.

In fact, First-UIP backjumping constitutes a "radical" strategy to recover from conflicts: It jumps directly to the point where a conflict-driven assertion takes effect, undoing all portions of the search space in between. The undone part of the search space is not necessarily exhausted, and some portions of it can be reconstructed in the future. On the one hand, the possibility to revisit parts of the search space makes the termination of CDCL less obvious than it is for other search procedures. (For a proof of termination, see for instance [14].) On the other hand, avoiding repetitions in the enumeration of solutions becomes non-trivial: When a solution has been found and a conflict occurs after flipping the value of some variable(s) in it, then a conflict-driven assertion might reestablish a literal from the already enumerated solution, and after backjumping, the same solution might be feasible again. This is avoided in CDCL solvers by recording "pseudo" nogoods for prohibiting already enumerated solutions. Such a nogood must not be removed, which is different from conflict nogoods that can be deleted once they are obsolete. Of course, an enumeration strategy that records nogoods for prohibiting solutions runs into trouble if there are numerous solutions, in which case the solver blows up in space.

Unlike First-UIP backjumping, CBJ, which has been designed for CSP and is also used in the SAT solver *relsat* [15], makes sure that backjumping only undoes exhausted search spaces. In particular, if there is a solution, then an unflipped decision literal of

Algorithm 1. NOGOODPROPAGATION

```
Input  : A program $\Pi$, a set $\nabla$ of nogoods, and an assignment $A$.
Output : An extended assignment and set of nogoods.

1  $U \leftarrow \emptyset$                                   // set of unfounded atoms
2  loop
3      while $\varepsilon \not\subseteq A$ for all $\varepsilon \in \Delta_\Pi \cup \nabla$ and
4      there is some $\delta \in \Delta_\Pi \cup \nabla$ st $\delta \setminus A = \{\sigma\}$ and $\overline{\sigma} \notin A$ do
5          $A \leftarrow A \circ (\overline{\sigma})$
6          $dlevel(\overline{\sigma}) \leftarrow max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
7      if $\varepsilon \subseteq A$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ or TIGHT($\Pi$) then
8          return $(A, \nabla)$
9      else
10         $U \leftarrow U \setminus A^{\mathbf{F}}$
11         if $U = \emptyset$ then $U \leftarrow$ UNFOUNDEDSET($\Pi, A$)
12         if $U = \emptyset$ then return $(A, \nabla)$
13         else let $p \in U$ in
14             $\nabla \leftarrow \nabla \cup \{\lambda(p, U)\}$
15             if $\mathbf{T}p \in A$ then return $(A, \nabla)$
16             else
17                 $A \leftarrow A \circ (\mathbf{F}p)$
18                 $dlevel(\mathbf{F}p) \leftarrow max(\{dlevel(\rho) \mid \rho \in \lambda(p, U) \setminus \{\mathbf{T}p\}\} \cup \{0\})$
```

that solution cannot be jumped over, as no nogood excludes the search space below it. Only the fact that all solutions containing a certain set of decision literals have been enumerated justifies retracting one of them. This is reflected by CBJ, where a decision literal can only be retracted if the search space below it is exhausted.

Our strategy to enumerate solutions combines First-UIP learning and backjumping with CBJ. As long as no solution has been found, we apply the First-UIP scheme as usual (cf. [7]). Once we have found a solution, its decision literals must be backtracked chronologically. That is, we cannot jump over any unflipped decision literal contributing to a solution. (Other decision literals are treated as usual.) Only if a search space is exhausted, we flip the value of the last decision literal contained in a solution. Note that the First-UIP scheme can be applied even if some decision literals belong to a solution as long as only other decision literals are jumped over.

Algorithm 1 refines the propagation algorithm introduced in [7]. The major change is given in ll. 3–6: For every unit-resulting literal $\overline{\sigma}$ that is added to A, the value of $dlevel(\overline{\sigma})$ is explicated. Instead of the current decision level, we assign the greatest value $dlevel(\rho)$ of any literal $\rho \in \delta \setminus \{\sigma\}$. So $dlevel(\overline{\sigma})$ is the smallest decision level such that $\overline{\sigma}$ is unit-resulting for δ wrt A. In Line 18, $dlevel(\mathbf{F}p)$ is determined in the same way for $\lambda(p, U)$. See [7] for details on the unchanged parts of Algorithm 1.

Algorithm 2 implements our approach to enumerating a given number of answer sets. Its key element is the chronological backtracking level bl. At any state of the computation, its value holds the greatest decision level such that (1) the corresponding

Algorithm 2. CDNL-ENUM-ASP

```
Input : A program $\Pi$ and a number $s$ of solutions to enumerate.
$A \leftarrow \emptyset$                                      // assignment over $atom(\Pi) \cup body(\Pi)$
$\nabla \leftarrow \emptyset$                                      // set of (dynamic) nogoods
$dl \leftarrow 0$                                     // decision level
$bl \leftarrow 0$                                     // (systematic) backtracking level
loop
    $(A, \nabla) \leftarrow$ NogoodPropagation($\Pi, \nabla, A$)
    if $\varepsilon \subseteq A$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ then
        if $dl = 0$ then exit
        else if $bl < dl$ then
            $(\delta, \sigma_{UIP}, k) \leftarrow$ ConflictAnalysis($\varepsilon, \Pi, \nabla, A$)
            $\nabla \leftarrow \nabla \cup \{\delta\}$
            $dl \leftarrow max(\{k, bl\})$
            $A \leftarrow A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$
            $A \leftarrow A \circ (\overline{\sigma_{UIP}})$
            $dlevel(\overline{\sigma_{UIP}}) \leftarrow k$
        else
            $\sigma_d \leftarrow dliteral(dl)$
            $dl \leftarrow dl - 1$
            $bl \leftarrow dl$
            $A \leftarrow A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$
            $A \leftarrow A \circ (\overline{\sigma_d})$
            $dlevel(\overline{\sigma_d}) \leftarrow dl$
    else if $A^{\mathbf{T}} \cup A^{\mathbf{F}} = atom(\Pi) \cup body(\Pi)$ then
        print $A^{\mathbf{T}} \cap atom(\Pi)$
        $s \leftarrow s - 1$
        if $s = 0$ or $dl = 0$ then exit
        else
            $\sigma_d \leftarrow dliteral(dl)$
            $dl \leftarrow dl - 1$
            $bl \leftarrow dl$
            $A \leftarrow A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$
            $A \leftarrow A \circ (\overline{\sigma_d})$
            $dlevel(\overline{\sigma_d}) \leftarrow dl$
    else
        $\sigma_d \leftarrow$ Select($\Pi, \nabla, A$)
        $dl \leftarrow dl + 1$
        $A \leftarrow A \circ (\sigma_d)$
        $dlevel(\sigma_d) \leftarrow dl$
        $dliteral(dl) \leftarrow \sigma_d$
```

decision literal has not (yet) been flipped and (2) some enumerated solution contains all decision literals up to decision level bl. To guarantee that no solution is repeated, we have to make sure that backjumping does not retract decision level bl without flipping a

Algorithm 3. ConflictAnalysis

```
Input  : A violated nogood $\delta$, a program $\Pi$, a set $\nabla$ of nogoods, and an assignment $A$.
Output : A derived nogood, a UIP, and a decision level.
1 let $\sigma \in \delta$ st $A = B \circ (\sigma) \circ B'$ and $\delta \setminus B = \{\sigma\}$
2 while $\{\rho \in \delta \mid dlevel(\rho) = dlevel(\sigma)\} \neq \{\sigma\}$ do
3     let $\varepsilon \in \Delta_\Pi \cup \nabla$ st $\overline{\sigma} \in \varepsilon$ and $\varepsilon \setminus B = \{\overline{\sigma}\}$
4     $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$
5     let $\sigma \in \delta$ st $B = C \circ (\sigma) \circ C'$ and $\delta \setminus C = \{\sigma\}$
6     $B \leftarrow C$
7 $k \leftarrow max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
8 return $(\delta, \sigma, k)$
```

decision literal whose decision level is smaller than or equal to bl.[1] We exclude such a situation in Algorithm 2 by denying backjumps "beyond" decision level bl, if a conflict is encountered at a decision level $dl > bl$, or by enforcing the flipping of the decision literal of decision level bl, if a conflict (or a solution) is encountered at decision level bl. The latter means that the search space below decision level bl is exhausted, that is, all its solutions have been enumerated, so that the decision literal of decision level bl needs to be flipped for enumerating any further solutions.

As in Algorithm 1, we explicitly assign $dlevel(\sigma)$ whenever some literal σ is added to assignment A in Algorithm 2. Also, we set $dliteral(dl)$ to σ_d in Line 39 when decision literal σ_d is added to A at decision level dl. In this way, no confusion about the decision level of a literal or the decision literal of a decision level is possible.[2]

Conflict analysis in Algorithm 3 follows the approach in [7]; it assumes that there is a *Unique Implication Point* (UIP) at the decision level where the conflict has been encountered. This is always the case: A look at ll. 8–22 in Algorithm 2 reveals that the conflict to be analyzed is a consequence of the last decision, and not caused by flipping a decision literal in order to enumerate more solutions. (Note that flipping a decision literal does not produce a new decision level, hence, we have $bl = dl$ if a deliberate flipping causes a conflict. In such a case, we do not analyze the respective conflict.)

We illustrate answer set enumeration by CDNL-ENUM-ASP on the schematic example in Figure 1. Thereby, we denote by σ_d^i the ith decision literal picked by Select in Line 35 of Algorithm 2. We denote by σ_a^i the complement of a UIP, asserted in Line 14 of Algorithm 2, after decision literal σ_d^i led to a conflict. For a literal σ, we write $\sigma[n]$ to indicate the decision level of σ, that is, $dlevel(\sigma) = n$. Note that, in Figure 1, we represent assignments only by their decision and asserted literals, respectively, and omit any literals derived by NogoodPropagation. We underline the decision literal of the chronological backtracking level bl. If an assignment contains such a literal,

[1] A backjump without flipping could happen if we would exclusively use the First-UIP scheme. An assertion at a decision level $dl<bl$ would be such that the complement of the corresponding UIP has been present in a solution enumerated before. Hence, reassigning all decision literals between dl (exclusive) and bl (inclusive) would lead to an already enumerated solution.

[2] We assume that $\sigma_d \notin A$ and $\overline{\sigma_d} \notin A$ for any decision literal σ_d returned by Select(Π, ∇, A) in Line 35 of Algorithm 2.

$$\begin{array}{ll}
A_1 = (\sigma_d^1[1], \sigma_d^2[2], \sigma_d^3[3], \sigma_d^4[4], \sigma_d^5[5]) & \text{conflict at } dl = 5 \\
A_2 = (\sigma_d^1[1], \sigma_d^2[2], \sigma_d^3[3], \sigma_a^5[3]) & \text{assertion at } dl = 3 \\[1ex]
A_3 = (\sigma_d^1[1], \sigma_d^2[2], \sigma_d^3[3], \sigma_a^5[3], \sigma_d^6[4]) & \text{solution at } dl = 4 \\
A_4 = (\sigma_d^1[1], \sigma_d^2[2], \underline{\sigma_d^3[3]}, \sigma_a^5[3], \overline{\sigma_d^6}[3]) & \text{backtracking to } bl = 3 \\[1ex]
A_5 = (\sigma_d^1[1], \sigma_d^2[2], \underline{\sigma_d^3[3]}, \sigma_a^5[3], \overline{\sigma_d^6}[3], \sigma_d^7[4], \sigma_d^8[5]) & \text{conflict at } dl = 5 \\
A_6 = (\sigma_d^1[1], \sigma_a^8[1], \sigma_d^2[2], \underline{\sigma_d^3[3]}, \sigma_a^5[3], \overline{\sigma_d^6}[3]) & \text{assertion at } dl = 1 \\
 & \text{backtracking to } bl = 3 \\[1ex]
A_7 = (\sigma_d^1[1], \sigma_a^8[1], \sigma_d^2[2], \underline{\sigma_d^3[3]}, \sigma_a^5[3], \overline{\sigma_d^6}[3], \sigma_d^9[4]) & \text{solution at } dl = 4 \\
A_8 = (\sigma_d^1[1], \sigma_a^8[1], \sigma_d^2[2], \underline{\sigma_d^3[3]}, \sigma_a^5[3], \overline{\sigma_d^6}[3], \overline{\sigma_d^9}[3]) & \text{backtracking to } bl = 3 \\
 & \text{solution/conflict at } dl = 3 = bl \\[1ex]
A_9 = (\sigma_d^1[1], \sigma_a^8[1], \underline{\sigma_d^2[2]}, \overline{\sigma_d^3}[2]) & \text{backtracking to } bl = 2 \ \ldots
\end{array}$$

Fig. 1. Answer set enumeration example

it must not be retracted unless the search space below it is exhausted, that is, unless a conflict or a(nother) solution is encountered at decision level bl.

Consider assignment A_1 in Figure 1, and assume that NOGOODPROPAGATION yields a violated nogood after decision literal σ_d^5 has been selected at decision level $dl = 5$. Let CONFLICTANALYSIS return a nogood δ such that $\overline{\sigma_a^5} \in \delta$ and $max(\{dlevel(\sigma) \mid \sigma \in \delta \setminus \{\overline{\sigma_a^5}\}\} \cup \{0\}) = 3$, that is, $\overline{\sigma_a^5}$ is a UIP. Given that no solution has been found yet, we have $bl = 0$. Thus, CDNL-ENUM-ASP jumps back to decision level 3 and asserts σ_a^5, yielding assignment A_2. Up to this point, the enumeration of solutions is similar to the search for a single solution. We next select decision literal σ_d^6 at decision level $dl = 4$. Assume that NOGOODPROPAGATION on assignment A_3 yields a solution. Since we are enumerating solutions, we cannot stop here. Rather, we continue with assignment A_4 obtained by flipping σ_d^6, and $bl = 3$ is the greatest decision level of any unflipped decision literal. Note that $\overline{\sigma_d^6}$ at decision level $dlevel(\overline{\sigma_d^6}) = 3 = bl$ is not asserted by any nogood. We continue by selecting decision literals σ_d^7 and σ_d^8, yielding assignment A_5. Suppose that NOGOODPROPAGATION yields again a violated nogood at decision level $dl = 5$ and that CONFLICTANALYSIS returns a nogood δ with $\overline{\sigma_a^8} \in \delta$ and $max(\{dlevel(\sigma) \mid \sigma \in \delta \setminus \{\overline{\sigma_a^8}\}\} \cup \{0\}) = 1$. That is, σ_a^8 is asserted by δ at decision level 1. Given that the previous solution included $\sigma_d^1 = dliteral(1)$, it must also have contained σ_a^8; otherwise, some nogood had been violated after NOGOODPROPAGATION. If we would now jump back to decision level 1 and assert σ_a^8, then the already enumerated solution would be feasible again, and CDNL-ENUM-ASP would repeat it. After asserting σ_a^8, we thus have to return to decision level $dl = 3 = bl$, rather than to 1, yielding assignment A_6. Note that A_6 still contains $\overline{\sigma_d^6}$, so that the solution encountered after selecting σ_d^6 (cf. A_3) cannot be repeated. Assume that selecting decision literal σ_d^9 at decision level $dl = 4$ yields a second solution for assignment A_7. Then, we backtrack to decision level $dl = 3 = bl$ and flip σ_d^9, yielding assignment A_8. Note that $\overline{\sigma_d^9}$ is not asserted by any nogood. If now NOGOODPROPAGATION yields a third solution, then decision level 3 is exhausted, that is, all solutions containing σ_d^1, σ_d^2, and σ_d^3 have been enumerated. Hence, we let $bl = 2$ and flip σ_d^3, yielding assignment A_9. Otherwise, if NOGOODPROPAGATION on A_8 leads to a violated nogood, then we do not analyze the conflict because $dl = 3 = bl$. In fact, flipped decision literals $\overline{\sigma_d^6}$ and $\overline{\sigma_d^9}$ lack asserting nogoods, so that the result of CONFLICTANALYSIS would be undefined. If NOGOODPROPAGATION yields a conflict for A_8, we thus proceed with A_9, as in the case that a solution is found for A_8.

We now introduce the notions of *correctness*, *completeness*, and *redundancy-freeness* for an answer set enumeration algorithm.

Definition 1. *For a logic program Π, we define an enumeration algorithm as*

- *correct, if every enumerated solution is an answer set of Π;*
- *complete, if all answer sets of Π are enumerated;*
- *redundancy-free, if no answer set of Π is enumerated twice.*

Furthermore, we need the following property **(UF)**:

> *For any assignment A, let* UNFOUNDEDSET(Π, A) *in Algorithm 1 return some non-empty unfounded set $U \subseteq atom(\Pi) \setminus A^{\mathbf{F}}$ for Π wrt A, if there is such a set U, and return the empty set $\emptyset$, otherwise.*[3]

By letting Π be a logic program and $s \in \mathbb{Z}$, we have the following correctness result.

Theorem 2. CDNL-ENUM-ASP(Π, s) *is correct, provided that **(UF)** holds.*

For a program Π and $X \subseteq atom(\Pi)$, we say that X *agrees* with a nogood δ if one of the following conditions holds, where $\overline{X} = atom(\Pi) \setminus X$:

- $\mathbf{F}p \in \delta$ for some $p \in X$,
- $\mathbf{T}p \in \delta$ for some $p \in \overline{X}$,
- $\mathbf{F}\beta \in \delta$ for some $\beta \in body(\Pi)$ such that $\beta \subseteq X \cup \{not\ p \mid p \in \overline{X}\}$, or
- $\mathbf{T}\beta \in \delta$ for some $\beta \in body(\Pi)$ such that $\beta \cap (\overline{X} \cup \{not\ p \mid p \in X\}) \neq \emptyset$.

Intuitively, the notion of agreement expresses that $\delta \not\subseteq A$ for the total assignment A corresponding to X. That is, $\mathbf{T}p \in A$ for all atoms $p \in X$, $\mathbf{F}p \in A$ for all atoms $p \in \overline{X}$, and for a body $\beta \in body(\Pi)$, $\mathbf{T}\beta \in A$ if all body literals of β are true wrt X and $\mathbf{F}\beta \in A$ otherwise. We can show that any answer set X of Π agrees with all nogoods dealt with by CDNL-ENUM-ASP, both static and dynamic ones.

We first consider static nogoods in $\Delta_\Pi \cup \Lambda_\Pi$, as given in (3) and (4).

Proposition 1. *Any answer set of Π agrees with all nogoods in $\Delta_\Pi \cup \Lambda_\Pi$.*

Given this, we can show that the answer sets of Π agree with all nogoods added to ∇.

Proposition 2. *At every state of* CDNL-ENUM-ASP(Π, s)*, any answer set of Π agrees with all nogoods in ∇, provided that **(UF)** holds.*[4]

This leads us to the completeness of CDNL-ENUM-ASP, when invoked with $s = 0$.

Theorem 3. CDNL-ENUM-ASP$(\Pi, 0)$ *is complete, provided that **(UF)** holds.*

Finally, we can show that CDNL-ENUM-ASP is redundancy-free.

Theorem 4. CDNL-ENUM-ASP(Π, s) *is redundancy-free.*

[3] A set U of atoms is unfounded for Π wrt A if we have $EB_\Pi(U) \subseteq A^{\mathbf{F}}$.

[4] We here stipulate **(UF)** for making sure that the result of CONFLICTANALYSIS is well-defined at every invocation.

Table 1. Experiments enumerating answer sets

No	Instance	#Sol	$clasp_a$	$clasp_{ar}$	$clasp_b$	$clasp_{br}$	*smodels*	$smodels_r$	$smodels_{cc}$	*cmodels*
1	hc_19	10^4	7.2	7.2	7.7	7.2	●	●	●	●
2	hc_19	10^5	71.4	77.1	83.5	91.2	●	●	●	●
3	hc_20	10^4	9.3	9.5	10.9	9.5	●	●	●	●
4	hc_20	10^5	103.4	117.2	115.8	109.9	●	●	●	●
5	mutex3IDFD	10^5	1.4	1.4	35.4	35.9	5.5	5.8	240.6	●
6	mutex3IDFD	10^6	14	13.9	●	●	55.9	52.8	●	●
7	mutex4IDFD	10^4	20.8	27.4	43.8	37	44.7	574.7	47.5	●
8	mutex4IDFD	10^5	52.2	63.2	596.7	585.7	273.4	●	●	●
9	pigeon_15	10^5	2.7	2.7	4	3.9	7.1	8.6	126.7	●
10	pigeon_15	10^6	26.1	26.5	53	54.7	71.8	73.6	●	●
11	pigeon_15	10^7	260.7	262.8	●	●	●	●	●	●
12	pigeon_16	10^5	3.2	3.1	4.4	4.6	7.8	9.9	175.2	●
13	pigeon_16	10^6	30.1	30.5	57.7	59.6	78.5	80.9	●	●
14	pigeon_16	10^7	303	304.5	●	●	●	●	●	●
15	queens_19	10^4	14.4	17.1	13.1	15.1	47.1	115	49	427.49
16	queens_19	10^5	141.5	143.8	135.9	162.7	265.1	358.1	●	●
17	queens_20	10^4	14.1	15.8	13.1	15.3	127	172.1	48.3	569.15
18	queens_20	10^5	147.2	170.5	149.6	178.6	380.3	●	●	●
19	schur-n29-m44	10^4	22.4	26.4	19.8	22.7	17.4	49.4	15.6	●
20	schur-n29-m44	10^5	203.1	212.5	177.2	246.4	132.4	175.9	353.2	●
21	schur-n29-m45	10^4	24.7	21.8	21.5	24.6	17.2	50.2	16.1	●
22	schur-n29-m45	10^5	231.6	265.6	190.7	199.9	133.3	176	397.3	●

5 Experiments

Our empirical evaluation addresses the following two questions: First, how does our algorithm improve on solution recording (via nogoods) and, second, in how far are backjumps hampered by the backtracking level. Our comparison considers *clasp* (RC4) in two different modes: (a) the one with bounded backjumping (and learning), using the algorithms from Section 4 (referred to by $clasp_a$), and (b) the one using unlimited backjumping (and learning) in conjunction with solution recording ($clasp_b$). Note that a *solution nogood* consists of decision literals only. The same strategy is pursued by $smodels_{cc}$ [16], but with decisions limited to atoms. In contrast, *cmodels* provides a whole answer set as solution nogood to the underlying (learning) SAT solver. Given that restarts are disabled in $clasp_a$ and $clasp_b$, our experiments also include both variants augmented with bounded and unbounded restarts, respectively (indicated by an additional subscript *r*). The bounded restart variant, $clasp_{ar}$, is allowed to resume search from the backtracking level (cf. Algorithm 2),[5] while $clasp_{br}$ can perform unlimited restarts. We also incorporate standard *smodels* (2.32) and the variant $smodels_r$ with activated restart option, $smodels_{cc}$ (1.08) with option "nolookahead" as recommended by the developers, and *cmodels* (3.67) using *zchaff* (2004.11.15). All experiments were run on a 2.2GHz PC on Linux. We report results in seconds, taking the average of 10 runs, each restricted to 600s time and 512MB memory. A timeout (in all 10 runs) is indicated by "●". The benchmark instances and extended results are available at [17].

In Table 1, we report results for enumerating a vast number of answer sets. The instances are from the areas of Hamiltonian cycles in complete graphs (1-4), bounded

[5] In order to guarantee redundancy-freeness, restarts must not discard the backtracking level with its flipped decision literals.

Table 2. Experiments illustrating backjumping and backtracking behavior

No	Instance	#Sol	Backtracks	Backjumps	Bounded Jumps	Skippable Levels	Skipped Levels (%)	Jump Length (max)	Bounded Length (max)	Jump Length (avg)	Bounded Length (avg)	Time
1	gryzzles.3	1	0	311	0	771	100	17	0	2.5	0	0.1
2	gryzzles.3	857913	618092	208373	4135	321000	97.6	21	15	1.5	1	178.4
3	gryzzles.7	1	0	675	0	1931	100	22	0	2.9	0	0.1
4	gryzzles.7	10^6	895612	215995	2783	324951	98	23	22	1.5	2	246.9
5	gryzzles.18	1	0	599	0	2026	100	27	0	3.4	0	0.1
6	gryzzles.18	10^6	811593	51219	1605	92953	96.1	27	18	1.7	2	235
7	mutex4IDFD	1	0	280	0	26698	100	590	0	95.4	0	17
8	mutex4IDFD	10^6	0	280	0	26698	100	590	0	95.4	0	579.7
9	sequence2-ss2	1	0	156	0	674	100	35	0	4.3	0	13.3
10	sequence2-ss2	38	64	2875	225	13915	53	73	43	2.6	29	17.9
11	sequence3-ss3	1	0	10921	0	40213	100	65	0	3.7	0	66.9
12	sequence3-ss3	332	315	55111	731	121435	97.8	65	20	2.2	3	361

model checking (5-8), pigeonhole (9-14), n-queens (15-18), and Schur numbers (19-22). We have chosen these combinatorial problems because of their large number of answer sets. This allows us to observe the effect of an increasing number of answer sets on the performance of the respective approaches. The number of requested (and successfully enumerated) solutions is given in the third column. Comparing the two variants of *clasp*, we observe that *clasp*$_a$ and *clasp*$_{ar}$ scale better than *clasp*$_b$ and *clasp*$_{br}$. This is most intelligible on examples from bounded model checking (5-8) and pigeonhole problems (9-14). Solutions for the former contain many decision literals, and the large solution nogoods significantly slow down *clasp*$_b$ and *clasp*$_{br}$. The pigeonhole problems are structurally simple, so that all decisions yield solutions. Since the number of easy-to-compute solutions is massive, the sheer number of recorded solution nogoods slows down *clasp*$_b$ and *clasp*$_{br}$. Also note that the time that *smodels* spends in lookahead is wasted here. With Hamiltonian cycles (1-4), n-queens (15-18), and Schur numbers (19-22), the picture is rather indifferent. That is, solving time tends to dominate enumeration time, and the recorded solution nogoods are not as critical as with the aforementioned problems. Notably, *smodels* is very effective on Schur numbers. We verified that all *clasp* variants make the same number of decisions (or choices) as *smodels*, so we conjecture that different run-times come from implementation differences: counter-based propagation in *smodels* versus watched literals in *clasp*. Regarding the other systems, we see that *smodels*$_{cc}$ is slower than *smodels* as regards enumeration (9-14) but sometimes faster if search is needed (17), and *cmodels* is clearly outperformed. Comparing *clasp*$_a$ to *clasp*$_{ar}$ and *clasp*$_b$ to *clasp*$_{br}$, we see that restarts make (almost) no difference on the problems in Table 1. Indeed, *clasp* hardly ever restarts on these problems, so that the effect is negligible. However, this indifference does not account for *smodels*$_r$, where restarts turn out to be quite counterproductive on our combinatorial problems.

Table 2 provides statistics regarding the backjumping and backtracking of *clasp*$_a$ upon the enumeration of answer sets. The first three instances are Hamiltonian path problems (1-6), the fourth is from bounded model checking (7,8), and the last two from compiler superoptimization (9-12). For every instance, we provide two rows: the backjump statistics for one answer set versus that for a certain number of answer sets. The "Backtracks" column shows the number of chronological backtracks, that is, conflicts on the backtracking level, while "Backjumps" indicates conflicts on greater decision levels. The number of backjumps that were forced to stop at the backtracking level is given by "Bounded Jumps". The "Skippable Levels" are the sum of backjump lengths (not counting backtracks), and "Skipped Levels" shows the percentage of levels that have effectively been skipped. We also provide the maximum "Jump Length" and the maximum "Bounded Length". The latter is the maximum number of skippable levels that have not been retracted in a jump because of hitting the backtracking level. Finally, we show the average "Jump Length", the average "Bounded Length", and the time. On the Hamiltonian path problems (1-6), we observe that the number of backtracks dominates that of backjumps. Indeed, we also observed on other problems, not shown here, that the "hard" part of the search was before finding the first answer set; afterwards, the number of conflicts above the backtracking level decreased significantly. We see this very drastically on the bounded model checking instance (7,8) where 280 long backjumps are performed (jump length 95 on average). After this "warm-up" phase, no further conflicts are encountered, even not on the backtracking level (0 backtracks). Finally, the superoptimization examples (9-12) are rather sparse regarding answer sets, and backjumps are still noticeable after the first solution has been found. In Line 10, we observe an exceptionally low percentage of skipped levels, approximately half of the skippable levels are kept. The few bounded jumps that are done have a significant length (29 unskipped levels on average). However, on all instances in Table 2, the jump of maximum length was unbounded and thus effectively executed. The average "Jump Length" and the average "Bounded Length" are usually small, except for 7, 8, and 10.

6 Discussion

We introduced a new approach to enumerating answer sets, centered around First-UIP learning and backjumping. To the best of our knowledge, our solution enumeration approach is novel even in the context of SAT. Unlike *relsat* [15], applying the *Last*-UIP scheme, our approach uses First-UIP backjumping as long as systematic backtracking is unnecessary. Different from the #SAT solver *cachet* [18], using so-called "component caching", our approach combines CDCL with CBJ for avoiding the repetition of solutions. Recent approaches to adopt SAT and CSP techniques in ASP solving [16,19,20] are rather implementation-specific and lack generality. Unlike this, we provided a uniform CSP-based approach by viewing ASP inferences as unit propagation on nogoods, which allowed us to directly incorporate techniques from CSP and SAT.

The *clasp* system implements state-of-the-art techniques from Boolean constraint solving, avoiding a SAT translation as done by *assat* [5], *cmodels* [6], and *sag* [19]. Also, *clasp* records loop nogoods only when ultimately needed for unit propagation; this is different from *assat* and *sag*, which determine loop formulas for all "terminating" loops. Unlike genuine ASP solvers *smodels* [2] and *dlv* [3], *clasp* does not determine greatest unfounded sets. Rather, it applies local propagation directly after an unfounded set has been found. Different from *smodels*$_{cc}$ [16] and *dlv* with backjumping [20], the usage of rule bodies in nogoods allows for a straightforward extension of unit propagation to ASP, abolishing the need for multiple inference rules. Notably, our novel approach allows *clasp* to enumerate answer sets of a program without explicitly prohibiting already computed solutions by nogoods, as done by *cmodels* and *smodels*$_{cc}$.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
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) (2006) 499–562
4. Mitchell, D.: A SAT solver primer. Bulletin of EATCS **85** (2005) 112–133
5. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
6. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning (2007) To appear.
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proceedings IJCAI'07, AAAI Press/The MIT Press (2007) 386–392
8. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
9. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
10. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1987) 89–148
11. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
12. Lee, J.: A model-theoretic counterpart of loop formulas. Proceedings IJCAI'05, Professional Book Center (2005) 503–508
13. Dechter, R., Frost, D.: Backjump-based backtracking for constraint satisfaction problems. Artificial Intelligence **136**(2) (2002) 147–188
14. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. MSc thesis, Simon Fraser University (2004)
15. Bayardo, R., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. Proceedings AAAI'97, AAAI Press/The MIT Press (1997) 203–208
16. Ward, J., Schlipf, J.: Answer set programming with clause learning. Proceedings LPNMR'04. Springer (2004) 302–313
17. (`http://www.cs.uni-potsdam.de/clasp`)
18. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. Proceedings SAT'04. (2004)
19. Lin, Z., Zhang, Y., Hernandez, H.: Fast SAT-based answer set solver. Proceedings AAAI'06, AAAI Press/The MIT Press (2006)
20. Ricca, F., Faber, W., Leone, N.: A backjumping technique for disjunctive logic programming. AI Communications **19**(2) (2006) 155–172

Head-Elementary-Set-Free Logic Programs

Martin Gebser[1], Joohyung Lee[2], and Yuliya Lierler[3]

[1] Institut für Informatik, Universität Potsdam, Germany
[2] School of Computing and Informatics, Arizona State University, USA
[3] Department of Computer Sciences, University of Texas at Austin, USA
gebser@cs.uni-potsdam.de, joolee@asu.edu, yuliya@cs.utexas.edu

Abstract. The recently proposed notion of an elementary set yielded a refinement of the theorem on loop formulas, telling us that the stable models of a disjunctive logic program can be characterized by the loop formulas of its elementary sets. Based on the notion of an elementary set, we propose the notion of head-elementary-set-free (HEF) programs, a more general class of disjunctive programs than head-cycle-free (HCF) programs proposed by Ben-Eliyahu and Dechter, that can still be turned into nondisjunctive programs in polynomial time and space by ”shifting” the head atoms into the body. We show several properties of HEF programs that generalize earlier results on HCF programs. Given an HEF program, we provide an algorithm for finding an elementary set whose loop formula is not satisfied, which has a potential for improving stable model computation by answer set solvers.

1 Introduction

Disjunctive logic programs under the stable model semantics are more expressive than nondisjunctive programs. The problem of deciding whether a disjunctive program has a stable model is Σ_2^P-complete [1], while the same problem for a nondisjunctive program is NP-complete.

However, Ben-Eliyahu and Dechter [2] showed that a class of disjunctive programs called “head-cycle-free (HCF)” programs can be turned into nondisjunctive programs in polynomial time and space, by “shifting” the head atoms into the body—a simple operation defined in [3]. This tells us that an HCF program is an “easy” disjunctive program, which is merely a syntactic shortcut of a nondisjunctive program. Thus, HCF programs play an important role in efficient computation of stable models for disjunctive programs. Indeed, the HCF property is exploited by answer set solvers DLV[1] [4] and CMODELS[2] [5].

In this paper, we propose the notion of head-elementary-set-free (HEF) programs, a more general class of disjunctive programs than HCF programs, that can still be turned into nondisjunctive programs in polynomial time and space by shifting. This is motivated by the recent study on elementary sets [6], which yielded a refinement of the theorem on loop formulas by Lin and Zhao [7]. All elementary sets are loops, but not all loops are elementary sets; still stable models can be characterized by elementary

[1] http://www.dbai.tuwien.ac.at/proj/dlv/
[2] http://www.cs.utexas.edu/users/tag/cmodels/

sets' loop formulas. Our definition of an HEF program is similar to the definition of an HCF program except that the former refers to elementary sets instead of loops. We observe that some other properties of nondisjunctive programs and HCF programs can be extended to HEF programs, including the main results by Lin and Zhao [8] characterizing the stable models of a nondisjunctive program by "inherent tightness," and the operational characterization of stable models of HCF programs by Leone *et al.* [9].

The properties of HEF programs studied here may be useful for improving the computation of disjunctive answer set solvers, such as DLV and CMODELS. As a first step, we provide an algorithm for finding an elementary set whose loop formula is not satisfied for a given HEF program, which is simpler and more efficient than the algorithm described in [10].

The outline of the paper is as follows. In Section 2, we review the definition of an elementary set introduced in [6] and show some of its properties. In Section 3, we introduce the notion of HEF programs and show that shifting preserves their stable models. In Section 4, we demonstrate that the notion of inherent tightness can be generalized to HEF programs, but not to general disjunctive programs. This section also includes simplifications of earlier notions. In Section 5, we show that the operational characterization of stable models by Leone *et al.* [9] can be extended to HEF programs as well. We also define "bounding" loops that allow for enhancing the model checking approach for disjunctive programs introduced in [9,11]. In Section 6, we present an algorithm for computing an elementary set for a given HEF program.

2 Review of Elementary Sets for Disjunctive Programs

We begin with a review of elementary sets, introduced in [6], which are a reformulation and generalization of elementary loops [12].

A *disjunctive program* is a finite set of *(disjunctive) rules* of the form

$$a_1; \dots; a_k \leftarrow a_{k+1}, \dots, a_l, \mathit{not}\ a_{l+1}, \dots, \mathit{not}\ a_m, \mathit{not}\ \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ \mathit{not}\ a_n \quad (1)$$

where $n \geq m \geq l \geq k \geq 0$ and $a_1, \dots, a_n$ are propositional atoms. We will identify a rule of the form (1) with the propositional formula

$$(a_{k+1} \wedge \dots \wedge a_l \wedge \neg a_{l+1} \wedge \dots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \dots \wedge \neg\neg a_n) \rightarrow (a_1 \vee \dots \vee a_k)\ .$$

We will also write (1) as

$$A \leftarrow B, F \quad (2)$$

where A is $a_1; \dots; a_k$, B is $a_{k+1}, \dots, a_l$, and F is

$$\mathit{not}\ a_{l+1}, \dots, \mathit{not}\ a_m, \mathit{not}\ \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ \mathit{not}\ a_n\ ,$$

and we identify A and B with their corresponding sets of atoms.

Let Π be a disjunctive program. A nonempty set X of atoms occurring in Π is called a *loop* of Π if, for all nonempty proper subsets Y of X, there is a rule (2) in Π such that $A \cap Y \neq \emptyset$ and $B \cap (X \setminus Y) \neq \emptyset$. As shown in [6], this definition of a loop is equivalent to the definition based on a positive dependency graph given in [13].

We say that a subset Y of X is *outbound* in X for Π if there is a rule (2) in Π such that $A \cap Y \neq \emptyset$, $B \cap (X \setminus Y) \neq \emptyset$, $A \cap (X \setminus Y) = \emptyset$, and $B \cap Y = \emptyset$. A nonempty set X of atoms that occur in Π is *elementary* for Π if all nonempty proper subsets of X are outbound in X for Π. It is clear that every elementary set is also a loop, but the converse does not hold. The definition of an elementary set above remains equivalent even if we restrict Y to be loops or even elementary sets.

Proposition 1. *For any disjunctive program Π and any nonempty set X of atoms that occur in Π, X is elementary for Π iff all proper subsets of X that are elementary for Π are outbound in X for Π.*

For any set Y of atoms, the *external support formula* of Y, denoted by $ES_\Pi(Y)$, is the disjunction of conjunctions $B \wedge F \wedge \bigwedge_{a \in A \setminus Y} \neg a$ for all rules (2) of Π such that $A \cap Y \neq \emptyset$ and $B \cap Y = \emptyset$.

The following proposition describes the relationship between the external support formula of an arbitrary set of atoms and the external support formulas of its subsets.

Proposition 2. *Let Π be a disjunctive program, and let X, Y, Z be sets of atoms such that $X \supseteq Y \supseteq Z$. If Z is not outbound in Y for Π and $X \models ES_\Pi(Z)$, then $X \models ES_\Pi(Y)$.*

This proposition is similar to Lemma 5 in [14], which states that $ES_\Pi(Z) \models ES_\Pi(Y)$ holds if there is no rule (2) in Π such that $A \cap Z \neq \emptyset$ and $B \cap (Y \setminus Z) \neq \emptyset$. Proposition 2 is more general in the sense that it refers to the stronger condition of "outboundness."

For any set Y of atoms, by $LF_\Pi(Y)$ we denote the following formula:

$$\bigwedge_{a \in Y} a \rightarrow ES_\Pi(Y) \ . \tag{3}$$

Formula (3) is called the *(conjunctive) loop formula* of Y for Π. Note that we still call (3) a loop formula even when Y is not a loop.

From Proposition 2, we derive the following relationship among loop formulas.

Proposition 3. *For any disjunctive program Π and any nonempty set X of atoms that occur in Π, there is a subset Y of X such that Y is elementary for Π and $LF_\Pi(Y) \models LF_\Pi(X)$.*

Proposition 3 allows us to restrict the attention to loop formulas of elementary sets only, rather than those of arbitrary sets or even loops. This yields the following theorem.

Theorem 1. *[6] For any disjunctive program Π and any model X of Π whose atoms occur in Π, the following conditions are equivalent:*

(a) X is stable for Π;[3]
(b) X satisfies $LF_\Pi(Y)$ for all nonempty sets Y of atoms occurring in Π;
(c) X satisfies $LF_\Pi(Y)$ for all loops Y of Π;
(d) X satisfies $LF_\Pi(Y)$ for all elementary sets Y of Π.

[3] For a model of Π, we will say that it is "stable for Π" if it is a stable model of Π.

3 Head-Elementary-Set-Free Logic Programs

Ben-Eliyahu and Dechter [2] defined a class of disjunctive programs called “head-cycle-free” programs that can be mapped in polynomial time and space to nondisjunctive programs, preserving the stable models. A disjunctive program Π is called *Head-Cycle-Free* (HCF) if, for every rule (2) in Π, there is no loop Y of Π such that $|A \cap Y| > 1$.

By referring to elementary sets in place of loops in the definition, we can define a class of programs that is more general than HCF programs. We will call a program Π *Head-Elementary-set-Free* (HEF) if, for every rule (2) in Π, there is no elementary set Y of Π such that $|A \cap Y| > 1$. From the fact that every elementary set is a loop, it is clear that every HCF program is an HEF program as well. However, not all HEF programs are HCF. For example, consider the following program Π_1:

$$\begin{array}{l} p \leftarrow r \\ q \leftarrow r \\ r \leftarrow p, q \\ p \,;q \leftarrow . \end{array} \tag{4}$$

The program has 6 loops, $\{p\}$, $\{q\}$, $\{r\}$, $\{p, r\}$, $\{q, r\}$, $\{p, q, r\}$. Since the head of the last rule contains two atoms from loop $\{p, q, r\}$, the program is not HCF. However, it is HEF since $\{p, q, r\}$ is not elementary for Π_1 (its subsets $\{p, r\}$ and $\{q, r\}$ are not outbound in $\{p, q, r\}$ for Π_1).

Let us write rule (2) in the following form:

$$a_1; \ldots ; a_k \leftarrow B, F \, . \tag{5}$$

Gelfond *et al.* [3] defined a mapping of a disjunctive program Π into a nondisjunctive program Π_{sh}, the “shifted” variant of Π, by replacing each rule (5) with k new rules:

$$a_i \leftarrow B,\ F,\ not\ a_1, \ldots,\ not\ a_{i-1},\ not\ a_{i+1}, \ldots,\ not\ a_k \, . \tag{6}$$

They showed that every stable model of Π_{sh} is also a stable model of Π, but not vice versa. Ben-Eliyahu and Dechter [2] showed that the other direction holds as well if Π is HCF. Here we extend the result to HEF programs.

Theorem 2. *If a program Π is HEF, then Π and Π_{sh} have the same stable models.*

For instance, one can check that both Π_1 and $(\Pi_1)_{sh}$ have $\{p\}$ and $\{q\}$ as their only stable models. Theorem 2 shows that HEF programs are not more expressive than nondisjunctive programs, so that one can regard the use of disjunctive rules in such programs as a syntactic shortcut. Another consequence is that the problem of deciding whether a model is stable for an HEF program is tractable, as in the case of nondisjunctive and HCF programs. (In the general disjunctive case, it is coNP-complete [4].)

Comparing the elementary sets of Π and the elementary sets of Π_{sh} gives the following result.

Proposition 4. *For any disjunctive program Π, if X is an elementary set of Π, then X is an elementary set of Π_{sh}.*

The converse of Proposition 4 does not hold, even if Π is HEF. For instance, consider the following HEF program Π_2:

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

Set $\{p, q, r\}$ is not elementary for Π_2 since, for instance, $\{p\}$ is not outbound in $\{p, q, r\}$. On the other hand, $\{p, q, r\}$ is elementary for $(\Pi_2)_{sh}$:

$$\begin{array}{l} p \leftarrow r, \mathit{not}\ q \\ q \leftarrow r, \mathit{not}\ p \\ r \leftarrow p \\ r \leftarrow q\,. \end{array} \tag{7}$$

However, there is a certain subset of Π_{sh} whose elementary sets are also elementary sets of Π. For a set X of atoms, by Π_X we denote the set of all rules in Π whose bodies are satisfied by X.

Proposition 5. *Let Π be a disjunctive program, X a set of atoms that occur in Π, and Y a subset of X. If Y is elementary for $(\Pi_{sh})_X$, then Y is elementary for Π as well.*

For $X = \{p, q, r\}$ and $(\Pi_2)_{sh}$, we have that $[(\Pi_2)_{sh}]_X$ consists of the last two rules of (7) only. Only singletons $\{p\}$, $\{q\}$, and $\{r\}$ are elementary for $[(\Pi_2)_{sh}]_X$, and they are elementary for Π_2 as well.

4 HEF Programs and Inherent Tightness

When we add more rules to a program, a stable model of the original program remains to be a stable model of the extended program as long as it satisfies the new rules.

Proposition 6. *For any disjunctive program Π and any model X of Π, X is stable for Π iff there is a subset Π' of Π such that X is stable for Π'.*

In view of Theorem 1, Proposition 6 tells us that, provided that X is a model of Π, it is sufficient to find a subset Π' of Π such that X is stable for Π', in order to verify that X is stable for Π. Of course, one can trivially take Π itself as the subset Π', but there are nontrivial subsets that deserve attention. If Π is nondisjunctive in Proposition 6, it is known that the subset Π' can be further restricted to a "tight" program [15,16]—the result known as "inherently tight", or "weakly tight" programs [8,17]. We will reformulate these results and show that they can be extended to HEF programs.

As in [13], we call a set of atoms occurring in Π *trivial* if it consists of a single atom a that has no rule (2) in Π such that $a \in A \cap B$. Recall that by Π_X we denote the set of all rules in Π whose bodies are satisfied by X.

Definition 1. *[16,13] A disjunctive program Π is called tight if every loop of Π is trivial. Program Π is called tight on a set X of atoms if every loop of Π_X is trivial.*

As defined in [18], a set X of atoms is *supported* by a nondisjunctive program Π if, for every atom $a \in X$, there is a rule (2) in Π_X such that $A = \{a\}$. We reformulate Lin and Zhao's notion of inherent tightness [8] as follows.

Definition 2. *A nondisjunctive program Π is called* inherently tight *on a set X of atoms if there is a subset Π' of Π such that Π' is tight and X is supported by Π'.*

Theorem 1 from [8] can be reformulated as follows.

Proposition 7. *For any nondisjunctive program Π and any model X of Π, X is stable for Π iff Π is inherently tight on X.*

One may wonder whether Proposition 7 can be extended to disjunctive programs as well, since the definition of a tight program (Definition 1) applies to disjunctive programs as well, and the notion of support was already extended to disjunctive programs [19,20,13]: a set X of atoms is *supported* by a disjunctive program Π if, for every atom $a \in X$, there is a rule (2) in Π_X such that $A \cap X = \{a\}$. We extend Definition 2 to disjunctive programs with these extended notions.

Unfortunately, for disjunctive programs, this straightforward extension of inherent tightness is not sufficient to characterize the stability of a model. In other words, only one direction of Proposition 7 holds for disjunctive programs.

Proposition 8. *For any disjunctive program Π and any model X of Π, if Π is inherently tight on X, then X is stable for Π.*

The following program Π_3 illustrates that the converse does not hold:

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

Set $\{p, q\}$ is the only stable model of Π_3, but there is no subset Π' of Π_3 such that Π' is tight and $\{p, q\}$ is supported by Π'.

However, one may expect that Proposition 7 can be extended to HEF programs since, as we noted in Section 3, HEF programs are merely a syntactic shortcut of nondisjunctive programs. Indeed, the following proposition holds.

Proposition 9. *For any HEF program Π and any model X of Π, X is stable for Π iff Π is inherently tight on X.*

Since every HCF program is HEF, the proposition also holds for HCF programs.

We observed that by turning to the notion of an elementary set in place of a loop, we can get generalizations of results known for loops, such as Theorem 2 and Proposition 9. This brings our attention to the following question. Can the notion of a tight program, which is based on loops, be generalized by referring to elementary sets instead? To answer this, let us modify Definition 1 as follows.

Definition 3. *A disjunctive program Π is called* e-tight *if every elementary set of Π is trivial. Program Π is called* e-tight *on a set X of atoms if every elementary set of Π_X is trivial.*

Since every elementary set is a loop, it is clear that a tight program is e-tight as well. But is the class of e-tight programs strictly more general than the class of tight programs? The reason why this is an interesting question to consider is because, if so, it would lead to a generalization of Fages' theorem [15], which would provide a more general class of programs for which the stable model semantics and the completion semantics coincide. However, it turns out that e-tight programs are not truly more general than tight programs.

Proposition 10. *(a) A disjunctive program is e-tight iff it is tight.*
(b) A disjunctive program is e-tight on a set X of atoms iff it is tight on X.

This result also indicates that the notion of an inherently tight program does not become more general by referring to elementary sets. That is, replacing "Π' is tight" in the statement of Definition 2 by "Π' is e-tight" does not affect the definition.

In the remainder of this section, we compare our reformulation of inherent tightness above with the original definition by Lin and Zhao.

Definition 4. *[8] A nondisjunctive program Π is called* inherently tight *on a set X of atoms if there is a subset Π' of Π such that Π' is tight on X and X is a stable model of Π'.*

There are two differences between our reformulation (Definition 2) and Definition 4. The former does not rely on the relative notion of tightness ("tight on a set of atoms") and uses a weaker condition of supportedness. Nevertheless it is not difficult to check that the two definitions are equivalent.

Proposition 7 above is a simplification of Theorem 1 from [8].

Proposition 11. *[8, Theorem 1] For any nondisjunctive program Π and any set X of atoms, X is a stable model of Π iff X is a model of the completion of Π and Π is inherently tight on X.*

Our reformulation of inherently tight programs is closely related to what Fages' called "well-supported" models [15]. We do not reproduce Fages' definition here due to lack of space, but it is not difficult to check that, for a nondisjunctive program Π and a set X of atoms, X is well-supported by Π iff Π is inherently tight on X. Proposition 7 is similar to Theorem 3.1 from [15], which showed that well-supported models coincide with stable models.

The notion of an inherently tight program is also closely related to the notion of a weakly tight program presented in [17].

5 Checking the Stability of Models for HEF Programs

The problem of deciding whether a given model is stable is coNP-complete for a disjunctive program, while it is tractable for HCF programs [9]. Leone *et al.* [9] presented an operational framework for checking the stability of a model in polynomial time for HCF programs. Given a disjunctive program Π and sets X, Y of atoms, they defined a sequence $R^0_{\Pi,X}(Y), R^1_{\Pi,X}(Y), \dots$ that converges to a limit $R^\omega_{\Pi,X}(Y)$ as follows:

– $R^0_{\Pi,X}(Y) = Y$ and
– $R^{i+1}_{\Pi,X}(Y)$ is obtained from $R^i_{\Pi,X}(Y)$ by removing every atom a for which there is a rule (2) in Π_X such that $A \cap X = \{a\}$ and $B \cap R^i_{\Pi,X}(Y) = \emptyset$.[4]

[4] Recall that Π_X consists of all rules (2) in Π such that $X \models B, F$.

A set Y of atoms is called *unfounded* by Π w.r.t. X if $X \not\models ES_{\Pi}(Y)$. Set X is *unfounded-free* for Π if it contains no nonempty subset that is unfounded by Π w.r.t. X. As shown in Corollary 2 from [21] and Theorem 4.6 from [9], unfounded-free models coincide with stable models.

Proposition 6.5 from [9] shows that X is unfounded-free for Π if $R^{\omega}_{\Pi,X}(X) = \emptyset$. The converse also holds if Π is restricted to be a HCF program, as shown in Theorem 6.9 from the same paper. That theorem can be extended to HEF programs.[5]

Proposition 12. *For any HEF program Π and any set X of atoms, X is unfounded-free for Π iff $R^{\omega}_{\Pi,X}(X) = \emptyset$.*

As an example, consider again program Π_1 ((4) in Section 3), which is HEF but not HCF. Theorem 6.9 from [9] does not apply since it is limited to HCF programs. However, for set $X_1 = \{p, q, r\}$, it holds that $R^{\omega}_{\Pi_1,X_1}(X_1) = X_1$, and in accordance with Proposition 12, X_1 is not a stable model of Π_1. For set $X_2 = \{p\}$, the limit $R^{\omega}_{\Pi_1,X_2}(X_2) = \emptyset$, and X_2 is a stable model of Π_1.

The following proposition shows how the HEF property and $R^{\omega}_{\Pi,X}$ can be used to decide whether a set Y of atoms contains a nonempty unfounded set for Π w.r.t. X. By $\Pi_{X,Y}$ we denote the set of all rules (2) in Π_X such that $X \cap (A \setminus Y) = \emptyset$.

Proposition 13. *For any disjunctive program Π, any set X of atoms, and any subset Y of X such that $\Pi_{X,Y}$ is HEF, $R^{\omega}_{\Pi,X}(Y) \neq \emptyset$ iff Y contains a nonempty unfounded subset for Π w.r.t. X.*

If we replace "$R^{\omega}_{\Pi,X}(Y) \neq \emptyset$" by "$R^{\omega}_{\Pi,X}(Y) = Y$ and Y is nonempty" in Proposition 13, only the left-to-right direction still holds. In the next section, we present an algorithm based on this for finding a non-trivial unfounded set for a HEF (sub)program.

As defined in [6], we say that a set Y of atoms occurring in a disjunctive program Π is *elementarily unfounded* by Π w.r.t. a set X of atoms if

- Y is an elementary set of $\Pi_{X,Y}$ that is unfounded by Π w.r.t. X, or
- Y is a singleton that is unfounded by Π w.r.t. X.

For a model X of Π, Theorem 1(e′) from [6] states that X is stable for Π iff no subset of X is elementarily unfounded by Π w.r.t. X. Thus stability checking can be cast into a problem of ensuring the absence of elementarily unfounded sets. Since every elementarily unfounded set is a loop, every elementarily unfounded set is clearly contained in a maximal loop, which allows us to split the search for elementarily unfounded sets by maximal loops. Below we describe a notion called "bounding loops," which give tighter bounds than maximal loops. We remark that the idea of using maximal loops for partitioning the program and splitting stability checking by subprograms was already presented by Leone *et al.* [9] and Koch *et al.* [11]. Their results can be enhanced by referring to bounding loops.

For a disjunctive program Π and a set X of atoms, let S be the set of all sets Y of atoms such that Y is a loop of $\Pi_{X,Y}$ and $R^{\omega}_{\Pi,X}(Y) = Y$. We call a maximal element

[5] We here consider slightly more general rules than those considered in [9], since the body of a rule may contain double negation (*not not*).

of S a *bounding* loop for Π w.r.t. X. The following two propositions describe properties of bounding loops, that are similar to maximal loops used for modular stability checking.

Proposition 14. *For any disjunctive program Π and any set X of atoms, bounding loops for Π w.r.t. X are disjoint.*

Proposition 15. *For any disjunctive program Π and any set X of atoms, every non-singleton elementarily unfounded set for Π w.r.t. X belongs to a bounding loop for Π w.r.t. X.*

Clearly, every bounding loop is contained in a maximal loop. However, as shown in the example below, bounding loops provide tighter bounds than maximal loops for locating elementarily unfounded sets. Propositions 14 and 15 tell us that the process of checking the absence of elementarily unfounded sets can be split by bounding loops.

Proposition 16. *For any disjunctive program Π and any model X of Π, X is stable for Π iff X is supported by Π and X contains no bounding loop Y for Π w.r.t. X such that Y has a nonempty unfounded subset for Π w.r.t X.*

We note that computing all bounding loops for Π w.r.t. X that are contained in X can be done in polynomial time using the following method:

1. Let $Y := X$.
2. Let $Z := R^{\omega}_{\Pi,X}(Y)$. (Note that $Z = R^{\omega}_{\Pi,X}(Z)$ holds.)
3. If $Z \neq \emptyset$, then consider the following cases:
   (a) If Z is a loop of $\Pi_{X,Z}$, then mark Z as a bounding loop for Π w.r.t. X.
   (b) Otherwise, proceed with step 2 for every maximal loop Y of $\Pi_{X,Z}$ that is contained in Z.

For example, consider program Π_4,

$$\begin{array}{lll} p \leftarrow r & s\,;t \leftarrow & p\,;q \leftarrow s \\ q \leftarrow r & s \leftarrow t & t\,;u \leftarrow q \\ r \leftarrow p, q & t \leftarrow s, u & u\,;v \leftarrow\,, \end{array}$$

and its model $X = \{p, q, r, s, t, u\}$. It holds that $(\Pi_4)_{X,X} = \Pi_4$, and X is a maximal loop of Π_4. Note that $R^{\omega}_{\Pi_4,X}(X) = \{p, q, r, s, t\} \neq X$, so that X is not a bounding loop for Π_4 w.r.t. X. Set $Z = \{p, q, r, s, t\}$ is not a loop of $(\Pi_4)_{X,Z}$; the maximal loops of $(\Pi_4)_{X,Z}$ contained in Z are $Y_1 = \{p, q, r\}$ and $Y_2 = \{s, t\}$. Indeed, Y_1 and Y_2 are the two bounding loops for Π_4 w.r.t. X.

From Proposition 13 and the definition of a bounding loop, we derive the following.

Corollary 1. *Let Π be a disjunctive program, X a set of atoms, and Y a bounding loop for Π w.r.t. X that is contained in X. If $\Pi_{X,Y}$ is HEF, then there is a nonempty subset of Y that is unfounded by Π w.r.t. X.*

Recall program Π_4, its model X, and bounding loop Y_1. Note that $(\Pi_4)_{X,Y_1}$ is HEF. By Corollary 1, the fact that $(\Pi_4)_{X,Y_1}$ is HEF implies that X is not stable for Π_4. In fact, Y_1 contains $\{p, r\}$ and $\{q, r\}$, which are both elementarily unfounded by Π_4 w.r.t. X.

6 Computing Elementarily Unfounded Sets

It is inevitable that exponentially many loop formulas have to be considered in the worst case [22]. Hence, SAT-based answer set solvers do not try to find all loop formulas at once; loop formulas are added incrementally until a stable model is found (if there is any). As shown in [6], it is sufficient to consider only loop formulas of elementarily unfounded sets in this process. Thus, it is important to design an efficient algorithm for finding elementarily unfounded sets.

For a general disjunctive program, it has been shown that deciding whether a given set of atoms is elementary is coNP-complete [6]. While we do not expect a tractable algorithm for computing elementarily unfounded sets of general disjunctive programs, it is possible for HEF programs. Below we present a tractable algorithm for HEF programs, which is simpler and more efficient than the one described in [10].[6]

For any disjunctive program Π and any set Y of atoms, we define $(Y, EC_\Pi(Y))$ as a directed graph where:

$$\begin{aligned} EC^0_\Pi(Y) &= \emptyset \\ EC^{i+1}_\Pi(Y) &= \{\, (a,b) \mid \text{there is a rule (2) in } \Pi \text{ such that } A \cap Y = \{a\} \text{ and} \\ &\qquad\quad \text{all atoms } b \text{ in } B \cap Y \text{ belong to the same} \\ &\qquad\quad \text{strongly connected component of } (Y, EC^i_\Pi(Y)) \,\} \\ EC_\Pi(Y) &= \textstyle\bigcup_{i \geq 0} EC^i_\Pi(Y)\ . \end{aligned}$$

This graph is equivalent to the "elementary subgraph" defined in [6], and it is closer to the algorithm for computing an elementarily unfounded set described below.

We first note that Theorem 2 in [6] can be extended to HEF programs.

Proposition 17. *For any HEF program Π and any nonempty set Y of atoms that occur in Π, Y is elementary for Π iff $(Y, EC_\Pi(Y))$ is a strongly connected graph.*

Given a disjunctive program Π, a set X of atoms occurring in Π, and a nonempty subset Y of X such that $\Pi_{X,Y}$ is HEF and $R^\omega_{\Pi,X}(Y) = Y$, Figure 1 shows an algorithm for computing an elementarily unfounded set by Π w.r.t. X that is contained in Y.[7]

Due to Step I, E-SET never considers any rule (2) of $\Pi_{X,Y}$ such that $|A \cap Y| > 1$. This is similar to the definition of $EC^{i+1}_\Pi(Y)$ above, where only rules (2) satisfying $A \cap Y = \{a\}$ contribute to any edge. In a bottom-up manner, Step 1(a) of E-SET adds edges to $EC_{\Pi_{X,Y}}(Y)$ for rules (2) such that $|B \cap Y| = 1$. This ensures that all rules contributing to edges depend on a single SCC of $(Y, EC_{\Pi_{X,Y}}(Y))$. In rules (2) of $\Pi_{X,Y}$ such that B contains multiple atoms from a recently computed SCC, Step 1(b) replaces all atoms of the SCC by a single representative. If this leads to $|B \cap Y| = 1$, rule (2) contributes an edge in the next iteration of Step 1(a). The described process is iterated until no further edges can be added. If a single SCC is obtained, i.e., if $(Y, EC_{\Pi_{X,Y}}(Y))$ is strongly connected, then Y is elementarily unfounded by Π w.r.t. X. Otherwise, in Step 2, we remove atoms from Y that belong to some SCC C that is not reached ($Y \setminus C$ still contains an elementarily unfounded set for Π w.r.t. X). In the next iteration

[6] That algorithm was designed for nondisjunctive programs, but also applies to HEF programs.

[7] "SCC" is used as a shorthand for "Strongly Connected Component."

```
E-SET($\Pi_{X,Y}, Y$)
I. $\Pi_{X,Y} := \Pi_{X,Y} \setminus \{(A \leftarrow B, F) \in \Pi_{X,Y} \mid |A \cap Y| > 1\}$
II. $EC_{\Pi_{X,Y}}(Y) := \emptyset$
III. While $(Y, EC_{\Pi_{X,Y}}(Y))$ is not strongly connected Do
  1. While there is a rule $(A \leftarrow B, F)$ in $\Pi_{X,Y}$ such that $|A \cap Y| = 1$ and $|B \cap Y| = 1$ Do
    (a) For each rule $(A \leftarrow B, F)$ in $\Pi_{X,Y}$ such that $|A \cap Y| = 1$ and $|B \cap Y| = 1$ Do
      i. $EC_{\Pi_{X,Y}}(Y) := EC_{\Pi_{X,Y}}(Y) \cup \{(a, b) \mid A \cap Y = \{a\}, B \cap Y = \{b\}\}$
      ii. $\Pi_{X,Y} := \Pi_{X,Y} \setminus \{(A \leftarrow B, F)\}$    /* the rule needs not be considered further */
    (b) For each (non-trivial) SCC $(C, EC_{\Pi_{X,Y}}(Y) \cap (C \times C))$ of $(Y, EC_{\Pi_{X,Y}}(Y))$ Do
      i. Select an atom $b \in C$
      ii. $\Pi_{X,Y} := (\Pi_{X,Y} \setminus \{(A \leftarrow B, F) \in \Pi_{X,Y} \mid |B \cap C| > 1\}) \cup$
                $\{(A \leftarrow b, B \setminus C, F) \mid (A \leftarrow B, F) \in \Pi_{X,Y}, |B \cap C| > 1\}$
  2. If $(Y, EC_{\Pi_{X,Y}}(Y))$ is not strongly connected Then
    (a) Select some SCC $(C, EC_{\Pi_{X,Y}}(Y) \cap (C \times C))$ of $(Y, EC_{\Pi_{X,Y}}(Y))$ that is not reached
        in $(Y, EC_{\Pi_{X,Y}}(Y))$
    (b) $Y := Y \setminus C$        /* some $Z \subseteq Y \setminus C$ is elementarily unfounded by $\Pi$ w.r.t. $X$ */
    (c) $EC_{\Pi_{X,Y}}(Y) := EC_{\Pi_{X,Y}}(Y) \setminus \{(a, b) \in EC_{\Pi_{X,Y}}(Y) \mid a \in C\}$
IV. Return $Y$
```

Fig. 1. E-SET: An algorithm to compute an elementarily unfounded set

of Step 1, this might allow to add more edges to $EC_{\Pi_{X,Y}}(Y)$ for rules (2) of $\Pi_{X,Y}$ such that $B \cap C \neq \emptyset$. The process is repeated until $(Y, EC_{\Pi_{X,Y}}(Y))$ becomes a strongly connected graph. Note that the computed set Y can be a proper subset of the Y in the invocation of E-SET($\Pi_{X,Y}, Y$).

When we apply E-SET to Π_1 ((4) in Section 3) and $Y = \{p, q, r\}$, it adds edges (p, r) and (q, r) to $EC_{\Pi_1}(Y)$. As the resulting graph is not strongly connected, either q or p is removed from Y. After this, adding edge (r, p) or (r, q), respectively, to $EC_{\Pi_1}(Y)$ leads to a strongly connected graph. The result of E-SET is thus either $\{p, r\}$ or $\{q, r\}$, which are the two elementarily unfounded sets for Π_1 w.r.t. $\{p, q, r\}$.

The following proposition states the correctness of the E-SET algorithm.

Proposition 18. *Let Π be a disjunctive program, X a set of atoms that occur in Π, and Y a nonempty subset of X. If $\Pi_{X,Y}$ is HEF and $R^{\omega}_{\Pi,X}(Y) = Y$, then E-SET($\Pi_{X,Y}$,$Y$) returns an elementarily unfounded set for Π w.r.t. X.*

It is reasonable to take a bounding loop Y for Π w.r.t. X such that $\Pi_{X,Y}$ is HEF as input for E-SET since every elementarily unfounded set is a subset of some bounding loop. For the correctness of E-SET, it is however sufficient that $\Pi_{X,Y}$ is HEF and $R^{\omega}_{\Pi,X}(Y) = Y$.

Finally, we comment on the complexity of E-SET. Note that E-SET successively merges atoms from an input set Y into SCCs until finally obtaining a single SCC.

Whenever a new SCC C is produced, all its atoms are replaced by a single element of C in rules (2) such that $|B \cap C| > 1$. This can be regarded as counting down body elements until only one atom from Y is left, in which case a rule "fires." This behavior is similar to the *Dowling-Gallier* algorithm [23], also used to compute the minimal model of a set of Horn clauses. Since the computation of SCCs and the Dowling-Gallier algorithm have linear complexity, the same is concluded for E-SET. In contrast, the elementary set computation algorithm in [10] has complexity $O(n \times log\ n)$.

7 Conclusion

The main contribution of this paper is identifying the class of HEF programs, a more general class of disjunctive programs than HCF programs, that can be turned into nondisjunctive programs in polynomial time and space by shifting head atoms into the body. We showed that several properties of nondisjunctive programs and HCF programs can be extended to HEF programs in a straightforward way. Since HCF programs have played an important role in the computation of stable models for disjunctive programs, we expect that HEF programs can be useful as well. As a first step, we have provided an algorithm for finding an elementarily unfounded set for a HEF program, which has a potential for improving the stable model computation for disjunctive programs.

As a future work, we plan to implement algorithm E-SET, presented in this paper, in CMODELS for an empirical evaluation. It is an open question whether identifying HEF programs is tractable, while it is known that identifying HCF programs can be done in linear time.

Acknowledgments

We are grateful to Selim Erdoğan, Tomi Janhunen, Dan Lessin, Vladimir Lifschitz, Torsten Schaub, Jicheng Zhao, and the anonymous referees for their useful comments. Joohyung Lee was partially supported by DTO AQUAINT.

References

1. Eiter, T., Gottlob, G.: Complexity results for disjunctive logic programming and application to nonmonotonic logics. In: Proceedings of International Logic Programming Symposium. (1993) 266–278
2. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial Intelligence **12** (1994) 53–87
3. Gelfond, M., Lifschitz, V., Przymusińska, H., Truszczyński, M.: Disjunctive defaults. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning. (1991) 230–237
4. 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** (2006) 499–562
5. Lierler, Y.: Cmodels: SAT-based disjunctive answer set solver. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning. (2005) 447–452

6. Gebser, M., Lee, J., Lierler, Y.: Elementary sets for logic programs. In: Proceedings of National Conference on Artificial Intelligence. (2006)
7. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157** (2004) 115–137
8. Lin, F., Zhao, J.: On tight logic programs and yet another translation from normal logic programs to propositional logic. In: Proceedings of International Joint Conference on Artificial Intelligence. (2003) 853–858
9. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation **135** (1997) 69–112
10. Anger, C., Gebser, M., Schaub, T.: Approaching the core of unfounded sets. In: Proceedings of International Workshop on Nonmonotonic Reasoning. (2006) 58–66
11. Koch, C., Leone, N., Pfeifer, G.: Enhancing disjunctive logic programming systems by SAT checkers. Artificial Intelligence **151** (2003) 177–212
12. Gebser, M., Schaub, T.: Loops: Relevant or redundant? In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning. (2005) 53–65
13. Lee, J.: A model-theoretic counterpart of loop formulas. In: Proceedings of International Joint Conference on Artificial Intelligence. (2005) 503–508
14. Ferraris, P., Lee, J., Lifschitz, V.: A generalization of the Lin-Zhao theorem. Annals of Mathematics and Artificial Intelligence **47** (2006) 79–101
15. Fages, F.: Consistency of Clark's completion and existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
16. Erdem, E., Lifschitz, V.: Tight logic programs. Theory and Practice of Logic Programming **3** (2003) 499–518
17. You, J.H., Yuan, L.Y., Zhang, M.: On the equivalence between answer sets and models of completion for nested logic programs. In: Proceedings of International Joint Conference on Artificial Intelligence. (2003) 859–866
18. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1988) 89–148
19. Baral, C., Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming **19/20** (1994) 73–148
20. Inoue, K., Sakama, C.: Negation as failure in the head. Journal of Logic Programming **35** (1998) 39–78
21. Saccá, D., Zaniolo, C.: Stable models and non-determinism in logic programs with negation. In: Proceedings of ACM Symposium on Principles of Database Systems. (1990) 205–217
22. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic **7** (2006) 261–268
23. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming **1** (1984) 267–284

A Deductive System for PC(ID)[⋆]

Ping Hou, Johan Wittocx[⋆⋆], and Marc Denecker

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{Ping.Hou,Johan.Wittocx,Marc.Denecker}@cs.kuleuven.be

Abstract. The logic FO(ID) uses ideas from the field of logic programming to extend first order logic with non-monotone inductive definitions. This paper studies a deductive inference method for PC(ID), its propositional fragment. We introduce a formal proof system based on the sequent calculus (Gentzen-style deductive system) for this logic. As PC(ID) is an integration of classical propositional logic and propositional inductive definitions, our deductive system integrates inference rules for propositional calculus and definitions. We prove the soundness and completeness of this deductive system for a slightly restricted fragment of PC(ID). We also give a counter-example to show that cut-elimination does not hold in this proof system.

1 Introduction

Inductive definitions are common in mathematical practice. For instance, the non-monotone inductive definition of the satisfaction relation $\models$ can be found in most textbooks on first order logic (FO). This prevalence of inductive definitions indicates that these offer a natural and well-understood way of representing knowledge. It is well-known that, in general, inductive definitions cannot be expressed in first order logic. For instance, the transitive closure of a graph is one of the simplest concepts typically defined by induction – the relation is defined by two inductive rules: (a) if (x, y) is an edge of the graph, (x, y) belongs to the transitive closure and (b) if there exists a z such that both (x, z) and (z, y) belongs to the transitive closure, then (x, y) belongs to the transitive closure – yet it cannot be defined in first order logic.

It turns out, however, that certain knowledge representation logics do allow a natural and uniform formalization of the most common forms of inductive definitions. Recently, the authors of [5,6] pointed out that semantical studies in the area of logic programming might contribute to a better understanding of such generalized forms of induction. In particular, it was argued that the well-founded semantics of logic programming [10] extends monotone induction and formalizes induction over well-founded sets and iterated induction. The language of FO(ID) uses the well-founded semantics to extend classical first order logic

[⋆] This work is supported by FWO-Vlaanderen and by GOA/2003/08.

[⋆⋆] Research Assistant of the Fund for Scientific Research-Flanders (Belgium) (FWO-Vlaanderen).

with a new "inductive definition" primitive. In the resulting formalism, all kinds of definitions regularly found in mathematical practice – e.g., monotone inductive definitions, non-monotone inductive definitions over a well-ordered set, and iterated inductive definitions – can be represented in a uniform way. Moreover, this representation neatly coincides with the form such definitions would take in a mathematical text. For instance, in FO(ID) the transitive closure of a graph can be defined as:

$$\left\{\begin{array}{c}\forall x, y\ TransCl(x,y) \leftarrow Edge(x,y) \\ \forall x, y\ TransCl(x,z) \leftarrow (\exists z\ TransCl(x,y) \wedge TransCl(y,z))\end{array}\right\}$$

However, FO(ID) is able to handle more than only mathematical concepts. Indeed, inductive definitions are also crucial in declarative Knowledge Representation. Not only non-inductive definitions are frequent in common-sense reasoning as argued in [2], also inductive definitions are. For instance, in [7], it was shown that situation calculus can be given a natural representation as an iterated inductive definition. The resulting theory is able to correctly handle tricky issues such as recursive ramifications, and is in fact, to the best of our knowledge, the most general representation of this calculus to date. Also does FO(ID) have strong links to several KR-logics. For instance, it can be classified in the family of Description Logics, which it extends by allowing n-ary predicates and non-monotone inductive definitions.

As for every formal logical system, the development of deductive inference methods is an important research topic. For instance, it is well-known that deductive reasoning is a distinguished feature of Description Logics. Description Logics support inference on ontologies without using Closed World Assumption and Unique Name Assumption [1]. As FO(ID) generalizes Description Logics, one could investigate how these inference methods can be extended to FO(ID). In this paper we take a first step towards the development of a proof system for FO(ID). However, because FO(ID) is not even semi-decidable, it is clear that a sound and complete proof system for FO(ID) cannot exist. As such, we will have to investigate deductive systems for FO(ID) and the subclasses of FO(ID) for which these systems are complete.

The goal of this paper is to present a proof system for PC(ID), as the initial investigation to that for FO(ID). Our work is inspired by the one of Compton, who used sequent calculus (Gentzen-style deductive system) methods in [3,4] to investigate sound and complete deductive inference methods for existential least fixpoint logic and stratified least fixpoint logic. Indeed, these two logics can be viewed as fragments of FO(ID). We introduce a sequent calculus for PC(ID) and prove its soundness and completeness for a restricted fragment of PC(ID). We also prove that cut-elimination does not hold in our proof system by showing a counter-example.

By developing a proof system for PC(ID), we want to enhance the understanding of proof-theoretic foundations of FO(ID). The proof-theoretic perspective also allows us to investigate the possibility of FO(ID) as an assertion language for program verification, which can overcome the limitations of first order logic.

Another application of this work could be the development of tools to check the correctness of the outputs generated by PC(ID) model generators such as MIDL [14]. Given a PC(ID) theory T as input, MIDL outputs a model for T or concludes that T is unsatisfiable. In the former case, an independent *model checker* can be used to check whether the output is indeed a model of T. However, when MIDL concludes that T is unsatisfiable, it is less obvious how to check the correctness of this answer. One solution is to transform a trace of MIDL's computation into a proof of unsatisfiability in some PC(ID) proof system. An independent *proof checker* can be used to check this formal proof. Model and proof checkers can be a great help to detect bugs in model generators. An analogous checker for the Boolean Satisfiability problem (SAT) solvers was described in [18].

The structure of this paper is as follows. We introduce PC(ID) in Section 2. We present a deductive system for PC(ID) in Section 3. The main results of the soundness and completeness of the deductive system are investigated in Section 4. We finish with conclusions, related and future work.

2 Preliminaries

In this section, we introduce PC(ID), the propositional fragment of FO(ID), and explain its semantics.

A propositional vocabulary τ is a set of propositional atoms. A *definition* D over τ is a finite set of rules of the form $P \leftarrow \varphi$, where $P \in \tau$ and φ is a propositional formula over τ. An atom appearing in the head of a rule of D is called a *defined* atom of D, any other atom is called an *open* atom of D. The set of defined symbols is denoted by τ_D^d, and the set of open symbols by τ_D^o. A PC(ID)-formula over τ is a boolean combination of propositional formulae and definitions over τ. A literal is an atom P or its negation $\neg P$.

A three-valued τ-interpretation is a function I from τ to the set of truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. An interpretation is called two-valued if it maps no atom to $\mathbf{u}$. When $\tau' \subseteq \tau$, we denote the restriction of a τ-interpretation I to the symbols of τ' by $I|_{\tau'}$. For a τ-interpretation I, a truth value v and an atom $P \in \tau$, we denote by $I[P/v]$ the τ-interpretation that assigns v to P and coincides with I on all other atoms. We extend this notation to sets of atoms.

The *truth order* $\leq$ on $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ is induced by $\mathbf{f} \leq \mathbf{u} \leq \mathbf{t}$; the *precision order* $\leq_p$ is induced by $\mathbf{u} \leq_p \mathbf{f}$ and $\mathbf{u} \leq_p \mathbf{t}$. Both truth and precision order pointwise extend to interpretations. Define $\mathbf{f}^{-1} = \mathbf{t}$, $\mathbf{u}^{-1} = \mathbf{u}$ and $\mathbf{t}^{-1} = \mathbf{f}$.

A three-valued interpretation I on τ can be extended to all propositional formulae over τ by induction on formulae:

- $P^I = I(P)$ if $P \in \tau$;
- $(\varphi \wedge \psi)^I = min_{\leq}(\{\varphi^I, \psi^I\})$;
- $(\varphi \vee \psi)^I = max_{\leq}(\{\varphi^I, \psi^I\})$;
- $(\neg\varphi)^I = (\varphi^I)^{-1}$.

It can be shown that if $I \leq_p J$, then $\varphi^I \leq_p \varphi^J$.

Let D be a definition over τ and let I_O be a three-valued τ^o_D-interpretation. Consider any sequence of three-valued τ-interpretations $(I^\alpha)_{\alpha \geq 0}$ extending I_O such that $I^0(P) = \mathbf{u}$ for every $P \in \tau^d_D$, and for every natural number α, $I^{\alpha+1}$ relates to I^α in one of the following ways:

- $I^{\alpha+1} = I^\alpha[P/\mathbf{t}]$ where P is a defined atom such that $P^{I^\alpha} = \mathbf{u}$ and for some rule $P \leftarrow \varphi \in D, \varphi^{I^\alpha} = \mathbf{t}$.
- $I^{\alpha+1} = I^\alpha[U/\mathbf{f}]$, where U is a non-empty set of defined atoms, such that for each $P \in U$, $I^\alpha(P) = \mathbf{u}$ and for each rule $P \leftarrow \varphi \in D$, $\varphi^{I^{\alpha+1}} = \mathbf{f}$.

We call such a sequence a *well-founded induction.* A well-founded induction is *terminal* if it cannot be extended anymore. It can be shown that each terminal well-founded induction is a sequence of increasing precision and its limit is the well-founded model extending I_O [8].

We say that an interpretation I satisfies a definition D and define $D^I = \mathbf{t}$ if I is the well-founded model extending $I|_{\tau^o_D}$. Otherwise, we define $D^I = \mathbf{f}$. By adding this as a new base case to the definition of the truth function of formulae, we can extend the truth function inductively to all PC(ID)-formulae. For an arbitrary PC(ID)-formula φ, we say that an interpretation I satisfies φ, or I is a model of φ, denoted by $I \models \varphi$, if I is two-valued and $\varphi^I = \mathbf{t}$.

Example 1. Consider the definition $D_1 = \left\{ P \leftarrow Q \right\}$. Then $P \in \tau^d_{D_1}$, $Q \in \tau^o_{D_1}$. There are only two interpretations satisfying D_1. One maps both P and Q to $\mathbf{t}$, the other one maps both P and Q to $\mathbf{f}$. These are also the models of $D_2 = \left\{ Q \leftarrow P \right\}$, where $Q \in \tau^d_{D_2}$ and $P \in \tau^o_{D_2}$. Note that $D_3 = \left\{ \begin{array}{c} P \leftarrow Q \\ Q \leftarrow P \end{array} \right\}$ has only one model, namely the model mapping both P and Q to false.

Remark that we are only interested in two-valued interpretations. We call a definition D *total* if for every interpretation I_O of its open atoms, the well-founded model of D extending I_O is two-valued.

3 The Deductive System for PC(ID)

In this section, we present **LPC(ID)**, a proof system for PC(ID) based on the propositional part of Gentzen's sequent calculus **LK** [11,16]. First, we introduce some basic definitions and notations. Let capital Greek letters $\Gamma, \Delta, \ldots$ denote finite (possibly empty) sequences of PC(ID)-formulae separated by commas. By $\bigwedge \Gamma$, respectively $\bigvee \Gamma$, we denote the conjunction, respectively disjunction of all formulae in Γ. By $\Gamma \setminus \Delta$, we denote the sequence obtained by deleting from Γ all occurrences of formulae that occur in Δ. A sequence Γ of literals is called *consistent* if Γ does not contain any complementary literals P and $\neg P$.

A *sequent* is an expression of the form $\Gamma \rightarrow \Delta$, where Γ and Δ are sequences of PC(ID)-formulae. Γ and Δ are respectively called the *antecedent* and *succedent* of the sequent and each formula in Γ and Δ is called a *sequent formula*. We will denote sequents by $S, S_1, \ldots$. A sequent $\Gamma \rightarrow \Delta$ is *valid*, denoted by $\models \Gamma \rightarrow \Delta$, if every two-valued model of $\bigwedge \Gamma$ satisfies $\bigvee \Delta$. A *counter-model* for $\Gamma \rightarrow \Delta$ is a two-valued interpretation I such that $I \models \bigwedge \Gamma$ but $I \not\models \bigvee \Delta$. The sequent $\Gamma \rightarrow$ is equivalent to $\Gamma \rightarrow \bot$ and $\rightarrow \Delta$ is equivalent to $\top \rightarrow \Delta$, where $\bot, \top$ are logical constants for *false* and *true*, respectively.

An *inference rule* is an expression of the form $\dfrac{S_1; \ldots ; S_n}{S}$ $n \geq 0$, where $S_1, \ldots , S_n$ and S are sequents. The S_i are called the *premises* of the inference rule, S is called the *consequence*. Intuitively, an inference rule means that S can be inferred, given that all $S_1, \ldots , S_n$ have already been inferred.

The *initial sequents*, or *axioms* of **LPC(ID)** are the sequents of the form $\Gamma, A \rightarrow A, \Delta$ or $\bot \rightarrow \Delta$ or $\Gamma \rightarrow \top$, where A is any formula, Γ and Δ are arbitrary sequences of formulae.

The inference rules for **LPC(ID)** consist of the structural rules, logical rules and definition rules. The structural and logical rules deal with the propositional calculus part of PC(ID) and are given as follows.

Structural rules

– Weakening rules: left: $\dfrac{\Gamma \rightarrow \Delta}{A, \Gamma \rightarrow \Delta}$; right: $\dfrac{\Gamma \rightarrow \Delta}{\Gamma \rightarrow \Delta, A}$.
– Contraction rules: left: $\dfrac{A, A, \Gamma \rightarrow \Delta}{A, \Gamma \rightarrow \Delta}$; right: $\dfrac{\Gamma \rightarrow \Delta, A, A}{\Gamma \rightarrow \Delta, A}$.
– Exchange rules: left: $\dfrac{\Gamma_1, A, B, \Gamma_2 \rightarrow \Delta}{\Gamma_1, B, A, \Gamma_2 \rightarrow \Delta}$; right: $\dfrac{\Gamma \rightarrow \Delta_1, A, B, \Delta_2}{\Gamma \rightarrow \Delta_1, B, A, \Delta_2}$.
– Cut rule: $\dfrac{\Gamma \rightarrow \Delta, A; \quad A, \Gamma \rightarrow \Delta}{\Gamma \rightarrow \Delta}$.

Logical rules

– $\neg$ rules: left: $\dfrac{\Gamma \rightarrow A, \Delta}{\neg A, \Gamma \rightarrow \Delta}$; right: $\dfrac{A, \Gamma \rightarrow \Delta}{\Gamma \rightarrow \Delta, \neg A}$.
– $\wedge$ rules: left: $\dfrac{A, B, \Gamma \rightarrow \Delta}{A \wedge B, \Gamma \rightarrow \Delta}$; right: $\dfrac{\Gamma \rightarrow \Delta, A; \ \Gamma \rightarrow \Delta, B}{\Gamma \rightarrow \Delta, A \wedge B}$
– $\vee$ rules: left: $\dfrac{A, \Gamma \rightarrow \Delta; \ B, \Gamma \rightarrow \Delta}{A \vee B, \Gamma \rightarrow \Delta}$; right: $\dfrac{\Gamma \rightarrow \Delta, A, B}{\Gamma \rightarrow \Delta, A \vee B}$.

By a left (right) *generalized logical rule*, we mean a left (right) logical rule where the formula of interest in the consequence (i.e. $\neg A$, $A \wedge B$ or $A \vee B$) can occur in an arbitrary position in Γ (Δ) instead of only left (right) from it. A generalized logical rule can be obtained by combining the exchange and logical rules.

The *definition rules* of **LPC(ID)** consist of the right definition rule, the left definition rule and the definition introduction rule. Without loss of generality, we assume from now on that there is only one rule with head P in a definition D for every $P \in \tau_D^d$. We refer to this rule as *the rule for P in D*. Indeed, any set of rules $\{P \leftarrow \varphi_1, \ldots , P \leftarrow \varphi_n\}$ can be transformed into a single rule $P \leftarrow \varphi_1 \vee \ldots \vee \varphi_n$.

The *right definition rule* allows inferring the truth of a defined atom from a definition D. It is closely related to the first rule for extending a well-founded induction. Let D be a definition and $P \leftarrow \varphi$ the rule for P in D. Then the right definition rule for P is given as follows.

Right definition rule for P $\dfrac{\Gamma, D \rightarrow \varphi, \Delta}{\Gamma, D \rightarrow P, \Delta}$.

We illustrate this inference rule with an example.

Example 2. Consider the definition $D = \left\{ \begin{array}{c} P \leftarrow P \wedge \neg Q \\ Q \leftarrow \neg P \end{array} \right\}$. The right definition rule for P is $\dfrac{\Gamma, D \rightarrow P \wedge \neg Q, \Delta}{\Gamma, D \rightarrow P, \Delta}$.

The next two inference rules are somewhat more involved. To state them, we first introduce some notations. Given an arbitrary set A of atoms, let P^A be a new atom for every atom $P \in A$. The vocabulary τ augmented with these symbols is denoted by τ_A. Given a propositional formula φ, φ^A denotes the formula obtained by replacing all occurrences of every atom $P \in A$ in φ by P^A. We call φ^A the *renaming* of φ with respect to A. For a set of propositional formulae F, F^A denotes $\{\varphi^A | \varphi \in F\}$. For a definition D, we denote by D^A the definition obtained by replacing every occurrence of every atom $P \in A$ by P^A.

The *left definition rule* allows inferring the falsity of a defined atom from a definition D and is therefore related to the second rule for extending a well-founded induction. Let D be a definition and $U \subseteq \tau_D^d$. Denote by N^U the set $\{\neg P^U \mid P \in U\}$ and let Γ and Δ be sequences of PC(ID)-formulae such that none of the renamed atoms P^U occurs in them. The left definition rule for every $P_i \in U$ is given as follows.

Left definition rule for $P_i \in U$ $\dfrac{\Gamma, N^U \rightarrow \neg(\varphi_1^U), \Delta; \ldots ; \Gamma, N^U \rightarrow \neg(\varphi_n^U), \Delta}{\Gamma, D, P_i \rightarrow \Delta}$,
where $P_1, \ldots, P_n$ are all atoms in U and $P_j \leftarrow \varphi_j$ is the rule for P_j in D for every $j \in [1, n]$.

We illustrate this inference rule with an example.

Example 3. Given a definition $D = \left\{ \begin{array}{c} P \leftarrow P \wedge \neg Q \\ Q \leftarrow Q \end{array} \right\}$,

– $U = \{P\}$, the left definition rule for $P \in U$ is

$$\frac{\Gamma, \neg P^U \rightarrow \neg(P^U \wedge \neg Q), \Delta}{\Gamma, D, P \rightarrow \Delta}.$$

– $U = \{P, Q\}$, the left definition rule for $P \in U$ is

$$\frac{\Gamma, \neg P^U, \neg Q^U \rightarrow \neg(P^U \wedge \neg Q^U), \Delta; \quad \Gamma, \neg P^U, \neg Q^U \rightarrow \neg Q^U, \Delta}{\Gamma, D, P \rightarrow \Delta}.$$

The *definition introduction rule* allows inferring the truth of a total definition from PC(ID)-formulae. Let D be a total definition and let $R = \tau_D^d$. Let Γ and Δ be sequences of PC(ID)-formulae such that for every $P \in R$, P^R does not occur in Γ or Δ. The definition introduction rule for D is given as follows.

Definition introduction rule for D

$$\frac{\Gamma, D^R \rightarrow P_1^R \equiv P_1, \Delta; \ldots; \Gamma, D^R \rightarrow P_n^R \equiv P_n, \Delta}{\Gamma \rightarrow D, \Delta},$$

where $P_1, \ldots, P_n$ are all defined atoms of D.

We illustrate this inference rule with an example.

Example 4. Given a definition $D = \left\{ \begin{array}{c} P \leftarrow O \\ Q \leftarrow Q \wedge \neg P \end{array} \right\}$, the definition introduction rule for D is

$$\frac{\Gamma, D^R \rightarrow P^R \equiv P, \Delta; \quad \Gamma, D^R \rightarrow Q^R \equiv Q, \Delta}{\Gamma \rightarrow D, \Delta}$$

where $D^R = \left\{ \begin{array}{c} P^R \leftarrow O \\ Q^R \leftarrow Q^R \wedge \neg P^R \end{array} \right\}$.

We denote by $\mathbf{LPC(ID)}^*$ the proof system containing all the inference rules of **LPC(ID)** except the definition introduction rule. We now come to the notion of an **LPC(ID)**-*proof* for a sequent, which is applicable for $\mathbf{LPC(ID)}^*$ as well.

Definition 1. *A* proof *in* **LPC(ID)** *or* **LPC(ID)**-proof *for a sequent S, is a tree T of sequents with root S. Moreover, each leaf of T must be an axiom and for each interior node S' there exists an inference rule such that S' is the consequence of that inference rule while the children of S' are precisely the premises of that inference rule. T is called a* proof tree *for S. A sequent S is called* provable *in* **LPC(ID)***, or* **LPC(ID)**-provable*, if there is an* **LPC(ID)***-proof for it.*

Example 5. Given a definition $D = \{ P \leftarrow O \}$ and $R = \tau_D^d$, the following is an **LPC(ID)**-proof for $O, P \rightarrow D$.

$$\text{definition introduction}\;\dfrac{\text{right }\vee\;\dfrac{\text{right weakening}\;\dfrac{\dfrac{\text{left weakening}\;\dfrac{\text{right definition}\;\dfrac{\text{left weakening}\;\dfrac{O \rightarrow O}{O, D^R \rightarrow O}}{O, D^R \rightarrow P^R}}{O, P, D^R \rightarrow P^R} \qquad \dfrac{P \rightarrow P}{O, P, D^R \rightarrow P}\;\text{left weakening}}{O, P, D^R \rightarrow P \wedge P^R}\;\text{right }\wedge}{O, P, D^R \rightarrow P \wedge P^R, \neg P \wedge \neg P^R}}{O, P, D^R \rightarrow P^R \equiv P}}{O, P \rightarrow D}$$

4 Main Results

In this section, we outline the proof of the soundness and completeness of **LPC(ID)**. We also show that cut-elimination is not possible for this deductive system.

4.1 Soundness

To prove the soundness of **LPC(ID)**, it is sufficient to prove that all axioms of **LPC(ID)** are valid and that every inference rule of **LPC(ID)** is sound, i.e. if all premises of an inference rule are valid then the consequence of that rule is valid. It is trivial to verify that the axioms are valid and that the structural and logical rules are sound. Hence, only the soundness of the right definition rule, the left definition rule and the definition introduction rule must be proven.

Lemma 1 (Soundness of the right definition rule). *Let D be a definition, containing the rule $P \leftarrow \varphi$. If $\models \Gamma, D \rightarrow \varphi, \Delta$, then $\models \Gamma, D \rightarrow P, \Delta$.*

Proof. Assume $\models \Gamma, D \rightarrow \varphi, \Delta$ but $\not\models \Gamma, D \rightarrow P, \Delta$. Then there exists a counter-model I for $\Gamma, D \rightarrow P, \Delta$. Therefore, $I \models \varphi$. Let $(I^\alpha)_{\alpha \leq \xi}$ be a terminal well-founded induction for D with limit $I^\xi = I$. This sequence is strictly increasing in precision, hence there is no $\alpha \leq \xi$ such that $\varphi^{I^\alpha} = \mathbf{f}$. As such, $P^I \neq \mathbf{f}$ and because I is two-valued, we can conclude $P^I = \mathbf{t}$, which is a contradiction to the assumption that I is a counter-model of $\Gamma, D \rightarrow P, \Delta$.

Lemma 2 (Soundness of the left definition rule). *Let D be a definition and $U = \{P_1, \ldots, P_n\}$ be a subset of τ_D^d. Denote for every $i \in [1, n]$ the body of the rule for P_i by φ_i. If $\models \Gamma, N^U \rightarrow \neg(\varphi_i^U), \Delta$ for every $i \in [1, n]$, then $\models \Gamma, D, P_i \rightarrow \Delta$ for all $i \in [1, n]$.*

Proof. Assume $\models \Gamma, N^U \rightarrow \neg(\varphi_i^U), \Delta$ for every $i \in [1, n]$, but $\not\models \Gamma, D, P_j \rightarrow \Delta$ for some $j \in [1, n]$. Then there exists a counter-model I for $\Gamma, D, P_j \rightarrow \Delta$. Let $(I^\alpha)_{\alpha \leq \xi}$ be a terminal well-founded induction for D with limit $I^\xi = I$. There exists a smallest $\alpha \leq \xi$ such that for some $i \in [1, n]$, $P_i^{I^\alpha} = \mathbf{u}$ and $P_i^{I^{\alpha+1}} = \mathbf{t}$. Consequently, for each $k \in [1, n]$, $P_k^{I^\alpha} = \mathbf{f}$ or $P_k^{I^\alpha} = \mathbf{u}$. Denote by I'^α the interpretation that assigns $P_k^{I^\alpha}$ to P_k^U for every $k \in [1, n]$ and coincides with I^α on all other atoms. So, for any formula φ not containing renamed atoms, $\varphi^{I^\alpha} = (\varphi^U)^{I'^\alpha}$. Denote by I' the interpretation that assigns $\mathbf{f}$ to every P_k^U, $k \in [1, n]$ and coincides with I on all other atoms. Remark that I' is more precise than I'^α. Also, $I' \models N_U$ and because neither Γ nor Δ contain any of the P_k^U, $I' \models \bigwedge \Gamma$ and $I' \not\models \bigvee \Delta$. Therefore, $I' \models \neg(\varphi_i^U)$ and because I' is more precise than I'^α, $(\varphi_i^U)^{I'^\alpha} \neq \mathbf{t}$. It follows that $\varphi_i^{I^\alpha} \neq \mathbf{t}$, which is a contradiction to $P_i^{I^{\alpha+1}} = \mathbf{t}$. Therefore $\models \Gamma, D, P_j \rightarrow \Delta$ for all $j \in [1, n]$.

Having the soundness of the left definition rule, we can explain the introduction of renaming formulae in the left definition rule. Our original idea for the left definition rule was of the form

$$\frac{\Gamma, \neg P_1, \ldots, \neg P_n \rightarrow \neg\varphi_1, \Delta; \ldots; \Gamma, \neg P_1, \ldots, \neg P_n \rightarrow \varphi_n, \Delta}{\Gamma, D, P_i \rightarrow \Delta} \tag{1}$$

where $\{P_1, \ldots, P_n\} \subseteq \tau_D^d$, P_i is an arbitrary atom in $\{P_1, \ldots, P_n\}$ and for every $j \in [1, n]$, $P_j \leftarrow \varphi_j$ is the rule for P_j in D.

However, this inference rule is not sound. For an arbitrary definition D and any defined atom P of D, $D \rightarrow \neg P$ can be inferred using this rule. We illustrate this with the next example.

Example 6. Consider the definition $D = \{ P \leftarrow \top \}$ and $\Gamma = P$. Since $P, \neg P \rightarrow \neg\top$, we can prove $D \rightarrow \neg P$ by using the inference rule (1), the right $\neg$ rule and the right contraction rule. However, for the same definition D and an empty sequence Γ, it is obvious that $D \rightarrow P$ by using the left definition rule, which shows that (1) is not a sound inference rule.

From the viewpoint of semantics, since the left definition rule corresponds to the second case of the well-founded induction, we have to adopt the approach of renaming to represent that the defined atoms of U are unknown in I^α and false in $I^{\alpha+1}$.

Lemma 3 (Soundness of the definition introduction rule). *Let D be a total definition and $R = \tau_D^d = \{P_1, \ldots, P_n\}$. If $\models \Gamma, D^R \rightarrow P_i^R \equiv P_i, \Delta$ for every $i \in [1, n]$, then $\models \Gamma \rightarrow D, \Delta$.*

Proof. Assume $\models \Gamma, D^R \rightarrow P_i^R \equiv P_i, \Delta$ for every $i \in [1, n]$, but $\not\models \Gamma \rightarrow D, \Delta$. Then there exists a τ-interpretation I such that $I \models \bigwedge \Gamma$, but $I \not\models D, I \not\models \bigvee \Delta$. Denote by I' the well-founded model of D extending $I|_{\tau_D^o}$. Because $I \not\models D$, there exists a defined atom P_j of D, such that $P_j^I \neq P_j^{I'}$. Let J be the $\tau \cup \tau_R$-interpretation such that J is the well-founded model of D^R extending I. Because Γ nor Δ contains an occurrence of an atom P_i^R, $J \models \bigwedge \Gamma$ and $J \not\models \bigvee \Delta$. Therefore, $J \models P_i^R \equiv P_i$ for every $i \in [1, n]$. Also, because D^R is obtained by renaming all defined atoms and none of the open atoms, it holds that $P_i^{I'} = (P_i^R)^J$ for every $i \in [1, n]$. Hence $P_j^I = (P_j^R)^J = P_j^{I'}$, which is a contradition. Therefore, $\models \Gamma \rightarrow D, \Delta$.

Note that the definition introduction rule is not sound if D is not total. We illustrate it with an example.

Example 7. Consider the definition $D = \{ P \leftarrow \neg P \}$, $R = \{P\}$ and Γ is the empty sequence. It is obvious that D^R is non-total. Thus, $\models D^R \rightarrow P^R \equiv P$ but $\not\models \rightarrow D$, which shows that the definition introduction rule is not sound when D is non-total.

By the fact that all inference rules in **LPC(ID)** except the definition introduction rule are sound with respect to non-total definitions and a straightforward induction, the soundness of **LPC(ID)*** and **LPC(ID)** can now be proven.

Theorem 1 (Soundness). *If a sequent $\Gamma \rightarrow \Delta$ is provable in* **LPC(ID)****, then $\models \Gamma \rightarrow \Delta$. If a sequent $\Gamma \rightarrow \Delta$ is provable in* **LPC(ID)** *and all definitions in Γ and Δ are total, then $\models \Gamma \rightarrow \Delta$.*

4.2 Completeness

In this subsection, we will prove that the deductive system **LPC(ID)** is complete, given that all definitions are total. The main difficulty in the completeness proof for **LPC(ID)** is to handle definitions in the sequents (We already know that the propositional calculus part of **LPC(ID)** is complete. See e.g. [16]).

The lemmas that follow show the completeness of the sequents of the form $\Gamma, D_1, \ldots, D_n \rightarrow \Delta$, where Γ and Δ are consistent sequences of literals.

Lemma 4. *Let D be a total definition and let Γ be a sequence of open literals of D, such that for every atom $Q \in \tau_D^o$ either $Q \in \Gamma$ or $\neg Q \in \Gamma$. Let L be a defined literal of D. If $\models \Gamma, D \rightarrow L$, then $\Gamma, D \rightarrow L$ is provable in* **LPC(ID)**.

Proof. Let I_O be the unique τ_D^o-interpretation such that $I_O \models \bigwedge \Gamma$ and let $(I^\alpha)_{\alpha \leq \xi}$ be a well-founded induction for D extending I_O. Denote by Δ^α the sequence of all defined literals L such that $L^{I^\alpha} = \mathbf{t}$. We prove that $\Gamma, D, \Delta^\alpha \rightarrow L$ is provable for all $L \in \Delta^{\alpha+1} \setminus \Delta^\alpha$. We distinguish between the case where $\Delta^{\alpha+1} \setminus \Delta^\alpha$ contains one positive literal and the case where it contains a set of negative literals.

- Assume $\Delta^{\alpha+1} \setminus \Delta^\alpha$ consists of one defined atom P and denote by φ the body of the rule for P in D. Because $(I^\alpha)_{\alpha \leq \xi}$ is a well-founded induction, $\models \Gamma, \Delta^\alpha \rightarrow \varphi$. Therefore, by the completeness of the propositional part of **LPC(ID)**, the sequent $\Gamma, \Delta^\alpha \rightarrow \varphi$ is provable. Hence, by left weakening and the right definition rule, $\Gamma, D, \Delta^\alpha \rightarrow P$ is provable.
- For the other case, assume $\Delta^{\alpha+1} \setminus \Delta^\alpha$ is a sequence of negative literals and denote it by $N = \{\neg P_1, \ldots, \neg P_n\}$. Denote $\{P_1, \ldots, P_n\}$ by U and denote the rule of each P_i in D by $P_i \leftarrow \varphi_i$. Because $(I^\alpha)_{\alpha \leq \xi}$ is a well-founded induction, $\models \Gamma, \Delta^\alpha, N \rightarrow \neg\varphi_i$ for every $i \in [1, n]$. Hence also $\models \Gamma, \Delta^\alpha, N^U \rightarrow \neg(\varphi_i^U)$. By the completeness of the propositional part of **LPC(ID)**, left weakening, the left definition rule and right $\neg$ rule, the sequent $\Gamma, \Delta^\alpha, D \rightarrow \neg P_i$ is provable for every $i \in [1, n]$.

Since D is total, it is obvious that the set of literals L for which $\models \Gamma, D \rightarrow L$ is exactly the set of all literals inferred during the well-founded induction $(I^\alpha)_{\alpha \leq \xi}$. Thus, by using the cut rule, it is easy to show by induction on α that if $\models \Gamma, D \rightarrow L$ for a defined literal L of D, the sequent $\Gamma, D \rightarrow L$ is provable in **LPC(ID)**.

Lemma 5. *Let D be a total definition and Γ an arbitrary consistent sequence of literals. If L is a defined literal of D such that $\models \Gamma, D \rightarrow L$, then $\Gamma, D \rightarrow L$ is provable in* **LPC(ID)**.

Proof. For every extension Γ' of Γ such that for every open atom P of D, either $P \in \Gamma'$ or $\neg P \in \Gamma'$, $\models \Gamma', D \rightarrow L$. Consider Γ'' as the sequence of open literals of D in Γ'. If $\models \Gamma'', D \rightarrow L$, then by the previous lemma, $\Gamma'', D \rightarrow L$ is provable in **LPC(ID)**, and by the left weakening rule, so is $\Gamma', D \rightarrow L$. If $\not\models \Gamma'', D \rightarrow L$, then by totality of D, $\models \Gamma'', D \rightarrow \neg L$ and hence, $\models \Gamma', D \rightarrow \neg L$. This means that $\Gamma' \wedge D$ is unsatisfiable, which implies that for some defined literal L' in Γ', $\models \Gamma'', D \rightarrow \neg L'$. By the previous lemma and the left weakening rule, $\Gamma', D \rightarrow \neg L'$ is provable in **LPC(ID)**. Given that $\Gamma', D \rightarrow L'$ is an axiom, we can use the left $\neg$ rule, the cut rule and the right weakening rule to show that $\Gamma', D \rightarrow L$ is provable in **LPC(ID)**. By the right $\neg$ rule and the cut rule, an **LPC(ID)**-proof for $\Gamma, D \rightarrow L$ can be constructed from the **LPC(ID)**-proofs of all $\Gamma', D \rightarrow L$.

Lemma 5 can be extended to the case where more than one definition is allowed in the antecedent of a sequent and more than one literal is allowed in the succedent.

Lemma 6. *Let $D_1, \dots, D_n$ be total definitions and let Γ, Δ be consistent sequences of literals. If $\models \Gamma, D_1, \dots, D_n \rightarrow \Delta$, then $\Gamma, D_1, \dots, D_n \rightarrow \Delta$ is provable.*

For lack of space, we omit the proof. It relies on the fact that by using the structural rules and the $\neg$ rules, $\Gamma, D_1, \dots, D_n \rightarrow \Delta$ can be proven from all valid sequents $\Gamma_i, D_i \rightarrow L_i$ where Γ_i is a sequence of open literals of D_i and L_i is a defined literal of D_i.

The remainder of the completeness proof follows the approach in [17]. We first introduce some terminology.

Definition 2. *A* reduction tree *for a sequent S is a tree T_S of sequents with root S such that the following two conditions hold.*

1. *For each non-leaf node S' of T_S there exists either a generalized logical rule or a definition introduction rule such that S' is the consequence of that inference rule while the children of S' are precisely the premises of that inference rule.*
2. *T_S is maximal, i.e. it is not the subtree of any tree that satisfies (1).*

Observe that the structural rules, the right definition rule and the left definition rule cannot be used in constructing a reduction tree. Remark that each leaf node of a reduction tree is either an axiom or a sequent of the form $\Gamma, D_1, \dots, D_n \rightarrow \Delta$ where Γ and Δ are sequences of atoms and $D_1, \dots, D_n$ are definitions.

It can easily be verified that the generalized logical rules and the definition introduction rule *preserve counter-models*, i.e., a counter-model for one of the premises of such an inference rule is also a counter-model to the consequence of that inference rule. Given this observation, the following property of reduction trees can also easily be verified.

Proposition 1. *For each sequent $S = \Gamma \rightarrow \Delta$, (a) there exists a reduction tree T_S, (b) if all leaf nodes of a reduction tree T_S are provable in* **LPC(ID)***, then the root sequent is provable in* **LPC(ID)***, and (c) a counter-model for a leaf node of T_S is a counter-model for the root.*

Theorem 2 (Completeness). *If $\models \Gamma \rightarrow \Delta$ and all definitions occurring in Γ and Δ are total, then $\Gamma \rightarrow \Delta$ is provable in* **LPC(ID)***.*

Proof. Let $\Gamma \rightarrow \Delta$ be a valid sequent and let T_S be a reduction tree with root $\Gamma \rightarrow \Delta$. Then by (c) of proposition 1, all leaves of T_S are valid. Since all leaves of T_S are of the form $\Pi, D_1, \dots, D_n \rightarrow \Sigma$, where Π and Σ are sequences of atoms and $D_1, \dots, D_n$ are definitions, it follows from lemma 6 that they are provable. Hence, by (b) of proposition 1, $\Gamma \rightarrow \Delta$ is provable.

4.3 Cut-Elimination

An important property of the propositional sequent calculus and many other formal proof systems is the cut-elimination, i.e. removing the cut rule from the system does not lead to incompleteness. However, for **LPC(ID)**, the cut-elimination does not hold. The following is a counter-example.

Example 8. Let D_1, respectively D_2 be the definitions $\{P \leftarrow O_1\}$, respectively $\{P \leftarrow O_2\}$. Then $D_1, D_2, O_1, \neg O_2 \rightarrow \bot$ can be proven in **LPC(ID)** as follows.

$$\dfrac{\text{weakening}\ \dfrac{\text{right definition}\ \dfrac{D_1, O_1 \rightarrow O_1}{D_1, O_1 \rightarrow P}}{D_1, D_2, O_1, \neg O_2 \rightarrow P} \qquad \dfrac{\dfrac{D_2, \neg O_2, \neg P^U \rightarrow \neg O_2}{D_2, \neg O_2, P \rightarrow \bot}\ \text{left definition}}{D_1, D_2, O_1, \neg O_2, P \rightarrow \bot}\ \text{weakening}}{D_1, D_2, O_1, \neg O_2 \rightarrow \bot}\ \text{cut}$$

It can be proven that the use of the cut rule can not be avoided for this example. The idea of the proof is as follows. In a proof tree for $D_1, D_2, O_1, \neg O_2 \rightarrow \bot$, both D_1 and D_2 must be used in an application of the right definition or left definition rule. For if not, all occurrences of them could be removed from the proof tree, yielding a correct proof tree with a non-valid root. Therefore, at some point in the proof tree, P must appear as an atomic formula in the antecedent or the succedent of a sequent. But in the root, P does only occur inside D_1 and D_2. So the other occurrences of P must have been removed by some rule. There are only four rules that remove formulae: the left, right and definition introduction rules, and the cut rule. But the left and right definition rules using D_1 or D_2 will introduce new occurrences of P, and the definition introduction rule will introduce a definition in the succedent, which can only be removed by the cut rule. Hence the cut rule is needed to remove the occurrences of P.

5 Conclusions, Related and Further Work

We presented a deductive system for the propositional fragment of FO(ID) which extends the sequent calculus for propositional logic. The main technical results are the soundness and completeness theorems of **LPC(ID)**. We also showed a counter-example to cut-elimination for PC(ID).

Related work is provided by Hagiya and Sakurai in [12]. They proposed to interpret a (stratified) logic program as iterated inductive definitions of Martin-Löf [15] and developed a proof theory which is sound with respect to the perfect model, and hence, the well-founded semantics of logic programming. A formal proof system based on tableau methods for analyzing computation for Answer Set Programming (ASP) was given as well by Gebser and Schaub [9]. As shown in [13], ASP is closely related to FO(ID). Their approach furnishes declarative and fine-grained instruments for characterizing operations as well as strategies of ASP-solvers and provides a uniform proof-theoretic framework for analyzing and comparing different algorithms, which is the first of its kind for ASP.

The first topic for future work, as mentioned in Section 1, is the development and implementation of a proof checker for MidL. This would require more study on resolution-based inference rules since MidL is basically an adaption of the DPLL-algorithm for SAT.

On the theoretical level, we plan to extend the deductive system for PC(ID) to FO(ID). As mentioned in the Introduction, a sound and complete proof system for FO(ID) does not exist. Therefore, we hope to build useful proof systems for FO(ID) that can solve a broad class of problems and investigate subclasses of FO(ID) for which they are complete.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook. Theory, Implementation and Applications.* Cambridge University Press, 2002.
2. R.J. Brachman and H.J. Levesque. Competence in Knowledge Representation. In *National Conference on Artificial Intelligence (AAAI'82)*, pages 189–192, 1982.
3. K.J. Compton. A deductive system for existential least fixpoint logic. *Journal of Logic and Computation*, 3(2):197–213, 1993.
4. K.J. Compton. Stratified least fixpoint logic. *Theoretical Computer Science*, 131(1):95–120, 1994.
5. M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. Fariñas del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence (JELIA'98)*, volume 1489 of *Lecture Notes in Artificial Intelligence*, pages 1–16, Schloss Dagstuhl, October 12-15 1998. Springer-Verlag.
6. M. Denecker, M. Bruynooghe, and V. Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654, October 2001.
7. M. Denecker and E. Ternovska. Inductive Situation Calculus. In *Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, pages 545–553, 2004.
8. M. Denecker and J. Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. Accepted for LPNMR07.
9. M. Gebser and T. Schaub. Tableau calculi for answer set programming. In S. Etalle and M. Truszczynski, editors, *Logic Programming, 22nd International Conference, ICLP 2006*, volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 2006.
10. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
11. G. Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.
12. M. Hagiya and T. Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2(1):59–77, 1984.
13. M. Mariën, D. Gilis, and M. Denecker. On the Relation Between ID-Logic and Answer Set Programming. In J.J. Alferes and J.A. Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2004.
14. M. Mariën, R. Mitra, M. Denecker, and M. Bruynooghe. Satisfiability checking for PC(ID). In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 565–579. Springer, 2005.
15. P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J.e. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, 1971.
16. M.E. Szabo, editor. *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Co., Amsterdam, 1969.
17. G. Takeuti. *Proof Theory.* North-Holland Publishing Co., Amsterdam.
18. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of Design, Automation and Test in Europe (DATE2003), Munich, Germany*, March 2003.

Modularity Aspects of Disjunctive Stable Models[⋆]

Tomi Janhunen[1], Emilia Oikarinen[1], Hans Tompits[2], and Stefan Woltran[2]

[1] Helsinki University of Technology
Department of Computer Science and Engineering
P.O. Box 5400, FI-02015 TKK, Finland
{Tomi.Janhunen,Emilia.Oikarinen}@tkk.fi
[2] Technische Universität Wien
Institut für Informationssysteme, 184/3
Favoritenstraße 9–11, A-1040 Vienna, Austria
{tompits,stefan}@kr.tuwien.ac.at

Abstract. Practically all programming languages used in software engineering allow to split a program into several modules. For fully declarative and nonmonotonic logic programming languages, however, the modular structure of programs is hard to realise, since the output of an entire program cannot in general be composed from the output of its component programs in a direct manner. In this paper, we consider these aspects for the stable-model semantics of disjunctive logic programs (DLPs). We define the notion of a *DLP-function*, where a well-defined input/output interface is provided, and establish a novel module theorem enabling a suitable compositional semantics for modules. The module theorem extends the well-known splitting-set theorem and allows also a generalisation of a shifting technique for splitting shared disjunctive rules among components.

1 Introduction

Practically all programming languages used in software engineering allow the user to split a program into several modules, which are composed by well-defined semantics over the modules' input/output interface. This not only helps towards a good programming style, but admits also to delegate coding tasks among several programmers, which then realise the specified input/output behaviour in terms of concrete modules.

The paradigm of *answer-set programming* (ASP), and in particular the case of *disjunctive logic programs* (DLPs) under the *stable-model semantics* [1], which we deal with herein, requires a fully declarative nonmonotonic semantics which is defined only over complete programs and therefore *prima facie* not directly applicable to modular programming. Due to this obstacle, the concept of a module has not raised much attention yet in nonmonotonic logic programming, and, except for a few dedicated papers [2,3,4], modules mostly appeared as a by-product in investigations of formal properties like stratification, splitting, or, more recently, in work on equivalence between

⋆ This work was partially supported by the Academy of Finland under project #211025 ("Advanced Constraint Programming Techniques for Large Structured Problems") and by the Austrian Science Foundation (FWF) under project P18019-N04 ("Formal Methods for Comparing and Optimizing Nonmonotonic Logic Programs").

programs [5,6,7,8]. The approach by Oikarinen and Janhunen [8] accommodates the module architecture discussed by Gaifman and Shapiro [2] for non-disjunctive programs and establishes a *module theorem* for stable models. This result indicates that the compositionality of stable models can be achieved in practice if positively interdependent atoms are not scattered among several modules.

In this paper, we deal with the formal underpinnings for modular programming in the context of disjunctive logic programs under the stable-model semantics. To begin with, we introduce the notion of a *DLP-function* which can roughly be described as a disjunctive logic program together with a well-defined input/output interface providing *input atoms*, *output atoms*, and *hidden* (*local*) *atoms*. In that, we follow Gelfond [9] who introduced *lp-functions* for specifying (partial) definitions of new relations in terms of old, known ones. This functional view of programs is also apparent if the latter are understood as *queries* over an input (i.e., a database). Indeed, several authors (like, e.g., Eiter, Gottlob, and Mannila [6]) introduce logic programs as an extension of Datalog, which poses no major problems with respect to stable semantics as long as a single program with a specified input/output behaviour is considered.

The latter point leads us to the second main issue addressed in our framework, viz. the question of a (semantically meaningful) method for the *composition* of modules. If the underlying semantics is inherently nonmonotonic, as is the case for stable semantics, this generates several problems, which were first studied in detail by Gaifman and Shapiro [2] for logic programs (without default negation) under minimal Herbrand models. As they observed, it is necessary to put certain syntactical restrictions on the programs to be composed, in order to make the semantics act accordingly. We shall follow their approach closely but extend it to programs permitting both default negation and disjunction. Notably, the problem of compositional semantics also arises in relation to the so-called *splitting-set theorem* [5,6,7], which aims at computing the stable models of a composed program by a suitable combination of the models of two programs which result from the split of the entire program.

After having the basic syntactical issues of DLP-functions laid out, we then define their model theory in terms of a generalisation of the stable-model semantics, where particular care is taken regarding the input of a DLP-function. The adequacy of our endeavour is witnessed by the main result of our paper, viz. the *module theorem*, providing the foundation for a fully compositional semantics: It shows how stable models of entire programs can be composed by joining together compatible stable models of their respective component programs. We round off our results with some applications of the module theorem. First, the module theorem readily extends the splitting-set theorem [5,6]. Second, it leads to a general *shifting principle* that can be used to simplify programs, i.e., to split shared disjunctive rules among components. Third, it gives rise to a notion of modular equivalence for DLP-functions that turns out to be a proper congruence relation supporting program substitutions.

2 The Class $\mathcal{D}$ of DLP-Functions

A *disjunctive rule* is an expression of the form

$$a_1 \lor \cdots \lor a_n \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_k, \tag{1}$$

where $n, m, k \geq 0$, and $a_1, \ldots, a_n$, $b_1, \ldots, b_m$, and $c_1, \ldots, c_k$ are propositional atoms. Since the order of atoms is considered insignificant, we write $A \leftarrow B, {\sim}C$ as a shorthand for rules of form (1), where $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_m\}$, and $C = \{c_1, \ldots, c_k\}$. The basic intuition behind (1) is that if each atom in the positive body B can be inferred and none of the atoms in the negative body C, then some atom in the head A can be inferred. When both B and C are empty, we have a *disjunctive fact*, written $A \leftarrow$. If A is empty, then we have a *constraint*, written $\bot \leftarrow B, {\sim}C$.

A *disjunctive logic program* (DLP) is conventionally formed as a set of disjunctive rules. Additionally, we want a distinguished *input and output interface* for each DLP. To this end, we extend a definition originally proposed by Gaifman and Shapiro [2] to the case of disjunctive programs.[1] Given a set R of disjunctive rules, we write $\mathrm{At}(R)$ for the *signature* of R, i.e., the set of (ground) atoms appearing in the rules of R. The set $\mathrm{Head}(R)$ consists of those elements of $\mathrm{At}(R)$ having head occurrences in R. This is exactly the set of atoms which are *defined* by the rules of R.

Definition 1. *A* DLP-function, *Π, is a quadruple $\langle R, I, O, H\rangle$, where I, O, and H are pairwise distinct sets of atoms, and R is a set of disjunctive rules such that*

$$\mathrm{At}(R) \subseteq I \cup O \cup H \textit{ and } \mathrm{Head}(R) \subseteq O \cup H.$$

The elements of I are called input atoms*, the elements of O* output atoms*, and the elements of H* hidden atoms.

Given a DLP-function $\Pi = \langle R, I, O, H\rangle$, we write, with a slight abuse of notation, $A \leftarrow B, {\sim}C \in \Pi$ to denote that the rule $A \leftarrow B, {\sim}C$ is contained in the set R. The atoms in $I \cup O$ are considered to be *visible* and hence accessible to other DLP-functions conjoined with Π; either to produce input for Π or to utilise the output of Π. On the other hand, the *hidden* atoms in H are used to formalise some auxiliary concepts of Π which may not make sense in the context of other DLP-functions but may save space substantially (see, e.g., Example 4.5 of Janhunen and Oikarinen [11]). The condition $\mathrm{Head}(R) \subseteq O \cup H$ ensures that a DLP-function may not interfere with its own input by defining input atoms of I in terms of its rules. In spite of this, the rules of Π may be conditioned by input atoms appearing in the bodies of rules. Following previous ideas [9,8], we define the signature $\mathrm{At}(\Pi)$ of a DLP-function $\Pi = \langle R, I, O, H\rangle$ as $I \cup O \cup H$.[2] For notational convenience, we distinguish the *visible* and *hidden parts* of $\mathrm{At}(\Pi)$ by setting $\mathrm{At}_{\mathrm{v}}(\Pi) = I \cup O$ and $\mathrm{At}_{\mathrm{h}}(\Pi) = H = \mathrm{At}(\Pi) \setminus \mathrm{At}_{\mathrm{v}}(\Pi)$, respectively. Additionally, $\mathrm{At}_{\mathrm{i}}(\Pi)$ and $\mathrm{At}_{\mathrm{o}}(\Pi)$ provide us a way of referring to the sets I and O of input and output atoms of Π, respectively. Lastly, for any set $S \subseteq \mathrm{At}(\Pi)$ of atoms, we denote the projections of S on $\mathrm{At}_{\mathrm{i}}(\Pi)$, $\mathrm{At}_{\mathrm{o}}(\Pi)$, $\mathrm{At}_{\mathrm{v}}(\Pi)$, and $\mathrm{At}_{\mathrm{h}}(\Pi)$ by S_{i}, S_{o}, S_{v}, and S_{h}, respectively.

In formal terms, a DLP-function $\Pi = \langle R, I, O, H\rangle$ provides a mapping from subsets of I to a set of subsets of $O \cup H$ in analogy to the method by Gelfond [9]. However, the exact definition of this mapping is deferred until Section 3 where the semantics of DLP-functions will be anchored. In the sequel, the (syntactic) class of DLP-functions is

[1] There are already similar approaches within the area of ASP [9,10,11,8].

[2] Consequently, the *length* of Π in symbols, denoted by $\|\Pi\|$, gives an upper bound for $|\mathrm{At}(\Pi)|$ which is important when one considers the computational cost of translating programs [10].

denoted by $\mathcal{D}$. It is assumed for the sake of simplicity that $\mathcal{D}$ spans over a fixed (at most denumerable) signature $\mathrm{At}(\mathcal{D})$[3] so that $\mathrm{At}(\Pi) \subseteq \mathrm{At}(\mathcal{D})$ holds for each DLP-function $\Pi \in \mathcal{D}$.

The composition of DLP-functions takes place as set out in Definition 2 below. We say that a DLP-function Π_1 *respects the hidden atoms* of another DLP-function Π_2 iff $\mathrm{At}(\Pi_1) \cap \mathrm{At_h}(\Pi_2) = \emptyset$, i.e., Π_1 does not use any atoms from $\mathrm{At_h}(\Pi_2)$.

Definition 2 (Gaifman and Shapiro [2]). *The* composition *of two DLP-functions Π_1 and Π_2 that* respect the hidden atoms of each other *is the DLP-function*

$$\Pi_1 \oplus \Pi_2 = \langle R_1 \cup R_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle. \qquad (2)$$

As discussed by Gaifman and Shapiro [2], program composition can be generalised for pairs of programs not respecting each other's hidden atoms. The treatment of atom types under Definition 2 is summarised in Table 1 where the intersections of the sets of the input, output, and hidden atoms of Π_1 and Π_2 are represented by the cells in the respective intersections of rows and columns (e.g., an atom $a \in O_1 \cap I_2$ becomes an output atom in $\Pi_1 \oplus \Pi_2$). Given that hidden atoms are mutually respected, ten cases arise in all. The consequences of Definition 2 should be intuitive to readers acquainted with the principles of object-oriented programming: (i) Although Π_1 and Π_2 must not share hidden atoms, they may share input atoms, i.e., $I_1 \cap I_2 \neq \emptyset$ is allowed. For now, the same can be stated about output atoms but this will be excluded by further conditions as done by Gaifman and Shapiro [2], where $O_1 \cap O_2 = \emptyset$ is assumed directly. (ii) An input atom of Π_1 becomes an output atom in $\Pi_1 \oplus \Pi_2$ if it appears as an output atom in Π_2, i.e., Π_2 provides the input for Π_1 in this setting. The input atoms of Π_2 are treated in a symmetric fashion. (iii) The hidden atoms of Π_1 and Π_2 retain their status in $\Pi_1 \oplus \Pi_2$.

Table 1. Division of atoms under $\oplus$ into input (i), output (o), or hidden (h) atoms

$\oplus$		I_2	O_2	H_2
		i	o	h
I_1	i	i	o	-
O_1	o	o	o	-
H_1	h	-	-	-

Given DLP-functions Π_1, Π_2, and Π_3 that pairwise respect the hidden atoms of each other, it holds that $\Pi_1 \oplus \Pi_2 \in \mathcal{D}$ (closure), $\Pi_1 \oplus \varnothing = \varnothing \oplus \Pi_1 = \Pi_1$ for the empty DLP-function $\varnothing = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ (identity), $\Pi_1 \oplus \Pi_2 = \Pi_2 \oplus \Pi_1$ (commutativity), and $\Pi_1 \oplus (\Pi_2 \oplus \Pi_3) = (\Pi_1 \oplus \Pi_2) \oplus \Pi_3$ (associativity). However, the notion of *modular equivalence* [8] is based on a more restrictive operator for program composition. The basic idea is to forbid positive dependencies between programs. Technically speaking, we define the *positive dependency graph* $\mathrm{DG}^+(\Pi) = \langle \mathrm{At}(\Pi), \leq_1 \rangle$ for each DLP-function Π in the standard way [12] using only positive dependencies: an atom $a \in A$ in the head of a rule $A \leftarrow B, {\sim}C \in \Pi$ depends positively on each $b \in B$ and each pair $\langle b, a \rangle$ belongs to the edge relation $\leq_1$ in $\mathrm{DG}^+(\Pi)$, i.e., $b \leq_1 a$ holds. The reflexive and transitive closure of $\leq_1$ gives rise to the dependency relation $\leq$ over $\mathrm{At}(\Pi)$.

A *strongly connected component* (SCC) S of $\mathrm{DG}^+(\Pi)$ is a maximal set $S \subseteq \mathrm{At}(\Pi)$ such that $b \leq a$ holds for every $a, b \in S$. Given that $\Pi_1 \oplus \Pi_2$ is defined, we say that Π_1 and Π_2 are *mutually dependent* iff $\mathrm{DG}^+(\Pi_1 \oplus \Pi_2)$ has a SCC S shared by Π_1 and Π_2 such that $S \cap \mathrm{At_o}(\Pi_1) \neq \emptyset$ and $S \cap \mathrm{At_o}(\Pi_2) \neq \emptyset$ [8].

[3] In practice, this set could be the set of all identifiers (names for propositions or similar objects).

Definition 3. *The* join, *$\Pi_1 \sqcup \Pi_2$, of two DLP-functions Π_1 and Π_2 is $\Pi_1 \oplus \Pi_2$, providing $\Pi_1 \oplus \Pi_2$ is defined and Π_1 and Π_2 are not mutually dependent.*

It is worth pointing out that $\mathrm{At_o}(\Pi_1) \cap \mathrm{At_o}(\Pi_2) = \emptyset$ follows in analogy to Gaifman and Shapiro [2] when $\Pi_1 \sqcup \Pi_2$ is defined. At first glance, this may appear rather restrictive, e.g., the head of a disjunctive rule $A \leftarrow B, \sim C$ cannot be shared by modules: either $A \subseteq \mathrm{At_o}(\Pi_1)$ or $A \subseteq \mathrm{At_o}(\Pi_2)$ must hold but not both. The general shifting technique to be presented in Section 5 allows us to circumvent this problem by viewing shared rules as syntactic sugar. Moreover, since Π_1 and Π_2 are not mutually dependent in $\Pi_1 \sqcup \Pi_2$, we have (i) $S \subseteq \mathrm{At_i}(\Pi_1 \sqcup \Pi_2)$, (ii) $S \subseteq \mathrm{At_o}(\Pi_1) \cup \mathrm{At_h}(\Pi_1)$, or (iii) $S \subseteq \mathrm{At_o}(\Pi_2) \cup \mathrm{At_h}(\Pi_2)$, for each SCC S of $\mathrm{DG}^+(\Pi_1 \oplus \Pi_2)$. The first covers joint input atoms $a \in \mathrm{At_i}(\Pi_1 \sqcup \Pi_2)$ which do not depend on other atoms by definition and which end up in singleton SCCs $\{a\}$.

The dependency relation $\leq$ lifts to the level of SCCs as follows: $S_1 \leq S_2$ iff there are $a_1 \in S_1$ and $a_2 \in S_2$ such that $a_1 \leq a_2$. In the sequel, a total order $S_1 < \cdots < S_k$ of the strongly connected components in $\mathrm{DG}^+(\Pi_1 \oplus \Pi_2)$ is also employed. Such an order $<$ is guaranteed to exist but it is not necessarily unique. E.g., the relative order of S_2 and S_3 can be freely chosen given that $S_1 \leq S_2 \leq S_4$ and $S_1 \leq S_3 \leq S_4$ hold for four components under $\leq$. Nevertheless $<$ is consistent with $\leq$, i.e., $S_i < S_j$ implies $S_j \not\leq S_i$ but either $S_i \leq S_j$ or $S_i \not\leq S_j$ may hold depending on $\leq$. Given that $\Pi_1 \sqcup \Pi_2$ is defined, we may project $S_1 < \cdots < S_k$ for Π_1 and Π_2 as follows. In case of Π_1, for instance, $S_{1,i} = S_i$, if $S_i \subseteq \mathrm{At_o}(\Pi_1) \cup \mathrm{At_h}(\Pi_1)$ or $S_i \subseteq \mathrm{At_i}(\Pi_1 \sqcup \Pi_2)$, and $S_{1,i} = S_i \cap \mathrm{At_i}(\Pi_1)$, if $S_i \subseteq \mathrm{At_o}(\Pi_2) \cup \mathrm{At_h}(\Pi_2)$. In the latter case, it is possible that $S_{1,i} = \emptyset$ or $S_{1,i}$ contains several input atoms of Π_1 which are independent of each other. This violates the definition of a SCC but we do not remove or split such exceptional components—also called SCCs in the sequel—to retain a uniform indexing scheme for the components of Π, Π_1, and Π_2. Thus, we have established the respective component structures $S_{1,1} < \cdots < S_{1,k}$ and $S_{2,1} < \cdots < S_{2,k}$ for Π_1 and Π_2.

3 Model Theory and Stable Semantics

Given any DLP-function Π, by an *interpretation*, M, for Π we understand a subset of $\mathrm{At}(\Pi)$. An atom $a \in \mathrm{At}(\Pi)$ is *true* under M (symbolically $M \models a$) iff $a \in M$, otherwise *false* under M. For a negative literal $\sim a$, we define $M \models \sim a$ iff $M \not\models a$. A set L of literals is satisfied by M (denoted by $M \models L$) iff $M \models l$, for every $l \in L$. We also define $M \models \bigvee L$, providing $M \models l$ for some $l \in L$.

To begin with, we cover DLP-functions with a pure classical semantics, which treats disjunctive rules as classical implications.

Definition 4. *An interpretation $M \subseteq \mathrm{At}(\Pi)$ is a* (classical) model *of a DLP-function $\Pi = \langle R, I, O, H \rangle$, denoted $M \models \Pi$, iff $M \models R$, i.e., for every rule $A \leftarrow B, \sim C \in R$,*

$$M \models B \cup \sim C \text{ implies } M \models \bigvee A.$$

The set of all classical models of Π is denoted by $\mathrm{CM}(\Pi)$.

Classical models provide a suitable level of abstraction to address the role of input atoms in DLP-functions. Given a DLP-function Π and an interpretation $M \subseteq \mathrm{At}(\Pi)$, the projection M_i can be viewed as the actual input for Π which may (or may not) produce the respective output M_o, depending on the semantics assigned to Π. The treatment of input atoms in the sequel will be based on *partial evaluation*: the idea is to pre-interpret input atoms appearing in Π with respect to M_i.

Definition 5. *For a DLP-function $\Pi = \langle R, I, O, H\rangle$ and an actual input $M_\mathrm{i} \subseteq I$ for Π, the* instantiation of Π with respect to M_i, *denoted by Π/M_i, is the quadruple $\langle R', \emptyset, I \cup O, H\rangle$ where R' consists of the following rules:*

1. *the rule $A \leftarrow (B \setminus I), {\sim}(C \setminus I)$, for each rule $A \leftarrow B, {\sim}C \in \Pi$ such that $M_\mathrm{i} \models B_\mathrm{i} \cup {\sim}C_\mathrm{i}$,*
2. *the fact $a \leftarrow$, for each atom $a \in M_\mathrm{i}$, and*
3. *the constraint $\bot \leftarrow a$, for each atom $a \in I \setminus M_\mathrm{i}$.*

The rules in the first item are free of input atoms since $A \cap I = \emptyset$ holds for each rule $A \leftarrow B, {\sim}C$ in R by Definition 1. The latter two items list rules that record the truth values of input atoms of Π in the resulting program Π/M_i. The reduct Π/M_i is a DLP-function without input whereas the visibility of atoms is not affected by instantiation.

Proposition 1. *Let Π be a DLP-function and $M \subseteq \mathrm{At}(\Pi)$ an interpretation that defines an actual input $M_\mathrm{i} \subseteq \mathrm{At}_\mathrm{i}(\Pi)$ for Π. Then, for all interpretations $N \subseteq \mathrm{At}(\Pi)$,*

$$N \models \Pi \text{ and } N_\mathrm{i} = M_\mathrm{i} \iff N \models \Pi/M_\mathrm{i}.$$

Thus, the input reduction, as given in Definition 5, is fully compatible with classical semantics and we may characterise the semantic operator CM by pointing out the fact that $\mathrm{CM}(\Pi) = \bigcup_{M_\mathrm{i} \subseteq I} \mathrm{CM}(\Pi/M_\mathrm{i})$. Handling input is slightly more complicated in the case of minimal models but Lifschitz's *parallel circumscription* [13] provides us a standard approach to deal with it. The rough idea is to keep the interpretation of input atoms fixed while minimising, i.e., falsifying others as far as possible.

Definition 6. *Let Π be a DLP-function and $F \subseteq \mathrm{At}(\Pi)$ a set of atoms assumed to have fixed truth values. A model $M \subseteq \mathrm{At}(\Pi)$ of Π is* F-minimal *iff there is no model N of Π such that $N \cap F = M \cap F$ and $N \subset M$.*

The set of F-minimal models of Π is denoted by $\mathrm{MM}_F(\Pi)$. In the sequel, we treat input atoms by stipulating $\mathrm{At}_\mathrm{i}(\Pi)$-minimality of models of Π. Then, the condition $N \cap F = M \cap F$ in Definition 6 becomes equivalent to $N_\mathrm{i} = M_\mathrm{i}$. Using this idea, Proposition 1 lifts for minimal models as follows. Recall that $\mathrm{At}_\mathrm{i}(\Pi/M_\mathrm{i}) = \emptyset$.

Proposition 2. *Let Π be a DLP-function and $M \subseteq \mathrm{At}(\Pi)$ an interpretation that defines an actual input $M_\mathrm{i} \subseteq \mathrm{At}_\mathrm{i}(\Pi)$ for Π. Then, for all interpretations $N \subseteq \mathrm{At}(\Pi)$,*

$$N \in \mathrm{MM}_{\mathrm{At}_\mathrm{i}(\Pi)}(\Pi) \text{ and } N_\mathrm{i} = M_\mathrm{i} \iff N \in \mathrm{MM}_\emptyset(\Pi/M_\mathrm{i}).$$

The set $\mathrm{MM}_{\mathrm{At}_\mathrm{i}(\Pi)}(\Pi)$ of models is sufficient to determine the semantics of a *positive* DLP-function, i.e., whose rules are of the form $A \leftarrow B$ where $A \neq \emptyset$ and only B may involve atoms from $\mathrm{At}_\mathrm{i}(\Pi)$. Therefore, due to non-empty heads of rules, a positive DLP Π is guaranteed to possess classical models since, e.g., $\mathrm{At}(\Pi) \models \Pi$, and thus also $\mathrm{At}_\mathrm{i}(\Pi)$-minimal models. To cover arbitrary DLP-functions, we interpret negative body literals in the way proposed by Gelfond and Lifschitz [1].

Definition 7. *Given a DLP-function $\Pi = \langle R, I, O, H\rangle$ and an interpretation $M \subseteq \mathrm{At}(\Pi)$, the* Gelfond-Lifschitz reduct of Π with respect to M *is the positive DLP-function*

$$\Pi^M = \langle \{A \leftarrow B \mid A \leftarrow B, {\sim}C \in \Pi,\ A \neq \emptyset,\ \text{and}\ M \models {\sim}C\},\ I,\ O,\ H\rangle. \quad (3)$$

Definition 8. *An interpretation $M \subseteq \mathrm{At}(\Pi)$ is a* stable model *of a DLP-function Π iff $M \in \mathrm{MM}_{\mathrm{At_i}(\Pi)}(\Pi^M)$ and $M \models \mathrm{CR}(\Pi)$, where $\mathrm{CR}(\Pi)$ is the set of constraints $\bot \leftarrow B, {\sim}C \in \Pi$.*

Hidden atoms play no special role in Definition 8 and their status will be clarified in Section 5 when the notion of *modular equivalence* is introduced. Definition 8 gives rise to the respective semantic operator $\mathrm{SM} : \mathcal{D} \rightarrow \mathbf{2}^{\mathbf{2}^{\mathrm{At}(\mathcal{D})}}$ for DLP-functions:

$$\mathrm{SM}(\Pi) = \{M \subseteq \mathrm{At}(\Pi) \mid M \in \mathrm{MM}_{\mathrm{At_i}(\Pi)}(\Pi^M) \text{ and } M \models \mathrm{CR}(\Pi)\}. \quad (4)$$

As a consequence of Proposition 2, a stable model M of Π is a minimal model of $\Pi^M/M_\mathrm{i} = (\Pi/M_\mathrm{i})^M$ which enables one to dismiss $\mathrm{At_i}(\Pi)$-minimality if desirable.

Example 1. Consider a DLP-function

$$\Pi = \langle \{a \vee b \leftarrow {\sim}c;\ a \leftarrow c, {\sim}b;\ b \leftarrow c, {\sim}a\}, \{c\}, \{a, b\}, \emptyset\rangle,$$

which has four stable models, $M_1 = \{a\}$, $M_2 = \{b\}$, $M_3 = \{a, c\}$, and $M_4 = \{b, c\}$, which are minimal models of the respective reducts of Π:

$$\Pi^{M_1}/(M_1)_\mathrm{i} = \Pi^{M_2}/(M_2)_\mathrm{i} = \langle \{a \vee b \leftarrow;\ \bot \leftarrow c\}, \emptyset, \{a, b, c\}, \emptyset\rangle,$$
$$\Pi^{M_3}/(M_3)_\mathrm{i} = \langle \{a \leftarrow;\ c \leftarrow\}, \emptyset, \{a, b, c\}, \emptyset\rangle, \text{ and}$$
$$\Pi^{M_4}/(M_4)_\mathrm{i} = \langle \{b \leftarrow;\ c \leftarrow\}, \emptyset, \{a, b, c\}, \emptyset\rangle.$$

□

An immediate observation is that we loose the general antichain property of stable models when input signatures are introduced. For instance, we have $M_1 \subset M_3$ and $M_2 \subset M_4$ in Example 1. However, since the interpretation of input atoms is fixed by the semantics, we perceive antichains *locally*, i.e., the set of stable models $\{N \in \mathrm{SM}(\Pi) \mid N_\mathrm{i} = M_\mathrm{i}\}$ forms an antichain, for each input $M_\mathrm{i} \subseteq \mathrm{At_i}(\Pi)$. In Example 1, the sets associated with actual inputs $\emptyset$ and $\{c\}$ are $\{M_1, M_2\}$ and $\{M_3, M_4\}$, respectively.

4 Module Theorem for DLP-Functions

Our next objective is to show that stable semantics allows substitutions under joins of programs as defined in Section 2. Given two DLP-functions Π_1 and Π_2, we say that interpretations $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$ are *mutually compatible* (*with respect to Π_1 and Π_2*), or just *compatible* for short, iff $M_1 \cap \mathrm{At_v}(\Pi_1) = M_2 \cap \mathrm{At_v}(\Pi_2)$, i.e., M_1 and M_2 agree about the truth values of their joint visible atoms. A quick inspection of Table 1 reveals the three cases that may arise if the join $\Pi = \Pi_1 \sqcup \Pi_2$ is defined and joint *output atoms* for Π_1 and Π_2 are disallowed: There are shared input atoms in $\mathrm{At_i}(\Pi) = \mathrm{At_i}(\Pi_1) \cap \mathrm{At_i}(\Pi_2)$ and atoms in $\mathrm{At_o}(\Pi_1) \cap \mathrm{At_i}(\Pi_2)$ and $\mathrm{At_i}(\Pi_1) \cap \mathrm{At_o}(\Pi_2)$ that are output atoms in one program and input atoms in the other program. Recall that according to Definition 3 such atoms end up in $\mathrm{At_o}(\Pi)$ when $\Pi_1 \sqcup \Pi_2$ is formed. Our first modularity result deals with the classical semantics of DLP-functions.

Proposition 3. *Let Π_1 and Π_2 be two positive DLP-functions with the respective input signatures $\mathrm{At_i}(\Pi_1)$ and $\mathrm{At_i}(\Pi_2)$ so that $\Pi_1 \sqcup \Pi_2$ is defined. Then, for any mutually compatible interpretations $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$,*

$$M_1 \cup M_2 \models \Pi_1 \sqcup \Pi_2 \iff M_1 \models \Pi_1 \text{ and } M_2 \models \Pi_2. \tag{5}$$

The case of minimal or stable models, respectively, is much more elaborate. The proof of Theorem 1 (see below) is based on *cumulative projections* defined for a join $\Pi_1 \sqcup \Pi_2$ of DLP-functions Π_1 and Π_2 and a pair of compatible interpretations $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$. It is clear that $M_1 = M \cap \mathrm{At}(\Pi_1)$ and $M_2 = M \cap \mathrm{At}(\Pi_2)$ hold for $M = M_1 \cup M_2$ in this setting. Next, we use a total order $S_1 < \cdots < S_k$ of the SCCs in $\mathrm{DG}^+(\Pi_1 \oplus \Pi_2)$ to define an increasing sequence of interpretations

$$N^j = N_{\mathrm{i}} \cup (N \cap (\textstyle\bigcup_{i=1}^{j} S_i)), \tag{6}$$

for each interpretation $N \in \{M, M_1, M_2\}$ and $0 \leq j \leq k$. Furthermore, let $\Pi = \langle R, I, O, H\rangle = \Pi_1 \sqcup \Pi_2$. The relative complement $\overline{M} = \mathrm{At}(\Pi) \setminus M$ contains atoms *false* under M and we may associate a set $R[S_i]$ of rules with each SCC S_i using $\overline{M}$:

$$R[S_i] = \{(A \cap S_i) \leftarrow B \mid A \leftarrow B \in R,\ A \cap S_i \neq \emptyset, \text{ and } A \setminus S_i \subseteq \overline{M}\}. \tag{7}$$

For each rule $A \leftarrow B \in R$, the reduced rule $(A \cap S_i) \leftarrow B$ is the contribution of $A \leftarrow B$ for the component S_i in case $\bigvee(A \setminus S_i)$ is false under M, i.e., $\bigvee A$ is not eventually satisfied by some other component of Π; note that $M \models \Pi$ will be assumed in the sequel. Although $R[S_i]$ depends on M, we omit M in the notation for the sake of conciseness. For each $0 \leq j \leq k$, we may now collect rules associated with the first j components and form a DLP-function with the same signature as Π:

$$\Pi^j = \langle \textstyle\bigcup_{i=1}^{j} R[S_i], I, O, H\rangle. \tag{8}$$

This implies that non-input atoms in $\bigcup_{i=j+1}^{k} S_i$ are false under interpretations defined by (6). Since each rule of Π is either contained in Π_1 or Π_2, we may use $\overline{M_1} = \mathrm{At}(\Pi_1) \setminus M_1$ and $\overline{M_2} = \mathrm{At}(\Pi_2) \setminus M_2$ to define Π_1^j and Π_2^j analogously, using (7) and (8) for Π_1 and Π_2, respectively. It follows that $\Pi_1^j \sqcup \Pi_2^j$ is defined and $\Pi^j = \Pi_1^j \sqcup \Pi_2^j$ holds for every $0 \leq j \leq k$ due to the compatibility of M_1^j and M_2^j and the fact that $\Pi = \Pi_1 \sqcup \Pi_2$. Moreover, it is easy to inspect from the equations above that, by definition, $M^{j-1} \subseteq M^j$ and the rules of Π^{j-1} are contained in Π^j, for every $0 < j \leq k$.

Finally, we may accommodate the definitions from above to the case of a single DLP-function by substituting Π for Π_1 and $\varnothing$ for Π_2. Then, $\mathrm{DG}^+(\Pi)$ is partitioned into strongly connected components $S_1 < \cdots < S_k$ of Π and the construction of cumulative projections is applicable to an interpretation $M \subseteq \mathrm{At}(\Pi)$, giving rise to interpretations M^j and DLP-functions Π^j for each $0 \leq j \leq k$. Lemmas 1 and 2 deal with a structure of this kind associated with Π and describe how the satisfaction of rules and $\mathrm{At_i}(\Pi)$-minimality are conveyed under cumulative projections.

Lemma 1. *Let Π be a positive DLP-function with an input signature $\mathrm{At_i}(\Pi)$ and strongly connected components $S_1 < \cdots < S_k$. Given a model $M \subseteq \mathrm{At}(\Pi)$ for Π, the following hold for the cumulative projections M^j and Π^j, with $0 \leq j \leq k$:*

1. *For every* $0 \leq j \leq k$, $M^j \models \Pi^j$.
2. *If* $N^j \models \Pi^j$, *for some interpretation* $N^j \subseteq M^j$ *of* Π^j, *where* $j > 0$, *then* $N^{j-1} \models \Pi^{j-1}$, *for the interpretation* $N^{j-1} = N^j \setminus S_j$ *of* Π^{j-1}.
3. *If* M^j *is an* $\mathrm{At_i}(\Pi)$*-minimal model of* Π^j, *for* $j > 0$, *then* M^{j-1} *is an* $\mathrm{At_i}(\Pi)$*-minimal model of* Π^{j-1}.

Example 2. To demonstrate cumulative projections in a practical setting, let us analyse a DLP-function $\Pi = \langle R, \emptyset, \{a,b,c,d,e\}, \emptyset \rangle$, where R contains the following rules:

$$\begin{array}{ll} a \vee b \leftarrow; & d \leftarrow c; \\ a \leftarrow b; & e \leftarrow d; \\ b \leftarrow a; & d \leftarrow e; \\ a \leftarrow c; & c \vee d \vee e \leftarrow a, b. \end{array}$$

The SCCs of Π are $S_1 = \{a,b,c\}$ and $S_2 = \{d,e\}$ with $S_1 < S_2$. The classical models of Π are $M = \{a,b,d,e\}$ and $N = \{a,b,c,d,e\}$. Given M, Π^1 and Π^2 have the respective sets of rules $R[S_1] = \{a \vee b \leftarrow;\ a \leftarrow b;\ b \leftarrow a;\ a \leftarrow c\}$ and $R[S_1] \cup R[S_2]$ where $R[S_2] = \{d \leftarrow c;\ e \leftarrow d;\ d \leftarrow e;\ d \vee e \leftarrow a, b\}$. According to (6), we have $M^0 = \emptyset$, $M^1 = \{a,b\}$, and $M^2 = M$. Then, e.g., $M^1 \models \Pi^1$ and $M^2 \models \Pi^2$ by the first item of Lemma 1. Since M^2 is an $\emptyset$-minimal model of Π^2, the last item of Lemma 1 implies that M^1 is an $\emptyset$-minimal model of Π^1. □

Lemma 2. *Let* Π *be a positive DLP-function with an input signature* $\mathrm{At_i}(\Pi)$ *and strongly connected components* $S_1 < \cdots < S_k$. *Then, an interpretation* $M \subseteq \mathrm{At}(\Pi)$ *of* Π *is an* $\mathrm{At_i}(\Pi)$*-minimal model of* Π *iff* M *is an* $\mathrm{At_i}(\Pi)$*-minimal model of* Π^k.

Example 3. For Π from Example 2, the rule $d \vee e \leftarrow a, b$ forms the only difference between Π^2 and Π but this is insignificant: M is also an $\emptyset$-minimal model of Π. □

Proposition 4. *Let* Π_1 *and* Π_2 *be two* positive *DLP-functions with the respective input signatures* $\mathrm{At_i}(\Pi_1)$ *and* $\mathrm{At_i}(\Pi_2)$ *so that* $\Pi_1 \sqcup \Pi_2$ *is defined. Then, for any mutually compatible models* $M_1 \subseteq \mathrm{At}(\Pi_1)$ *and* $M_2 \subseteq \mathrm{At}(\Pi_2)$ *of* Π_1 *and* Π_2*, respectively,*

$M_1 \cup M_2$ *is* $\mathrm{At_i}(\Pi_1 \sqcup \Pi_2)$*-minimal* $\iff$ M_1 *is* $\mathrm{At_i}(\Pi_1)$*-minimal and* M_2 *is* $\mathrm{At_i}(\Pi_2)$*-minimal.*

Proof sketch. The proof of this result proceeds by induction on the cumulative projections M^j, M_1^j, and M_2^j induced by the SCCs $S_1 < \cdots < S_k$ of $\mathrm{DG}^+(\Pi_1 \sqcup \Pi_2)$, i.e., M^j is shown to be an $\mathrm{At_i}(\Pi)$-minimal model of Π^j iff M_1^j is an $\mathrm{At_i}(\Pi_1)$-minimal model of Π_1 and M_2^j is an $\mathrm{At_i}(\Pi_2)$-minimal model of Π_2, where Π^j, Π_1^j, and Π_2^j, for $0 \leq j \leq k$, are determined by (7) and (8) when applied to Π, Π_1, and Π_2. Lemma 2 closes the gap between the $\mathrm{At_i}(\Pi)$-minimality of $M^k = M$ as a model of Π^k from (8) with $j = k$ and as that of Π. The same can be stated about $M_1^k = M_1$ and $M_2^k = M_2$ but in terms of the respective projections $S_{i,1} < \cdots < S_{i,k}$ obtained for $i \in \{1,2\}$. □

Lemma 3. *Let* Π_1 *and* Π_2 *be two DLP-functions with the respective input signatures* $\mathrm{At_i}(\Pi_1)$ *and* $\mathrm{At_i}(\Pi_2)$ *so that* $\Pi = \Pi_1 \sqcup \Pi_2$ *is defined. Then, also* $\Pi_1^{M_1} \sqcup \Pi_2^{M_2}$ *is defined for any mutually compatible interpretations* $M_1 \subseteq \mathrm{At}(\Pi_1)$ *and* $M_2 \subseteq \mathrm{At}(\Pi_2)$*, and* $\Pi^M = \Pi_1^{M_1} \sqcup \Pi_2^{M_2}$ *holds for their union* $M = M_1 \cup M_2$.

Theorem 1 (Module Theorem). *Let Π_1 and Π_2 be two DLP-functions with the respective input signatures $\mathrm{At_i}(\Pi_1)$ and $\mathrm{At_i}(\Pi_2)$ so that $\Pi_1 \sqcup \Pi_2$ is defined. Then, for any mutually compatible interpretations $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$,*

$$M_1 \cup M_2 \in \mathrm{SM}(\Pi_1 \sqcup \Pi_2) \iff M_1 \in \mathrm{SM}(\Pi_1) \textit{ and } M_2 \in \mathrm{SM}(\Pi_2).$$

Proof. Let $M_1 \subseteq \mathrm{At}(\Pi_1)$ and $M_2 \subseteq \mathrm{At}(\Pi_2)$ be compatible interpretations and $M = M_1 \cup M_2$. Due to compatibility, we can recover $M_1 = M \cap \mathrm{At}(\Pi_1)$ and $M_2 = M \cap \mathrm{At}(\Pi_2)$ from M. Additionally, we have $\mathrm{CR}(\Pi) = \mathrm{CR}(\Pi_1) \cup \mathrm{CR}(\Pi_2)$ and $\Pi^M = \Pi_1^{M_1} \sqcup \Pi_2^{M_2}$ is defined by Lemma 3. Now, $M \in \mathrm{SM}(\Pi)$ iff

$$M \text{ is an } \mathrm{At_i}(\Pi)\text{-minimal model of } \Pi^M \text{ and } M \models \mathrm{CR}(\Pi). \tag{9}$$

By Proposition 4, we get that (9) holds iff (i) M_1 is an $\mathrm{At_i}(\Pi_1)$-minimal model of $\Pi_1^{M_1}$ and $M_1 \models \mathrm{CR}(\Pi_1)$, and (ii) M_2 is an $\mathrm{At_i}(\Pi_2)$-minimal model of $\Pi_2^{M_2}$ and $M_2 \models \mathrm{CR}(\Pi_2)$. Thus, $M \in \mathrm{SM}(\Pi)$ iff $M_1 \in \mathrm{SM}(\Pi)$ and $M_2 \in \mathrm{SM}(\Pi)$. □

The moral of Theorem 1 and Definition 3 is that stable semantics supports modularisation as long as positive dependencies remain within program modules. The proof of the theorem reveals the fact that such modules may involve several strongly connected components. Splitting them into further modules is basically pre-empted by hidden atoms which cannot be placed in separate modules. Theorem 1 can be easily extended for DLP-functions consisting of more than two modules. In view of this, we say that a sequence $M_1, \ldots, M_n$ of stable models for modules $\Pi_1, \ldots, \Pi_n$, respectively, is *compatible*, iff M_i and M_j are pairwise compatible, for all $1 \leq i, j \leq n$.

Corollary 1. *Let $\Pi_1, \ldots, \Pi_n$ be a sequence of DLP-functions such that $\Pi_1 \sqcup \cdots \sqcup \Pi_n$ is defined. Then, for all compatible sequences $M_1, \ldots, M_n$ of interpretations,*

$$\textstyle\bigcup_{i=1}^{n} M_i \in \mathrm{SM}(\Pi_1 \sqcup \cdots \sqcup \Pi_n) \iff M_i \in \mathrm{SM}(\Pi_i), \textit{ for all } 1 \leq i \leq n.$$

5 Applications

In this section, we demonstrate the applicability of Theorem 1 on three issues, viz. splitting DLP-functions, shifting disjunctions, and checking equivalence.

Splitting DLP-Functions. Theorem 1 is strictly stronger than the splitting-set theorem [5]. Given a DLP-function of form $\Pi = \langle R, \emptyset, O, \emptyset \rangle$ (which is essentially an "ordinary" DLP), a *splitting set* $U \subseteq O$ for Π satisfies, for each $A \leftarrow B, {\sim}C \in R$, $A \cup B \cup C \subseteq U$, whenever $A \cap U \neq \emptyset$. Given a splitting set U for Π, the *bottom*, $\mathrm{b}_U(R)$, of R with respect to U contains all rules $A \leftarrow B, {\sim}C \in R$ such that $A \cup B \cup C \subseteq U$, whereas the *top*, $\mathrm{t}_U(R)$, of R is $R \setminus \mathrm{b}_U(R)$. Thus, we may define $\Pi = \Pi_B \sqcup \Pi_T$, where $\Pi_B = \langle \mathrm{b}_U(R), \emptyset, U, \emptyset \rangle$ and $\langle \mathrm{t}_U(R), U, O \setminus U, \emptyset \rangle$. Then, Theorem 1 implies for any interpretation $M \subseteq \mathrm{At}(\Pi) = O$ that $M \cap U \in \mathrm{SM}(\Pi_B)$ and $M \in \mathrm{SM}(\Pi_T)$ iff $\langle M \cap U, M \setminus U \rangle$ is a *solution* for Π with respect to U, i.e., M is a stable model of Π. On the other hand, as demonstrated in previous work [8], the splitting-set theorem can be applied to DLP-functions like $\langle \{a \leftarrow {\sim}b;\ b \leftarrow {\sim}a\}, \emptyset, \{a, b\}, \emptyset \rangle$ only in a trivial way, i.e., for $U = \emptyset$ or $U = \{a, b\}$. In contrast, Theorem 1 applies to the preceding DLP-function, i.e., $\langle \{a \leftarrow {\sim}b\}, \{b\}, \{a\}, \emptyset \rangle \sqcup \langle \{b \leftarrow {\sim}a\}, \{a\}, \{b\}, \emptyset \rangle$ is defined.

Shifting Disjunctions. A further application of our module theorem results in a general shifting principle, defined as follows.

Definition 9. *Let $\Pi = \langle R, I, O, H\rangle$ be a DLP-function with strongly connected components $S_1 < \cdots < S_k$. The* general shifting *of Π is the DLP-function* $\mathrm{GSH}(\Pi) = \langle R', I, O, H\rangle$, *where R' is*

$$\{(A \cap S_i) \leftarrow B, {\sim}C, {\sim}(A \setminus S_i) \mid A \leftarrow B, {\sim}C \in \Pi,\ 1 \leq i \leq k, \text{ and } A \cap S_i \neq \emptyset\}.$$

This is a proper generalisation of the *local shifting* transformation [14] which is not applicable to the program Π given below because of head cycles involved.

Example 4. Consider Π with the following rules:

$$\begin{array}{ll} a \vee b \vee c \vee d \leftarrow; & \\ a \leftarrow b; & c \leftarrow d; \\ b \leftarrow a; & d \leftarrow c. \end{array}$$

For $\mathrm{GSH}(\Pi)$, the first rule is replaced by $a \vee b \leftarrow {\sim}c, {\sim}d$ and $c \vee d \leftarrow {\sim}a, {\sim}b$. It is easy to verify that both Π and $\mathrm{GSH}(\Pi)$ have $\{a, b\}$ and $\{c, d\}$ as their stable models. □

Theorem 2. *For any DLP-function $\Pi = \langle R, I, O, H\rangle$,* $\mathrm{SM}(\Pi) = \mathrm{SM}(\mathrm{GSH}(\Pi))$.

Proof. Let $S_1 < \cdots < S_k$ be the strongly connected components of Π and $M \subseteq \mathrm{At}(\Pi) = I \cup O \cup H$ an interpretation. By applying the construction of cumulative projections for both Π^M and $\mathrm{GSH}(\Pi)^M$, we obtain

$$\begin{aligned}(\Pi^M)^k = \{(A \cap S_i) \leftarrow B \mid A \leftarrow B, {\sim}C \in \Pi,\\ M \cap C = \emptyset, 1 \leq i \leq k, A \cap S_i \neq \emptyset, \text{and } A \setminus S_i \subseteq \overline{M}\},\end{aligned}$$

which coincides with $(\mathrm{GSH}(\Pi)^M)^k$. It follows by Lemma 2 that M is an $\mathrm{At_i}(\Pi)$-minimal model of Π^M iff M is an $\mathrm{At_i}(\Pi)$-minimal model of $\mathrm{GSH}(\Pi)^M$. □

Theorem 2 provides us a technique to split disjunctive rules among components that share them in order to get joins of components defined.

Example 5. For the DLP-function Π from Example 4, we obtain $R_1 = \{a \vee b \leftarrow {\sim}c, {\sim}d;\ a \leftarrow b;\ b \leftarrow a\}$ and $R_2 = \{c \vee d \leftarrow {\sim}a, {\sim}b;\ c \leftarrow d;\ d \leftarrow c\}$ as the sets of rules associated with $\Pi_1 = \langle R_1, \{c, d\}, \{a, b\}, \emptyset\rangle$ and $\Pi_2 = \langle R_2, \{a, b\}, \{c, d\}, \emptyset\rangle$, for which $\Pi_1 \sqcup \Pi_2 = \langle R_1 \cup R_2, \emptyset, \{a, b, c, d\}, \emptyset\rangle$ is defined. □

Checking Equivalence. Finally, we briefly mention how DLP-functions can be compared with each other at the level of modules as well as entire programs.

Definition 10. *Two DLP-functions Π_1 and Π_2 are* modularly equivalent, *denoted by $\Pi_1 \equiv_{\mathrm{m}} \Pi_2$, iff*

1. $\mathrm{At_i}(\Pi_1) = \mathrm{At_i}(\Pi_2)$ *and* $\mathrm{At_o}(\Pi_1) = \mathrm{At_o}(\Pi_2)$, *and*
2. *there is a bijection $f : \mathrm{SM}(\Pi_1) \rightarrow \mathrm{SM}(\Pi_2)$ such that for all interpretations $M \in \mathrm{SM}(\Pi_1)$, $M \cap \mathrm{At_v}(\Pi_1) = f(M) \cap \mathrm{At_v}(\Pi_2)$.*

Using $\equiv_{\mathrm{m}}$, we may reformulate the content of Theorem 2 as $\Pi \equiv_{\mathrm{m}} \mathrm{GSH}(\Pi)$. The proof for a congruence property lifts from the case of normal programs using the module theorem strengthened to the disjunctive case (i.e., Theorem 1).

Corollary 2. *Let Π_1, Π_2, and Π be DLP-functions. If $\Pi_1 \equiv_{\mathrm{m}} \Pi_2$ and both $\Pi_1 \sqcup \Pi$ and $\Pi_2 \sqcup \Pi$ are defined, then $\Pi_1 \sqcup \Pi \equiv_{\mathrm{m}} \Pi_2 \sqcup \Pi$.*

Applying Corollary 2 in the context of Theorem 2 indicates that shifting can be localised to a particular component $\Pi_2 = \mathrm{GSH}(\Pi_1)$ in a larger DLP-function $\Pi_1 \sqcup \Pi$.

A broader discussion which relates modular equivalence with similar notions proposed in the literature [15] is subject of future work but some preliminary comparisons in the case of disjunction-free programs are given by Oikarinen and Janhunen [8].

6 Conclusion and Discussion

In this paper, we discussed a formal framework for modular programming in the context of disjunctive logic programs under the stable-model semantics. We introduced syntax and semantics of DLP-functions, where input/output interfacing is realised, and proved a novel module theorem, establishing a suitable compositional semantics for program modules. Although our approach is not unique in the sense that there are different possibilities for defining the composition of modules, it nevertheless shows the limits of modularity in the context of a nonmonotonic declarative programming language. In any case, we believe that research in this direction not only yields results of theoretical interest but also could serve as a basis for future developments addressing practicably useful methods for software engineering in ASP.

Concerning previous work on modularity in ASP, Eiter, Gottlob, and Mannila [6] consider the class of disjunctive Datalog as query programs over relational databases. In contrast to our results, their module architecture is based on both *positive and negative dependencies* and no recursion between modules is tolerated. These constraints enable a straightforward generalisation of the splitting-set theorem for that architecture. Eiter, Gottlob, and Veith [3] address modularity within ASP by viewing program modules as *generalised quantifiers* allowing nested calls. This is an abstraction mechanism typical to programming-in-the-small approaches. Finally, Faber *et al.* [16] apply the *magic set method* in the evaluation of Datalog programs with negation, introducing the concept of an *independent set*, which is a specialisation of a splitting set. The module theorem put forward by Faber *et al.* [16] is, however, weaker than Theorem 1 presented in Section 4.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
2. Gaifman, H., Shapiro, E.: Fully Abstract Compositional Semantics for Logic Programs. In: Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89). (1989) 134–142
3. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97). Volume 1265 of LNCS, Springer (1997) 290–309

4. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In: Proceedings of the 22nd International Conference on Logic Programming (ICLP'06). Volume 4079 of LNCS, Springer (2006) 376–390
5. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: Proceedings of the 11th International Conference on Logic Programming (ICLP'94), MIT Press (1994) 23–37
6. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (1997) 364–418
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with External Evaluations for Semantic Web Reasoning. In: Proceedings of the 3rd European Semantic Web Conference (ESWC'06). Volume 4011 of LNCS, Springer (2006) 273–287
8. Oikarinen, E., Janhunen, T.: Modular Equivalence for Normal Logic Programs. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06). (2006) 412–416.
9. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A., Sadri, F., eds.: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Volume 2408 of LNCS, Springer (2002) 413–451
10. Janhunen, T.: Some (In)translatability Results for Normal Logic Programs and Propositional Theories. Journal of Applied Non-Classical Logics **16**(1–2) (2006) 35–86
11. Janhunen, T., Oikarinen, T.: Automated Verification of Weak Equivalence within the SMODELS System. Theory and Practice of Logic Programming (2006). To appear
12. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. Annals of Mathematics and Artificial Intelligence **12**(1–2) (1994) 53–87
13. Lifschitz, V.: Computing Circumscription. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85), Morgan Kaufmann (1985) 121–127
14. Eiter, T., Fink, M., Tompits, H., Woltran, T.: Simplifying Logic Programs under Uniform and Strong Equivalence. In: Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'03). Volume 2923 of LNAI, Springer (2004) 87–99
15. 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'05), Professional Book Center (2005) 97–102
16. Faber, W., Greco, G., Leone, N.: Magic Sets and Their Application to Data Integration. In: Proceedings of the 10th International Conference on Database Theory (ICDT'05). Volume 3363 of LNCS, Springer (2005) 306–320

A Characterization of Strong Equivalence for Logic Programs with Variables

Vladimir Lifschitz[1,⋆], David Pearce[2,⋆⋆], and Agustín Valverde[3,⋆⋆⋆]

[1] Department of Computer Sciences,
University of Texas at Austin, USA
vlcs.utexas.edu

[2] Computing Science and Artificial Intelligence,
Univ. Rey Juan Carlos, (Móstoles, Madrid), Spain
davidandrew.pearceurjc.es

[3] Dept. of Applied Mathematics, Univ. of Málaga, Spain
a_valverdectima.uma.es

Abstract. Two sets of rules are said to be strongly equivalent to each other if replacing one by the other within any logic program preserves the program's stable models. The familiar characterization of strong equivalence of grounded programs in terms of the propositional logic of here-and-there is extended in this paper to a large class of logic programs with variables. This class includes, in particular, programs with conditional literals and cardinality constraints. The first-order version of the logic of here-and-there required for this purpose involves two additional non-intuitionistic axiom schemas.

1 Introduction

The concept of a stable model was originally defined in [6] for sets of rules of a very special syntactic form. Later it was extended to arbitrary propositional formulas [13,5] and to arbitrary first-order sentences [10,11,4]. The extension to formulas with quantifiers is important, in particular, in view of its close relation to conditional literals—an LPARSE construct widely used in answer set programming [14]. For instance, according to [4], the choice rule

$$\{q(x)\ :\ p(x)\}$$

can be viewed as shorthand for the first-order formula

$$\forall x(p(x) \to (q(x) \vee \neg q(x))).$$

⋆ Partially supported by the National Science Foundation under Grant IIS-0412907.

⋆⋆ Partially supported by CICyT projects TIC-2003-9001-C02 and TIN2006-15455-CO3.

⋆⋆⋆ Partilly supported by CICyT project TIC-2003-9001-C01, TIN2006-15455-CO1 and Junta de Andalucia project TIC-115.

Similarly, the LPARSE rule

$$2\,\{q(x)\,:\,p(x)\}$$

can be thought of as shorthand for the formula

$$\begin{array}{l}\forall x(p(x) \to (q(x) \vee \neg q(x)))\wedge\\ \quad\exists xy(p(x) \wedge q(x) \wedge p(y) \wedge q(y) \wedge x \neq y).\end{array}$$

In this paper we extend the main theorem of [8] to stable models of first-order sentences. That theorem relates strong equivalence of propositional (grounded) logic programs to the propositional logic of here-and-there. Recall that two sets of rules are said to be strongly equivalent to each other if replacing one by the other within any logic program preserves the program's stable models; the propositional logic of here-and-there is the extension of propositional intuitionistic logic obtained by adding the axiom schema

HOS $\quad F \vee (F \to G) \vee \neg G.$

This is a simplified form of an axiom from [7], proposed in [2]. It is weaker than the law of the excluded middle, but stronger than the weak law of the excluded middle

WEM $\neg F \vee \neg\neg F.$

(To derive WEM from HOS, take G to be $\neg F$.)

Such characterizations of strong equivalence are interesting because they tell us which transformations can be used to simplify rules, or groups of rules, in a logic program. For instance, if we replace the pair of rules

$$\begin{array}{l}q \leftarrow \mathit{not}\ p\\ q \leftarrow \{\mathit{not}\ p\}\,0\end{array}$$

in a logic program with the fact q then the stable models of the program will remain the same. Indeed, the formula

$$(\neg p \to q) \wedge (\neg\neg p \to q)$$

is equivalent to q in the propositional logic of here-and-there. (Proof: use WEM with p as F.)

There are several natural extensions of the logic of here-and-there to first-order formulas; all of them include the axioms and inference rules of intuitionistic predicate logic, axiom schema HOS, and some other axioms. Our goal here is to determine which of these extensions corresponds to the strong equivalence of first-order sentences in the sense of [4].

The next section is a review the definition of a stable model from [4]. In Section 3 we state our main theorem, which characterizes strong equivalence in terms of a first-order version of the logic of here-and-there, and give examples of the use of that logic for establishing the strong equivalence of formulas and corresponding programs in the language of LPARSE. Section 4 describes a characterization of our first-order logic of here-and-there in terms of Kripke models; the soundness and completeness theorem stated in that section is a key element of the proof of the main theorem. Proofs are outlined in Sections 5 and 6. In Section 7 the theorem on strong equivalence is extended from formulas to theories—sets of formulas, possibly infinite. Related work is discussed in Section 8.

2 Stable Models of a First-Order Sentence

If p and q are predicate constants of the same arity then $p = q$ stands for the formula

$$\forall \mathbf{x}(p(\mathbf{x}) \leftrightarrow q(\mathbf{x})),$$

and $p \leq q$ stands for

$$\forall \mathbf{x}(p(\mathbf{x}) \rightarrow q(\mathbf{x})),$$

where $\mathbf{x}$ is a tuple of distinct object variables. If $\mathbf{p}$ and $\mathbf{q}$ are tuples $p_1, \dots, p_n$ and $q_1, \dots, q_n$ of predicate constants then $\mathbf{p} = \mathbf{q}$ stands for the conjunction

$$p_1 = q_1 \wedge \cdots \wedge p_n = q_n,$$

and $\mathbf{p} \leq \mathbf{q}$ for

$$p_1 \leq q_1 \wedge \cdots \wedge p_n \leq q_n.$$

Finally, $\mathbf{p} < \mathbf{q}$ is an abbreviation for $\mathbf{p} \leq \mathbf{q} \wedge \neg(\mathbf{p} = \mathbf{q})$. In second-order logic, we will apply the same notation to tuples of predicate variables.

According to [4], for any first-order sentence (closed formula) F, SM$[F]$ stands for the second-order sentence

$$F \wedge \neg \exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})),$$

where $\mathbf{p}$ is the list of all predicate constants $p_1, \dots, p_n$ occurring in F, $\mathbf{u}$ is a list of n distinct predicate variables $u_1, \dots, u_n$, and $F^*(\mathbf{u})$ is defined recursively, as follows:

- $p_i(t_1, \dots, t_m)^* = u_i(t_1, \dots, t_m)$;
- $(t_1 = t_2)^* = (t_1 = t_2)$;
- $\bot^* = \bot$;
- $(F \odot G)^* = F^* \odot G^*$, where $\odot \in \{\wedge, \vee\}$;
- $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
- $(QxF)^* = QxF^*$, where $Q \in \{\forall, \exists\}$.

(There is no clause for negation here, because we treat $\neg F$ as shorthand for $F \rightarrow \bot$.) A model of F is *stable* if it satisfies SM$[F]$.

This definition looks very different from the original definition of a stable model from [6], but it is actually a generalization of that definition, in the following sense. Let F be (the sentence corresponding to) a finite set of rules of the form

$$A_0 \leftarrow A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n,$$

where $A_0, \ldots, A_n$ are atomic formulas not containing equality. According to Proposition 1 from [4], the Herbrand stable models of F in the sense of the definition above are identical to the stable models of F in the sense of the original definition. For instance, the sentence

$$p(a) \wedge q(b) \wedge \forall x((p(x) \wedge \neg q(x)) \rightarrow r(x)), \tag{1}$$

representing the logic program

$$\begin{array}{l} p(a), \\ q(b), \\ r(x) \leftarrow p(x), not\ q(x), \end{array} \tag{2}$$

has a unique Herbrand stable model

$$\{p(a), q(b), r(a)\},$$

which is the stable model of (2) in the sense of the 1988 definition.

Here is an example illustrating the relationship between the definition above and the semantics of programs with conditional literals and choice rules proposed in [14]. The sentence

$$p(a) \wedge p(b) \wedge \forall x(p(x) \rightarrow (q(x) \vee \neg q(x))),$$

representing the program

$$\begin{array}{l} p(a), \\ p(b), \\ \{q(x) : p(x)\}, \end{array} \tag{3}$$

has 4 Herbrand stable models

$$\begin{array}{l} \{p(a),\ p(b)\}, \\ \{p(a),\ p(b),\ q(a)\}, \\ \{p(a),\ p(b),\ q(b)\}, \\ \{p(a),\ p(b),\ q(a),\ q(b)\}, \end{array}$$

which are identical to the stable models of (3) in the sense of [14].

3 Theorem on Strong Equivalence

About first-order sentences F and G we say that F is *strongly equivalent* to G if, for every first-order sentence H (possibly of a larger signature), $F \wedge H$ has the same stable models as $G \wedge H$ [4].

By $\mathbf{INT}^=$ we denote first-order intuitionistic logic with the usual axioms for equality:

$$x = x$$

and

$$x = y \rightarrow (F(x) \rightarrow F(y))$$

for every formula $F(x)$ such that y is substitutable for x in $F(x)$.

Our characterization of strong equivalence refers to the axiom schema

SQHT $\exists x(F(x) \rightarrow \forall x F(x))$.

The notation SQHT stands for "static quantified here-and-there"; see Section 4 below for an explanation. We also need the "decidable equality" axiom

DE $x = y \vee x \neq y$.

Theorem on Strong Equivalence. *A sentence F is strongly equivalent to a sentence G iff the equivalence $F \leftrightarrow G$ is provable in*

$$\mathbf{INT}^{=} + \mathrm{HOS} + \mathrm{SQHT} + \mathrm{DE}. \tag{4}$$

We will denote system (4) by $\mathbf{SQHT}^{=}$.

Example 1. In any program containing the rules

$$\begin{array}{l} \{q(x)\ :\ p(x)\} \\ p(a) \\ q(a) \end{array}$$

replacing the fact $q(a)$ with the constraint

$$\leftarrow \textit{not}\ q(a)$$

would not change the program's stable models. Indeed, the formula

$$\forall x(p(x) \rightarrow (q(x) \vee \neg q(x))) \wedge p(a) \wedge q(a) \tag{5}$$

is intuitionistically equivalent to

$$\forall x(p(x) \rightarrow (q(x) \vee \neg q(x))) \wedge p(a) \wedge \neg\neg q(a).$$

Proof: The first two conjunctive terms of (5) imply $q(a) \vee \neg q(a)$, and consequently $\neg\neg q(a) \leftrightarrow q(a)$.

Replacing a fact by a constraint can be viewed as a simplification, because the effect of adding a constraint to a program on its stable models is easy to describe. For instance, adding the constraint $\leftarrow \textit{not}\ q(a)$ to a program eliminates its stable models that do not satisfy $q(a)$. Adding the fact $q(a)$ to a program may affect its stable models, generally, in a very complicated way.

Example 2. Dropping $x \neq y$ from the body of the rule

$$p(y) \leftarrow p(x), q(x, y), x \neq y$$

would not change a program's stable models, because the formula

$$\forall xy(p(x) \wedge q(x, y) \wedge x \neq y \rightarrow p(y)) \tag{6}$$

is equivalent to

$$\forall xy(p(x) \wedge q(x, y) \rightarrow p(y)) \tag{7}$$

in $\mathbf{INT}^= + \mathrm{DE}$. Proof: By DE, (7) is equivalent to the conjunction of (6) and

$$\forall xy(p(x) \wedge q(x,y) \wedge x = y \rightarrow p(y)).$$

The last formula is provable in $\mathbf{INT}^=$.

Example 3. A different characterization of strong equivalence is used in [4, Section 4] to show that $\neg\forall x F(x)$ is strongly equivalent to $\exists x \neg F(x)$. To prove this fact using the theorem above, observe that the equivalence

$$\neg\forall x F(x) \leftrightarrow \exists x \neg F(x) \tag{8}$$

is provable in $\mathbf{INT} + \mathrm{SQHT}$. (Proof: the implication right-to-left is provable intuitionistically; the implication left-to-right is an intutionistic consequence of SQHT.) Furthemore, $\neg\neg\exists x F(x)$ is strongly equivalent to $\exists x \neg\neg F(x)$. (Proof: the formula

$$\neg\neg\exists x F(x) \leftrightarrow \exists x \neg\neg F(x). \tag{9}$$

is intuitionistically equivalent to the instance of (8) in which $\neg F(x)$ is taken as $F(x)$.)

4 Kripke Models

Our proof of the theorem on strong equivalence refers to the class of Kripke models introduced in [4]. In this section we discuss two reasons why this class of models is relevant. On the one hand, system $\mathbf{SQHT}^=$, introduced above in connection with the problem of strong equivalence, turns out to be a sound and complete axiomatization of this class of models. On the other hand, according to Proposition 4 from [4], the first-order equilibrium logic based on this class of models provides a characterization of the concept of a stable model that we are interested in.

The definition of this class of models uses the following notation. If I is an interpretation of a signature σ (in the sense of classical logic) then by σ^I we denote the extension of σ obtained by adding pairwise distinct symbols ξ^*, called *names*, for all elements ξ of the universe of I as object constants. We will identify I with its extension to σ^I defined by $I(\xi^*) = \xi$. The value that I assigns to a ground term t of signature σ^I will be denoted by t^I. By σ_f we denote the part of σ consisting of its function constants (including object constants, which are viewed as function constants of arity 0).

An *HT-interpretation* of σ is a triple $\langle I^{\mathrm{f}}, I^{\mathrm{h}}, I^{\mathrm{t}} \rangle$, where

– I^{f} is an interpretation of σ_f, and
– I^{h}, I^{t} are sets of atomic formulas formed using predicate constants from σ and object constants ξ^* for arbitrary elements ξ of the universe of I^{f}, such that $I^{\mathrm{h}} \subseteq I^{\mathrm{t}}$.

The symbols h, t are called *worlds*; they are ordered by the relation h<t.

Note that according to this definition the two worlds share a common universe. In this sense, our Kripke models are static; this explains the use of the word "static" in the name of the axiom SQHT (Section 3). The worlds share also a common equality relation: in both of them, equality is understood as identity.

The *satisfaction* relation between an HT-interpretation $I = \langle I^{\mathrm{f}}, I^{\mathrm{h}}, I^{\mathrm{t}} \rangle$, a world w and a sentence F of the signature σ^U, where U is the universe of I^{f}, is defined recursively:

- $I, w \models p(t_1, \dots)$ if $p((t_1^I)^*, \dots) \in I^w$,
- $I, w \models t_1 = t_2$ if $t_1^I = t_2^I$,
- $I, w \not\models \bot$,
- $I, w \models F \wedge G$ if $I, w \models F$ and $I, w \models G$,
- $I, w \models F \vee G$ if $I, w \models F$ or $I, w \models G$,
- $I, w \models F \rightarrow G$ if, for every world w' such that $w \leq w'$, $I, w' \not\models F$ or $I, w' \models G$,
- $I, w \models \forall x F(x)$ if, for each ξ from the universe of I^{f}, $I, w \models F(\xi^*)$,
- $I, w \models \exists x F(x)$ if, for some ξ from the universe of I^{f}, $I, w \models F(\xi^*)$.

We write $I \models F$ if $I, \mathrm{h} \models F$.[1]

For any set Γ of sentences and any sentence F, we write $\Gamma \models F$ if every HT-interpretation satisfying all formulas in Γ satisfies F also. We write $\Gamma \vdash F$ if F is derivable from Γ in $\mathbf{SQHT}^=$.

Soundness and Completeness Theorem. *$\Gamma \models F$ iff $\Gamma \vdash F$.*

In the next section we outline a proof of the more difficult part of this claim, the implication left-to-right.

The corresponding concept of an equilibrium model is defined as follows. An HT-interpretation $\langle I^{\mathrm{f}}, I^{\mathrm{h}}, I^{\mathrm{t}} \rangle$ is *total* if $I^{\mathrm{h}} = I^{\mathrm{t}}$. A total HT-interpretation $\langle I, J, J \rangle$ is an *equilibrium model* of F if

(i) $\langle I, J, J \rangle \models F$, and
(ii) for any proper subset J' of J, $\langle I, J', J \rangle \not\models F$.

We can represent an interpretation I of σ in the sense of classical logic as the pair $\langle I|_{\sigma^{\mathrm{f}}}, I' \rangle$, where I' is the set of all atomic formulas, formed using predicate constants from σ and names ξ^*, which are satisfied by I. According to Proposition 4 from [4], an interpretation $\langle I, J \rangle$ is a stable model of a sentence F iff $\langle I, J, J \rangle$ is an equilibrium model of F. Thus stable models of a sentence are essentially identical to its equilibrium models.

5 Proof of Completeness

Assume that $\Gamma \not\vdash F$. We will define an HT-interpretation I that satisfies all formulas in Γ but does not satisfy F.

There exists a signature σ', obtained from σ by adding new object constants, and a set Γ_{h} of sentences of this signature, satisfying the following conditions:

[1] This definition looks different from the definition of satisfaction proposed in [4], but it easy to check that they are equivalent to each other.

(i) $\Gamma \subseteq \Gamma_{\rm h}$,
(ii) $F \notin \Gamma_{\rm h}$,
(iii) $\Gamma_{\rm h}$ is closed under $\vdash$,
(iv) for any sentence of the form $G \vee H$ in $\Gamma_{\rm h}$, $G \in \Gamma_{\rm h}$ or $H \in \Gamma_{\rm h}$,
(v) for any sentence of the form $\exists x F(x)$ in $\Gamma_{\rm h}$ there exists an object constant c in σ' such that $F(c) \in \Gamma_{\rm h}$.

(Conditions (iii)–(v) can be expressed by saying that $\Gamma_{\rm h}$ is *prime*.) The proof is the same as in Henkin's proof of completeness of intuitionistic logic [1, Section 3]. For any ground terms t_1 and t_2 of the signature σ', we write $t_1 \approx t_2$ if the formula $t_1 = t_2$ belongs to $\Gamma_{\rm h}$. Let $\Gamma_{\rm t}$ be a maximal superset of $\Gamma_{\rm h}$ that is consistent and closed under classical logic; such a superset exists by the Lindenbaum lemma. Now $I = \langle I^{\rm f}, I^{\rm h}, I^{\rm t}\rangle$ is defined as follows:

(i) the universe of $I^{\rm f}$ is the set of equivalence classes of the relation $\approx$;
(ii) for each object constant c from σ, $I^{\rm f}[c]$ is the equivalence class of $\approx$ that contains c,
(iii) for each function constant g from σ of arity $n > 0$, $I^{\rm f}[g](\xi_1, \dots, \xi_n)$ is the equivalence class of $\approx$ that contains $g(t_1, \dots, t_n)$ for all terms $t_1 \in \xi_1, \dots, t_n \in \xi_n$.
(iv) for each world w, I^w is the set of formulas of the form $p(\xi_1^*, \dots, \xi_n^*)$ such that Γ_w contains $p(t_1, \dots, t_n)$ for all terms $t_1 \in \xi_1, \dots, t_n \in \xi_n$.

Note that I can be viewed as an HT-interpretation of the extended signature σ' if we extend clause (ii) to all objects constants from σ'. In the rest of this section, terms and formulas are understand as terms and formulas of the extended signature.

Lemma 1. *For any ground terms t_1, t_2, $(t_1 = t_2) \in \Gamma_{\rm t}$ iff $(t_1 = t_2) \in \Gamma_{\rm h}$.*

Proof. The if part follows from the fact that $\Gamma_{\rm h} \subseteq \Gamma_{\rm t}$. Only if: Assume that $(t_1 = t_2) \notin \Gamma_{\rm h}$. Since $\Gamma_{\rm h}$ contains the instance $t_1 = t_2 \vee t_1 \neq t_2$ of DE and is prime, it follows that $(t_1 \neq t_2) \in \Gamma_{\rm h}$. Since $\Gamma_{\rm t}$ is a consistent superset of $\Gamma_{\rm h}$, we can conclude that $(t_1 = t_2) \notin \Gamma_{\rm t}$.

Lemma 2. *For any sentence of the form $\exists x G(x)$ there exists an object constant c such that the formula*

$$\exists x G(x) \rightarrow G(c) \tag{10}$$

belongs to $\Gamma_{\rm t}$.

Proof. *Case 1:* $\exists x G(x) \in \Gamma_{\rm t}$. Since $\Gamma_{\rm h}$ contains the instance

$$\neg\exists x G(x) \vee \neg\neg\exists x G(x)$$

of WEM and is prime, $\Gamma_{\rm h}$ contains one of its disjunctive terms. But the first disjunctive term cannot belong to $\Gamma_{\rm h}$ because the consistent superset $\Gamma_{\rm t}$ of $\Gamma_{\rm h}$ contains $\exists x G(x)$. Consequently $\neg\neg\exists x G(x) \in \Gamma_{\rm h}$. Since equivalence (9) belongs to $\Gamma_{\rm h}$, it follows that $\exists x \neg\neg G(x) \in \Gamma_{\rm h}$. Since $\Gamma_{\rm h}$ is prime, it follows that there exists an object constant c such that $\neg\neg G(c) \in \Gamma_{\rm h}$. Since $\Gamma_{\rm h} \subseteq \Gamma_{\rm t}$ and (10) is a classical consequence of $\neg\neg G(c)$, it follows that (10) belongs to Γ_{t}. *Case 2:* $\exists x G(x) \notin \Gamma_t$. Since Γ_{t} is maximally consistent, it follows that $\neg\exists x G(x) \in \Gamma_{\mathrm{t}}$. Since (10) is a classical consequence of $\neg\exists x G(x)$, it follows that (10) belongs to Γ_{t}.

Recall that our goal is to prove two properties of the interpretation I: it satisfies all formulas in Γ but does not satisfy F. By the choice of Γ_{h}, this is immediate from the following lemma:

Lemma 3. *For any sentence G of signature σ' and any world w,*

$$I, w \models G \ \ \textit{iff} \ \ G \in \Gamma_w.$$

Proof. by induction on G. We will consider the three cases where reasoning is different than in the similar proof for intuitionistic logic [1, Section 3]: $t_1 = t_2$, $G \to H$, and $\forall x G(x)$.

1. To check that

$$I, w \models t_1 = t_2 \text{ iff } t_1 = t_2 \in \Gamma_w$$

we show that each side is equivalent to $t_1 \approx t_2$. For the left-hand side, this follows from the fact that for every ground term t, t^{I_f} is the equivalence class of $\approx$ that contains t (by induction on t). For the right-hand side, if $w = \mathrm{h}$ then this is immediate from the definition of $\approx$; if $w = \mathrm{t}$ then use Lemma 1.

2. Assume that

$$I, w \models G \ \text{ iff } \ G \in \Gamma_w$$

and

$$I, w \models H \ \text{ iff } \ H \in \Gamma_w;$$

we want to show that

$$I, w \models G \to H \ \text{ iff } \ G \to H \in \Gamma_w.$$

The if part follows from the induction hypothesis and the clause for $\to$ in the definition of satisfaction. To prove the only if part for $w = \mathrm{t}$, use the induction hypothesis and the fact that, by the maximal consistency of Γ_{t}, this set contains either G or $\neg G$. For $w = \mathrm{h}$, we conclude from the induction hypothesis and the assumption $I, \mathrm{h} \models G \to H$ that

$$G \notin \Gamma_{\mathrm{h}} \text{ or } H \in \Gamma_{\mathrm{h}} \tag{11}$$

and

$$G \notin \Gamma_{\mathrm{t}} \text{ or } H \in \Gamma_{\mathrm{t}}. \tag{12}$$

Case 1: $G \in \Gamma_{\mathrm{h}}$. Then, by (11), $H \in \Gamma_{\mathrm{h}}$ and consequently $G \to H \in \Gamma_{\mathrm{h}}$. *Case 2:* $\neg G \in \Gamma_{\mathrm{h}}$. Since $\neg G \vdash G \to H$, $G \to H \in \Gamma_{\mathrm{h}}$. *Case 3:* $G, \neg G \notin \Gamma_{\mathrm{h}}$. Since Γ_{h} contains the instance $\neg G \vee \neg\neg G$ of WEM and is prime, it follows that $\neg\neg G \in \Gamma_{\mathrm{h}} \subseteq \Gamma_{\mathrm{t}}$. Then $G \in \Gamma_{\mathrm{t}}$ and, by (12), $H \in \Gamma_{\mathrm{t}}$. Since Γ_{t} is consistent and contains Γ_{h}, $\neg H \notin \Gamma_{\mathrm{h}}$. On the other hand, Γ_{h} contains the instance

$G \vee (G \to H) \vee \neg H$ of HOS and is prime; since neither G nor $\neg H$ belongs to Γ_{h}, $G \to H \in \Gamma_{\mathrm{h}}$.

3. Assume that for every object constant c

$$I, w \models G(c) \text{ iff } G(c) \in \Gamma_w;$$

we need to show that

$$I, w \models \forall x G(x) \text{ iff } \forall x G(x) \in \Gamma_w.$$

The if part follows from the induction hypothesis and the clause for $\forall$ in the definition of satisfaction. To prove the only if part for $w = \mathrm{t}$, take an object constant c such that the formula

$$\exists x \neg G(x) \to \neg G(c) \tag{13}$$

belongs to Γ_{t} (Lemma 2). It follows from the induction hypothesis and the clause for $\forall$ in the definition of satisfaction that $G(c)$ belongs to Γ_{t} too; $\forall x G(x)$ is a classical consequence of (13) and $G(c)$. To prove the only if part for $w = \mathrm{h}$, consider the instance

$$\exists x (G(x) \to \forall x G(x))$$

of SQHT. Since Γ_{h} is prime, there exists an object constant c such that Γ_{h} contains the formula

$$G(c) \to \forall x G(x). \tag{14}$$

By the induction hypothesis and the clause for $\forall$ in the definition of satisfaction, $G(c) \in \Gamma_{\mathrm{h}}$; $\forall x G(x)$ is an intuitionistic consequence of $G(c)$ and (14).

6 Proof of the Strong Equivalence Theorem

In view of the soundness and completeness theorem, the theorem on strong equivalence can be rewritten as: a sentence F is strongly equivalent to a sentence G iff F and G are satisfied by the same HT-interpretations.

If F and G are satisfied by the same HT-interpretations then $F \wedge H$ and $G \wedge H$ are satisfied by the same HT-interpretations; then $F \wedge H$ and $G \wedge H$ have the same equilibrium models, and consequently the same stable models.

For the converse, let us assume that $F \wedge H$ and $G \wedge H$ have the same stable models for every first order sentence H. Then these formulas have the same equilibrium models. For any predicate constant P, let $C(P)$ stand for the sentence

$$\forall \mathbf{x}(\neg\neg P(\mathbf{x}) \to P(\mathbf{x})).$$

Note first that F and G have the same total models. Indeed, let H_0 be the conjunction of sentences $C(P)$ for all predicate constants P occurring in F or G; the total models of F can be characterized as the equilibrium models of $F \wedge H_0$, and the total models of G can be characterized as the equilibrium models of $G \wedge H_0$.

Assume that $\langle I^{\rm f}, I^{\rm h}, I^{\rm t}\rangle$ satisfies F but not G, and consider the total HT-interpretation $\langle I^{\rm f}, I^{\rm t}, I^{\rm t}\rangle$. It is clear that the total interpretation $\langle I^{\rm f}, I^{\rm t}, I^{\rm t}\rangle$ satisfies F, and consequently G. Let H_0 be the conjunction of sentences $C(P)$ for all predicate constants P occurring in G, and let H_1 be the implication $G \rightarrow H_0$. The HT-interpretation $\langle I^{\rm f}, I^{\rm h}, I^{\rm t}\rangle$ satisfies H_1. Indeed, it does not satisfy its antecedent G, and I^t satisfies its consequent H_0. Therefore $\langle I^{\rm f}, I^{\rm t}, I^{\rm t}\rangle$ is not an equilibrium model of $F \wedge H_1$. Then $\langle I^{\rm f}, I^{\rm t}, I^{\rm t}\rangle$ is not an equilibrium model of $G \wedge H_1$ either. But this is impossible, because $G \wedge H_1$ implies H_0, so that all models of that set are total.

7 Strong Equivalence for Theories

The definition of a stable model from [4], reproduced here in Section 2, can be extended to finite sets of first-order sentences: a model of such a set Γ is called stable if it satisfies $\mathrm{SM}\big[\bigwedge_{F\in\Gamma} F\big]$. The relationship between stable models and equilibrium models, discussed at the end of Section 4, suggests a way to further extend this definition to "theories"—arbitrary sets of first-order sentences, possibly infinite. We say that a total HT-interpretation $\langle I, J, J\rangle$ is an *equilibrium model* of a set Γ of first order sentences if

(i) $\langle I, J, J\rangle \models \Gamma$, and
(ii) for any proper subset J' of J, $\langle I, J', J\rangle \not\models \Gamma$.

An interpretation $\langle I, J\rangle$ is a *stable model* of Γ iff $\langle I, J, J\rangle$ is an equilibrium model of Γ.

About sets Γ, Δ of first-order sentences we say that Γ is *strongly equivalent* to Δ if, for any set Σ of first-order sentences (possibly of a larger signature), $\Gamma\cup\Sigma$ has the same stable models as $\Delta\cup\Sigma$. This relation between sets of formulas can be characterized in the same way as the strong equivalence relation between formulas:

Theorem on Strong Equivalence for Theories. *A set Γ of sentences is strongly equivalent to a set Δ of sentences iff Γ is equivalent to Δ in* $\mathbf{SQHT}^=$.

This theorem, in combination with the theorem on strong equivalence for formulas, shows that the new definition of strong equivalence is a generalization of the definition from Section 3: F and G are strongly equivalent to each other as sentences iff $\{F\}$ and $\{G\}$ are strongly equivalent to each other as theories.

The proof is similar to that in the previous section, but considering the (possibly infinite) set Σ_0 of formulas $C(P)$ instead of the conjunction H_0 of these formulas, and the set of implications $G \rightarrow C(P)$ instead of H_1.

8 Related Work

An alternative proof that the logic $\mathbf{SQHT}^=$ captures strong equivalence for theories in the sense of the previous section can be found in [12]. The two proofs highlight different properties. In each case it is shown that when two theories Γ and Δ are not equivalent in $\mathbf{SQHT}^=$, an extension Σ can be constructed such that the equilibrium models of $\Gamma \cup \Sigma$ and $\Delta \cup \Sigma$ differ. From the proof given in Section 7 above it is clear that Σ can be constructed in the same signature, without additional constants. From the proof given in [12] the extension may use object constants not appearing in Γ or Δ. However, Σ is shown there to have a very simple form: its elements are "unary" formulas, that is, ground atomic formulas and implications $F \rightarrow G$ where F and G are ground atomic formulas. From this observation it follows that if Γ and Δ consist of logic program rules and are not strongly equivalent then there exists a set of program rules Σ (of a simple form) such that $\Gamma \cup \Sigma$ and $\Delta \cup \Sigma$ have different equilibrium or stable models.

Strong equivalence for non-ground logic programs under the answer set semantics has also been defined and studied in [9,3]. In the case of [3] the concept is similar to the one presented in the previous section, except that the equivalence is defined with respect to a somewhat different notion of stable model than the one used here, and equality is not explicitly treated. In general the two concepts are different since stable or equilibrium models as defined here are not required to satisfy the unique name assumption. As a consequence not every equilibrium model need be a stable model or answer set in the sense of [3]. However for the safe programs without equality studied in [3] we can establish a simple characterisation of strong equivalence. Let us denote by **SQHT** the logic $\mathbf{SQHT}^=$ without an equality predicate and axioms for equality:

$$\mathbf{SQHT} = \mathbf{INT} + \mathrm{HOS} + \mathrm{SQHT} + \mathrm{DE}.$$

Disjunctive logic programs are defined in the usual way, and rules where each variable appears in at least one positive body atom are called *safe*; a program is *safe* if all its rules are safe. According to a theorem from [12], two safe disjunctive programs are strongly equivalent in the sense of [3] if and only if they are equivalent in the logic **SQHT**. The proof makes use of the fact that if Γ and Δ are non-strongly equivalent safe programs then there exists a set of Σ of unary program rules such that $\Gamma \cup \Sigma$ and $\Delta \cup \Sigma$, which are both safe, have different stable models. For safe programs, this theorem also encompasses the notion of strong equivalence found in [9].

9 Conclusion

In this paper we understand logic programs with variables in a very general way, as arbitrary first-order formulas with equality. The logic $\mathbf{SQHT}^=$ is the first-order version of the logic of here-and-there that characterizes strong equivalence for such programs. This logic is an extension of the intuitionistic first-order logic with equality. One of its three additional postulates is the axiom schema HOS, familiar from the propositional logic of here-and-there. Another is the well-known decidable equality axiom DE. The third, SQHT, is apparently new; it can be thought of as the result the first step towards converting the trivial implication $\forall x F(x) \rightarrow \forall x F(x)$ to prenex form. Studying properties of this intermediate logic is a topic for future work.

References

1. Dirk van Dalen. Intuitionistic logic. In Dov Gabbay and Franz Guenther, editors, *Handbook of Philosophical Logic, Volume III: Alternatives in Classical Logic*, Dordrecht, 1986. D. Reidel Publishing Co.
2. Dick De Jongh and Lex Hendriks. Characterization of strongly equivalent logic programs in intermediate logics. *Theory and Practice of Logic Programming*, 3:250–270, 2003.
3. T. Eiter, M. Fink, H. Tompits, and S. Woltran Strong and Uniform Equivalence in Answer-Set Programming: Characterizations and Complexity Results for the Non-Ground Case. KR 2005, AAAI, 2005.
4. Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.
5. Paolo Ferraris. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 119–131, 2005.
6. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, 1988.
7. T. Hosoi. The axiomatization of the intermediate propositional systems s_n of gödel. *Journal of the Faculty of Science of the University of Tokyo*, 13:183–187, 1996.
8. Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
9. F. Lin. Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic. In *Proc. KR'02*, 170-176.
10. David Pearce and Agustin Valverde. Towards a first order equilibrium logic for nonmonotonic reasoning. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, pages 147–160, 2004.
11. David Pearce and Agustin Valverde. A first order nonmonotonic extension of constructive logic. *Studia Logica*, 80:323–348, 2005.
12. David Pearce and Agustin Valverde. Quantified Equilibrium Logic and the First Order Logic of Here-and-There. Technical Report, Univ. Rey Juan Carlos, 2006, available at http://www.satd.uma.es/matap/investig/tr/ma06_02.pdf.
13. David Pearce. A new logical characterization of stable models and answer sets. In Jürgen Dix, Luis Pereira, and Teodor Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216)*, pages 57–70. Springer-Verlag, 1997.
14. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

A Unified Semantics for Constraint Handling Rules in Transaction Logic

Marc Meister[1,⋆], Khalil Djelloul[1,⋆⋆], and Jacques Robin[2,⋆]

[1] Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm, Germany
[2] Centro de Informática, Universidade Federal de Pernambuco
Caixa Postal 7851, 50732-970 Recife PE, Brazil

Abstract. Reasoning on Constraint Handling Rules (CHR) programs and their executional behaviour is often ad-hoc and outside of a formal system. This is a pity, because CHR subsumes a wide range of important automated reasoning services. Mapping CHR to Transaction Logic ($\mathcal{TR}$) combines CHR rule specification, CHR rule application, and reasoning on CHR programs and CHR derivations inside one formal system which is executable. This new $\mathcal{TR}$ semantics obviates the need for disjoint declarative and operational semantics.

1 Introduction

Constraint Handling Rules (CHR) [6] is a concurrent, committed-choice, rule-based language which was originally created as a declarative logic constraint language to implement monotonic reasoning services. Its main features are guarded rules which transform multi-sets of constraints (atomic formulas) into simpler ones until they are solved.

Over the last decade, CHR has become available for most Prolog systems, Java, Haskell, and Curry and has matured into a general-purpose programming language with many applications [12]: Nonmonotonic reasoning services can be implemented in CHR, e.g. the fluent executor (FLUX) [13] which provides general reasoning facilities about actions and sensor information under incomplete information. Also, classic algorithms like the union-find, which rely on inherently nonmonotonic updates, have been implemented with optimal complexity in CHR [11].

The *operational semantics* of CHR is specified by a state transition system. Although applicability of a CHR rule is defined within predicate logic, the operational semantics is not integrated into a logic and is different from the declarative semantics in predicate logic. Basically the problem is that there is no elegant *predicate logic-based semantics* for changing the constraint store. Hence, reasoning on CHR programs and their executional behaviour is often ad-hoc and outside of a formal logic-based system.

⋆ Supported by the DAAD project ROARS.
⋆⋆ Funded by the DFG project GLOB-CON.

We integrate the operational semantics of CHR into Transaction Logic ($\mathcal{TR}$) [3,4,5] which extends predicate logic with – among other things – a declarative account for state changes in logic programs (cf. [3] for a list of failed attempts to formalise updates in a logic programming language). Transaction Logic naturally enjoys nonmonotonic behaviour due to the dynamics of a database which represents a current state [8].

Contributions and overview of the paper. By mapping the core of CHR to $\mathcal{TR}$, we combine CHR rule specification, CHR rule application, and reasoning on CHR programs and CHR derivations inside one formal system which is executable. We show that a CHR rule applies if and only if the $\mathcal{TR}$ query of the mapping of this CHR rule succeeds in $\mathcal{TR}$ and extend this result to CHR derivations by integrating the CHR run-time system. A formal statement then links the procedural aspect of execution (the operational semantics) with a new model-theoretic (declarative) reading. Thus our semantics covers both operational and declarative aspects elegantly. An efficient proof system in $\mathcal{TR}$ executes CHR programs and reasons on CHR derivations mechanically.

– We present the aspect of a missing unified semantics for CHR through an easy example in Section 2 and propose our solution to overcome this missing aspect in Section 3.
– We explain the most basic instantiation of $\mathcal{TR}$ to give a logical account for range-restricted ground CHR programs in Section 4.
– We map the constraint store to a database, the CHR program to a serial-Horn $\mathcal{TR}$ program that updates this database, and the CHR run-time system to a generalised-Horn $\mathcal{TR}$ program. The details of our CHR-to-$\mathcal{TR}$ mapping in Section 5 are necessary for our sound- and completeness result which is our main contribution.
– In Section 6 we apply our approach to two examples, showing how to execute and reason on them in the framework of Transaction Logic. We use the FLORA-2 system [14] for implementation.

Complete proofs and full CHR and FLORA-2 sources of the examples are available at `http://www.informatik.uni-ulm.de/pm/index.php?id=138`.

2 The Problem: Reasoning on Constraint Handling Rules

Example 1. Consider a coin-throw simulation program[1], consisting of two CHR rules r_1 and r_2.

$$r_1 \ @ \ \texttt{throw} \Leftrightarrow \texttt{caput} \qquad r_2 \ @ \ \texttt{throw} \Leftrightarrow \texttt{nautica}$$

Intuitively, as both rules are applicable for the goal `throw`, the answer constraint is `caput` or `nautica` depending on the rule selection. Clearly, we have the two

[1] To avoid misunderstandings with the *head* of a rule, we replaced the good old "head" and "tail" with the ancient "caput" and "nautica".

possible state transitions (`throw`) $\rightarrowtail_{r_1}$ (`caput`) and (`throw`) $\rightarrowtail_{r_2}$ (`nautica`) for the goal `throw`.

What we are missing is one logic-based formal system for mechanical *execution* and *reasoning*, which should be implemented to also allow automatic reasoning. Available CHR run-time systems (e.g. the reference implementation in SICStus Prolog for CHR) come as black-boxes and offer no means for reasoning. For example, we want to prove the following three properties automatically:

(P1) Throwing a coin can yield `caput`.
(P2) Throwing a coin cannot yield both `caput` and `nautica`.
(P3) Application of r_1 cannot yield `nautica`.

Because the constraint `throw` is interpreted as a trigger (and not as static truth) in the coin-throw simulation program, the gap between the predicate logic declarative semantics [6] of this general-purpose CHR program – the meaningless formula `caput` $\leftrightarrow$ `nautica` – and its executions is especially large. The underlying problem is that predicate logic is a static logic, unable to express the dynamics of deletion and insertion directly. Here, reasoning has to be done ad-hoc (outside of a logic) along the operational semantics of CHR [1].

The linear logic semantics [2] overcomes this restriction of the classic predicate logic semantics and gives a meaningful declarative semantics also for general-purpose CHR programs. While the linear logic notion of a resource models the necessary dynamics, it does not cover all aspects of the operational semantics: Linear logic has no inherent notion of execution and we cannot reason on the execution itself but only on the result of an execution. Similar to the classic declarative semantics, the linear logic semantics links initial and final state with a logical reading of the program. As CHR derivations are mimicked inside its proofs, reasoning on derivations is not possible directly.

Summarising, both predicate and linear logic *declarative* semantics allow reasoning on the properties of the program, but lack the possibility to actually execute the rules, reason on the execution, and are not readily mechanised. Thus, reasoning on execution lacks a formal logic-based framework. Most importantly, specification (as CHR rules), execution (by a CHR run-time system), and reasoning are not integrated and reasoning is either done by hand or by special-purpose tools (e.g. for *confluence* [1]). The need to integrate the operational semantics into a logic was recognised by Maher [9]: Besides a “logical” (declarative) semantics, also a data/control-flow analysis is highly desirable, e.g. to prove termination of a program. Clearly this data/control-flow analysis aspect is inherently absent in [6,2] which cover the “logical” (declarative) semantics only. Maher continues, that “there is possibility that this analysis can be carried out within a logic framework” [9, p. 870]. We argue that Transaction Logic ($\mathcal{TR}$) provides this missing aspect in the next section.

3 The Idea: Map CHR to Transaction Logic

We map CHR to Transaction Logic to simulate the operational semantics of CHR by logic programming with state changes and use executional entailment

– a formal statement in $\mathcal{TR}$ – to execute and to reason on CHR derivations. In their seminal work on Transaction Logic [3], Bonner and Kifer extend predicate logic with a declarative account for state changes in logic programming. As the operational semantics of CHR is formalised by a state transition system, where a CHR rule application changes the constraints store, we map CHR programs to serial-Horn $\mathcal{TR}$ programs and identify the application of a CHR rule by the state transition system with a successful query of the $\mathcal{TR}$ program. To this end, we map the constraint store to a database with the elementary database updates *insertion* and *deletion*. A CHR derivation is then the side-effect on the database when the $\mathcal{TR}$ proof system infers the $\mathcal{TR}$ query to be true.

Example 1 (Cont.). We show the basic ideas for the coin-throw simulation program with a non-deterministic rule selection strategy (and review this example in Section 6 in detail). We map rule r_1 to the serial-Horn $\mathcal{TR}$ rule $r_1^{\mathcal{TR}}$.

$$r_1^{\mathcal{TR}} \equiv \mathrm{chr}(r_1) \leftarrow \texttt{throw} \otimes \texttt{throw}.\mathit{del} \otimes \texttt{caput}.\mathit{ins}$$

To make the $\mathcal{TR}$-predicate $\mathrm{chr}(r_1)$ true, we have to execute the serial conjunction on its right hand side: First check that `throw` is present, then delete it, and then insert `caput`. The order in the serial conjunction $\otimes$ is crucial, as the $\mathcal{TR}$-predicates $\texttt{throw}.\mathit{del}$ and $\texttt{caput}.\mathit{ins}$ have side-effects on the database[2]. If we execute $\mathrm{chr}(r_1)$ on the (initial) database $\{\texttt{throw}\}$, we pass through the empty database $\{\}$, and arrive at the (final) database $\{\texttt{caput}\}$. For $P = \{r_1^{\mathcal{TR}}\}$, we have the following *executional entailment* statement $\models_x$ in $\mathcal{TR}$, which states, that the successful invocation of program P by $\mathrm{chr}(r_1)$ can successfully update the database along the given *execution path* $\{\texttt{throw}\}, \{\}, \{\texttt{caput}\}$.

$$P, \{\texttt{throw}\}, \{\}, \{\texttt{caput}\} \models_x \mathrm{chr}(r_1)$$

The executional entailment statement has both a procedural (operational) and a model-theoretic (declarative) semantics in $\mathcal{TR}$. On the one hand, an available efficient $\mathcal{TR}$ inference system for the subclass of serial-Horn programs actually computes the necessary updates of an initial database $\{\texttt{throw}\}$ when establishing the truth of $\mathrm{chr}(r_1)$ and implements the procedural aspect of $\mathcal{TR}$. Integrating the operational semantics of CHR into $\mathcal{TR}$ by executional entailment, we have – on the other hand – a new model-theoretic (declarative) semantics which captures the possible executions of a CHR program.

We show in Section 5, that a CHR rule r is applicable iff we can establish the truth of the head of a $\mathcal{TR}$ rule $r^{\mathcal{TR}}$ and then extend our mapping to cover the CHR run-time system. The changes caused on the constraint store are mapped one-to-one to updates of the database as we simulate CHR rule application by the $\mathcal{TR}$ inference system.

We can then prove properties **(P1-3)** from Section 2 mechanically. Even better, the FLORA-2 system allows us to both execute and reason on this example automatically (cf. Section 6).

[2] The symbol $\otimes$ stands for serial conjunction in $\mathcal{TR}$ and *not* for join of views on databases.

4 Preliminaries

We provide necessary background for readers not familiar with CHR and $\mathcal{TR}$.

4.1 Constraint Handling Rules

Constraint Handling Rules (CHR) [6,12] is a concurrent, committed-choice, rule-based logic programming language. We distinguish between two different kinds of constraints: *built-in constraints* which are solved by a given constraint solver, and *user-defined (CHR) constraints* which are defined by the rules in a CHR program. This distinction allows one to embed and utilise existing constraint solvers as well as side-effect-free host language statements. As we trust the built-in black-box constraint solvers, there is no need to modify or inspect them.

A *CHR program* is a finite set of rules. There are two main kinds of rules: *Simplification rules* $R \ @ \ H \Leftrightarrow G \mid B$ and *propagation rules* $R \ @ \ H \Rightarrow G \mid B$. Each rule has a unique name R, the *head* H is a non-empty multi-set conjunction of CHR constraints, the *guard* G is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a multi-set conjunction of built-in and CHR constraints. A trivial guard expression "*true* |" can be omitted.

Since we do not focus on propagation rules in this paper, it suffices to say that they are equivalent (in the standard semantics) to simplification rules of the form $R \ @ \ H \Leftrightarrow G \mid (H \wedge B)$.

The *operational semantics* of CHR is defined by a state transition system where states are conjunctions of constraints. To a conjunction of constraints, rules are applied until a fixpoint is reached. Note that conjunctions in CHR are considered as *multi-sets* of atomic constraints. Any of the rules that are applicable can be applied and rule application cannot be undone since CHR is a committed-choice language. A simplification rule $R \ @ \ H \Leftrightarrow G \mid B$ is applicable in state $(H' \wedge C)$, if the built-in constraints C_b of C imply that H' matches the head H and the guard G is entailed under this matching, cf. (1). The consistent, predicate logic, built-in constraint theory CT contains Clark's syntactic equality.

$$\begin{array}{ll} \text{IF} & R \ @ \ H \Leftrightarrow G \mid B \text{ is a fresh variant of rule } R \text{ with variables } \bar{X} \\ \text{AND} & CT \models (\forall)\, C_b \rightarrow \exists \bar{X}\, (H = H' \wedge G) \\ \text{THEN} & (H' \wedge C) \rightarrowtail_R (B \wedge G \wedge H = H' \wedge C) \end{array} \tag{1}$$

If applied, a simplification rule *replaces* the matched CHR constraints in the state by the body of the rule. In the operational semantics, rules are applied until exhaustion, i.e. the CHR run-time system (which actually runs a CHR program by selecting applicable rules and matching constraints) computes the reflexive transitive closure $\rightarrowtail^*$ of $\rightarrowtail$. The CHR run-time system should stop immediately, when insertion of a built-in constraint makes C_b inconsistent. However, this *termination at failure* is not explicitly addressed in the operational semantics.

4.2 Transaction Logic

Transaction Logic ($\mathcal{TR}$) [3,4,5] is a conservative extension of classical predicate logic, where predicates can have side-effects on a database, allowing to model state changes. Similar to predicate logic, $\mathcal{TR}$ features a Horn fragment which supports logic programming. While $\mathcal{TR}$ is an extremely versatile logic to handle specification, execution, and reasoning on logic programs with updates, it suffices for this work to use a basic instantiation of $\mathcal{TR}$ which restricts side-effects to the updates *insertion* and *deletion* on a relational, ground database.

A database is a set of ground atoms. A sequence of databases $D_0, \ldots, D_n$ is called a *path* $\pi = \langle D_0, \ldots, D_n \rangle$ which can be *split* into sub-paths $\langle D_0, \ldots, D_i \rangle \circ \langle D_i, \ldots, D_n \rangle$ (for $0 \leq i \leq n$). Access to the database is restricted by two oracles: The *data oracle* $\mathcal{O}^d$ maps a database D to a set of ground atoms that are considered to be true along the path $\langle D \rangle$. Elementary database updates are captured by the *transition oracle* $\mathcal{O}^t$ which maps two databases D and D' to a set of ground atoms considered to be true along the path $\langle D, D' \rangle$.

Definition 1 (Path Structure with Relational Oracles). *A* path structure M *assigns a classical Herbrand structure (or* $\top$ *which satisfies everything) to every path and is subject to the following restrictions for ground atoms* p.

$$\begin{array}{lll} M(\langle D \rangle) \models p & \textit{if } p \in D & (p \in \mathcal{O}^d(D)) \\ M(\langle D, D' \rangle) \models p.ins & \textit{if } D' = D \cup \{p\} & (p.ins \in \mathcal{O}^t(D, D')) \\ M(\langle D, D' \rangle) \models p.del & \textit{if } D' = D \setminus \{p\} & (p.del \in \mathcal{O}^t(D, D')) \end{array} \tag{2}$$

Quantification of $\mathcal{TR}$ formulas and satisfaction of composed $\mathcal{TR}$ formulas are defined analogously to predicate logic: A $\mathcal{TR}$ formula with $\neg, \wedge, \vee$, or $\leftarrow$ as main connective is *satisfied along a path* π if the appropriate property holds between its sub-formulas *along the same path* π. Satisfaction from the basic properties (2) extends to the case of longer paths by the new *serial conjunction* operator: A serial conjunction $\phi \otimes \psi$ is satisfied along the path π iff ϕ is satisfied along π_1 and ψ is satisfied along π_2 for *some* split of the path $\pi = \pi_1 \circ \pi_2$. The modal *possibility* $\Diamond\phi$ expresses that ϕ is satisfiable along *some* path starting from the current database, formally $M(\langle D \rangle) \models \Diamond\phi$ iff there is a path π starting at database D with $M(\pi) \models \phi$.

The formal statement *executional entailment* links a program, a possible sequence of databases which captures the side-effects of the program, and the invocation of the program.

Definition 2 (Executional Entailment). *Consider a set of* $\mathcal{TR}$ *formulas* P *and an* execution path *consisting of a sequence of databases* $D_0, \ldots, D_n$*. A path structure* M_P *is a* model of P *iff* $M_P(\pi) \models \phi$ *for every* $\mathcal{TR}$ *formula* $\phi \in P$ *and every path* π*.*

$$P, D_0, \ldots, D_n \models_x \phi \quad \textit{iff} \quad M_P(\langle D_0, \ldots, D_n \rangle) \models \phi \textit{ for every model } M_P \textit{ of } P$$

Executional entailment, $\models_x$, selects one of (possibly several) valid execution paths, for which ϕ is true for all models M_P of P. A model M_P of P is a path structure that respects the oracles and satisfies every formula of P along every path. The execution path $D_0, \ldots, D_n$ records all side-effects when establishing the truth of ϕ, i.e. the "successful program invocation of ϕ for program P *can* update the database along execution path from D_0 to $D_1 \ldots$ to D_n" [3, p. 31].

Example 2. Consider the $\mathcal{TR}$ program $P = \{r \leftarrow p \otimes p.del\}$. Invocation of r deletes p from the database, but only if p was initially present and we have $P, \{p, q\}, \{q\} \models_x r$ but $P, \{q\}, \{\} \not\models_x r$. Also, deletion of p should be the only side-effect of the invocation of r, hence $P, \{p, q\}, \{\} \not\models_x r$ as $M_P(\langle\{p, q\}, \{\}\rangle) \models r$ is not correct for *all* models M_P of P. When we insert q under the condition that r *can* execute, e.g. $P, \{p\}, \{p, q\} \models_x \Diamond r \otimes q.ins$, we keep p in the database.

For the class of *serial-Horn programs* (i.e. sets of Horn rules with only serial conjunctions in the r.h.s) and *serial queries*, $\mathcal{TR}$ features an *executional deduction inference system* [3]. For a serial-Horn program P, an initial database D_0, and an existentially quantified serial query $(\exists)\,\phi$, it infers the *sequent* $P, D_0\text{---}\vdash (\exists)\,\phi$ iff there is an executional entailment of $(\exists)\,\phi$ along an *execution path* starting from D_0. Most importantly the system both tries to infer the truth of $(\exists)\,\phi$ and computes the necessary changes to D_0 which we record in $D_0, \ldots, D_n$. Formally, the following fundamental sound- and completeness result links the model-theoretic executional entailment with the mechanised executional deduction.

Theorem 1 (Bonner and Kifer [3]). $P, D_0, \ldots, D_n \models_x (\exists)\phi$ *iff there is an executional deduction of* $(\exists)\phi$ *with execution path* $D_0, \ldots, D_n$.

For our serial-Horn program P from Example 2 we have $P, \{p, q\}\text{---}\vdash r$ and the successful inference of the query r computes the execution path $\{p, q\}, \{q\}$ from the initial database $\{p, q\}$. Of course, we cannot infer $P, \{q\}\text{---}\vdash r$ as there exists no execution path for the query r starting from $\{q\}$. By the definition of executional entailment, an execution either succeeds or all tentative side-effects are rolled back. Due to this *transaction property* of $\mathcal{TR}$ we cannot infer $P, \{p\}\text{---}\vdash r \otimes r$ and the tentative deletion of p by the first call to r is not manifested as the second call to r fails.

5 The Details: CHR-to-$\mathcal{TR}$-Mapping

We map CHR states to databases, adapt the data oracle $\mathcal{O}^d$, map CHR rules to serial-Horn $\mathcal{TR}$ rules, and specify the CHR run-time system as a generalised-Horn $\mathcal{TR}$ program. We then show our sound- and completeness result that links CHR derivations with executional entailment statements of $\mathcal{TR}$.

For this paper, we restrict ourselves to *range-restricted ground CHR*. Range-restricted CHR rules have no local variables, i.e. every variable in each rule already occurs in the head of the rule and all CHR states are ground as there are no variables in the goal.

5.1 Mapping CHR States to Valid Databases

We map each ground, user-defined constraint c_i of a CHR state (recall that a CHR state is a multi-set conjunction) to a $\mathcal{TR}$-predicate $u(c_i, i)$ where the second argument is a unique identifier – we use a natural number. We trail a new, unique identifier k in a bookkeeping $\mathcal{TR}$-predicate $n(k)$ (assuming that

$u/2$ and $n/1$ are not defined by CT). Reflecting user-defined constraints as $\mathcal{TR}$-function symbols allows us to specify the necessary bookkeeping for the insertion and deletion of user-defined constraints as a serial-Horn program.

Definition 3. *A* valid database *D contains one bookkeeping predicate $n(k)$, predicates $u/2$ with unique identifiers that are all smaller than k, and built-ins b_i. The mapping $m_\mathtt{s}$ is defined from the set of CHR states $\mathcal{S}$ (consisting of user-defined constraints c_i and built-ins b_i) to the set of valid databases $\mathcal{D}$ by*

$$(\bigwedge_{0\le i<k} c_i \wedge \bigwedge_{0\le i<l} b_i) \mapsto \{u(c_i,i) : 0 \le i < k\} \cup \{n(k)\} \cup \{b_i : 0 \le i < l\} \ .$$

Two valid databases are equivalent, *$\sim$, iff they differ only in the set of identifiers (including the argument of the bookkeeping predicate) and there is a bijective mapping between these sets.*

Clearly, $m_\mathtt{s}(c(a)) = \{u(c(a),0), n(1)\}$ and $\{u(c(a),5), n(9)\}$ are equivalent. We update valid databases through the serial-Horn program $P_\mathtt{basic}$:

$$\begin{aligned} &\mathrm{udel}(U) \leftarrow u(U,K) \otimes u(U,K).del \\ &\mathrm{uins}(U) \leftarrow n(K) \otimes n(K).del \otimes n(K+1).ins \otimes u(U,K).ins \end{aligned} \tag{3}$$

Deletion of a ground user-defined constraint is conditional (cf. Example 2) and insertion requires some bookkeeping.

Property 1 (Conditional Deletion of User-Defined Constraints). Invocation of $\mathrm{udel}(c)$ deletes a copy of the ground, user-defined constraint c from the valid database $D + \{u(c,k)\}$[3] and terminates in the valid database D.

$$P_\mathtt{basic}, D + \{u(c,k)\}, D \models_x (\exists)\ \mathrm{udel}(c) \text{ with } k \in \mathbf{N}$$

Property 2 (Insertion of User-Defined Constraints). Invocation of $\mathrm{uins}(c)$ inserts a new copy of the ground, user-defined constraint c into the valid database $D + \{n(k)\}$ and terminates in the valid database $D + \{n(k+1), u(c,k)\}$.

$$P_\mathtt{basic}, D + \{n(k)\}, \ldots, D + \{n(k+1), u(c,k)\} \models_x (\exists)\ \mathrm{uins}(c) \text{ with } k \in \mathbf{N}$$

5.2 Mapping the Built-In Theory *CT* to the Data Oracle $\mathcal{O}^d$

For range-restricted ground CHR the entailment condition of the state transition system, defined in (1) can be simplified as there are no local variables. Because we match the head H with the ground constraints H' from the store, the formula $\exists \bar{X}(H = H' \wedge G)$ is ground. We extend the relational data oracle $\mathcal{O}^d$, defined in (2), to implement the built-in constraint theory CT.

Definition 4 (Data Oracle as Built-in Solver). *For any database D and ground atomic built-in ϕ, the data oracle respects CT:*
$\phi \in \mathcal{O}^d(D)$ if $CT \models D_b \rightarrow \phi$ for the conjunction D_b of built-in predicates of D.

[3] We use "+" to denote disjoint set union.

5.3 Mapping CHR Rules to Serial-Horn Rules in $\mathcal{TR}$

We map CHR rules to $\mathcal{TR}$ rules that update the database through $P_{\texttt{basic}}$ as defined in (3). We normalise any range-restricted CHR rule to contain no function symbols in the head by introducing a new variable for any implicit equality in the head and adding an explicit (new) equality to the guard, e.g. $r \ @ \ p(a) \Leftrightarrow \mathit{true}$ normalises to $r \ @ \ p(X) \Leftrightarrow X = a \mid \mathit{true}$.

Definition 5. *Consider a normalised range-restricted simplification rule r. The head H of rule r is the (multiset) conjunction $\bigwedge_{i=0}^{n_h} h_i$ of user-defined constraints h_i, the guard G is the conjunction $\bigwedge_{i=1}^{n_g} g_i$ of built-in constraints g_i ($n_g = 0$ represents true), and the body B is the (multiset) conjunction $\bigwedge_{i=1}^{n_u} u_i$ of constraints u_i ($n_u = 0$ represents true). The auxiliary t maps user-defined constraints u_i to* $\mathrm{uins}(u_i)$ *and built-in constraints to $u_i.ins$. We define the mapping $m_{\mathtt{r}} : r \ @ \ H \Leftrightarrow G \mid B \mapsto r^{\mathcal{TR}}$ by*

$$r^{\mathcal{TR}} \equiv \mathrm{chr}(r) \leftarrow \left(\bigotimes_{i=0}^{n_h} \mathrm{udel}(h_i)\right) \otimes \left(\bigotimes_{i=1}^{n_g} g_i\right) \otimes \left(\bigotimes_{i=1}^{n_u} t(u_i)\right) \ . \tag{4}$$

Our mapping $m_{\mathtt{r}}$ is guided by the intuition that establishing the truth of $\mathrm{chr}(r)$ should have the same effect on the database as rule application by CHR's state transition system on the constraint store. The body of $\mathrm{chr}(r)$ consists of parts corresponding to head, guard, and body of the CHR rule r: First, we succinctly query the database for copies of each head constraint and delete them. Then we pass the check for the guard (as r is range-restricted, all variables in the guard are now bound) to our data oracle which respects the built-in constraint theory CT. Finally, we add the body constraints, labelling each inserted user-defined constraint with a new identifier.

By the transaction property of $\mathcal{TR}$ we can safely intertwine applicability checks with updates of the database, e.g. if the guard fails, the tentative deletions of the user-defined head constraints are undone.

Formally, application of r by the state transition system is equivalent to executional entailment of $\mathrm{chr}(r)$ modulo identifier renaming.

Lemma 1. *Consider two ground CHR states S and S', two valid databases D and D', a normalised range-restricted CHR simplification rule r, and its mapping $r^{\mathcal{TR}}$ as defined in (4). For $D \sim m_{\mathtt{s}}(S)$ and $D' \sim m_{\mathtt{s}}(S')$ we have*

$$S \rightarrowtail_r S' \ \ \textit{iff} \ \ P_{\texttt{basic}} + \{r^{\mathcal{TR}}\}, D, \ldots, D' \models_x (\exists) \ \mathrm{chr}(r) \ .$$

5.4 Sound and Complete: CHR Run-Time System in $\mathcal{TR}$

We now extend Lemma 1 from a single rule step of a single CHR rule to a CHR derivation of a CHR program by integrating the fixpoint computation, i.e. the operational semantics of CHR, into $\mathcal{TR}$.

Our main result shows that our mapping from CHR to $\mathcal{TR}$ is sound- and complete w.r.t. the operational semantics of CHR. To this end, we express *applicability of a CHR rule* in state S by $P, D \models_x (\exists) \Diamond \operatorname{chr}(R)$ with $D \sim m_{\mathtt{s}}(S)$ and use induction on the derivation length for the extension from $\rightarrowtail$ to $\rightarrowtail^n$.

Theorem 2 (Sound- and Completeness). *Consider two ground CHR states S and S', two valid databases D and D', a CHR program P consisting of range-restricted simplification rules, and its mapping $P^{\mathcal{TR}} = P_{\mathtt{basic}} + \{r^{\mathcal{TR}} : r \in P\}$. For $D \sim m_{\mathtt{s}}(S)$, $D' \sim m_{\mathtt{s}}(S')$, and an execution path π starting in D and ending in D' we have*

$$S \rightarrowtail_P^* S' \;\; \textit{iff} \;\; P^{\mathcal{TR}}, \pi \models_x (\exists) \left(\bigotimes_{i=1}^{n} \operatorname{chr}(R_i) \right) \otimes [\neg \Diamond \operatorname{chr}(R)] \;\; \textit{with } n \in \mathbf{N}$$

where $[\neg \Diamond \operatorname{chr}(R)]$ restricts satisfaction of $\neg \Diamond \operatorname{chr}(R)$ to paths of length one.

We now sketch how to implement the CHR run-time system as $\mathcal{TR}$ program with hypothetic goals (to express possibility) and negated goals (to check that no rule is applicable). We capture the fixpoint semantics of CHR as

$$\text{fixpoint} \leftarrow \textit{while} \;\; \text{applicable} \;\; \textit{do} \;\; \operatorname{chr}(R) \;\; \textit{od}$$

and implement the imperative *while*-loop programming construct as a simple *generalised-Horn program* $P_{\mathtt{runTime}}$ in $\mathcal{TR}$.[4]

$$\text{fixpoint} \leftarrow \operatorname{chr}(R) \otimes \text{fixpoint} \tag{5}$$

$$\text{applicable} \leftarrow \Diamond \operatorname{chr}(R) \tag{6}$$

$$\text{fixpoint} \leftarrow [\neg \, \text{applicable}] \tag{7}$$

Rule (5) succeeds if the call $\operatorname{chr}(R)$ – which successfully applies a CHR rule R – succeeds. In this case we call fixpoint (tail-recursively). We need two generalised-Horn rules to express that no CHR rule is applicable: Rule (6) succeeds if a CHR rule is applicable and this test leaves the database D untouched and rule (7) succeeds if no CHR rule is applicable at the current state using *negation-as-failure* to compute $[\neg \, \text{applicable}]$.

Bonner and Kifer give an extended (sound- and complete) executional deduction inference system that integrates the $\Diamond$ operator. Negation $\neg$ is then treated (outside of the proof system) as negation-as-failure. A slight modification of the model-theoretic executional statement allows to give a declarative account for locally stratified generalised-Horn programs. Compared to Definition 2, we no longer look at *all* but only at the *perfect models* of the program. As $P_{\mathtt{runTime}}$ is stratified we can use this executional entailment statement, $\models_x^{\text{perf}}$, cf. [3].

Corollary 1. *Under the premises of Theorem 2 we have*

$$S \rightarrowtail_P^* S' \;\; \textit{iff} \;\; P^{\mathcal{TR}} + P_{\mathtt{runTime}}, \pi \models_x^{\text{perf}} (\exists) \text{ fixpoint} \quad .$$

[4] Note that we can add *termination at failure* by adding “D_b consistent” to the loop condition easily, allowing to reason also on *failed derivations.*

Our *declarative $\mathcal{TR}$ semantics* of the CHR program P is the *perfect-model semantics of the generalised-Horn $\mathcal{TR}$ program $P^{\mathcal{TR}} + P_{\mathtt{runTime}}$*. Invocation of fixpoint – on the other hand – computes $\rightarrowtail^*$ as side-effect on the database, i.e. captures the *operational semantics of CHR*. This brings together the operational and declarative semantics of CHR in $\mathcal{TR}$.

6 Examples

We use the FLORA-2 system [14], a sophisticated object-oriented, knowledge management environment that implements the executional deduction inference system of $\mathcal{TR}$ by offering backtrackable deletion and insertion of facts, to execute and reason on CHR. Similar to Prolog, but with handling updates in a declarative way, serial queries are treated from left to right and one database is kept at any time. We have $P, D_0, \text{---} \vdash (\exists)\,\phi$ iff the query "?- ϕ" succeeds for the program P from the initial database D_0. In this case, FLORA-2 updates the database according to the computed executional path as side-effect.

Example 3 (Coin-Throw Simulation Program). We revisit Example 1 in detail. For the program $P^{\mathcal{TR}}_{\mathtt{coin}} = P_{\mathtt{basic}} + \{r_1^{\mathcal{TR}}, r_2^{\mathcal{TR}}\}$, defined in (3) and by (4), and the initial database $m_{\mathtt{s}}(\mathtt{throw}) = \{u(\mathtt{throw}, 0), n(1)\}$ we throw a coin by querying "?- chr(R)". This query succeeds, returns an answer substitution for R, and updates the database. In a subsequent query "?- $u(S, I)$" we query the current database state for the side S of the coin.

We now prove properties **(P1-3)** from Section 2 automatically:

(P1) The query "?- chr$(R), u(\mathtt{nautica}, I)$"[5] succeeds from D_0, i.e. we have a mechanical proof that a computation $(\mathtt{throw}) \rightarrowtail (\mathtt{nautica})$ exists. Due to the post-condition $u(\mathtt{nautica}, I)$ the FLORA-2 system backtracks over rule-application if rule r_1 is selected in the first try.

(P2) Throwing a coin cannot yield both `caput` and `nautica` is true because the query "?- chr$(R), u(\mathtt{nautica}, I), u(\mathtt{caput}, J)$" fails (from database D_0).

(P3) Applying rule r_1 cannot yield `nautica` as "?- chr$(r_1), u(\mathtt{nautica}, I)$" fails (from database D_0).

For complex CHR programs this knowledge is much less trivial and very valuable for understanding. While CHR programs are usually very concise, debugging is often tedious, and automatised reasoning is highly desirable.

Example 4 (Greatest Common Denominator). Euclid's algorithm to compute the greatest common denominator (gcd) is probably the first algorithm in history that is still commonly used. The CHR implementation of the gcd consists of only two rules, where the built-in theory CT also contains the order between natural numbers.

$$r_1 \;@\; \mathtt{gcd}(0) \Leftrightarrow \mathit{true}$$

$$r_2 \;@\; \mathtt{gcd}(X_1) \wedge \mathtt{gcd}(X_2) \Leftrightarrow 0 < X_1 \wedge X_1 \leq X_2 \mid \mathtt{gcd}(X_1) \wedge \mathtt{gcd}(X_2 \% X_1)$$

[5] The serial conjunction operator $\otimes$ is written as comma in FLORA-2.

The CHR derivation $(\mathtt{gcd}(24) \wedge \mathtt{gcd}(30) \wedge \mathtt{gcd}(42)) \rightarrowtail^* (\mathtt{gcd}(6))$ computes the gcd of 24, 30, and 42. The gcd algorithm can be seen as a (very basic) nonmonotonic reasoning service, as e.g. adding $\mathtt{gcd}(7)$ to the goal invalidates the original answer constraint $\mathtt{gcd}(6)$.

As FLORA-2 does not implement the possibility operator $\Diamond$, we carry out the next two inferences mechanically – but not automatically. We now assume $D_0 = m_{\mathtt{s}}(\mathtt{gcd}(24) \wedge \mathtt{gcd}(30) \wedge \mathtt{gcd}(42))$ as initial database and simulate one (of several possible) CHR derivations – recall that CHR is a committed-choice language – by inferring the sequent $P^{\mathcal{TR}}_{\mathtt{gcd}} + P_{\mathtt{runTime}}, D_0\text{---} \vdash (\exists)$ fixpoint. Then we inspect the final database D_n of the computed executional path $D_0, \ldots, D_n$ which contains $u(\mathtt{gcd}(6), k)$ with some identifier $k \in \mathbf{N}$. Similarly, we have a mechanical proof that no $\mathtt{gcd}(0)$ constraint is in the final constraint store as we cannot infer $P^{\mathcal{TR}}_{\mathtt{gcd}} + P_{\mathtt{runTime}}, D_0\text{---} \vdash (\exists)$ fixpoint $\otimes u(\mathtt{gcd}(0), I)$. Here the post-condition $u(\mathtt{gcd}(0), I)$ forces us to backtrack over all possible execution paths, i.e. CHR derivations, due to non-deterministic constraint and rule selection.

We can reason automatically on the derivation length: There is no CHR derivation of the gcd with only 4 CHR rule applications because the FLORA-2 query "?- $\mathrm{chr}(r_2), \mathrm{chr}(r_2), \mathrm{chr}(r_1), \mathrm{chr}(r_1)$" fails. Similarly, we prove that the gcd can be computed with derivation length 5 and that another CHR derivation with length 8 exists, e.g. $(\underline{24}, 30, \underline{42}) \rightarrowtail_{r_2} (24, \underline{30}, \underline{18}) \rightarrowtail_{r_2} (\underline{24}, 12, \underline{18}) \rightarrowtail_{r_2} (6, \underline{12}, \underline{18}) \rightarrowtail_{r_2} (\underline{6}, \underline{12}, 6) \rightarrowtail_{r_2} (6, \underline{0}, 6) \rightarrowtail_{r_1} (\underline{6}, \underline{6}) \rightarrowtail_{r_2} (6, \underline{0}) \rightarrowtail_{r_1} (6)$.

7 Conclusion

We showed how we can execute and reason on execution of CHR programs within one logical framework by integrating the operational and declarative semantics of CHR into $\mathcal{TR}$. We introduced rule names into the formalism, mapped CHR states and CHR rules to databases and $\mathcal{TR}$ rules, and mapped the CHR runtime system for non-deterministic rule application to a recursively defined $\mathcal{TR}$-predicate. The *perfect-model semantics* of a generalised-Horn $\mathcal{TR}$ program is our new *declarative $\mathcal{TR}$ semantics* of CHR. The model-theoretical executional entailment statement ("one possible execution sequence") brings together $\mathcal{TR}$ program, execution path, and program invocation. The executional deduction inference system mechanically infers a $\mathcal{TR}$-query and computes the necessary updates to the database. We showed execution and automatic reasoning on CHR using the FLORA-2 system.

By bringing the operational semantics of CHR into $\mathcal{TR}$, we merged operational and declarative semantics of CHR in one formal system which allows both execution and reasoning. Our approach is more practical than the one taken for the available declarative semantics of CHR. Both the declarative classic predicate logic semantics and its recent extension to linear logic are more theoretical. They cannot execute a CHR program, cannot reason on its execution, and offer only limited help to mechanise reasoning.

We plan to extend our mapping and to investigate the connections between the formalisms of CHR, constraint programming, and $\mathcal{TR}$ in more detail:

- Lift the restrictions on ground, range-restricted CHR by encoding variables in the database, introduce propagation rules $H \Rightarrow G \mid B$ (which do not remove H upon application), and avoid trivial non-termination, by encoding the *propagation history* in the database.
- Use full $\mathcal{TR}$ to reason on the effect properties [5] of a CHR program starting from our new declarative $\mathcal{TR}$ semantics.
- Another direction is to extend $\mathcal{TR}$ with constraints according to the *general CLP-scheme* [7] which would then allow constraint solving over a side-effect-full, logic programming host language.

As CHR enables the direct implementation of many important monotonic and nonmonotonic reasoning services, this work can be seen as very first step towards a unifying framework to specify, execute, and reason about the semantics of rule-based programs, knowledge bases, and inference engines as envisioned in [10].

Acknowledgements. We thank Thom Frühwirth and the anonymous reviewers for their valuable comments which helped us to improve this paper.

References

1. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.
2. H. Betz and T. Frühwirth. A linear-logic semantics for Constraint Handling Rules. In *CP 2005*, volume 3709 of *LNCS*, pages 137–151. Springer, 2005.
3. A. J. Bonner and M. Kifer. Transaction Logic programming (or, a logic of procedural and declarative knowledge). Technical Report CSRI-323, Computer Systems Research Institute, University of Toronto, Canada, 1995.
4. A. J. Bonner and M. Kifer. A logic for programming database transactions. In *Logics for Databases and Information Systems*, pages 117–166. Kluwer, 1998.
5. A. J. Bonner and M. Kifer. Results on reasoning about updates in Transaction Logic. In *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 166–196. Springer, 1998.
6. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3):95–138, 1998.
7. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *J. Logic Programming*, 37(1–3):1–46, 1998.
8. M. Kifer. Nonmonotonic reasoning in FLORA-2. In *LPNMR 2005*, volume 3662 of *LNCS*, pages 1–12. Springer, 2005.
9. M. J. Maher. Logic semantics for a class of committed-choice programs. In *ICLP 87*, pages 858–876. The MIT Press, 1987.
10. J. Robin. Reused Oriented Automated Reasoning Software (ROARS) project web page, 2005. http://www.cin.ufpe.br/~jr/mysite/RoarsProject.html.
11. T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules. *J. Theory and Practice of Logic Programming*, 6(1&2):213–224, 2006.
12. T. Schrijvers et al. The Constraint Handling Rules (CHR) web page, 2007. http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/.
13. M. Thielscher. FLUX: A logic programming method for reasoning agents. *J. Theory and Practice of Logic Programming*, 5(4&5):533–565, 2005.
14. G. Yang, M. Kifer, C. Zhao, and V. Chowdhary. *FLORA-2: User's Manual, Version 0.94 (Narumigata)*, 2005. http://flora.sourceforge.net/docs/.

Conditional Planning with External Functions

Davy Van Nieuwenborgh[1,⋆], Thomas Eiter[2,⋆⋆], and Dirk Vermeir[1]

[1] Vrije Universiteit Brussel, VUB
Dept. of Computer Science
Pleinlaan 2, B-1050 Brussels, Belgium
{dvnieuwe,dvermeir}@vub.ac.be
[2] Institute of Information Systems,
Vienna University of Technology, Austria
eiter@kr.tuwien.ac.at

Abstract. We introduce the logic-based planning language $\mathcal{K}^c$ as an extension of $\mathcal{K}$ [5]. $\mathcal{K}^c$ has two advantages upon $\mathcal{K}$. First, the introduction of external function calls in the rules of a planning description allows the knowledge engineer to describe certain planning domains, e.g. involving complex action effects, in a more intuitive fashion then is possible in $\mathcal{K}$. Secondly, in contrast to the conformant planning framework $\mathcal{K}$, $\mathcal{K}^c$ is formalized as a conditional planning system, which enables $\mathcal{K}^c$ to solve planning problems that are impossible to express in $\mathcal{K}$, e.g. involving sensing actions. A prototype implementation of conditional planning with $\mathcal{K}^c$ is build on top of the DLV$^{\mathcal{K}}$system, and we illustrate its use by some small examples.

1 Introduction

In general, the task of a planning system consists of finding, dependent on the initial state, a sequence of actions such that a certain goal will be reached if, starting from that initial state, the actions in the sequence are executed in the correct order. In the context of logic-based languages, a number of frameworks have been proposed in the literature to logically describe and reason about such planning problems, e.g. [8,9,3,13,11,4,15,5,16].

In [4,5], the planning language $\mathcal{K}$ was introduced as a system for planning under incomplete knowledge, i.e. rather than describing transitions between states of the world (complete knowledge), one can describe in $\mathcal{K}$ transitions between states of (incomplete) knowledge. $\mathcal{K}$'s ability to deal with incomplete knowledge comes from the fact that its semantics is close in spirit to the answer set semantics [7], thus allowing negation as failure (naf) to be used in the causation rules of the planning description. As for answer sets, the semantics of $\mathcal{K}$ is defined in a two step process. First, the semantics is defined for planning descriptions without naf. Next, for arbitrary planning descriptions, a reduction is introduced, similar to the GL-reduct [7], which removes naf from the planning descriptions w.r.t. a candidate state transition. Finally, if the state transition

⋆ Supported by the Flemish Fund for Scientific Research (FWO-Vlaanderen).
⋆⋆ Supported by the Austrian Science Fund project FWF P16536-N04.

is valid w.r.t. the reduct of the planning description, the state transition is valid for the planning description.

Although $\mathcal{K}$ is an expressive framework, it suffers from two shortcomings. The first problem encountered is a direct consequence of the fact that $\mathcal{K}$ is a conformant planning system. Consider e.g. the problem (taken from [15]) of defusing a bomb. To defuse the bomb, a special lock has to be placed in the locked position. If one defuses the bomb while the lock is unlocked, the bomb explodes and the person defusing the bomb is killed. The person defusing the bomb can determine if the bomb is locked or unlocked by looking at it, and she can switch the lock from the locked to the unlocked position and vice-versa. Obviously, no action can be undertaken once the bomb has exploded. A possible encoding of this problem in $\mathcal{K}$ is depicted below.

```
fluents: exploded. locked. unlocked. disarmed. dead.
actions: disarm. turn. look.
always: caused exploded after disarm, unlocked.
        caused disarmed after disarm, locked.
        caused unlocked after turn, locked.
        caused locked after turn, unlocked.
        caused dead if exploded.
        caused locked if not unlocked after look.
        caused unlocked if not locked after look.
        executable disarm if not exploded, not dead.
        nonexecutable disarm if not locked, not unlocked.
        executable turn if not exploded, not dead.
        executable look if not exploded, not dead.
        inertial dead.
        noConcurrency.
goal: disarmed?(3)
```

One can check that no conformant plan[1] for the above problem exists. Indeed, if the action *look* is performed we know that the bomb is either locked or unlocked, but both cases need a different, but incompatible, strategy for defusing the bomb. In such cases, one needs to switch from conformant to conditional planning. A conditional plan for the above problem could be

$$look; case \begin{cases} \{locked\} \rightarrow disarm \\ \{unlocked\} \rightarrow turn; disarm \end{cases} \quad (1)$$

Thus, the person defusing the bomb first looks at the bomb. If it is locked, she disarms it; but if it is unlocked, she first turns the switch and then disarms the bomb.

A second problem with $\mathcal{K}$ is encountered when one needs to describe effects of actions that are not easily captured by logic programming rules. Consider the following variant of the Bubble Breaker game. We have a grid of a certain height and certain width and on each position in the grid we have a colored bubble. We can tap on any position in the grid, but if we tap a position, the biggest connected (left/right/up/down directions) region of bubbles with the same color and including the tapped position, is removed from the board. In case the biggest connected region only contains the tapped position itself, nothing is removed. After the bubbles are removed, the remaining bubbles in each column fall down, such that there are no holes between the bubbles in the same column. An illustration of the course of the game is provided in Figure 1. The goal of the game is to tap certain positions in such a way that all the bubbles are removed from the board.

[1] A conformant (or secure) plan is a plan that always leads to a goal state when it is executed.

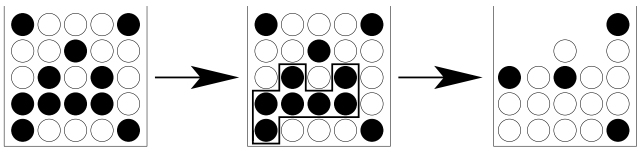


Fig. 1. Left: the initial configuration of the board. Middle: the region that is selected after tapping position $(1, 1)$, i.e. the lower left corner. Right: the configuration after the selected block is removed.

Clearly, it is not so easy to describe the effects of the *tap* action using causation rules. First, one has to compute the biggest connected region, afterwards that region has to be removed and finally the blocks on top of the removed region have to fall down. One solution is to add an additional action *update* and some extra fluents to take care of this complicated process of computing the effects of the *tap* action. The complete encoding in $\mathcal{K}$ can be found in [17], but we will briefly describe parts of it here.

First, some (non)executability statements are needed for the tap and update action, i.e. when there are holes between bubbles in the same column, the update action is executable and the tap action is not.

```
executable update if pos(X,Y,C), not placed(X,Yn), succ(Yn,Y).
nonexecutable tap(X,Y) if pos(A,B,C), not placed(A,Bn), succ(Bn,B).
```

Next, to compute the region that has to be removed from the board and to actually remove that region, one can use the following set of causation rules.

```
caused remove(X,Y,C) after tap(X,Y), pos(X,Y,C).
caused remove(Xn,Y,C) if remove(X,Y,C) after pos(Xn,Y,C), succ(X,Xn).
caused remove(Xn,Y,C) if remove(X,Y,C) after pos(Xn,Y,C), succ(Xn,X).
caused remove(X,Yn,C) if remove(X,Y,C) after pos(X,Yn,C), succ(Y,Yn).
caused remove(X,Yn,C) if remove(X,Y,C) after pos(X,Yn,C), succ(Yn,Y).
caused pos(X,Y,C) if not remove(X,Y,C) after pos(X,Y,C), tap(A,B).
```

Finally, to encode that the bubbles have to fall down, the update action will be executed a number of times and each time all the bubbles with a free position below them are moved one position lower.

```
caused pos(X,Yn,C) after update, pos(X,Y,C), not placed(X,Yn), succ(Yn,Y).
caused pos(X,1,C) after update, pos(X,1,C).
caused pos(X,Y,C) after update, pos(X,Y,C), pos(X,Yn,Cn), succ(Yn,Y).
```

Note that a plan for this problem using the encoding in $\mathcal{K}$ will always be of the form

$$tap(x, y); update; \ldots, update; tap(v, w); update; \ldots; tap(k, l); \ldots; update; tap(a, b).$$

To solve problems like the above, we propose to extend the action language $\mathcal{K}$ by allowing external functions to be called[2] in the causation rules of an action description. Intuitively, an external function call allows a knowledge engineer to import fluent information from an external source in response to an action that is performed. These external function calls are inspired by the DLVHEX system [6], i.e. a system for answer

[2] We assume that the external functions have no side effects when they are called.

set programming with higher-order atoms and external evaluations. For this reason, our external functions will use the same notation as the ones in DLVHEX.

Reconsider e.g. the Bubble Breaker game and the following causation rule involving an external function $\&nbc$, which stands for $new\_board\_configuration$

$$caused\ pos(Xn, Yn, Cn)\ after\ \&nbc[X, Y](Xn, Yn, Cn), tap(X, Y)\ .$$

When the action tap is executed for a certain position (X, Y), the external function $\&nbc$ will return a set of tuples of the form (Xn, Yn, Cn), with (Xn, Yn) a position and Cn a color, that correspond to the new configuration of the grid when a tap action is executed on the given position. The above rule imports this output into the fluent pos, yielding that the above rule can be intuitively read as "executing the action tap on position X and Y of the grid causes the fluents $pos(Xn, Yn, Cn)$ to become true if the tuple (Xn, Yn, Cn) is an output of the external function call $\&nbc$".

The external function in the previous rule is deterministic, i.e. for a given configuration of the grid and a position (X, Y), the external function always produces the same output tuples. However, when we reconsider the defusing a bomb problem with external functions $\&look\_effect$ and $\&disarm\_effect$ computing the effects of the *look* (resp. *disarm*) action, we get non-deterministic outcomes. E.g., consider the following causation rules

$$caused\ X\ after\ \&look\_effect()[X], look.$$
$$caused\ X\ after\ \&disarm\_effect()[X], disarm.$$

Intuitively, the external functions will return, when their corresponding action gets executed, some fluents as output which are imported into the planning process by the above causation rules. Clearly, the outcomes of these functions are non-deterministic. E.g., disarming the bomb when you don't know if it is locked or not can either yield *disarmed* or *exploded*. Further, this example on non-deterministic external functions demonstrates that the proposed extension to $\mathcal{K}$ can be used to simulate sensing actions [12,10,14,15,16]. By combining an ordinary action together with an external function that materializes the sensed values of executing the "sensing" action, we have very flexible means to encode sensing, e.g. sensing more than one fluent, or a combination of fluents at the same time; or sensing that depends on what is known (or not known) in the current (incomplete) state, ...

The rest of the paper is organized as follows. In Section 2 we introduce the syntax of $\mathcal{K}^c$, while its semantics is defined in Section 3. Before concluding in Section 5, we discuss our prototype implementation in section 4.

2 Syntax of $\mathcal{K}^c$

A *signature* of a planning domain with external actions PD is a tuple $PD_{sig} = (\sigma^{act}, \sigma^{fl}, \sigma^{ec}, \sigma^{typ}, \sigma^{con}, \sigma^{var})$, where σ^{act}, σ^{fl}, σ^{ec} and σ^{typ} are mutually disjoint sets of respectively action, fluent, external function and type names. The names in σ^{act}, σ^{fl} and σ^{typ} actually correspond to predicate symbols, so we associate with each of them an arity $n \geq 0$. On the other hand, the names in σ^{ec} correspond to external predicate symbols, with whom we associate both an input arity i and an output arity o $(i, o \geq 0)$[3]. Further, σ^{con} and σ^{var} are mutually disjoint sets of respectively constants and variable symbols[4].

For a given signature PD_{sig}, an *action atom* is defined as $p(t_1, \ldots, t_n)$, where $p \in \sigma^{act}$, n is the arity associated with p and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var} \cup \sigma^{fl}$. We define *fluent atoms* and *type atoms* similarly by substituting $p \in \sigma^{act}$ by $p \in \sigma^{fl}$ or $p \in \sigma^{typ}$ respectively. An *external function call* is defined as $\&p[i_1, \ldots, i_m](o_1, \ldots, o_n)$, where $p \in \sigma^{ec}$, m, n are the input and output arities associated with p and $i_1, \ldots, i_m, o_1, \ldots, o_n \in \sigma^{con} \cup \sigma^{var} \cup \sigma^{fl}$. A *variable atom* is defined as $X(t_1, \ldots, t_n)$, $n \geq 0$, where $X \in \sigma^{var}$ and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var} \cup \sigma^{fl}$.

An action (resp. fluent, external function call, type, variable) literal is an action (resp. fluent, external function call, type, variable) atom a or its classical negation $\neg a$. For a set of literals X we use $\neg X$ to denote the set $\{\neg p \mid p \in X\}$, where $\neg(\neg a) \equiv a$ for an atom a. Furthermore, we use X^+ (resp. X^-) to denote the set of positive (resp. negative) literals in X. To denote the set of all action (resp. fluent, external function call, type, variable) literals that can be formed using the signature, we use $\mathcal{L}_{act}$ (resp. $\mathcal{L}_{fl}, \mathcal{L}_{ec}, \mathcal{L}_{typ}, \mathcal{L}_{vat}$). In addition, we use $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$, $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup {\mathcal{L}_{act}}^+ \cup {\mathcal{L}_{ec}}^+$ and $\mathcal{L} = \mathcal{L}_{fl,typ} \cup {\mathcal{L}_{act}}^+ \cup {\mathcal{L}_{ec}}^+$.

All actions, fluents, and external function calls that can be used in a planning description have to be declared using the following declaration rules.

Definition 1. *An **action** (resp. **fluent**) **declaration** is an expression of the form*

$$p(X_1, \ldots, X_m) \textit{ requires } t_1, \ldots, t_n$$

where either $p \in \sigma^{act}$ (resp. $p \in \sigma^{fl}$) or $p \in \sigma^{var}$, i.e. either $p(X_1, \ldots, X_m) \in {\mathcal{L}_{act}}^+$ (resp. $p(X_1, \ldots, X_m) \in {\mathcal{L}_{fl}}^+$) or $p(X_1, \ldots, X_m) \in {\mathcal{L}_{vat}}^+$, and $X_1, \ldots, X_m \in \sigma^{var}$. Further, $t_1, \ldots, t_n \in \mathcal{L}_{typ}$, $n \geq 0$, and every X_i (and p if $p \in \sigma^{var}$) occurs in $t_1, \ldots, t_n$. Whenever $n = 0$, the keyword requires may be omitted.

*An **external function call declaration** is an expression of the form*

$$\&p[I_1, \ldots, I_m](O_1, \ldots, O_n) \textit{ requires } t_1, \ldots, t_k \textit{ ranges } r_1, \ldots, r_l$$

where $\&p[I_1, \ldots, I_m](O_1, \ldots, O_n) \in {\mathcal{L}_{ec}}^+$ and $I_1, \ldots, I_m, O_1, \ldots, O_n \in \sigma^{var}$. Further, $t_1, \ldots, t_k, r_1, \ldots, r_l \in \mathcal{L}_{typ}$, $k, l \geq 0$, and every I_i occurs in $t_1, \ldots, t_k$ and every O_i occurs in $r_1, \ldots, r_l$.

To describe the static and dynamic dependencies of fluents on other fluents, external functions, and actions, we introduce causation rules, while initial state constraints are used to describe the initial state of a planning problem.

Definition 2. *A **causation rule** (**rule**, for short) is an expression of the form*

$$\textit{caused } f \textit{ if } b_1, \ldots, b_k, \textit{not } b_{k+1}, \ldots, \textit{not } b_l \textit{ after } a_1, \ldots, a_m, \textit{not } a_{m+1}, \ldots, \textit{not } a_n$$

where $f \in \mathcal{L}_{fl} \cup \mathcal{L}_{vat} \cup \{false\}$, $b_1, \ldots, b_l \in \mathcal{L}_{fl,typ} \cup {\mathcal{L}_{ec}}^+$, $a_1, \ldots, a_n \in \mathcal{L}$, $l \geq k \geq 0$, and $n \geq m \geq 0$. Whenever $l = 0$ (resp. $n = 0$), the keyword if (resp. after) may be

[3] Note that predicate symbols with the same name, but different arities, are not allowed.

[4] As usual constants begin with a lower case letter, while variables start with an upper case letter.

*omitted. When both $l = n = 0$, the keyword caused may also be dropped. Rules where $n = 0$ are called **static rules**, while all other rules are called **dynamic rules**. A static rule preceded by the keyword initially, is called an **initial state constraints**.*

To access the different parts of a causation rule r (or an initial state constraint), we define $h(r) = f$, $post^+(r) = \{b_1, \ldots, b_k\}$, $post^-(r) = \{not\ b_{k+1}, \ldots, not\ b_l\}$, $pre^+(r) = \{a_1, \ldots, a_m\}$, $pre^-(r) = \{not\ a_{m+1}, \ldots, not\ a_n\}$, and $lit(r) = \{f, b_1, \ldots, b_l, a_1, \ldots, a_n\}$.

To allow conditional execution of actions, we define executability conditions.

Definition 3. *An **executability condition** is an expression of the form*

$$executable\ a\ if\ b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l$$

where $a \in \mathcal{L}_{act}{}^+ \cup \mathcal{L}_{vat}$, $b_1, \ldots, b_l \in \mathcal{L}$ and $l \geq k \geq 0$. Whenever $l = 0$, i.e. the execution is unconditional, the keyword if may be omitted.

To access the different parts of an executability condition e, we define $h(e) = a$, $post^+(e) = post^-(e) = \emptyset$, $pre^+(e) = \{b_1, \ldots, b_k\}$, $pre^-(e) = \{not\ b_{k+1}, \ldots, not\ b_l\}$, and $lit(e) = \{a, b_1, \ldots, b_l\}$.

Furthermore, we define, for any rule, initial state constraint, or executability condition r, that $post(r) = post^+(r) \cup post^-(r)$ and $pre(r) = pre^+(r) \cup pre^-(r)$.

From $\mathcal{K}$, we also adopt the *safety restriction* notion, i.e. all rules (including initial state constraints) and executability conditions have to satisfy the syntactic restriction that all variables in a naf type literal must also occur in some literal which is not a naf type literal.

Definition 4. *An **action description** is a pair (D, R) where D is a finite set of action, fluent and external function declarations and R is a finite set of safe executability conditions, safe causation rules, and safe initial state constraints.*

*A **planning domain** is a pair $PD = (\Pi, AD)$, where Π is a logic program over the literals of $\mathcal{L}_{typ}$ admitting exactly one answer set, and AD is an action description.*

*A **query** is of the form*

$$g_1, \ldots, g_m, not\ g_{m+1}, \ldots, not\ g_n ?(i)$$

where $g_1, \ldots, g_n \in \mathcal{L}_{fl}$ are variable-free, and $i, j \geq 0, n \geq m \geq 0$.

*A **planning problem** is a pair (PD, q), where PD is a planning domain and q is a query.*

In appendix B and C of [17], one can find the $\mathcal{K}^c$ encodings of the Bubble Breaker game and the defusing a bomb problem respectively. Especially note the difference in readability between the encoding, of the same problem, in $\mathcal{K}$ from Appendix A of [17] and the one in $\mathcal{K}^c$ from Appendix B of [17].

3 Semantics of $\mathcal{K}^c$

3.1 Instantation

Similar to the grounding of a logic program, we instantiate a planning problem such that the semantics can be defined more easily. The main difference with classical grounding is that we only allow correctly typed action, fluent, and external function call literals to be generated.

A substitution is any function $\theta : \sigma^{var} \mapsto \sigma^{con} \cup \sigma^{fl}$, i.e. a function assigning constants or fluent names to variables. The application of a substitution θ can be extended to any syntactic object x by defining $\theta(x)$ as the object x' obtained from x by replacing every $X \in \sigma^{var}$ that occurs in x, and that is defined by θ, with $\theta(x)$.

First, we define the valid instantiations of the fluents, actions, and external functions.

Definition 5. *Let $PD = (\Pi, (D, R))$ be a planning domain, and let M be the unique answer set of Π.*

*For a fluent (resp. action) declaration $d \in D$ and a substitution θ that is at least defined over $X_1, \ldots, X_m$ (and p if $p \in \sigma^{var}$), we say that $\theta(p(X_1, \ldots, X_m))$ is a **legal fluent** (resp. **action**) **instantiation** if $\{\theta(t_1), \ldots, \theta(t_n)\} \subseteq M$ and $\theta(p) \in \sigma^{fl}$ (resp. $\theta(p) \in \sigma^{act}$) if $p \in \sigma^{var}$.*

*For an external function declaration $d \in D$ and a substitution θ that is at least defined over $I_1, \ldots, I_m, O_1, \ldots, O_n$, $\theta(\&p[I_1, \ldots, I_m](O_1, \ldots, O_n))$ is a **legal external function call instantiation** if $\{\theta(I_1), \ldots, \theta(I_m), \theta(O_1), \ldots, \theta(O_n)\} \subseteq M$.*

To denote the set of all legal fluent (resp. action, external function call) instantiations of PD and their classical negations, we will use $\mathcal{L}^{fl}_{PD}$ (resp. $\mathcal{L}^{act}_{PD}, \mathcal{L}^{ec}_{PD}$). In addition, we use $\mathcal{L}_{PD} = \mathcal{L}^{fl}_{PD} \cup \mathcal{L}^{act}_{PD}{}^{+} \cup \mathcal{L}^{ec}_{PD}{}^{+}$.

Using the above, we can define the instantiation of a planning domain.

Definition 6. *Let $PD = (\Pi, (D, R))$ be a planning domain. The instantiation of PD, denoted $PD\downarrow$, is defined as $PD\downarrow = (\Pi\downarrow, (D, R\downarrow))$, where $\Pi\downarrow$ is the grounding of Π over σ^{con} and $R\downarrow = \{\theta(r) \mid r \in R, \theta \in \Theta_r\}$, where Θ_r is the set of all substitutions θ that define all the variables in r, such that*

- *$lit(\theta(r)) \cap \mathcal{L}_{dyn} \subseteq \mathcal{L}_{PD}$;*
- *$(post^{+}(\theta(r)) \cup pre^{+}(\theta(r))) \cap \mathcal{L}_{typ} \subseteq M$;*
- *$h(r) \in \mathcal{L}^{fl}_{PD} \cup \{false\}$ if r is a causation rule or an initial state constraint; and*
- *$h(r) \in \mathcal{L}^{act}_{PD}{}^{+}$ if r is an executability condition.*

Intuitively, the above ensures that in the instantiation of PD all actions, fluents and external function calls agree with their declarations, that positive type literals agree with the background knowledge, that causation rules or initial state constraints should cause a fluent literal and that executability conditions should have an action in their head. A planning domain PD is said to be *ground* if PD and $PD\downarrow$ coincide.

From now on, we will assume that we are working with the grounded version of a given planning domain PD, i.e. we will implicitly replace PD by $PD\downarrow$.

3.2 Conditional Planning

A plan in $\mathcal{K}$ is a sequence of sets of actions. However, this approach is not feasible in the context of conditional planning, where one wants to cope with non-deterministic effects of actions. For this reason, we will introduce the concept of a conditional plan, i.e. a plan that allows to branch depending on the effects that are caused by executing an action. Our notion of a conditional plan is inspired by the one from [16], and is limited to conditional plans with only the case-endcase construct[5].

In what follows, we use $\mathcal{P}(X)$ to denote the powerset of X.

Definition 7. *Let PD be a planning domain. A **conditional plan** for PD is defined inductively as follows:*

1. *A sequence of sets of actions $A_1; \ldots; A_k$, with $A_i \subseteq {\mathcal{L}_{PD}^{act}}^+$, is a conditional plan.*
2. *If we have a sequence of sets of actions $A_1; \ldots; A_k \in {\mathcal{L}_{PD}^{act}}^+$ and conditional plans $c_1, \ldots, c_l$, with $1 \leq l \leq |\mathcal{P}(\mathcal{L}_{PD}^{fl})|$, then*
$$A_1; \ldots; A_k; case \begin{cases} o_1 & \rightarrow c_1 \\ & \ldots \\ o_l & \rightarrow c_l \end{cases} \tag{2}$$
is a conditional plan, where $o_i \in \mathcal{P}(\mathcal{L}_{PD}^{fl})$ and $o_i \neq o_j$ whenever $i \neq j$, i.e. each element of $\mathcal{P}(\mathcal{L}_{PD}^{fl})$ is associated to at most one c_i.
3. *Nothing else is a conditional plan.*

Intuitively, a conditional plan of the form[6] (2) in the above definition has to be read as "execute the actions in A_1, then the ones in A_2, ..., then the ones in A_k; and depending on which set of fluent literals that are true after executing these actions, execute the corresponding plan c_i".

The plan we introduced in the introduction, i.e. (1) on page 215, is a conditional plan for our running defusing a bomb example. Although in general, a conditional planner should consider all possible sets of fluent literals in a case construct, in practice this is not always necessary as certain sets cannot occur, given the current state and the actions performed. E.g., the external function *look_effect* will never return both *locked* and *unlocked* at the same time.

Definition 8. *For a planning domain PD, a **state** is any consistent subset $s \subseteq \mathcal{L}_{PD}^{fl}$.*

For each external function $p \in \sigma^{ec}$, we will use, with $t_1, \ldots, t_m \in \sigma^{con} \cup \sigma^{fl}$,

$$out(\&p[t_1, \ldots, t_m]) = \{(o_1, \ldots, o_n) \mid \&p[t_1, \ldots, t_m](o_1, \ldots, o_n) \in {\mathcal{L}_{PD}^{ec}}^+\} \ ,$$

i.e. the set containing all possible output tuples for a given input tuple. As not all input tuples are valid for the legal instantations of p, we will use

$$vit(\&p) = \{(t_1, \ldots, t_m) \mid \&p[t_1, \ldots, t_m](o_1, \ldots, o_n) \in {\mathcal{L}_{PD}^{ec}}^+\} \ .$$

Further, we associate with p an $(m+1)$-ary function $f_{\&p}$ that associates with each tuple $(s, t_1, \ldots, t_m)$ an element of $\mathcal{P}(\mathcal{P}(out(\&p[t_1, \ldots, t_m])))$, where s is a state and

[5] Although e.g. [12,14] introduce constructs as if-then-else or while-do in conditional plans, the former can be easily transformed to case-endcase statements, while the same holds for the latter in case one is interested in plans of bounded length.

[6] For practical purposes, multiple o_i (having the same c_i) might be compactly represented by a Boolean combination F of fluent literals using the connectives $\wedge$, $\vee$ and *not* , which is evaluated on a state s in the obvious way. A state s would correspond to the conjunction $\bigwedge_{l \in s} l \wedge \bigwedge_{l \in \mathcal{L}_{PD}^{fl} \setminus s} not\ l$.

$(t_1, \ldots, t_m) \in vit(\&p)$. Intuitively, the function $f_{\&p}$ returns, for an external function $p \in \sigma^{ec}$ and an input tuple $(t_1, \ldots, t_m)$ w.r.t. a state s, the combinations of output tuples that are possible as a return value when the function is executed. Clearly, when $|f_{\&p}(s, t_1, \ldots, t_m)| = 1$, the external function is deterministic, otherwise it is non-deterministic.

To handle the external functions correctly in a state transition, we need to take care that each external function is evaluated exactly once for each possible input tuple. If not, we would have undesired results in case of non-deterministic functions, e.g. having two different rules with the same external function evaluating to different sets of output tuples. For this reason, we define an *external function evaluation* w.r.t. a state s as a function g_s, such that for each $p \in \sigma^{ec}$ and for each $(t_1, \ldots, t_m) \in vit(\&p)$, $g_s(\&p[t_1, \ldots, t_m]) = o$, where $o \in f_{\&p}(s, t_1, \ldots, t_m)$. Thus, each external function evaluates in g_s to exactly one set of output tuples for each possible input w.r.t. a state s.

Definition 9. *Let PD be a planning domain. A* ***state transition*** *is a tuple*

$$t = \langle s, g_s, A, s', g_{s'} \rangle \ ,$$

where s, s' are states, $g_s, g_{s'}$ are external function evaluations w.r.t. s (resp. s') and A is a set of action atoms.

Similar to the answer set semantics[7], we define our semantics first for positive planning domains, i.e. planning domains that are free from negation as failure. Afterwards, we will define a reduction from a general planning domain to a positive one. In what follows, we consider a ground planning domain $PD = (\Pi, (D, R))$, where M the unique answer set of Π.

For a set of ground literals $X \subseteq \mathcal{L}_{fl,typ} \cup \mathcal{L}_{act}{}^{+}$, a ground literal $l \in \mathcal{L}_{fl,typ} \cup \mathcal{L}_{act}{}^{+}$ and an external function evaluation g_s, with $s = X \cap \mathcal{L}^{fl}_{PD}$, we use

- $X \models_{g_s} l$, when $l \in X$;
- $X \models_{g_s} not\ l$, when $l \notin X$;
- $X \models_{g_s} \&p[t_1, \ldots, t_m](o_1, \ldots, o_m)$, when $(o_1, \ldots, o_m) \in g_s(\&p[t_1, \ldots, t_n])$; and
- $X \models_{g_s} not\ \&p[t_1, \ldots, t_m](o_1, \ldots, o_m)$, when $(o_1, \ldots, o_m) \notin g_s(\&p[t_1, \ldots, t_n])$.

Finally, for a set of ground literals $Y \subseteq \mathcal{L}$, we use $X \models_{g_s} Y$ iff $X \models_{g_s} y$ for each $y \in Y$. As usual, we have $X \not\models_{g_s} Y$ if we have not $X \models_{g_s} Y$.

Definition 10. *For a positive PD, a state s_0 and an external function evaluation g_{s_0}, we call s_0 a* ***legal initial state*** *if s_0 is the smallest (w.r.t. subset inclusion) set such that $h(r) \in s_0$ whenever $s_0 \cup M \models_{g_{s_0}} post(r)$ for all initial state constraints and static rules $r \in R$.*

For a positive PD, a state s and an external function evaluation g_s, a set $A \subseteq \mathcal{L}^{act}_{PD}{}^{+}$ is called an *executable action set* w.r.t. s and g_s, if for each $a \in A$ there exists an executability condition $e \in R$ such that $h(e) = a$ and $s \cup M \cup A \models_{g_s} pre(e)$.[7]

[7] Note that we allow dependent actions, i.e. actions that depend on the execution of other actions.

Definition 11. *Let PD be a positive planning domain, let $t = \langle s, g_s, A, s', g_{s'} \rangle$ be a state transition and let $r \in R$ be a causation rule. We say that r is **satisfied** by s' w.r.t. t iff either $h(r) \subseteq s' \setminus \{false\}$ or we do not have both $s' \cup M \models_{g_{s'}} post(r)$ and $s \cup M \cup A \models_{g_s} pre(r)$.*

*A state transition $t = \langle s, g_s, A, s', g_{s'} \rangle$ is called a **legal state transition** if A is an executable action set w.r.t. s and g_s, and s' is a minimal (w.r.t. subset inclusion) consistent set that satisfies all causation rules in R, except initial state constraints, w.r.t. t.*

Next, we generalize the above to arbitrary ground planning domains PD, i.e. planning domains containing negation as failure in the rules. This is done by defining a reduction to positive planning domains, similar to the GL-reduct for the answer set semantics [7].

Definition 12. *Let PD be an arbitrary planning domain and consider a state transition $t = \langle s, g_s, A, s', g_{s'} \rangle$. The **reduction** of PD w.r.t. t, denoted PD^t, is defined by $PD^t = (\Pi, (D, R^t))$, where R^t is obtained from R by removing:*

- *every $r \in R$ for which either $s' \cup M \not\models_{g_{s'}} post^-(r)$ or $s \cup A \cup M \not\models_{g_s} pre^-(r)$;*
- *all literals not l, with $l \in \mathcal{L}$, from the remaining rules.*

Clearly, PD^t is a positive planning domain. Now we can define the concepts of legal initial states, executable action sets and legal state transitions in the case of arbitrary planning domains.

Definition 13. *Let PD be a planning domain. A state s_0 is a **legal initial state** if s_0 is a legal initial state for PD^t, where $t = \langle \emptyset, g_\emptyset, \emptyset, s_0, g_{s_0} \rangle$. A set A is an **executable action set** w.r.t. a state s, if A is an executable action set w.r.t. s in PD^t, where $t = \langle s, g_s, A, \emptyset, g_\emptyset \rangle$. A state transition $t = \langle s, g_s, A, s', g_{s'} \rangle$ is a **legal state transition** if it is a legal state transition for PD^t.*

Before we can define optimistic conditional plans, we need some additional notions. For a planning domain PD and two states s_0 and s_n, with $n \geq 0$, a sequence of state transitions

$$T = \langle \langle s_0, g_{s_0}, A_1, s_1, g_{s_1} \rangle, \langle s_1, g'_{s_1}, A_2, s_2, g_{s_2} \rangle, \ldots, \langle s_{n-1}, g'_{s_{n-1}}, A_n, s_n, g_{s_n} \rangle \rangle$$

is called a *trajectory* from s_0 to s_n for PD if all state transitions in T are legal. Further, we use $\mathcal{T}(s_0, s_n)$ to denote the set of all trajectories from s_0 to s_n; and for a sequence of sets of actions $A_1; \ldots; A_k$ and a set of states S, we use

$$\begin{aligned} PD(A_1; \ldots; A_k, S) = \{s_k \mid & \langle \langle s_0, g_{s_0}, A_1, s_1, g_{s_1} \rangle, \langle s_1, g'_{s_1}, A_2, s_2, g_{s_2} \rangle, \ldots, \\ & \langle s_{k-1}, g'_{s_{k-1}}, A_k, s_k, g_{s_k} \rangle \rangle \in \mathcal{T}(s_0, s_k) \wedge s_0 \in S\} \ . \end{aligned}$$

Now, we have all necessary means to define optimistic conditional plans.

Definition 14. *Let $PP = (PD, q)$ be a planning problem, let C be a conditional plan and let S be a set of states. We define C being **optimistic** w.r.t. S inductively as*

1. *if $C = A_1; \ldots; A_k$ and $\exists s \in S \cdot \exists x \in PD(C, \{s\}) \cdot \{g_{m+1}, \ldots, g_n\} \cap x = \emptyset \wedge \{g_1, \ldots, g_m\} \subseteq x$, then C is optimistic w.r.t. S.*

2. *if* $C = A_1; \ldots ; A_k;$ *case* $\begin{cases} o_1 & \rightarrow & c_1 \\ & \ldots & \\ o_l & \rightarrow & c_l \end{cases}$ *; and* $\{o_1, \ldots o_l\} \cap PD(A_1; \ldots ; A_k, S) \neq \emptyset$*; and* c_i *is optimistic w.r.t.* o_i *for each* $i \in [1 \ldots l]$*, then* C *is optimistic w.r.t.* S*.*

Now, C *is an* ***optimistic plan*** *for* PP *if it is optimistic w.r.t. the set of all legal initial states.*

Intuitively, condition (1) in the above definition demands that a goal state can be reached for each starting state in S, while condition (2) demands that each of the conditional states o_i can be reached, starting from the states in S, and that for each of these conditional states o_i a goal state can be reached by executing c_i.

Note that an optimistic conditional plan corresponds to an optimistic plan in $\mathcal{K}$ when the conditional plan does not contain case constructs. This implies that executing an optimistic conditional plan can yield situations where the goal is not reached. Similar to $\mathcal{K}$, we can define when a conditional plan is secure, i.e. when executing the conditional plan will always result in a goal state.

Definition 15. *Let* $PP = (PD, q)$ *be a planning problem, let* C *be a conditional plan and let* S *be a set of states. The* ***secureness*** *of* C *w.r.t.* S *is inductively defined as*

1. *if* $C = A_1; \ldots ; A_k$ *and* $\forall s \in S \cdot \forall i \in [1 \ldots k] \cdot \forall s' \in PD(A_1; \ldots ; A_{i-1}, \{s\}) \cdot PD(A_i, \{s'\}) \neq \emptyset$ *and* $\forall s \in S \cdot \forall x \in PD(C, \{s\}) \cdot \{g_{m+1}, \ldots , g_n\} \cap x = \emptyset \wedge \{g_1, \ldots , g_m\} \subseteq x$*, then* C *is secure w.r.t.* S*.*
2. *if* $C = A_1; \ldots ; A_k;$ *case* $\begin{cases} o_1 & \rightarrow & c_1 \\ & \ldots & \\ o_l & \rightarrow & c_l \end{cases}$ *; and* $PD(A_1; \ldots ; A_k, S) \subseteq \{o_1, \ldots , o_n\}$ *and* $\forall s \in S \cdot \forall i \in [1 \ldots k] \cdot \forall s' \in PD(A_1; \ldots ; A_{i-1}, \{s\}) \cdot PD(A_i, \{s'\}) \neq \emptyset$ *and* c_i *is secure w.r.t.* o_i *for each* $i \in [1 \ldots l]$*, then* C *is secure w.r.t.* S*.*

Now, C *is a* ***secure plan*** *for* PP *if it is secure w.r.t. the set of all legal initial states.*

Intuitively, the condition $\forall s \in S \cdot \forall i \in [1 \ldots k] \cdot \forall s' \in PD(A_1; \ldots ; A_{i-1}, \{s\}) \cdot PD(A_i, \{s'\}) \neq \emptyset$ in the above definition ensures that a secure plan never gets "stuck" in a state during execution.

The conditional plan (1) on page 215 is a secure plan for our defusing a bomb example. However, if we change the behavior of the external function *look_effect* such that it either returns *locked*, *unlocked* or neither *locked* nor *unlocked*, than one can check that the conditional plan is not any longer secure. Furthermore, it turns out that no secure plan exists for this modification.

On the other hand, consider the following extension of the defusing a bomb example. We can look at the bomb and if the light is on, we know that the bomb is either *locked* or *unlocked*, but if we do not have light (or we don't know if there is light or not) looking at the bomb can either yield *locked*, *unlocked* or neither *locked* nor *unlocked*. Further, we have an action *check_light* with a corresponding external function *check_light_effect* which materializes the effect of the "sensing" action *check_light*. Finally, using the action *switch* we can switch the state from the light. Now[8] one can see that the following

[8] The encoding in $\mathcal{K}^c$ of this example can be found in Appendix D of [17].

conditional plan

$$check\_light;\, case \begin{cases} \{light\} \rightarrow look;\, case \begin{cases} \{locked\} \rightarrow disarm \\ \{unlocked\} \rightarrow turn;\, disarm \end{cases} \\ \{no\_light\} \rightarrow switch;\, look;\, case \begin{cases} \{locked\} \rightarrow disarm \\ \{unlocked\} \rightarrow turn;\, disarm \end{cases} \end{cases}$$

is secure, while the conditional plan on page 215 is only an optimistic one.

4 Computing Conditional Plans Using $DLV^{\mathcal{K}}$

To demonstrate our conditional planning framework, we developed a prototype implementation of a part of the semantics of $\mathcal{K}^c$ on top of DLV$^{\mathcal{K}}$. As DLV$^{\mathcal{K}}$ does not support external evaluations, our prototype currently disregards[9] this feature of $\mathcal{K}^c$. Further, we only implemented optimistic plan generation so far, but in the next step a secure checker will be added, using the built-in security checker of DLV$^{\mathcal{K}}$. When provided with a classical $\mathcal{K}$ planning problem description, DLV$^{\mathcal{K}}_c$ will generate a conditional plan in a graphical representation. E.g., feeding the defusing a bomb planning description from the introduction (page 2) to the system, will yield the conditional plan depicted in Figure 2. To generate this plan, we use DLV$^{\mathcal{K}}$'s batch mode for generating optimistic plans. Each optimistic plan received from DLV$^{\mathcal{K}}$ is first compressed by removing useless planning steps, and afterwards this compressed optimistic plan is put into the graph representing the optimistic conditional plan, i.e. each optimistic plan from DLV$^{\mathcal{K}}$ can be seen as a valid trajectory in an optimistic conditional plan.

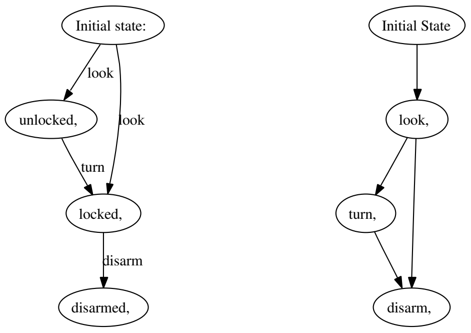


Fig. 2. The generated conditional plan for the defusing a bomb example from page 2. On the left, we have a conditional plan that also contains the states reached, while the plan on the right only contains the different actions that need to be taken to reach the goal. To increase readability, our implementation compacts the tree shape of a conditional plan into a dag whenever possible.

The prototype implementation is built using the Python programming language and can be run on any modern platform that DLV$^{\mathcal{K}}$ supports. The implementation, together

[9] One can of course, naïvely, introduce the behavior of external evaluations by adding mutually exclusive rules that introduce the possible outcomes of the external evaluations hard-coded.

with additional information and examples, can be found at `http://tinfpc2.vub.ac.be/cdlvk`.

5 Related Work and Conclusion

In this paper we presented $\mathcal{K}^c$, a conditional planning language that can use external functions to outsource the computation of certain effects when an action is executed. As $\mathcal{K}^c$ is a proper extension of the planning language $\mathcal{K}$, it relates to most other planning language in the same way, and we therefore refer to [5]. One exception here, are extensions of those languages that incorporate sensing actions to obtain non-deterministic, i.e. conditional, planning. For these extensions, e.g. [15,16], we clearly showed that external function calls are well-suited to simulate sensing actions by combining an ordinary action with an external function that materializes the effects of the "sensing" action.

In future work, we plan to extend our prototype so the generated conditional plans can be checked for secureness. We also want to incorporate external functions natively, such that the explicit introduction of such functions by using mutually exclusive rules can be dropped, improving the readability and robustness of the planning descriptions. Finally, we are currently employing our framework in the context of repairing web service workflows by planning [1], one of the topics of the ongoing WS-Diamond research project [2]. The high expressiveness and declarativity of $\mathcal{K}^c$, together with its conditional planning capabilities, turns out to be beneficial in that area of application.

References

1. Private Communications with Gerhard Friedrich, University of Klagenfurt, Austria.
2. Ws-diamond: Web-service diagnosability, monitoring & diagnosis (ist-516933). Project website at http://wsdiamond.di.unito.it.
3. Special issue on reasoning about action and change. *Journal of Logic Prog.*, 31(1-3), 1997.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In *Computational Logic*, volume 1861 of *LNCS*, pages 807–821. Springer, 2000.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *Transactions on Computational Logic*, 5(2):206–263, 2004.
6. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. 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)*, pages 90–96, 2005.
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
8. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3&4):301–321, 1993.
9. E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95(2):409–438, 1997.
10. K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In *Proc. of the 5th Intl. Conf. on Principles of KR and Reasoning*, pages 174–185, 1996.

11. L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing, concurrency, and exogenous events: logical framework and implementation. In *Proc. of the 7th Intl. Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 678–689, 2000.
12. H. J. Levesque. What is planning in the presence of sensing? In *AAAI/IAAI, Vol. 2*, pages 1139–1146, 1996.
13. V. Lifschitz. *The Logic Programming Paradigm - A 25-Year Perspective*. Springer, 1999.
14. J. Lobo, S. Taylor, and G. Mendez. Adding knowledge to the action description language A. In *Proc. of the 14th National Conf. on AI (AAAI97)*, pages 454–459. AAAI Press, 1997.
15. T. C. Son and C. Baral. Formalizing sensing actions a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, 2001.
16. T. C. Son, P. H. Tu, and C. Baral. Planning with sensing actions and incomplete information using logic programming. In *Proc. of the 7th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, volume 2923 of *LNAI*, pages 261–274, 2004.
17. D. Van Nieuwenborgh, T. Eiter, and D. Vermeir. Conditional planning with external functions. Technical report, 2007, http://tinf2.vub.ac.be/˜dvnieuwe/lpnmr2007technical.ps.

Logic Programs with Abstract Constraints: Representaton, Disjunction and Complexities

Jia-Huai You[1], Li Yan Yuan[1], Guohua Liu[1], and Yi-Dong Shen[2]

[1] Department of Computing Science
University of Alberta, Edmonton, Alberta, Canada
[2] Lab of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China

Abstract. We study logic programs with arbitrary abstract constraint atoms, called *c-atoms*. As a theoretical means to analyze program properties, we investigate the possibility of unfolding these programs to logic programs composed of ordinary atoms. This approach reveals some structural properties of a program with c-atoms, and enables characterization of these properties based on the known properties of the transformed program. Furthermore, this approach leads to a straightforward definition of answer sets for disjunctive programs with c-atoms, where a c-atom may appear in the head of a rule as well as in the body. We also study the complexities for various classes of logic programs with c-atoms.

1 Introduction

Logic programs with abstract constraints were introduced as a general framework for representing, and reasoning with, sets of atoms [12,13]. This is in contrast with traditional logic programs, which are used primarily to reason with individuals.

An abstract constraint atom, which is also called a *c-atom* following [21], is of the form (D, C), where D is a domain and C is a collection of subsets from the power set of D, which are intended to represent admissible solutions. By allowing c-atoms to appear anywhere in a rule, the framework of logic programs with c-atoms has become a highly expressive knowledge representation language. For example, many problems can be conveniently represented with constraints over sets of atoms, such as *weight* and *cardinality constraints* and *aggregates* (see, e.g. [2,3,4,5,7,15,16,17,18,19]).

In the study of logic programs with abstract constraints, a crucial assumption was made: an abstract constraint be monotone [12,13]. A constraint is *monotone* if it holds that whenever it is satisfied by a set of atoms I, it must be satisfied by any extension of I. It is observed that under this assumption, much of the basic concepts and techniques for characterizing the semantics of normal logic programs can be generalized to the new context. In addition, some important constraints, such as pseudo-boolean constraints and cardinality constraints, can be represented by abstract monotone atoms [11,12].

The assumption on monotone atoms, however, also limits the scope of applications that the framework supports. For example, many constraints involving aggregate functions are not monotone. More generally, when a c-atom is allowed to appear in the head of a rule, it is fully capable of expressing a constraint in the sense of *Constraint Satisfaction Problem* (CSP). Hence, a CSP can be represented by a collection of condition-free rules. In this way, the framework of logic programs with c-atoms can express complex constraint satisfaction problems, such as those involving conditional constraints [14][1], which are useful in modeling configuration and design problems.

Recently, Son et al. propose to define answer sets for programs with arbitrary c-atoms with a notion of conditional satisfaction [21]. They show that answer sets defined this way generalize the notion of *well-supported* models for normal logic programs [6], and the resulting semantics coincides with the previously introduced semantics for logic programs with monotone c-atoms. In addition, they point out the different behaviors between answer sets defined *by reduct* and those defined *by complement*.

In this paper, we extend the work of [21] in three directions. First, we study the possibility of representing logic programs with c-atoms by logic programs composed of ordinary atoms. This can be seen as an extension of the unfolding approach presented in [20], but we also consider unfolding c-atoms in the head of a rule as well as negative c-atoms in the body. In addition, we use the term *unfolding* in a more general sense in which new atoms are introduced in the transformed program in order to guarantee polynomial time transformation. The unfolding approach leads to a characterization of the existence of answer sets based on an extended *call-consistent* condition [6]. Second, the unfolding approach leads to an answer set semantics for disjunctive programs with c-atoms. Disjunctive programs with aggregates have previously been studied in [5,17], where aggregates do not appear in the heads of program rules. In our case, an arbitrary c-atom can appear in a disjunctive head as well as positively or negatively in a rule body. Since the unfolding approach reveals structural properties, it is possible to formulate the notion of *head cycle* for the new context. We know that if a disjunctive program is head cycle-free, it can be reduced to a normal program by *shifting* [1]. A similar result holds for disjunctive programs with c-atoms. Finally, we provide complexity results for classes of programs with c-atoms.

The next section provides background and defines the notations, followed by an example to show a usage of arbitrary c-atoms to represent CSPs in Section 3. Section 4 shows unfolding of (non-disjunctive) logic programs with c-atoms to normal programs while preserving the underlying answer set semantics. By further unfolding disjunctions in the rule heads, in Section 5 we define a semantics for disjunctive programs with c-atoms. Section 6 gives some complexity results. Section 7 is about related work, with Section 8 containing final remarks.

2 Background and Notation

We follow the notation used in [10,21]. The key definitions are taken from [21].

We assume a propositional language $\mathcal{L}$ with a countable set $\mathcal{A}$ of *propositional atoms*. A propositional atom is also called an *ordinary atom* or just an *atom*. An *abstract constraint atom*, or *c-atom* for abbreviation, is a pair (D, C) such that $D \subseteq \mathcal{A}$ and $C \subseteq 2^D$. Given a c-atom $A = (D, C)$, we use A_d and A_c to denote D and C, respectively.

A c-atom is called *elementary* if it is of the form $(\{a\}, \{\{a\}\})$, where a is an atom. In expressing such a c-atom, we may simply write the atom instead.

[1] Which were called *dynamic CSPs*.

A disjunctive program (with c-atoms) is a collection of rules of the form

$$A_1 \vee ... \vee A_k \leftarrow B_1, ..., B_m, \texttt{not}\ C_1, ..., \texttt{not}\ C_n. \tag{1}$$

where $k \geq 1$, $m, n \geq 0$, and A_i's, B_i's, and C_i's are all c-atoms. $\texttt{not}\ C_i$ is called a *negation-as-failure c-atom*, or simply a *negative c-atom*. A *literal* refers to a c-atom or a negative c-atom. For a rule r of the form (1), the left hand side of $\leftarrow$ is called the *head* and the right hand side the *body*. We use $head(r)$, $pos(r)$ and $neg(r)$ to denote $\{A_1, ..., A_k\}$, $\{B_1, ..., B_n\}$, and $\{\texttt{not}\ C_1, ..., \texttt{not}\ C_n\}$, respectively, and let $body(r) = pos(r) \cup neg(r)$. By abuse of notation, we may denote a rule r by $head(r) \leftarrow pos(r), neg(r)$. Note that by doing so we have assumed that when we write a set of literals in the body we mean a conjunction, and in the head we mean a disjunction.

C-atoms of the form $(D, \emptyset)$ are always *false* (to be defined shortly). When a c-atom of this type appears in the head of a rule, we will write $\perp$ instead, and call it a *constraint*.

A program P is called a *general program* (or a *non-disjunctive program*), if for every rule $r \in P$, $head(r)$ is singleton. A general program P is called *basic* if for any $r \in P$, either r is a constraint or the head of r is elementary. A *condition-free program* is a general program where all the rules have an empty body. A *positive program* is a general program without negative c-atoms. A *positive disjunctive program* is a disjunctive program without negative c-atoms.

Given a program P, $At(P)$ denotes the set of (ordinary) atoms appearing in P.

Sometimes we say *a model M restricted to the atoms appearing in a program P*. By this we mean $M \cap At(P)$, and denote it by $M|_{At(P)}$.

Let $I \subseteq \mathcal{A}$ and A be a c-atom. We say I satisfies A, denoted $I \models A$, if $I \cap A_d \in A_c$; we say I satisfies $\texttt{not}\ A$, denoted $I \models \texttt{not}\ A$, if $I \cap A_d \notin A_c$. I satisfies the body of a rule r if $I \models l$ for each $l \in body(r)$; I satisfies the disjunctive head of a rule r if $I \models A$ for some atom $A \in head(r)$. A set of atoms S is a *model* of a program P if for each rule $r \in P$, $S \models head(r)$ whenever $S \models body(r)$.

Answer sets for basic positive programs are defined with a notion of conditional satisfaction.

Definition 1. *Let R and S be two sets of atoms. The set R* conditionally satisfies *a c-atom A w.r.t. S, denoted by $R \models_S A$, if $R \models A$ and, for every I such that $R \cap A_d \subseteq I$ and $I \subseteq S \cap A_d$, we have that $I \in A_c$.*

Conditional satisfaction is used to define a generalized version of one-step provability operator. Given a basic positive program P, and sets of atoms R and S, define

$$T_P(R, S) = \{a \mid \exists r \in P,\ R \models_S body(r)\ \&\ head(r) = (\{a\}, \{\{a\}\})\}$$

Definition 2. *Let M be a model of a basic positive program P. M is an* answer set *for P iff $M = T_P^{\infty}(\emptyset, M)$, where $T_P^0(\emptyset, M) = \emptyset$ and $T_P^{i+1}(\emptyset, M) = T_P(T_P^i(\emptyset, M), M)$, for all $i \geq 0$.*

Answer sets for general programs are defined in two steps. In the first step, answer sets for basic programs are defined, and in the second, a general program is represented by a collection of basic programs, and the answer sets for the former are defined in terms of the answer sets for the latter.

There have been two different interpretations of negation-as-failure. The first treats a negative c-atom $\texttt{not}\ A$ by its complement, and replaces it with a positive c-atom A' where $A'_d = A_d$ and $A'_c = 2^{A_d} \backslash A_c$. Given a program P, the resulting program is called the *complement of* P. The second adopts the well-known technique of reduct, and defines the reduct of a program P w.r.t. a set of atoms M as

$$P^M = \{head(r) \leftarrow pos(r) \mid r \in P,\ \forall \texttt{not}\ C \in neg(r), M \not\models C\}$$

Definition 3. *Let P be a basic program and M a set of atoms. (i) M is an* answer set by complement *for P iff M is an answer set for its complement. (ii) M is an* answer set by reduct *for P iff M is an answer set for P^M.*

Next, a general program is represented by its instances in the form of a basic program, and the answer sets of the former are defined in terms of the answer sets of the latter.

Let P be a general program and $r \in P$. For each $\pi \in head(r)_c$ (when $|head(r)| = 1$, we write $head(r)$ to denote the only c-atom in it), the *instance* of r w.r.t. π is the set of rules consisting of

1. $b \leftarrow body(r)$, for each $b \in \pi$, and
2. $\bot \leftarrow d, body(r)$, for each $d \in head(r)_d \backslash \pi$.

An *instance of* P is a program obtained by replacing each rule of P with one of its instances. Note that an instance of P is a basic program.

Definition 4. *Let P be a general program and M a set of atoms. M is an answer set by reduct (by complement, resp.) for P iff M is an answer set by reduct (by complement, resp.) for one of its instances.*

For convenience, from now on, the term *answer set* may refer to either answer set by reduct or answer set by complement. Distinction will be made when necessary.

3 Representing Constraint Satisfaction Problem

C-atoms can be used to represent CSPs in a straightforward way. A CSP, denoted $A(\mathcal{X}, \mathcal{D}, \mathcal{C})$, consists of a finite set of variables $\mathcal{X}$, a collection of finite domains $\mathcal{D}$, where variable x's domain is denoted $D(x)$, and a collection of constraints $\mathcal{C}$. A constraint $R_{x_1 \dots x_j} \in \mathcal{C}$ is a subset of the Cartesian product $D(x_1) \times \dots \times D(x_j)$; the set of variables $\{x_1, ..., x_j\}$ is called the *scope* of the constraint. A *solution* to a CSP is an assignment where each variable is assigned a value from its domain such that all of the constraints are satisfied. A constraint $R_{x_1 \dots x_j}$ is satisfied if and only if the assignment to variables in its scope yields a tuple in the relation $R_{x_1 \dots x_j}$.

In the following, we will use an atom $x(v)$ to represent that the CSP variable x takes the value $v \in D(x)$. Given a CSP $A(\mathcal{X}, \mathcal{D}, \mathcal{C})$, we define the corresponding program P_A as the one that consists of the following condition-free rules:

– for each constraint $R_{x_1 \dots x_j} \in \mathcal{C}$, there is a condition-free rule $(D, C) \leftarrow$ in P_A, where

$$D = \{x(v) \mid x \in \{x_1, ..., x_j\},\ v \in D(x)\}$$
$$C = \{\{x_1(v_1), ..., x_j(v_j)\} \mid (v_1, ..., v_j) \in R_{x_1 \dots x_j}\}$$

- if a variable $x \in \mathcal{X}$ doesn't appear in any constraint, we have a condition-free rule $(D, C) \leftarrow$ in P_A, where
$$D = \{x(v) \mid v \in D(x)\} \quad \text{and} \quad C = \{\{x(v)\} \mid v \in D(x)\}.$$

Let $A(\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a CSP. We can show that an assignment of the variables in $\mathcal{X}$, denoted $\{x_1 \leftarrow v_1, ..., x_n \leftarrow v_n\}$, is a solution to the CSP if and only if $\{x_1(v_1), ..., x_n(v_n)\}$ is an answer set for P_A.

Thus, the class of general programs can be used to represent conditional CSPs, where the satisfaction of a constraint may be conditioned upon the satisfaction of some other constraints. The expressiveness can be further enhanced by allowing a disjunction of constraints in the head of a rule.

4 Unfolding General Programs

We will describe the process of unfolding in three steps: unfolding the heads of rules, unfolding positive c-atoms in rule bodies, and unfolding negative c-atoms. It turns out that these transformations are independent of each other. As such, the order of their applications is unimportant.

4.1 Unfolding the Head

Let P be a general program. Unfolding the rule heads in P will yield a basic program, which is denoted $U_{hd}(P)$.

Let A be a c-atom in the head of a rule. In our unfolding, each subset $\pi \in A_c$ will be named explicitly by a new atom, say ξ_π. For each rule $r \in P$, of the form $A \leftarrow \mathit{body}(r)$, where $A_c = \{\pi_1, ..., \pi_k\}$, the resulting rules in $U_{hd}(P)$ are:

1. $1\{\xi_{\pi_1}, ..., \xi_{\pi_k}\}1 \leftarrow \mathit{body}(r)$.
2. $h \leftarrow \xi_\pi$. for each $\pi \in A_c$, and for each $h \in \pi$, and
3. $\perp \leftarrow \xi_\pi, d, \mathit{body}(r)$. for each $\pi \in A_c$, and for each $d \in A_d \setminus \pi$.

Note that the choice rule used above is a convenient representation of a collection of normal rules; that is,

$$\xi_{\pi_i} \leftarrow \mathit{body}(r), \texttt{not}\ \xi_{\pi_1}, ..., \texttt{not}\ \xi_{\pi_{i-1}}, \texttt{not}\ \xi_{\pi_{i+1}}, ..., \texttt{not}\ \xi_{\pi_k}. \quad \text{for each } 1 \le i \le k$$

Example 1. Suppose we have a rule: $(\{a, b, c\}, \{\emptyset, \{a\}, \{a, b\}\}) \leftarrow$. The unfolded program consists of the rules:

$$\begin{array}{lll} 1\{\xi_\emptyset, \xi_{\{a\}}, \xi_{\{a,b\}}\}1 \leftarrow . & & \\ \perp \leftarrow \xi_\emptyset, a. & \perp \leftarrow \xi_\emptyset, b. & \perp \leftarrow \xi_\emptyset, c. \\ a \leftarrow \xi_{\{a\}}. & \perp \leftarrow \xi_{\{a\}}, b. & \perp \leftarrow \xi_{\{a\}}, c. \\ a \leftarrow \xi_{\{a,b\}}. & b \leftarrow \xi_{\{a,b\}}. & \perp \leftarrow \xi_{\{a,b\}}, c. \end{array}$$

Unfolding the head resembles the definition of answer sets for a general program in terms of its instances in the form of basic program. However, the use of new symbols is essential in representing all the instances by a program in a compact way.

In the following, and in the rest of this paper, given a program P, we will denote by $AS(P)$ the set of all answer sets of P, and by $AS_{|N}(P)$ the set of all answer sets of P restricted to the atoms in N.

Proposition 1. *For any general program P, the transformation from P to $U_{hd}(P)$ takes polynomial time (in the size of P), and satisfies $AS(P) = AS_{|At(P)}(U_{hd}(P))$.*

A condition-free program may not have an answer set. An advantage of unfolding is to understand this behavior in terms of its unfolded program.

Example 2. The following condition-free program has no answer set.

$$(\{a\}, \{\emptyset\}) \leftarrow . \qquad (\{a\}, \{\{a\}\}) \leftarrow .$$

The unfolded program is

$$\xi_{\emptyset} \leftarrow . \quad \bot \leftarrow \xi_{\emptyset}, a. \quad \xi_{\{a\}} \leftarrow . \quad a \leftarrow \xi_{\{a\}}.$$

where the conflict in the original program can be demonstrated by a derivation of $\bot$.

4.2 Unfolding Positive C-Atoms in Rule Body

Unfolding positive c-atoms in rule bodies can be applied to any program. Given a program P, the unfolded program will be denoted $U_{pos}(P)$.

For each rule $r \in P$ of the form

$$head(r) \leftarrow (D_1, C_1), ..., (D_n, C_n), neg(r). \tag{2}$$

where $n \geq 0$, we will have a rule

$$head(r) \leftarrow \phi_1, ..., \phi_n, neg(r).$$

in $U_{pos}(P)$, where ϕ_i's are new symbols, plus the following rules: for each $1 \leq j \leq n$,

$$\phi_j \leftarrow \pi, \texttt{not}\ d_1, ..., \texttt{not}\ d_k. \quad \text{for each } \pi \in C_j \text{ where } \{d_1, ..., d_k\} = D_j \setminus \pi.$$

Example 3. Consider the following program from [21]:

$$\begin{array}{ll} p(1). & p(-1) \leftarrow p(2). \\ p(2) \leftarrow \text{SUM}(\{X \mid p(X)\}) \geq 1. & \end{array}$$

SUM$(\{X \mid p(X)\}) \geq 1$ above denotes a constraint, say A, where $A_d = \{p(1), p(2), p(-1)\}$ and $A_c = \{\{p(1)\}, \{p(2)\}, \{p(1), p(2)\}, \{p(2), p(-1)\}, \{p(1), p(2), p(-1)\}\}$. As shown in [21], P has no answer set.

An important intuition in answer set programming is that a positive conclusion must be supported by a non-circular argument. E.g., that $\{p(1), p(2), p(-1)\}$ above is not an answer set is precisely due to this intuition. The conclusions of $p(2)$ and $p(-1)$ are drawn from a circular argument. This will be made apparent by unfolding, where five rules are generated out of the last rule in P, one for each $\pi \in C$. The unfolded program $U_{pos}(P)$ therefore consists of the following rules

$$\begin{array}{l} p(1). \quad p(-1) \leftarrow p(2). \quad p(2) \leftarrow \phi. \\ \phi \leftarrow p(1), \texttt{not}\, p(2), \texttt{not}\, p(-1). \\ \phi \leftarrow p(2), \texttt{not}\, p(1), \texttt{not}\, p(-1). \\ \phi \leftarrow p(1), p(2), \texttt{not}\, p(-1). \\ \phi \leftarrow p(-1), p(2), \texttt{not}\, p(1). \\ \phi \leftarrow p(1), p(2), p(-1). \end{array}$$

Suppose M is an answer set for P. Then, $p(2)$ is either in M or not in M. Assume $p(2) \in M$. Then $\phi \in M$ since ϕ is the only support for $p(2)$. For each rule with ϕ as the head, only the first doesn't have $p(2)$ in the body. But this rule relies on having $\texttt{not}\, p(2)$. Thus, $p(2)$ is not well-supported, and therefore $p(2) \notin M$. If $p(2)$ is not in M, then we have $p(-1) \notin M$, since $p(2)$ is the only support for $p(-1)$. However, the fourth rule now has a satisfied body, which derives ϕ and then $p(2)$. Contradiction.

Proposition 2. *For any general program P, the transformation from P to $U_{pos}(P)$ takes polynomial time, and satisfies $AS(P) = AS_{|At(P)}(U_{pos}(P))$.*

Remarks: Due to the introduction of new symbols, our unfolding in general results in much smaller programs than that of [20].

The paper [20] deals with logic programs with aggregates. Assume an aggregate α is represented by a c-atom (D, C), where D is the domain of α and $C \subseteq 2^D$ consists of all π such that π satisfies α. For the purpose of this discussion, assume program P consists of basic positive rules of the form

$$A \leftarrow (D_1, C_1), ..., (D_n, C_n). \tag{3}$$

The size of such a rule is bounded by expression $1 + (d + mk) * n$, where, for all i, d is the maximum size of D_i, m is the maximum size of any π, $\pi \in C_i$, and k is the maximum size of C_i.

In our unfolding, the size of the rules unfolded from r is bounded by $O(dkn)$, because each $\pi \in C_i$ is unfolded using every atom in D_i exactly once. Clearly, this is a polynomial time process. We therefore conclude that $U_{pos}(P)$ can be transformed from P in polynomial time.

In the approach given in [20], a rule r of the form (3) is unfolded to the collection of all instances of the rule, one for each combination of solutions to the n aggregates, in which case, the program unfolded from r is bounded by $O(dk^n)$.

Call-consistent condition: We can formulate a *call-consistent* condition [6] and describe sufficient conditions for the class of basic positive programs to possess an answer set and multiple answer sets, respectively. This is an an important class, since it includes basic programs under the by-complement semantics.

Let P be a basic positive program consisting of rules of the form (3). To be consistent with the original definition of stable model [8], we assume that a constraint $\bot \leftarrow$ *body* in a program is already replaced by a rule with an elementary head:

$$(\{f\}, \{\{f\}\}) \leftarrow \mathit{body}, (\{f\}, \{\emptyset\}).$$

where f is a new symbol.

We construct a *dependency graph* G_P as follows: for each rule of the form (3) in P, there is a positive edge from atom A to each atom $b \in \pi$, for all $\pi \in C_i$, $1 \leq i \leq n$; and a negative edge from A to each $d \in D_i$, $1 \leq i \leq n$, such that $\exists \pi \in C_i$ such that $d \notin \pi$.

We say that P has an *odd loop* if there is a path in G_P from an atom to itself via an odd number of negative edges, and P has an *even loop* if there is a path in G_P from an atom to itself via an even number of negative edges. P is said to be *call-consistent* if P has no odd loops.

Theorem 1. *Let P be a basic positive program. (i) P has an answer set if P is call-consistent. (ii) P has more than one answer set only if P has an even loop.*

Example 4. To illustrate the point (ii) above, consider the following program.

$$p \leftarrow . \quad a \leftarrow (\{p,b\},\{\{p\}\}). \quad b \leftarrow (\{p,a\},\{\{p\}\}).$$

The program has two answer sets $\{p,a\}$ and $\{p,b\}$. Then, according to the theorem, there must exist an even loop in its dependency graph. Indeed, the path from a to b negatively, and then from b to a negatively, is an even loop.

4.3 Unfolding Negative C-Atoms in Rule Bodies

Unfolding negative c-atoms is formulated for answer sets by reduct. Given a program P, the unfolded program will be denoted by $U_{neg}(P)$.

For each rule $r \in P$ of the form

$$head(r) \leftarrow pos(r), \texttt{not}\ (D_1, C_1), ..., \texttt{not}\ (D_n, C_n). \tag{4}$$

where $n \geq 0$, we will have a rule in $U_{neg}(P)$, which is obtained by replacing each negative c-atom $\texttt{not}\ (D_j, C_j)$ in (4), where $C_j = \{\pi_{j_1}, ..., \pi_{j_k}\}$, by a conjunction $\texttt{not}\ \psi_{j_1}, ..., \texttt{not}\ \psi_{j_k}$, where ψ_{j_i}'s are new symbols; in addition, for each $1 \leq j \leq n$, we will have the following rules in $U_{neg}(P)$:

$$\psi_{j_i} \leftarrow body_{j_i}. \qquad \text{for each } j_1 \leq j_i \leq j_k$$

where $body_{j_i} = \pi_{j_i} \cup \{\texttt{not}\ a \mid a \in D_j \setminus \pi_{j_i}\}$.

Example 5. Consider the following program P from [21]:

$$c \leftarrow \texttt{not}\ 1\{a,b\}1. \qquad a \leftarrow c. \qquad b \leftarrow a.$$

Note that $1\{a,b\}1$ represents the constraint $(\{a,b\},\{\{a\},\{b\}\})$. P has $\{a,b,c\}$ as its unique answer set by reduct. The unfolded program is:

$$c \leftarrow \texttt{not}\ \psi_1, \texttt{not}\ \psi_2. \quad \psi_1 \leftarrow a, \texttt{not}\ b. \quad \psi_2 \leftarrow b, \texttt{not}\ a. \quad a \leftarrow c. \quad b \leftarrow a.$$

Note that the unique answer set above is not an answer set by complement for P. In fact, the program has no answer sets by complement. As noticed in [21], an answer set by reduct may not be well-supported. The unfolded program provides a technical explanation to this phenomenon.

As noted in [21], for the program P above, $\{a, b, c\}$ is not a minimal model, since $\{b\}$ is a model. It is also known that every answer set by complement is an answer set by reduct, but the reverse does not hold. Thus, answer sets by complement can be computed by first computing answer sets by reduct, and then checking the minimality. Such an extra level of minimality checking sometimes may push the complexity to a higher level (e.g., this is the case from normal programs to disjunctive programs). An interesting question arises: Is the complexity of the semantics by complement higher than that of the semantics by reduct? We will see in Section 6 that the answer is NO.

Proposition 3. *For any general program P, the transformation from P to $U_{neg}(P)$ takes polynomial time and, under the by-reduct semantics, $AS(P) = AS_{|At(P)}(U_{neg}(P))$.*

5 Unfolding Disjunction

When a c-atom A appears in the head of a rule, there may be multiple specified solutions in A_c, and thus an answer set may not be a minimal model. Therefore, the traditional method of defining answer sets for disjunctive programs, namely requiring a set of atoms M to be a minimal model of P^M, is not applicable. On the other hand, when the head of a rule is a disjunction of c-atoms, minimal truth in these c-atoms is desirable, and when every c-atom in a disjunctive program is elementary, the semantics should reduce to the stable model semantics [9].

In this section, we describe unfolding of disjunction in the head, based on which we define answer sets for disjunctive programs. Given a disjunctive program P, the resulting program will be denoted $U_{dis}(P)$.

Consider any rule r in P of the form

$$(D_1, C_1) \vee ... \vee (D_k, C_k) \leftarrow body(r). \tag{5}$$

where $k \geq 1$. Let $C_i = \{\pi_{i_1}, ..., \pi_{i_m}\}$, where $1 \leq i \leq k$. The rule r can be unfolded to yield the following rules in $U_{dis}(P)$

$$\begin{array}{ll}
\Phi_1 \vee ... \vee \Phi_k \leftarrow body(r). & \\
1\{\xi_{\pi_{i_1}}, ..., \xi_{\pi_{i_m}}\}1 \leftarrow \Phi_i. & \text{for each } 1 \leq i \leq k, \\
h \leftarrow \xi_{\pi_{i_j}}. & \text{for each } \pi_{i_j} \in C_i \text{, and for each } h \in \pi_{i_j} \text{, and} \\
\bot \leftarrow \xi_{\pi_{i_j}}, d, \Phi_i. & \text{for each } \pi_{i_j} \in C_i \text{, and for each } d \in D_i \setminus \pi_{i_j}
\end{array}$$

where Φ_i and $\xi_{\pi_{i_j}}$ are all new symbols.

It's clear that unfolding of disjunction is a polynomial time process in the size of P.

Example 6. Consider the following rule where $D = \{sunny, windy, raining\}$.

$$(D, \{\{sunny\}\}) \vee (D, \{\{windy\}, \{raining\}, \{windy, raining\}\}) \leftarrow .$$

The rule can be unfolded to

$$\begin{array}{l}
pleasant \vee annoying \leftarrow . \quad 1\{nice\}1 \leftarrow pleasant. \\
sunny \leftarrow nice. \quad \bot \leftarrow nice, windy. \quad \bot \leftarrow nice, raining. \\
......
\end{array}$$

where the variables $pleasant$, $annoying$, and $nice$ are all new symbols. In the unfolding of disjunction, we first name the c-atoms in the head (e.g., $pleasant$ and $annoying$); when a disjunct in the head is satisfied, exactly one admissible set is implied (e.g., the second rule above), where each admissible set is explicitly named (e.g. $nice$ above); then the meaning of an admissible set is encoded (e.g., $nice$ implies $sunny$, and so on).

We now define answer sets for disjunctive programs with c-atoms.

Definition 5. *Let P be a disjunctive program and M be a set of atoms.*
Answer set by reduct: *Let $P' = U_{neg} \circ U_{pos} \circ U_{dis}(P)$. M is an* answer set by reduct *for P iff there is an answer set M' for P' such that $M = M'|_{At(P)}$.*
Answer set by complement: *Let P' be the complement of P, and $P'' = U_{pos} \circ U_{dis}(P')$. M is an* answer set by complement *for P iff there is an answer set M' for P'' such that $M = M'|_{At(P)}$.*

Theorem 2. *(i) If P is a general program, then the answer sets defined in Def. 5 coincide with those defined in Def. 4.*
(ii) If P is a disjunctive program, where any c-atom appearing in any rule head is elementary, then any answer set by complement for P is a minimal model.
(iii) If P is a disjunctive program where every c-atom is elementary, then the answer sets by reduct for P coincide with those by complement for P, and coincide with the stable models of P as defined in [9].

The result in (i) states that our semantics for disjunctive programs is a faithful extension of the semantics for non-disjunctive programs. The result stated in (ii) gives a condition for an answer set by complement to be minimal. In (iii), our semantics reduces to the standard stable model semantics for (conventional) disjunctive programs.

Head cycle-free condition: We can formulate a *head cycle-free* condition for a disjunctive program P. Let P consist of rules r of the form

$$(A_1, B_1) \vee ... \vee (A_k, B_k) \leftarrow (D_1, C_1), ..., (D_n, C_n), neg(r). \tag{6}$$

We construct a *positive dependency graph* G_P as follows: there is a positive edge from an atom p to an atom q if there is a rule $r \in P$ of the form (6) such that $p \in \xi$ for some $\xi \in B_i$, $1 \leq i \leq k$, and $q \in \pi$ for some $\pi \in C_j$, $1 \leq j \leq n$.

Two atoms a_1 and a_2 are said to be *head sharing* if there is a rule of the form 6 such that $a_1 \in B_i$ and $a_2 \in B_j$, for some i and j such that $1 \leq i, j \leq k$ and $i \neq j$. A *head cycle* refers to a path in G_P that goes through two *head-sharing* atoms. G_P is said to be *head cycle-free* if there is no head cycle in it.

Below, by *shifting*, we mean that, given program P, any disjunctive rule $r \in P$, of the form $A_1 \vee ... \vee A_k \leftarrow body(r)$ is replaced by the collection of rules

$$A_i \leftarrow body(r), \texttt{not}\ A_1, ..., \texttt{not}\ A_{i-1}, \texttt{not}\ A_{i+1}, ..., \texttt{not}\ A_k.$$

where $1 \leq i \leq k$. Let us denote the resulting program by P_{shift}.

Theorem 3. *Let P be a disjunctive program. If G_P is head cycle-free, then P can be reduced to a general program by shifting such that $AS(P) = AS(P_{shift})$.*

6 Complexity

Theorem 4. *For the following classes of programs with c-atoms, the problem of deciding whether an answer set exists for a program P in that class is NP-complete.*

(i) *P is a condition-free program.*
(ii) *P is a positive program.*
(iii) *P is a general program under the by-reduct semantics.*
(iv) *P is a general program under the by-complement semantics.*
(v) *P is a disjunctive program such that G_P is head cycle-free.*

Proof
Hardness: There is a polynomial time reduction from the NP-complete problem CSP to a condition-free program, and the class of condition-free programs is a subclass of all the other classes.

Membership in NP: The decision problems for (ii) & (iii) are no harder than the corresponding decision problem for normal logic programs (Propositions 1, 2, and 3). For (v), the conclusion follows from Theorem 3 and the result on shifting [1].

We prove it for (iv). Let's guess a set of atoms I for which we are to verify deterministically in polynomial time whether it is an answer set by complement for P. We can do this by checking whether $I = T_P^\infty(\emptyset, I)$, with a straightforward extension of conditional satisfaction to negative c-atoms: for any set of atoms S, $S \models_I \texttt{not}\ A$ iff $S \models_I A'$, where A' is the complement of A.

In computing $T_P^\infty(\emptyset, I)$, the whole process terminates after at most n iterations, where $n = |At(P)|$. Thus, we only need to show that each iteration in computing $T_P^n(\emptyset, I)$ takes polynomial time in the size of P. Clearly, this holds if checking the (extended) conditional satisfaction of a c-atom A as well as a negative c-atom $\texttt{not}\ A$ takes polynomial time in the size of A.

Let S be a set of atoms and A a c-atom. We check $S \models_I A$ as follows. By definition, $S \models_I A$ means $S \cap A_d \in A_c$, and any set in between $S \cap A_d$ and $I \cap A_d$ must be in A_c. We know that the number of the sets that are said to be in A_c here is 2^N, where $N = |I \cap A_d - S \cap A_d|$. (Note that $S \cap A_d$ may not be a subset of $I \cap A_d$, in which case, $N = 0$ and $S \cap A_d$ is the only set that is counted.) So, if the size of the set $\{S \cap A_d \mid S \cap A_d \in A_c\} \cup \{\pi \mid \pi \in A_c, I \cap \pi \supseteq S \cap A_d\}$ is also 2^N, then $S \models_I A$, otherwise $S \not\models_I A$. Clearly, this can be checked in polynomial time in the size of A.

Now consider $\texttt{not}\ A$, with A' being the complement of A. By definition, $S \models_I \texttt{not}\ A$ iff $S \models_I A'$ iff

(i) $S \models A'$, and for any S' such that $S \cap A_d \subseteq S' \subseteq I \cap A_d$, $S' \cap A_d \in A'_c$, iff
(ii) $S \cap A_d \notin A_c$, and for each $\pi \in A_c$, it is not the case that $S \cap A_d \subseteq \pi \subseteq I \cap A_d$.

Clearly, the last statement can be checked in polynomial time in the size of A. □

We can also show

Theorem 5. *For the following classes of programs with c-atoms, the problem of deciding whether an answer set exists for a program P in that class is Σ_2^P-complete.*

(i) *P is a disjunctive program under the by-reduct semantics.*
(ii) *P is a disjunctive program under the by-complement semantics.*

7 Relation to Previous Work

Logic programming with abstract constraints was proposed in [13], where the semantics were studied for the programs with monotone c-atoms. Son et al. [21] treated arbitrary c-atoms. In this paper, we further extended this line of work by treating disjunction. Our treatment is based on unfolding, which was used earlier in [20] to unfold positive c-atoms in rule bodies. We extend this approach by also unfolding the head, including the disjunctive head, and negative c-atoms in rule bodies (for the by-reduct semantics).

Disjunction has been considered in [5,17], where the head of a rule is a disjunction of elementary c-atoms. We have shown in Theorem 2 that when the head contains only elementary c-atoms, we will get the expected behavior, namely all answer sets by complement are minimal models.

Logic programs with aggregates have gained quite a bit attention lately. Usually, it is assumed that an aggregate only occurs positively in a rule body. By allowing c-atoms to appear anywhere in a disjunctive rule, we are able to represent conditional constraints and disjunctive constraints.

Our complexity results are derived for the language of logic programs with c-atoms, where the size of a c-atom A could be exponential in the size of A_d. It is important to note that this language differs from the language of logic programs with aggregates, where aggregates are expressed by pre-defined aggregate functions over some relational operators. The complexity for the latter has been reported in [19]. To the best of our knowledge, the complexity for programs with aggregates/c-atoms in the head has not been studied before.

8 Final Remarks

In this paper, we have studied the class of disjunctive programs with c-atoms, and various subclasses, where a c-atom can appear anywhere in a rule. We have also provided some complexity results for these classes of programs. We have shown that programs in these classes all have an intuitive, rather direct interpretation in terms of the familiar programs composed of ordinary atoms, which can be obtained by unfolding. We have shown that the insights into the semantical issues for logic programs with c-atoms can often be revealed by their unfolded counterparts.

An interesting question is how to compute answer sets for classes of programs with arbitrary c-atoms. The unfolding approach studied here is intended as a means for theoretical analysis. In practical systems, c-atoms are typically expressed in terms of some pre-defined functions and predicates, such as weight and cardinality constraints, and those involving aggregates. In this case, an implementation supported by special constraint propagation algorithms for effective space pruning is highly desirable, which we are studying at the present time.

Acknowledgement. The work by Jia-Huai You, Li Yan Yuan and Guohua Liu were partially supported by NSERC, and the first two were also assisted from Grant NSFC 60573009. Yi-Dong Shen is supported by the NSFC grants 60673103 and 60421001.

References

1. R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Math. and Artificial Intelligence*, 14:53–87, 1994.
2. F. Calimeri, W. Faber, N. Leone, and S. Perri. Declarative and computational properties of logic programs with aggregates. In *Proc. IJCAI'05*, pages 406–411, 2005.
3. T. Dell'Armi, W. Faber, G. Lelpa, and N. Leone. Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In *Proc. IJCAI'03*, pages 847–852, 2003.
4. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantic for logic programs with aggregates. In *Proc. ICLP '01*, pages 212–226, 2001.
5. W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs. In *Proc. JELIA'04*, pages 200–212, 2004.
6. F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
7. M. Gelfond. *Representing knowledge in A-Prolog*, chapter 413-451. Springer, 2002.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.
9. M. Gelfond and V. Lifschitz. Classic negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
10. L. Liu and M. Truszczyński. Properties of programs with monotone and convex constraints. In *Proc. AAAI '05*, pages 701–706, 2005.
11. L. Liu and M. Truszczyński. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research*, 7:299–334, 2006.
12. V. Marek, I. Niemelä, and M. Truszczyński. Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*. To appear.
13. V. Marek and M. Truszczyński. Logic programs with abstract constraint atoms. In *Proc. AAAI '04*, pages 86–91, 2004.
14. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. AAAI'90*, pages 25–32, 1990.
15. N. Pelov. *Semantics of Logic Programs with Aggregates*. PhD thesis, Ketholieke Universiteit Leuven, 2004.
16. N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*. To appear.
17. N. Pelov and M. Truszczyński. Semantics of disjunctive programs with monotone aggregates an operator-based approach. In *Proc. NMR '04*, pages 327–334, 2004.
18. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2), 2002.
19. T.C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*. To appear.
20. T.C. Son, E. Pontelli, and I. Elkabani. A translational semantics for aggregates in logic programs. Technical Report CS-2005-006, New Mexico State University, 2005.
21. T.C. Son, E. Pontelli, and P. Tu. Answer sets for logic programs with arbitrary abstract constraint atoms. In *Proc. AAAI'06*, pages 129–134, 2006.

General Default Logic

Yi Zhou[1], Fangzhen Lin[2], and Yan Zhang[1]

[1] School of Computing and Mathematics
University of Western Sydney
Penrith South DC NSW 1797, Australia
[2] Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract. In this paper, We propose a general default logic. It extends Reiter's default logic by adding rule connectives like disjunction in logic programming, and Ferraris's general logic program by allowing arbitrary propositional formulas to be the base in forming logic programs. We show the usefulness of this logic by applying it to formalizing rule constraints, generalized closed world assumptions, and conditional defaults.

1 Introduction

In this paper, we consider a language with connectives from both classical logic and logic programming, and provide a fixed-point semantics for the language so that our logic is both a generalization of Reiter's default logic [1] and Ferraris's general logic programs with stable model semantics [2].

The interplay between Reiter's default logic and logic programming with negation-as-failure has been going on since the beginning of nonmonotonic reasoning. Reiter's default logic was considered to be a formalization of default reasoning including the negation-as-failure mechanism used in Prolog. This is confirmed when Gelfond and Lifschitz [3] showed that their stable model semantics of normal logic programs [4] can be embedded in Reiter's default logic. In a normal logic program, the head of a rule must be an atom. If one allows disjunctions of atoms in the head of rule, then one obtains what has been called disjunctive logic programs, and Gelfond *et al.* [5] showed that the answer set semantics of these disjunctive logic programs can be embedded in their disjunctive default logic. Notice that Reiter's default logic also allows disjunction in the formulas occurring in default rules, but this disjunction is different from the disjunction used in disjunctive logic programming. One could say that the former is *classical* disjunction as in classical logic while the latter is *default* disjunction. The difference is that default disjunction means *minimality* while classical disjunction does not. For instance, the logic program $\{a|b \leftarrow\}$ has two answer sets $\{a\}$ and $\{b\}$, but the default theory $(\{\}, \{: a \vee b / a \vee b\})$ has one extension $\{a \vee b\}$ which has three models[1] $\{a\}$, $\{b\}$, and $\{a, b\}$.

[1] We identify a model with the set of atoms that are true in the model.

Another difference between disjunction in disjunctive logic programs and disjunction in formulas in classical logic is that the former is allowed only in the head of a rule. It cannot occur in the body of a rule nor can it be nested. This limitation in the use of disjunction in disjunctive logic programming has been addressed by Lifschitz *et al.* [6], Pearce [7] and recently by Ferraris [2]. He defined logic programs that look just like propositional formulas but are interpreted according to a generalized stable model semantics. In other words, in these formulas called general logic programs, negations are like negation-as-failure, implications are like rules, and disjunctions are like those in disjunctive logic programs. While Ferraris's general logic programs go well-beyond disjunctive logic programs by allowing disjunctions, negation-as-failure, and rules to occur anywhere, they do not allow any classical disjunctions and implications. Thus the natural question is whether a meaningful logic can be defined that allow both connectives from logic programming and classical logic. This paper answers this question positively. Before we proceed to define our logic formally, the following example illustrates the need for classical disjunction in default reasoning.

Example 1. Suppose that we have the following information about the students in a high school:

(*) A student good at math is normally good at physics. Conversely, a student good at physics is normally good at math.

In logic programming, this can be represented as follows:

1. $GoodAtPhysics(x) \leftarrow GoodAtMath(x),\ \texttt{not}\ \neg GoodAtPhysics(x)$[2];
2. $GoodAtMath(x) \leftarrow GoodAtPhysics(x),\ \texttt{not}\ \neg GoodAtMath(x)$.

Now suppose we are told that Mike is either good at math or good at physics. If we represent this disjunctive information as:

3. $GoodAtMath(Mike) \mid GoodAtPhisics(Mike)$

then we would conclude that Mike is both good at math and good at physics as the logic program $\{1, 2, 3\}$ has a unique answer set. One could argue whether this is a reasonable conclusion. If we do not wish the defaults to be applied here, then we could replace (3) by the following fact:

3'. $GoodAtMath(Mike) \vee GoodAtPhisics(Mike)$.

As we shall see, in our logic, the theory representing $\{1, 2, 3'\}$ will have a unique extension $\{GoodAtMath(Mike) \vee GoodAtPhisics(Mike)\}$.

2 General Default Logic

In this section, we define our logic $\mathcal{R}$ that allow both classical connectives and logic programming connectives. Then, we show that Reiter's default logic, Gelfond *et al.*'s disjunctive default and Ferraris's general logic programs are special cases of the extension semantics of general default logic.

[2] We use a first order notation here to denote the set of all grounded propositional rules.

2.1 Syntax and Basic Semantics

Let *Atom* be a set of atoms, also called propositional variables. By $\mathcal{L}$ we mean the classical propositional language defined recursively by *Atom* and classical connectives $\bot$, $\neg$, $\rightarrow$ as follows:

$$F ::= \bot \mid p \mid \neg F \mid F \rightarrow F,$$

where $p \in Atom$. $\top$, $F \wedge G$, $F \vee G$ and $F \leftrightarrow G$ are considered as shorthands of $\bot \rightarrow \bot$, $\neg(F \rightarrow \neg G)$, $\neg F \rightarrow G$ and $(F \rightarrow G) \wedge (G \rightarrow F)$ respectively, where F and G are formulas in $\mathcal{L}$. Formulas in $\mathcal{L}$ are called *facts*. The satisfaction relation between a set of facts and a fact is defined as usual. A *theory* T is a set of facts which is closed under the classical entailment. Let Γ be a set of facts, $Th(\Gamma)$ denotes the logic closure of Γ under classical entailment. We write Γ to denote the theory $Th(\{\Gamma\})$ if it clear from the context. For instance, we write $\emptyset$ to denote the theory of all tautologies; we write $\{p\}$ to denote the theory of all logic consequences of p. We say that a theory T is *inconsistent* if there is a fact F such that $T \models F$ and $T \models \neg F$, otherwise, we say that T is *consistent*.

We introduce a set of new connectives, called *rule connectives*. They are $-$ for *negation as failure* or *rule negation*, $\Rightarrow$ for *rule implication*, $\&$ for *rule and*, $\mid$ for *rule or* and $\Leftrightarrow$ for *rule equivalence* respectively. We define a propositional language $\mathcal{R}$ recursively by facts and rule connectives as follows:

$$R ::= F \mid R \Rightarrow R \mid R \,\&\, R \mid R \,|\, R,$$

where F is a fact. $-R$ and $R \Leftrightarrow S$ are considered as shorthands of $R \Rightarrow \bot$ and $(R \Rightarrow S) \,\&\, (S \Rightarrow R)$ respectively, where R and S are formulas in $\mathcal{R}$. Formulas in $\mathcal{R}$ are called *rules* or *rule formulas*. Particularly, facts are also rules. A *rule base* Δ is a set of rules.

The order of priority for these connectives are

$$\{\neg\} > \{\wedge, \vee\} > \{\rightarrow, \leftrightarrow\} > \{-\} > \{\,\&\,, \,|\,\} > \{\Rightarrow, \Leftrightarrow\}.$$

For example, $-p \vee \neg p \Rightarrow q$ is a well defined rule, which denotes the rule $(-(p \vee (\neg p))) \Rightarrow q$. Both $\neg p \vee p$ and $\neg p \,|\, p$ are well defined rules. The former is also a fact but the latter is not a fact. However, $\neg(p \,|\, p)$ is not a well defined rule.

We define the *subrule* relationship between two rules recursively as follows:

1. R is a subrule of R;
2. R and S are subrules of $R \Rightarrow S$;
3. R and S are subrules of $R \,\&\, S$;
4. R and S are subrules of $R \,|\, S$,

where R and S are rules. Thus, clearly, R is a subrule of $-R$. For example, p is a subrule of $\neg p \,|\, p$ but not a subrule of $\neg p \vee p$.

We now define the *satisfaction relation* $\models_R$ between a theory and a rule inductively:

- If R is a fact, then $T \models_R R$ iff $T \models R$.
- $T \models_R R \,\&\, S$ iff $T \models_R R$ and $T \models_R S$;

– $T \models_R R \,|\, S$ iff $T \models_R R$ or $T \models_R S$;
– $T \models_R R \Rightarrow S$ iff $T \not\models_R R$ or $T \models_R S$.

Thus, $T \models_R -R$ iff $T \models_R R \rightarrow \bot$ iff $T \not\models_R R$ or $T \models_R \bot$. If T is consistent, then $T \models_R -R$ iff $T \not\models_R R$. If T is inconsistent, then for every rule R, $T \models_R R$. $T \models_R R \Leftrightarrow S$ iff $T \models_R (R \Rightarrow S) \,\&\, (S \Rightarrow R)$ iff $T \models_R R \Rightarrow S$ and $T \models_R S \Rightarrow R$.

We say that T *satisfies* R, also T is a *model* of R iff $T \models_R R$. We say that T satisfies a rule base Δ iff T satisfies every rule in Δ. We say that two rule bases are *weakly equivalent* if they have the same set of models.

For example, let T be $\emptyset$. T is a model of $\neg p \vee p$, but T is not a model of $\neg p \,|\, p$. This example also shows a difference between the two connectives $\vee$ and $|$. As another example, T is a model of $-p$ but not a model of $\neg p$. This example also shows a difference between the two connectives $\neg$ and $-$.

Theorem 1. *Let T be a theory and R, S two rule formulas in $\mathcal{R}$.*

1. *$T \models_R - - R$ iff $T \models_R R$.*
2. *$T \models_R -(R \,\&\, S)$ iff $T \models_R -R \,|\, -S$.*
3. *$T \models_R -(R \,|\, S)$ iff $T \models_R -R \,\&\, -S$.*
4. *$T \models_R R \Rightarrow S$ iff $T \models_R -R \,|\, S$.*

Proof. These assertions follow directly from the definitions. As an example, we write down the proof of assertion 2 here. According to the definition, $T \models_R -(R \,\&\, S)$ iff T is not a model of $R \,\&\, S$, which holds iff a) T is not a model of R or b) T is not a model of S. On the other hand, $T \models_R -R \,|\, -S$ iff a) T is a model of $-R$ or b) T is a model of $-S$, which holds iff a) T is not a model of R or b) T is not a model of S. Hence, assertion 2 holds.

2.2 Extension

Let T be a theory and R a rule in $\mathcal{R}$. The reduction of R on T, denoted by R^T, is the formula obtained from R by replacing every maximal subrules of R which is not satisfied by T with $\bot$. It can also be defined recursively as follows:

– If R is a fact, then $R^T = \begin{cases} R \text{ if } T \models_R R \\ \bot \text{ otherwise} \end{cases}$,
– If R is $R_1 \odot R_2$, then $R^T = \begin{cases} R_1^T \odot R_2^T & \text{if } T \models_R R_1 \odot R_2 \\ \bot & \text{otherwise} \end{cases}$, where $\odot$ is $\&$, $\Leftrightarrow$, $|$ or $\Rightarrow$.

Thus, if R is $-S$ and T is consistent, then $R^T = \begin{cases} \bot \text{ if } T \models_R S \\ \top \text{ otherwise} \end{cases}$.

Let T be a theory and Δ a rule base, the reduction of Δ on T, denoted by Δ^T, is the set of all the reductions of rules in Δ on T.

For example, let T be $\{p\}$. The reduction of $-p \Rightarrow q$ on T is $\bot \Rightarrow \bot$, which is weakly equivalent to $\top$. The reduction of $p \,|\, q$ on T is p. The reduction of $p \vee q$ on T is $p \vee q$. This example also shows that although two rules are weakly equivalent (e.g. $-p \Rightarrow q$ and $p \,|\, q$), their reductions on a same theory might not be weakly equivalent.

This notion of reduction is similar to Ferraris's notion of reduction for general logic programs [2].

Definition 1. *Let T be a theory and Δ a rule base. We say that T is an extension of F iff:*

1. $T \models_R \Delta^T$.
2. *There is no theory T_1 such that $T_1 \subset T$ and $T_1 \models_R \Delta^T$.*

Example 2. Consider the rule base $\Delta_1 = \{-p \Rightarrow q\}$.

– Let T_1 be $\emptyset$. The reduction of Δ_1 on T_1 is $\{\bot\}$. T_1 is not a model of $\Delta_1^{T_1}$. Hence, T_1 is not an extension of Δ_1.
– Let T_2 be $\{p\}$. The reduction of Δ_1 on T_2 is $\bot \Rightarrow \bot$. T_2 is a model of $\Delta_1^{T_2}$. But $\emptyset$ is also a model of $\Delta_1^{T_2}$ and $\emptyset \subset T_2$. Hence, T_2 is not an extension of Δ_1.
– Let T_3 be $\{q\}$. The reduction of Δ_1 on T_3 is $\top \Rightarrow q$, which is weakly equivalent to q. T_3 is a model of q and there is no proper subset of T_3 which is also a model of q. Hence, T_3 is an extension of Δ_1.
– Let T_4 be $\{\neg p \wedge q\}$. The reduction of Δ_1 on T_4 is $\top \Rightarrow q$, which is also weakly equivalent to q. T_4 is a model of $\Delta_1^{T_4}$. But $\{q\}$ is also a model of $\Delta_1^{T_4}$. Hence T_4 is not an extension of Δ_1.
– We can examine that the only extension of Δ_1 is T_3.

Similarly, $p \mid q$ has two extensions: $\{p\}$ and $\{q\}$. $p \vee q$ has a unique extension $\{p \vee q\}$. This example shows that although two rules are weakly equivalent, their extensions might not be the same (e.g. $-p \Rightarrow q$ and $p \mid q$).

Example 3 (Example 1 continued). Consider the example mentioned in the introduction section again. Reformulated in general default logic with domain $D = \{Mike\}$, the rule base $1, 2, 3'$, denoted by Δ, is:

1. $GoodAtMath(Mike) \& -\neg GoodAtPhysics(Mike) \Rightarrow GoodAtPhysics(Mike)$;
2. $GoodAtPhysics(Mike) \; \& \; -\neg GoodAtMath(Mike) \Rightarrow GoodAtMath(Mike)$;
3'. $GoodAtMath(Mike) \vee GoodAtPhisics(Mike)$.

Let T_1 be the theory $Th(\{GoodAtMath(Mike), GoodAtPhysics(Mike)\})$. The reduction of Δ on T_1 is

1-1 $GoodAtMath(Mike) \Rightarrow GoodAtPhysics(Mike)$;
1-2 $GoodAtPhysics(Mike) \Rightarrow GoodAtPhysics(Mike)$;
1-3 $GoodAtMath(Mike) \vee GoodAtPhysics(Mike)$.

Although T_1 is a model of Δ^{T_1}, $GoodAtMath(Mike) \vee GoodAtPhysics(Mike)$ is also a model of Δ^{T_1} and it is a proper subset of T_1. Thus, T_1 is not a extension of Δ. And we can examine the only extension of Δ is $\{GoodAtMath(Mike) \vee GoodAtPhysics(Mike)\}$.

It is clear that if Δ is a set of facts, then Δ has exactly one extension, which is the deductive closure of itself.

Intuitively, the extensions of a rule base represent all possible beliefs which can be derived from the rule base. The following theorem shows that every extension of a rule base is also a model of it.

Theorem 2. *Let T be a consistent theory and Δ a rule base. If T is an extension of Δ, then T satisfies Δ.*

Proof. According to the definitions, it is easy to see that $T \models_R \Delta^T$ iff $T \models_R \Delta$. On the other hand, if T is an extension of Δ, then $T \models_R \Delta^T$. Hence, this assertion holds.

The converse of Theorem 2 does not hold in general. For instance, $\{p\}$ is a model of $\{- - p\}$, but not an extension of it.

2.3 Default Logic

In this paper, we only consider Reiter's default logic in propositional case. A default rule has the form

$$p : q_1, \dots, q_n / r,$$

where p, $q_i, 1 \le i \le n$ and r are propositional formulas. p is called the prerequisite, $q_i, 1 \le i \le n$ are called the justifications and r is called the consequent. A default theory is a pair $\Delta = (W, D)$, where W is a set of propositional formulas and D is a set of default rules. A theory T is called an extension of a default theory $\Delta = (W, D)$ if $T = \Gamma(T)$, where for any theory S, $\Gamma(S)$ is the minimal set (in the sense of subset relationship) satisfying the following three conditions:

1. $W \subseteq \Gamma(S)$.
2. $\Gamma(S)$ is a theory.
3. For any default rule $p : q_1, \dots, q_n / r \in D$, if $p \in \Gamma(S)$ and $\neg q_i \notin S, 1 \le i \le n$, then $r \in \Gamma(S)$.

We now show that Reiter's default logic in propositional case can be embedded into the logic $\mathcal{R}$. Let R be a default rule with the form $p : q_1, \dots, q_n / r$. By R^* we denote the following rule in $\mathcal{R}$

$$p \,\&\, -\neg q_1 \,\&\, \dots \,\&\, -\neg q_n \Rightarrow r.$$

Let $\Delta = (W, D)$ be a default theory, by Δ^* we denote the rule base

$$W \cup \{R^* \mid R \in D\}.$$

Theorem 3. *Let T be a theory and $\Delta = (W, D)$ a default theory. T is an extension of Δ iff T is an extension of Δ^*.*

Proof. $\Rightarrow$: Suppose that T is an extension of Δ. Then $T \models_R W^T$ since W is a set of facts. Moreover, for all rule $R \in D$ with the form $p : q_1, \dots, q_n / r$. There are three cases:

- $p \notin T$. In this case, R^{*T} is weakly equivalent to $\top$. Thus, $T \models_R R^{*T}$.
- There is a $q_i, 1 \leq i \leq n$ such that $\neg q_i \in T$. In this case, R^{*T} is also weakly equivalent to $\top$. Thus, $T \models_R R^{*T}$.
- $p \in T$ and there is no $q_i, 1 \leq i \leq n$ such that $\neg q_i \in T$. In this case, according to the definition of extensions in default logic, $r \in T$. Therefore, R^{*T} is weakly equivalent to $p \Rightarrow r$. Hence, $T \models_R R^{*T}$.

This shows that for all $R \in D$, $T \models_R R^{*T}$. Hence, $T \models_R \Delta^{*T}$. On the other hand, there is no consistent theory $T_1 \subset T$ and $T_1 \models_R \Delta^{*T}$. Otherwise, suppose there is such a T_1. T_1 must satisfy W since $W \subseteq \Delta^{*T}$. For all rule $R \in D$, T_1 satisfies R^{*T}. Therefore, T_1 satisfies the third condition in the definition of default extensions. Therefore, $\Gamma(T) \subseteq T_1$. Hence, $\Gamma(T) \neq T$. This shows that T is not an extension of Δ, a contradiction.

$\Leftarrow$: Suppose that T is an extension of Δ^*. We now show that T is the smallest theory satisfying condition 1 to 3 in the definition of default extensions. First, $T \models_R W$ since $W \subseteq \Delta^*$ and W is a set of facts. Second, T is a theory. Finally, for all rule $R \in \Delta$ with the form $p : q_1, \ldots, q_n/r$, if $p \in T$ and there is no $q_i, 1 \leq i \leq n$ such that $\neg q_i \in T$, then R^{*T} is $p \Rightarrow r$. And $T \models_R R^{*T}$, therefore $r \in T$. This shows that T satisfies all those conditions. Now suppose otherwise there is a proper subset T_1 of T also satisfies Condition 1 to 3. Then, similarly $T_1 \models_R W$ and for all rule $R \in D$, $T_1 \models_R R^{*T}$. Thus, $T_1 \models_R \Delta^{*T}$. This shows that T is not an extension of Δ^*, a contradiction.

Observe that R and R^* are essential the same except the syntax to represent them. Thus, Reiter's default logic in propositional case is a special case of general default logic. In the rest of this section, we shall also show that Gelfond *et al.*'s disjunctive default logic is a special case of $\mathcal{R}$ by a similar translation.

A disjunctive default rule has the form

$$p : q_1, \ldots, q_n/r_1, \ldots, r_k,$$

where p, $q_i, 1 \leq i \leq n$ and $r_j, 1 \leq j \leq k$ are propositional formulas. A disjunctive default theory is a pair $\Delta = (W, D)$, where W is a set of propositional formulas and D is a set of disjunctive default rules. A theory T is called an extension of a disjunctive default theory $\Delta = (W, D)$ if $T = \Gamma(T)$, where for any theory S, $\Gamma(S)$ is the minimal set (in the sense of subset relationship) satisfying the following three conditions:

1. $W \subseteq \Gamma(S)$.
2. $\Gamma(S)$ is a theory.
3. For any default rule $p : q_1, \ldots, q_n/r_1, \ldots, r_k \in D$, if $p \in \Gamma(S)$ and $\neg q_i \notin S, 1 \leq i \leq n$, then for some $j, 1 \leq j \leq k$, $r_j \in \Gamma(S)$.

We now show that Gelfond *et al.*'s default logic in propositional case can be embedded into the logic $\mathcal{R}$ as well. Let R be a disjunctive default rule with the form $p : q_1, \ldots, q_n/r_1, \ldots, r_k$. By R^* we denote the following rule in $\mathcal{R}$

$$p \,\&\, -\neg q_1 \,\&\, \ldots \,\&\, -\neg q_n \Rightarrow r_1 \mid \ldots \mid r_k.$$

Let $\Delta = (W, D)$ be a disjunctive default theory, by Δ^* we denote the rule base

$$W \cup \{R^* \mid R \in D\}.$$

Theorem 4. *Let T be a theory and $\Delta = (W, D)$ a disjunctive default theory. T is an extension of Δ iff T is an extension of Δ^*.*

Proof. This proof is quite similar with the proof of Theorem 3.

2.4 General Logic Programming

Ferraris's general logic programs are defined over propositional formulas. Given a propositional formula F and a set of atoms X, the reduction of F on X, denoted by F^X, is the proposition formula obtained from F by replacing every subformula which is not satisfied by F into $\bot$. Given a set of propositional formulas Δ, Δ^X is the set of all reductions of formulas in Δ on X. A set of atoms X is said to be a stable model of Δ iff X is the minimal set (in the sense of subset relationship) satisfying Δ^X.

Let F be a propositional formula. By F^* we denote the formula in $\mathcal{R}$ obtained from F by replacing every classical connectives into corresponding rule connectives, that is, from $\rightarrow$ to $\Rightarrow$, from $\neg$ to $-$, from $\wedge$ to $\&$, from $\vee$ to $\mid$ and from $\leftrightarrow$ to $\Leftrightarrow$. Let Δ be a general logic program, by Δ^* we denote the rule base

$$\{F^* \mid F \in \Delta\}.$$

Lemma 1. *Let X be a set of atoms and F a propositional formula. $Th(X)$ is a model of F^* iff X is a model of F.*

Proof. We prove this assertion by induction on the structure of F.

1. If F is $\top$ or $\bot$, it is easy to see that this assertion holds.
2. If F is an atom p, then $Th(X) \models_R F^*$ iff $p \in X$ iff X is a model of F.
3. If F is $\neg G$, then $Th(X) \models_R F^*$ iff $Th(X) \models_R -G^*$ iff $Th(X)$ is not a model of G^* iff X is not a model of G iff X is a model of $\neg G$.
4. If F is $G \wedge H$, then $Th(X) \models_R F^*$ iff $Th(X) \models_R G^* \,\&\, H^*$ iff $Th(X)$ is a model of G^* and $Th(X)$ is a model of H^* iff X is a model of G and X is a model of H iff X is a model of $G \wedge H$.
5. If F is $G \vee H$, then $Th(X) \models_R F^*$ iff $Th(X) \models_R G^* \mid H^*$ iff $Th(X)$ is a model of G^* or $Th(X)$ is a model of H^* iff X is a model of G or X is a model of H iff X is a model of $G \vee H$.
6. If F is $G \rightarrow H$, then $Th(X) \models_R F^*$ iff $Th(X) \models_R G^* \Rightarrow H^*$ iff $Th(X)$ is not a model of G^* or $Th(X)$ is a model of H^* iff X is not a model of G or X is a model of H iff X is a model of $G \rightarrow H$.
7. If F is $G \leftrightarrow H$, then $Th(X) \models_R F^*$ iff $Th(X) \models_R G^* \Leftrightarrow H^*$ iff a) $Th(X)$ is both a model of G^* and a model of H^* or b) $Th(X)$ is neither a model of G^* nor a model of H^* iff a) X is both a model of G and a model of H or b) X is neither a model of G and nor a model of H iff X is a model of $G \leftrightarrow H$.

This completes the induction proof.

Theorem 5. *Let X be a set of atoms and Δ a general logic program. X is a stable model of Δ iff $Th(X)$ is an extension of Δ^*.*

Proof. By Lemma 1, it is easy to see that $(\Delta^X)^*$ is the same as $(\Delta^*)^{Th(X)}$.

$\Rightarrow$: Suppose X is a stable model of Δ. Then X is the minimal set satisfying Δ^X. By Lemma 1, $Th(X)$ is a model of $(\Delta^X)^*$. Thus $Th(X)$ is a model of $(\Delta^*)^{Th(X)}$. And there is no proper subset T_1 of $Th(X)$ such that $T_1 \models_R (\Delta^*)^{Th(X)}$. Otherwise, T_1 is a model of $(\Delta^X)^*$. Let X_1 be the set of atoms $\{p \mid T_1 \models p\}$. By induction on the structure, it is easy to see that for any set of propositional formulas Γ, T_1 is a model of Γ^* iff $Th(X_1)$ is a model of Γ^*. Hence, $Th(X_1)$ is a model of $(\Delta^X)^*$. Therefore by Lemma 1, X_1 is a model of Δ^X. Moreover $X_1 \subset X$ since $T_1 \subset Th(X)$. This shows that X is not a stable model of Δ, a contradiction.

$\Leftarrow$: Suppose $Th(X)$ is an extension of Δ^*. Then $Th(X)$ is the minimal set satisfying $(\Delta^*)^{Th(X)}$. Therefore, $Th(X)$ is the minimal set satisfying $(\Delta^X)^*$. By Lemma 1, it is easy to see that X is the minimal set satisfying Δ^X. Therefore, X is a stable model of Δ.

Actually, although Ferraris used the notations of classical connectives to denote the connectives in general logic programs, those connectives are still connectives in answer set programming. They are essentially rule connectives. In this paper, we use a set of rule connectives to denote them. Hence, the answer set semantics for general logic programs is also a special case of general default logic. Moreover, it is a special case of general default logic which only allows the facts are atoms, while general logic programming with strong negation (namely classical negation) is also a special case of general default logic which allows the facts are literals.

In [3], Gelfond and Lifschitz showed that the answer set semantics for normal logic programs is a special case of Reiter's default logic; in [5], Gelfond *et al.* showed that the answer set semantics for disjunctive logic programs is a special case of their disjunctive default logic. Together with this work, one can observe that the series of semantics for answer set programs (with classical negation) are essentially special cases of corresponding semantics for default logics which restrict the facts into atoms (literals).

3 Applications

In this section, we show that this logic is flexible enough to represent several important situations in common sense reasoning, including rule constraints, general closed world assumptions and conditional defaults.

3.1 Representing Rule Constraints

Similar to constraints in answer set programming, constraints in general default logic eliminate the extensions which do not satisfy the constraints. Let R be a rule, the constraint of R can be simply represented as

$$-R.$$

Theorem 6. *T is an extension of $\Delta \cup \{-R\}$ iff T is an extension of Δ and $T \not\models_R R$.*

Proof. $\Rightarrow$: Suppose that T is an extension of $\Delta \cup \{-R\}$. Then by the definition, $T \models_R (-R)^T$. Thus, $T \models_R -R$. Therefore $T \not\models_R R$. On the other hand, $T \models_R \Delta^T$. We only need to prove that T is a minimal theory satisfying Δ^T. Suppose otherwise, there is a theory T_1 such that $T_1 \subset T$ and $T_1 \models_R \Delta^T$. Notice that $(\Delta \cup \{-R\})^T$ is $\Delta^T \cup \{(-R)^T\}$, which is $\Delta^T \cup \{\top\}$. Thus, $T_1 \models_R (\Delta \cup \{-R\})^T$. This shows that T is not an extension of $\Delta \cup \{-R\}$, a contradiction.

$\Leftarrow$: Suppose that T is an extension of Δ and $T \not\models_R R$. Then T is the minimal theory satisfying Δ^T. Thus, T is also the minimal theory satisfying $(\Delta \cup \{-R\})^T$ since $(-R)^T$ is $\top$. Therefore, T is an extension of $\Delta \cup \{-R\}$.

3.2 Representing General Closed World Assumptions

In answer set programming, given an atom p, closed world assumption for p is represented as follows:

$$\neg p \leftarrow \texttt{not}\, p.$$

Reformulated in general default logic, it is

$$-p \Rightarrow \neg p.$$

However, this encoding of closed world assumption may lead to counter-intuitive effects when representing incomplete information. Consider the following example [8,9].

Example 4. Suppose we are give the following information:

(*) If a suspect is violent and is a psychopath then the suspect is extremely dangerous. This is not the case if the suspect is not violent or not a psychopath.

This statement can be represented (in general default logic) as three rules:

1. $violent \,\&\, psychopath \Rightarrow dangerous.$
2. $\neg violent \Rightarrow \neg dangerous.$
3. $\neg psychopath \Rightarrow \neg dangerous.$

Let us also assume that the DB has complete positive information. This can be captured by closed world assumption. In the classical approach, it can be represented as follows:

4. $-violent \Rightarrow \neg violent.$
5. $-psychopath \Rightarrow \neg psychopath.$

Now suppose that we have a disjunctive information that a person is either violent or a psychopath. This can be represented as:

6. $violent \mid psychopath$.

It is easy to see that the rule base $1-6$ [3] has two extensions:

$$Th(\{\neg violent, psychopath, \neg dangerous\});$$

$$Th(\{violent, \neg psychopath, \neg dangerous\}).$$

Thus, we can get a result $\neg dangerous$. Intuitively, this conclusion is too optimistic.

In our point of view, the reason is that the closed world assumption (4 and 5) are too strong. It should be replaced by

7. $-(violent \vee psychopath) \Rightarrow \neg(violent \vee psychopath)$.

We can see that $1-3, 6, 7$ has two extensions

$$Th(\{violent\}) \text{ and } Th(\{psychopath\}).$$

Here, the answer of query *dangerous* is unknown.

Generally, given a fact F, the general closed world assumption of F can be represented as (in general default logic)

$$-F \Rightarrow \neg F.$$

Given a rule base Δ such that $-F \Rightarrow \neg F \in \Delta$, it is clear that if a theory T is an extension of Δ and $T \not\models F$, then $T \models \neg F$.

3.3 Representing Conditional Defaults

A conditional rule has the following form:

$$R \Rightarrow S,$$

where R and S are rules. R is said to be the condition and S is said to be the body. Of course, this yields a representation of conditional defaults in Reiter's default logic.

Let us consider the following example about New Zealand birds from [10]. We shall show how we can represent it using conditional defaults in a natural way.

Example 5. Suppose that we have the following information:

(*) Birds normally fly. However, in New Zealand, birds normally do not fly.

[3] There are fours kinds of formalization for this example in general default logic. It depends on the way of representing the conjunctive connective in 1 and the way of representing the disjunctive connective in 6. This kind of formalization is a translation from the representation in disjunctive logic program. However, all these four kinds of formalization are fail to capture the sense of this example if the classical approach of closed world assumption is adopted.

One can represent this information in Reiter's default logic as follows:

d_1 : $bird : fly \,/\, fly$;
d_2 : $bird \wedge newzealand : \neg fly \,/\, \neg fly$.

Given the fact

1. $bird$,

the default theory $(1, \{d_1, d_2\})$ has exactly one extension $Th(\{bird, fly\})$. However, given the fact

2. $newzealand, bird$,

the default theory $(2, \{d_1, d_2\})$ has two extensions $Th(\{bird, newzealand, fly\})$ and $Th(\{bird, newzealand, \neg fly\})$.

In [10], Delgrande and Schaub formalized this example by using dynamic priority on defaults. We now show that the information (∗) can be represented by using conditional defaults in a natural way as follows:

3. $newzealand \Rightarrow (bird \,\&\, -fly \Rightarrow \neg fly)$.
4. $-newzealand \Rightarrow (bird \,\&\, -\neg fly \Rightarrow fly)$.

We can see that the rule base 1, 3, 4 still has exactly one extension $Th(\{bird, fly\})$, and the rule base 2, 3, 4 has a unique extension $Th(\{newzealand, bird, \neg fly\})$.

4 Conclusion

We have proposed a general default logic, called $\mathcal{R}$. It extends Reiter's default logic by adding rule connectives, and Ferraris's general logic program by allowing arbitrary propositional formulas to be the base in forming logic programs. We also show that this logic is flexible enough to capture several important situations in common sense reasoning.

Just as Lin and Shoham [11] showed that propositional default logic can be embedded in the logic of GK, a non-standard modal logic with two modal operators K for knowledge and A for assumption, and Lin and Zhou [12] showed that Ferraris's general logic programs can be embedded in the logic of GK, it is possible to show that our new logic $\mathcal{R}$ can also be embedded in the logic of GK. One potential benefit of doing so would be to obtain a way to check strong equivalence in $\mathcal{R}$ in classical logic. Another important task is to compare the expressive power of $\mathcal{R}$ with other non-monotonic formalisms. It is also possible to extend our logic $\mathcal{R}$ to allow facts to be first order sentences as in Reiter's default logic. We leave these to future work.

Acknowledgment

The research of the first and third authors is supported in part by Australian Research Council (ARC) Discovery Projects grant DP0666540. The second author's work was supported in part by a Croucher Senior Research Fellowship and by HK RGC under grant HKUST CERG 616806. The authors would like to thank the anonymous reviewers for their useful comments.

References

1. Reiter, R.: A logic for default reasoning. Artificial Intelligence **13** (1980) 81–132
2. Ferraris, P.: Answer sets for propositional theories. In: LPNMR. (2005) 119–131
3. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
4. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. Fifth International Conference and Symposium on Logic Programming. (1988) 1070–1080
5. Gelfond, M., Lifschitz, V., Przymusinska, H., Truszczyński, M.: Disjunctive Defaults. In Allen, J., Fikes, R., Sandewall, B., eds.: Proceedings of the second Conference on Principles of Knowledge Representation and Reasoning, Cambridge, Massachusetts, Morgan Kaufmann (1991)
6. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 369–389
7. Pearce, D.: Equilibrium logic: an extension of answer set programming for non-monotonic reasoning. In: Proccedings of WLP2000. (2000) 17
8. Gelfond, M.: Logic programming and reasoning with incomplete information. Ann. Math. Artif. Intell. **12**(1-2) (1994) 89–116
9. Chan, E.P.F.: A possible world semantics for disjunctive databases. Knowledge and Data Engineering **5**(2) (1993) 282–292
10. Delgrande, J., Schaub, T.: Compiling reasoning with and about preferences into default logic. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI–97), IJCAI Inc. Distributed by Morgan Kaufmann, San Mateo, CA. (1997) 168–174
11. Lin, F., Shoham, Y.: A logic of knowledge and justified assumptions. Artificial Intelligence **57** (1992) 271–289
12. Lin, F., Zhou, Y.: From answer set logic programming to circumscription via logic of gk. In: Proceedings of the IJCAI'07. (2007)

The $\mathcal{LP}$-$\mathcal{OD}$ System: Logic Programming Meets Outlier Detection

Fabrizio Angiulli[1], Gianluigi Greco[2], Luigi Palopoli[1], and Domenico Trimboli[1]

[1] DEIS - University of Calabria, Via P. Bucci 41C, 87030 Rende (CS), Italy
{f.angiulli,palopoli,trimboli}@deis.unical.it
[2] Dip. di Matematica - University of Calabria, Via P. Bucci 30B, 87030 Rende (CS), Italy
ggreco@mat.unical.it

Abstract. The development of effective knowledge discovery techniques has become a very active research area in recent years due to the important impact it has had in several relevant application domains. One interesting task therein is that of singling out anomalous individuals from a given population, e.g., to detect rare events in time-series analysis settings, or to identify objects whose behavior is deviant w.r.t. a codified standard set of rules. Such exceptional individuals are usually referred to as *outliers* in the literature.

In this paper, the $\mathcal{LP}$-$\mathcal{OD}$ logic programming outlier detection system is described, based on the concept of outlier formally stated in the context of knowledge-based systems in [1]. The $\mathcal{LP}$-$\mathcal{OD}$ system exploits a rewriting algorithm that transforms any outlier detection problem into an equivalent inference problem under stable model semantics, thereby making outlier computation effective and realizable on top of any stable model solver.

1 Overview of Outlier Detection

Outlier detection, i.e., identifying anomalous individuals from a given population, has important applications in bioinformatics, fraud detection, and intrusion detection, just to cite a few. In fact, several approaches have already been developed to realize outlier detection, mainly by means of data mining techniques including clustering-based and proximity-based methods as well as domain density analysis [4]. Usually, these approaches model the "normal" behavior of individuals by performing some statistical kind of computation on the given data set (various methods basically differ on the basis of the way such computation is carried out) and, then, single out those individuals whose behavior or characteristics "significantly" deviate from normal ones.

However, while looking over a set of observations to discover outliers, it often happens that there is some "qualitative" description of the domain of interest encoding, e.g., what an expected normal behavior should be. This description might be, for instance, derived by an expert and might be formalized by means of a suitable language for knowledge representation. This is precisely the approach pursued in [1], where the formalization is carried out by exploiting logic programs under stable model semantics.

This means that the background knowledge, that is, what is known in general about the world (also called in the following *rule component*), and the observations, that is, what is currently perceived about (possibly, a portion of) the world (also called in the following *observation component*), are respectively encoded in the form of a logic program P^{rls} and a set of facts P^{obs} under stable models semantics. The structure $\mathcal{P} = \langle P^{\mathrm{rls}}, P^{\mathrm{obs}}\rangle$, called *rule-observation pair*, constitutes the input for outlier detection problems.

According to the formalization of [1], the fact that an individual is an outlier has to be witnessed by a suitable associated set of observations, the outlier witness, which shows why the individual "deviates" from normality. Specifically, the choice is to assume that outlier witnesses are sets of facts that are "normally" (that is in the absence of the outlier) explained by the domain knowledge which, in turns, entails precisely the opposite of the witnesses whenever outliers are there. In this way, witnesses are meant to precisely characterize the abnormality of outliers with respect to both the theory and the other data at hand. This intuition is formalized in the following definition.

Let $\mathcal{P} = \langle P^{\mathrm{rls}}, P^{\mathrm{obs}}\rangle$ be a rule-observation pair and let $\mathcal{O} \subseteq P^{\mathrm{obs}}$ be a set of facts. Then, $\mathcal{O}$ is an *outlier* in $\mathcal{P}$ if there is a non-empty set $\mathcal{W} \subseteq P^{\mathrm{obs}}$ with $\mathcal{W} \cap \mathcal{O} = \emptyset$, called *outlier witness* for $\mathcal{O}$ in $\mathcal{P}$, such that *(1)* $P(\mathcal{P})_{\mathcal{W}} \models \neg\mathcal{W}$, and *(2)* $P(\mathcal{P})_{\mathcal{W},\mathcal{O}} \not\models \neg\mathcal{W}$, where $P(\mathcal{P}) = P^{\mathrm{rls}} \cup P^{\mathrm{obs}}$, $P(\mathcal{P})_{\mathcal{W}} = P(\mathcal{P}) \setminus \mathcal{W}$, $P(\mathcal{P})_{\mathcal{W},\mathcal{O}} = P(\mathcal{P})_{\mathcal{W}} \setminus \mathcal{O}$, and $\models$ denotes entailment under either *cautious semantics* ($\models_c$) or *brave semantics* ($\models_b$).

Example 1. Suppose that during a visit to Australia you notice a *mammal*, say *Donald*, that you classify as a *platypus* because of its graceful, yet comical appearance. However, it seems to you that *Donald* is giving birth to a young, but this is very strange given that you know that a platypus is a peculiar mammal that lays eggs. Formalizing this scenario as an outlier detection problem is simple. Indeed, observations can be encoded as the facts {`Mammal(Donald)`, `GiveBirth(Donald)`, `Platypus(Donald)`} and the additional knowledge by means of the following logical rule:

$$\texttt{Platypus(X)} \leftarrow \texttt{Mammal(X), not GiveBirth(X).}$$

It is worthwhile noting that if you had not observed that *Donald* was a platypus, you would not have inferred this conclusion, given your background knowledge and the fact `GiveBirth(Donald)`. However, if for some reason you have doubted the fact that *Donald* was giving birth to its young, then it would come as no surprise that *Donald* was a platypus. Therefore, the fact `GiveBirth(Donald)` is precisely recognized to represent an outlier, whose anomaly is indeed witnessed by the fact `Platypus(Donald)`. $\lhd$

Clearly enough, detecting outliers by exploiting a logical characterization of the domain of interest is generally more complex than it appears from the previous example. And, indeed, outlier detection problems turn out to be generally intractable under stable model semantics: if P^{rls} is stratified, then deciding for the existence of an outlier in a given rule-observation pair is NP-complete; otherwise, if P^{rls} is a general logic program, then deciding for the same problem is Σ_2^P-complete under both the brave and the cautious semantics. Furthermore, if the basic notion of outliers is constrained to single out outliers of *minimum size*, which is desirable in many application scenarios, then the complexity of computing an arbitrary outlier (if defined) is in FΔ_2^P, for stratified P^{rls}, and in FΔ_3^P (under both brave and cautious semantics), for general P^{rls}. In particular, if P^{rls} is stratified, then the above problem is FNP//OptP[O(log *n*)]-complete.

Motivated by the above complexity results witnessing the need of high expressiveness to deal with outlier detection, we have implemented a system, named $\mathcal{LP}$- $\mathcal{OD}$,

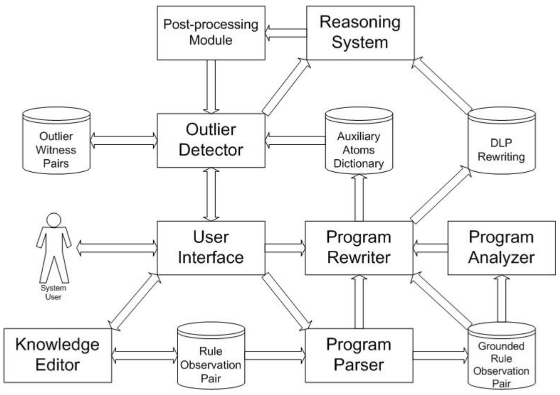


Fig. 1. Architecture of the $\mathcal{LP}$-$\mathcal{OD}$ system

supporting this mining task via logic programming. In fact, the system encodes the difficult part of the computation in suitable logic programs (eventually with negation and disjunction) under the stable model semantics. The rationale is to put outliers in one-to-one correspondence with the stable models of the associated encoding, by exploiting the ability of disjunctive logic programs to capture problems complete for the second level of the polynomial hierarchy.

Details on the architecture of $\mathcal{LP}$-$\mathcal{OD}$, and on its usage are illustrated in the remainder of the paper.

2 The $\mathcal{LP}$-$\mathcal{OD}$ System

An overview of the architecture of $\mathcal{LP}$-$\mathcal{OD}$ is reported in Figure 1.[1] The system is composed of eight main modules, whose description follows:

- **User Interface:** it is the module through which the user interacts with the system.
- **Knowledge Editor:** this is a text editor used to load, save, and edit the rule-observation pair. When writing down the program the user is allowed to use the syntax of first order logic programs.
- **Program Parser:** this module checks that the input knowledge is syntactically correct, determines all the constants in the rule observation components, computes the grounded version of the input knowledge, and stores this knowledge in an intermediate format, referred to as the *Grounded rule-observation pair*.
- **Program Analyzer:** this module determines the class of logic programs to which the input knowledge belongs to, i.e., either stratified logic programs or general logic programs, by exploiting Tarjan's algorithm [8] to determine the strongly connected components of the graph associated to the grounded pair. In fact, depending on the type of the input at hand, suited algorithms are to be selected for the rewriting (cf. Section 2.1).

[1] The actual implementation of $\mathcal{LP}$-$\mathcal{OD}$ has been carried out by using Java.

- **Program Rewriter:** the module takes in input the grounded rule-observation pair and outputs a new (disjunctive) logic program consisting in a rewriting of the original one (the *DLP Rewriting*, in the architecture of Figure 1). We use two different rewritings: one applicable to any logic program, yielding a disjunctive logic program, and one suitable for stratified logic programs, yielding an ordinary logic program. The *Program Rewriter* consults the *Program Analyzer* to decide the type of rewriting to apply. Note that the rewriting introduces auxiliary atoms, some of which are in one-to-one correspondence with atoms belonging to the observation part and are used to guess outlier and witness sets. Thus, the *Program Rewriter* stores this mapping in the *Auxiliary atoms dictionary*, since it is needed at outlier detection time.
- **Reasoning System:** the transformation carried out by the *Program Rewriter* can be evaluated on top of available stable models engine, such as GnT [5], DLV [6] and *Smodels* [7]. In the present version of $\mathcal{LP}$- $\mathcal{OD}$, this module is implemented through the DLV system.
- **Post-processing Module:** it gets the models computed by the *Reasoning System* as soon as they are generated, and extracts from each model the auxiliary atoms encoding the outlier set and the witness set computed, i.e., a solution of the outlier detection problem. Since for general logic programs, the same solution can be returned by several models, this module also filters out repeated solutions. Furthermore, if the user declares of being interested in *minimal* outliers only, the module is responsible for filtering out those which are not minimal.
- **Outlier Detector:** this module starts the execution of the *Reasoning System* module and collects the solutions returned by the *Post-processing Module*. These solutions are not immediately interpretable by the user, as they are encoded through auxiliary atoms. Thus, the *Auxiliary Atoms Dictionary* is exploited to replace auxiliary atoms with the original atoms belonging to the observation component. Solutions are presented to the user and can be stored in different formats to be further analyzed or integrated in the input knowledge.

We next discuss our rewriting strategy and the way users interact with the system.

2.1 Program Rewriter

The *Program Rewriter* module implements a transformation from any rule-observation pair $\mathcal{P} = \langle P^{\mathrm{rls}}, P^{\mathrm{obs}} \rangle$ to a suitable logic program $\mathcal{L}(\mathcal{P})$ such that its stable models are in correspondence with the outliers in $\mathcal{P}$.

Assume P^{rls} is a stratified logic program, then the rewriting $\mathcal{L}(\mathcal{P})$ outputs a logic program composed by the following groups of rules:

$i.$ two renamed copies of the original rule component, used for checking conditions (1) and (2) in the definition of outlier;
$ii.$ rules serving the purpose of guessing an outlier and its associated witness;
$iii.$ rules simulating the removal of the outlier and the witness from the observation component, which are needed for verifying whether the definition of outlier is satisfied;
$iv.$ rules actually evaluating the definition of outlier;
$v.$ constraints imposing that atoms cannot belong to an outlier and to witness at the same time.

Instead, in the case where P^{rls} is a general logic program, rules inserted in the rewriting for stratified logic programs do not suffice to check condition *(1)* of the outlier definition. Thus, we resort to disjunctive logic programs that capture the complexity class Σ_2^P (see, e.g., [3]). Specifically, a well-known characterization of minimal models for positive programs is exploited [9]: let P be a (non-disjunctive) positive propositional logic program, and let M be a model for it. Then, M is minimal if and only if there is a function ϕ assigning a natural number to each atom occurring in M, and a function r assigning a rule of P to each element p in M such that: (a) $\mathbf{body}(r(p)) \subseteq M$, (b) $\mathbf{head}(r(p)) = p$, and (c) $\phi(q) < \phi(p)$, for each $q \in \mathbf{body}(r(p))$.

Thus, besides the previous group of rules, the rewriting for general rule-observation pairs contains the following additional groups of rules:

$vi.$ rules guessing:
- a truth value assignment M to the atoms of the rule-observation pair;
- the rules $r(p)$ associated with each atom p evaluating true in M, that is rule $r(p)$ such that $\mathbf{head}(r(p)) = p$;
- the natural number $\phi(p)$ associated with each atom p in M.

$vii.$ rules checking if the assignment M is a stable model of $P(\mathcal{P})_{\mathcal{W}}$, i.e. if it satisfies all the conditions above described;

$viii.$ rules checking, for each stable models M of $P(\mathcal{P})_{\mathcal{W}}$, that $\neg\mathcal{W}$ is not contained in M (i.e. the condition (1) of the outlier definition under the cautious semantics). This is carried out with a *saturation* technique.

Minimum-size outliers. As a further functionality of the system, we have also provided support for detection problems aiming at singling out *minimum-size* outliers into a suitable logic program. To this end, we make use of *weak constraints*. Weak constraints, taking the form of rules such as $:\sim\ \mathtt{b_1}, \cdots, \mathtt{b_k}, \mathtt{not}\ \mathtt{b_{k+1}}, \cdots, \mathtt{not}\ \mathtt{b_{k+m}}$, express a set of desired conditions that may be however violated; their informal semantics is to minimize the number of violated instances. In fact, in [2] it is proved that the introduction of weak constraints allows the solution of optimization problems since each weak constraint can be regarded as an "objective function" of an optimization problem. Thus, by inserting the constraint $:\sim\ \mathtt{o_i}.$ into $\mathcal{L}(\mathcal{P})$, for each $\mathtt{o_i} \in P^{\mathrm{obs}}$, into the program $\mathcal{L}(\mathcal{P})$, we have that minimum-size outliers in $\mathcal{P}$ are in one-to-one correspondence with best stable models of the rewriting.

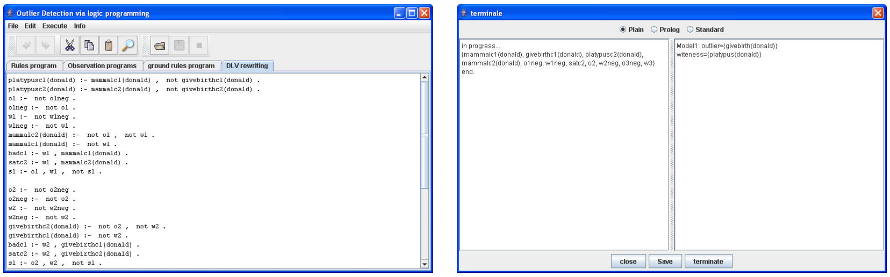
2.2 User Interface

Two snapshots for the graphical interface of $\mathcal{LP}$-$\mathcal{OD}$ are shown in Figure 2. The interface has some input panels to edit the rule and the observation components.

By entering the menu "Execute" and the submenu "Translate into DLV", the rewriting $\mathcal{L}(\mathcal{P})$ described in the preceding section is performed. In this preliminary phase, the user is in charge of deciding which kinds of outlier have to be singled out, i.e., if the focus is on all the outliers, on minimal ones, or on minimum-sized ones.

Having translated the rule-observation pair, it is possible to perform outlier detection by entering the menu "Execute" and then the submenu "Outlier detection".

The system starts executing the DLV interpreter in a separate process, and opens a terminal window. Moreover, as soon as they are computed by the DLV system, the



```
platypusc1(donald) :- mammalc1(donald) ,  not givebirthc1(donald) .
platypusc2(donald) :- mammalc2(donald) ,  not givebirthc2(donald) .
o1 :-  not o1neg .
o1neg :-  not o1 .
w1 :-  not w1neg .
w1neg :-  not w1 .
mammalc2(donald) :-  not o1 ,  not w1 .
mammalc1(donald) :-  not w1 .
badc1 :- w1 , mammalc1(donald) .
satc2 :- w1 , mammalc2(donald) .
s1 :- o1 , w1 ,  not s1 .

o2 :-  not o2neg .
o2neg :-  not o2 .
w2 :-  not w2neg .
w2neg :-  not w2 .
givebirthc2(donald) :-  not o2 ,  not w2 .
givebirthc1(donald) :-  not w2 .
badc1 :- w2 , givebirthc1(donald) .
satc2 :- w2 , givebirthc2(donald) .
s1 :- o2 , w2 ,  not s1 .
```

[✓ Plain] Prolog Standard

```
in progress...
{mammalc1(donald), givebirthc1(donald), platypusc2(donald),
mammalc2(donald), o1neg, w1neg, satc2, o2, w2neg, o3neg, w3}
end.
```

```
Model1: outlier={givebirth(donald)}
witeness={platypus(donald)}
```

Fig. 2. User Interface: Rewriting of the input rule-observation program (left), and the outlier detection phase (right)

system displays, in the left part of the window, the stable models of the program $\mathcal{L}(\mathcal{P})$, and, in the right part of the window, the outliers together with the associated witnesses. Note that the possibility of singling out minimal outliers only is implemented as a post-processing step. In this case, $\mathcal{LP}$- $\mathcal{OD}$ does not display the outliers as soon as they are available, but firstly collects them and filters out non-minimal ones.

The output of the system can eventually be saved in various formats, to be further analyzed or integrated in the input knowledge.

References

1. F. Angiulli, G. Greco, and L. Palopoli. Outlier detection by logic programming. *ACM Transactions on Computational Logic*, to appear. Available for download at *http://www.acm.org/pubs/tocl/accepted/253angiulli.pdf.*
2. F. Buccafurri, N. Leone, and P. Rullo. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.
3. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Survey*, 33(3):374–425, 2001.
4. J. Han and M. Kamber. *Data Mining, Concepts and Technique*. Morgan Kaufmann, San Francisco, 2001.
5. T. Janhunen, I. Niemelä, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. In *Proc. of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 411–419, Breckenridge, Colorado, USA, 2000.
6. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, To appear.
7. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 420–429, Berlin, 1997.
8. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
9. R. Ben-Eliyahu Zohary and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87, 1994.

clasp: A Conflict-Driven Answer Set Solver

Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub*

Institut für Informatik, Universität Potsdam,
August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We describe the conflict-driven answer set solver *clasp*, which is based on concepts from constraint processing (CSP) and satisfiability checking (SAT). We detail its system architecture and major features, and provide a systematic empirical evaluation of its features.

1 Introduction

Our new system *clasp* [1] combines the high-level modeling capacities of Answer Set Programming (ASP; [2]) with state-of-the-art techniques from the area of Boolean constraint solving. Unlike existing ASP solvers, *clasp* is originally designed and optimized for conflict-driven ASP solving [3,4], centered around the concept of a *nogood* from the area of constraint processing (CSP). Rather than applying a SAT(isfiability checking) solver to a CNF conversion, *clasp* directly incorporates suitable data structures, particularly fitting backjumping and learning. This includes dedicated treatment of binary and ternary nogoods [5], and watched literals for unit propagation on "long" nogoods [6]. Unlike *smodels*$_{cc}$ [7], which builds a material implication graph for keeping track of the multitude of inference rules found in ASP solving, *clasp* uses the more economical approach of SAT solvers: For a derived literal, it only stores a pointer to the responsible constraint. Despite its optimized data structures, the implementation of *clasp* provides an elevated degree of abstraction for handling different types of (static and dynamic) nogoods. This paves the way for the future support of language extensions, e.g., aggregates. Different from *smodels* [8] and *dlv* [9], unfounded set detection within *clasp* does not determine greatest unfounded sets. Rather, an identified unfounded atom is immediately falsified, before checking for any further unfounded sets.

We focus on *clasp*'s primary operation mode, viz., conflict-driven nogood learning; its second operation mode runs (systematic) backtracking without learning. Beyond backjumping and learning, *clasp* features a number of related techniques, typically found in SAT solvers based on *Conflict-Driven Clause Learning* (CDCL; [10]). *clasp* incorporates restarts, deletion of recorded conflict and loop nogoods, and decision heuristics favoring literals from conflict nogoods. All these features are configurable via command line options and subject to our experiments. The two major contributions of this paper consist, first, in a detailed description of the system architecture (in Section 2) and, second, in a systematic empirical evaluation of some selected run-time features (in Section 3). Many of these features are based on experiences made in the area of SAT; hence it is interesting to see how their variation affects solving ASP problems.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 System Architecture

The system architecture of *clasp* can be divided into three major components by following the underlying data flow (cf. Figure 1): *clasp* reads ground logic programs in *lparse* format [11], possibly including choice rules, cardinality and weight constraints. The latter constructs are compiled away during *parsing*. The resulting normal rules are then taken by the *program builder* to generate *nogoods* (capturing Clark's completion) and to create an initial positive atom-body-dependency graph (containing only distinct bodies). While all vertices of this graph are associated with assignable variables in the *static data*, only the non-trivial strongly connected components of the positive atom-body-dependency graph are kept and used to initialize the *unfounded set checker*. Note that *clasp* uses *hybrid assignments*, treating atoms and bodies equitably as assignable objects.

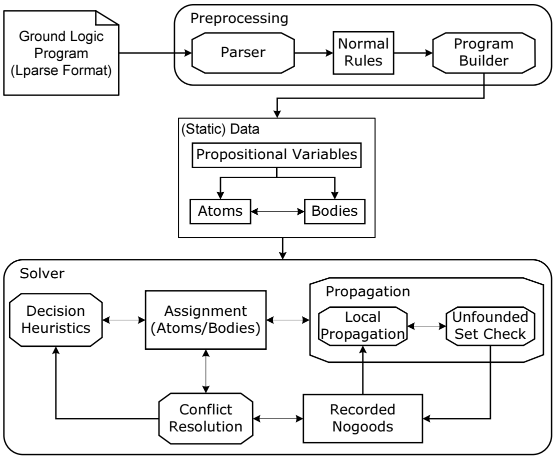


Fig. 1. The system architecture of *clasp*

The elementary data type used in the *solver* is that of a *Boolean constraint* (and thus not restricted to sets of literals). The solver distinguishes static nogoods (see above) that are excluded from nogood deletion and *recorded* nogoods (stemming from conflicts or loops) accumulated during the search. While the former are part of the static data, the latter are kept in a separate database. Also, a learnt nogood maintains an activity counter that is used as a parameter for nogood deletion (see below). Different data structures are used for binary, ternary, and longer nogoods (accounting for the large number of short nogoods capturing Clark's completion). This is complemented by maintaining two *watch lists* [5,6] for each variable, storing all longer nogoods that need to be updated if the variable becomes true or false, respectively.

Variable assignments are either done by *propagation* or via a decision heuristics. *clasp*'s *local* propagation amounts to applying the well-known *unit clause rule* to nogoods (cf. [3]). A variable assigned by local propagation has a pointer to the (unit) nogood it was derived from; this includes unfounded atoms derived from loop nogoods (see [3] for details). During propagation, binary nogoods are preferred over ternary ones, which are preferred over longer nogoods. Also, our propagation procedure is distinct in giving a clear preference to local propagation over *unfounded set* computations. Once an unfounded set U is determined, only a single atom from U is taken to generate a loop nogood that is added to the recorded nogoods. Then, local propagation resumes until a fixed point is reached. This is repeated until there are no non-false atoms left in U. Afterwards, either another unfounded set is found or propagation terminates.

Unfounded set detection within *clasp* combines *source pointers* [12] with the unfounded set computation algorithm in [13]. Notably, it aims at small and "loop-encompassing" rather than greatest unfounded sets, as determined by *smodels* and *dlv*.

Whenever propagation encounters a conflict, *clasp*'s *conflict resolution* is engaged. As described in [3], conflict resolution determines a conflict nogood (that is recorded) and a decision level to jump back to. Backjumping and nogood recording work similar to CDCL with *First-UIP scheme* [10]. The corresponding algorithms are detailed in [3]. For enumerating answer sets, *clasp* uses a novel bounded backjumping approach that is elaborated upon in [4]. Given that *clasp*'s learning and backjumping strategy have already been theoretically as well as experimentally elaborated upon elsewhere [3,4], let us concentrate in what follows on heuristic aspects and in particular investigate how established CSP and SAT strategies apply in the context of ASP. This also provides an overview of the variety of different strategies supported by *clasp*.

clasp's *decision heuristics* depends on whether learning is in effect or not. Without learning, *clasp* relies on *look-ahead* strategies (that extend unit propagation by *failed-literal detection* [14]). When learning, *clasp* uses *look-back* strategies derived from corresponding CDCL-based approaches in SAT, viz., *VSIDS* [6], *BerkMin* [15], and *VMTF* [5]. All of them are conflict-oriented and so primarily influenced by conflict resolution. The heuristic values mainly need to be updated when a new nogood is recorded. Notably, *clasp* leaves it to the user whether this includes loop nogoods or not.

clasp distinguishes two types of *restart policies*. The first starts with an initial number of conflicts after which *clasp* restarts; this threshold is then increased by a factor after each restart. The second policy goes back to Luby et al. [16] and is based on a sequence of numbers of conflicts (e.g., $32\ 32\ 64\ 32\ 32\ 64\ 128\ 32\ \ldots$ for unit 32) after each of which it restarts. The bounded restart strategy used when enumerating answer sets is described in [4]. Moreover, *clasp* allows for a limited number of *initial randomized runs*, typically with a small restart threshold, in the hope to discover putatively interesting nogoods before actual search starts.

clasp's *nogood deletion* strategy borrows ideas from *minisat* [17] and *berkmin* [15]. It associates an *activity* with each dynamic nogood and limits the number of recorded nogoods by removing nogoods whenever a threshold is reached. The limit is initialized with the size of the input program and increased by a factor every restart. Note that nogood deletion applies to both conflict as well as loop nogoods.

3 Experiments

We conducted experiments on a variety of problem classes. Our comparison includes *clasp* (RC4) in various modes: the normal mode (N) and variants of it changing either the heuristics (H), initial randomized runs (I), restarts (R), or nogood deletion (D):

N The standard mode of *clasp* (RC4) defaults to the following command line options:

`--heuristic=berkmin` indicates that choices (on atoms and bodies) are done according to an adaption of the *BerkMin* heuristics [15].

`--lookback-loops=no` indicates that the heuristics ignores loop nogoods.

`--restarts=simple(100,1.5)` makes *clasp* restart every 100×1.5^k conflicts for $k \geq 0$ (i.e., after $100\ 150\ 225\ 337\ 506\ \ldots$ conflicts).

`--deletion=3,1.1` fixes the size and growth factor of the dynamic nogood database. Initially, *clasp* allows for recording $(|atom(\Pi) \cup body(\Pi)|/3)$ nogoods before nogood deletion is invoked. The size is increased with each restart by the factor `1.1`, given as second parameter.

`--loops=common` uses a fixed set of bodies when composing the loop nogoods of an unfounded set (alternatives: `distinct` and `shared`; cf. [1]).

H1 **`--heuristic=berkmine`** modifies the initialization of `berkmin` by counting watched literals rather than taking the original randomized approach.

H2 **`--heuristic=vmtfe --lookback-loops=yes`** uses an extended adaption of the *VMTF* heuristics [5] and furthermore takes loop nogoods into account. (Actually, the other heuristics could use loop nogoods as well. But only with VMTF, they showed an improvement, while hampering the other heuristics.)

H3 **`--heuristic=vsids`** uses an adaption of the *VSIDS* heuristics [6].

I **`--randomize=50,20`** makes *clasp* perform `50` initial runs with a random choice policy before actual search commences; each run is stopped after `20` conflicts.

R1 **`--restarts=luby(64)`** uses Luby et al.'s restart strategy [16] with base `64`.

R2 **`--restarts=simple(16000,1)`** is similar to *siege*'s fixed-interval restart strategy [5], cutting of every `16000` conflicts.

R3 **`--restarts=simple(700,1)`** is similar to *chaff*'s fixed-interval restart strategy [18], cutting of every `700` conflicts.

R4 **`--no-restarts`** inhibits restarting.

D1 **`--deletion=25,1.1`** keeps the dynamic nogood database rather small.

D2 **`--no-deletion`** turns off deletion of dynamic nogoods.

For comparison, we include *smodels* with default settings (S; V2.32) and with its restart option (Sr). We also incorporate *smodels*$_{cc}$ (Scc; V1.08) with option "nolookahead", as recommended by the developers, and *cmodels* (C; V3.65) using *zchaff* (2004.11.15).

All experiments were run on a 800MHz PC on Linux. We report the average time (in seconds) on ten different *shuffles* of an input program. Each run was restricted to 300s time and 512MB RAM. Times exclude parsing, done off-line with *lparse* (V1.0.17). A timeout in *all* 10 runs is indicated by "•"; otherwise, it is taken to be 300s within statistics. The benchmark instances as well as extended results are available at [1,19]. The instances in Table 1 are random programs (1-10); computing bounded spanning trees (11-15), weighted spanning trees (16-20), and Hamiltonian cycles (21-25); game solving for Sokoban (26-35) and Gryzzles (36-40); from bounded model checking (41-52); Social Golfers scheduling (53-57); and machine code superoptimization (58-62). The problems have a variety of different characteristic properties, such as SAT vs UNSAT, random vs structured, tight vs non-tight, etc. Our aim is to give an overview of *clasp*'s performance on a broad palette of problems, from which instances are picked representatively with the only requirement that they are selective.

For brevity, we here only provide a summary of the benchmark results shown in Table 1. For each solver, the last 7 rows show statistics over all runs ($62 \times 10 = 620$ runs per solver). Let us focus on the number of "Timeouts" indicating robustness. (Recall that we shuffled the inputs in order to compensate for luckiness.) It turns out that all different features of *clasp*, that is, heuristics, restarts, and clause deletion, have an impact. Among heuristics, the BerkMin variants N and H1 turned out to be more reliable than VMTF (H2) and VSIDS (H3). Although VMTF is often best, it also leads to

Table 1. Experiments computing *one* answer set

No	Instance	N	H1	H2	H3	I	R1	R2	R3	R4	D1	D2	S	Sr	Scc	C
1	lp.200.00900.9	25.3	24.3	94.2	56.3	25.2	64.4	22.4	55.7	16.1	21	50.6	291.6	291.8	94.6	60.3
2	lp.200.00900.19	25.2	24.6	65.2	44.6	26.7	86.5	16.4	72.9	12.7	23.2	56.6	229.1	231.5	86.9	72.6
3	lp.200.00900.23	20.4	19.8	64	41	22	67.9	16.1	62.1	12.1	21.5	50.5	254.8	244.6	77.2	72.6
4	lp.200.01000.2	21.3	25.3	75.5	47.2	26	60.6	18.9	56.9	13.5	21.7	47	247.1	245.5	70.2	49.8
5	lp.200.01000.22	24.2	21.4	77.1	46.2	25.4	62.1	18.4	55.7	13.3	23.2	43.9	234	232.4	68.8	57.9
6	b5	43.4	45.5	67.6	56.5	45.6	●	158	●	19.4	35.8	216.6	108.5	103.4	●	279.5
7	b9	51	49	72.6	60.6	48.1	●	254.6	●	24.8	37.1	257.4	127.4	131	●	●
8	b10	54.7	52.4	87.8	61.6	55.6	●	243.3	●	25.1	43.4	276.4	165.7	164.9	●	●
9	b17	29.7	25.9	49.7	58	24.6	153.1	50.3	174.1	10.8	21.1	154.6	80.8	76.3	233.1	182.8
10	b26	32.4	36.7	26.7	60.2	55.9	114.7	26.5	151.2	27	17.4	103.8	81.3	177.9	269.9	172.8
11	104_rand_45_250_1727040059_0	48.8	45.9	24.1	52.8	40.6	52.6	45.8	41.7	50.9	48.9	48.9	●	●	296.1	152.3
12	104_rand_45_250_1727040917_0	56.3	51.7	24.9	57.6	39.8	60.7	42.1	42.3	42.2	56.3	56.3	●	●	297.6	184.2
13	104_rand_45_250_1727042043_0	61.7	65	27.5	108.5	40.2	78.8	41.6	56.2	41.6	61.7	61.7	●	●	287.5	102.3
14	104_rand_45_250_1727044175_0	47.4	43.6	24.8	47.8	38.7	53.6	42.3	42.7	42.2	47.5	47.4	●	●	298.7	163.3
15	104_rand_45_250_1727068226_0	48.8	49.1	26.5	57.8	40.7	57	41.2	41.9	41.2	48.8	48.8	●	●	296.9	122.1
16	207_rand_35_138_2077101081_0	12.8	16.3	8.2	18.9	12.9	13.4	11.7	10.9	11.7	12.8	12.8	191.8	225.7	85.8	11.6
17	207_rand_35_138_2077159055_0	10.2	10.6	7.6	13.3	12.8	10.2	10	10.1	10	10.2	10.2	290	296.5	76.7	10.5
18	209_rand_45_138_1119566817_0	23.3	23.1	15.4	31.2	23.4	24.1	25.4	23.6	30.3	23.3	23.3	●	●	215.9	21.4
19	209_rand_45_138_1119569108_0	26.8	27.9	21.4	44.1	24.5	27.7	25.4	25.1	25.5	26.8	26.8	●	●	222.3	19.4
20	209_rand_45_138_1119571853_0	24	25	15.4	37.4	22.5	24.2	22.7	21	22.7	24	24	●	●	202.7	30.4
21	rand_200_1800_1154991214_4	6.6	19.9	28	9.3	10	7.1	24.9	5.2	68.9	6.6	6.6	179.9	71.8	18.7	106.7
22	rand_200_1800_1154991214_7	5.1	18.6	4.1	5.4	9.4	5	27.1	5.4	95.1	5.1	5.1	219.5	123.7	16.8	127.2
23	rand_200_1800_1154991214_9	6	14.9	5.7	5.6	7.9	9.5	60.2	7.9	182	6	6	218.2	57.5	28	120.6
24	rand_200_1800_1154991214_11	5.6	19.7	8	6.9	9	6.5	40	6	93.3	5.6	5.6	●	206.1	17.3	91.9
25	rand_200_1800_1154991214_14	5.9	18.2	6.4	4.9	9	8.3	21.4	5.6	53.1	5.8	5.9	154.6	82.8	21.3	96.5
26	yoshio.2.n16.len15	24	26.5	32.9	56.7	19.3	26.4	26.1	29.4	26.1	32.3	26.1	●	●	63	78.6
27	yoshio.2.n16.len16	24	37.7	48.2	50.6	16.8	34.3	21.9	26.9	19.6	31.2	33.2	●	●	62.2	92.5
28	yoshio.11.n15.len14	4.8	4.9	7.3	28.1	5	4.6	4.7	4.8	4.7	4.9	4.8	●	●	12.8	18.7
29	yoshio.11.n15.len15	4.5	6.2	8.6	31.2	6.3	5.2	5.8	6.2	5.9	5.3	4.5	●	●	12.8	36.4
30	yoshio.36.n14.len13	13.8	13.7	15.7	38.9	9.1	10.4	12.1	12	12	15	43.8	●	●	18.2	22.7
31	yoshio.36.n14.len14	12.3	10.4	12.5	29.2	6.7	9.5	8.9	10.2	8.9	13.3	13.7	●	258.4	21.7	27.7
32	yoshio.46.n13.len12	13.5	15	18.2	24.5	12.9	14.6	13.6	12.7	13.6	17.6	13.4	●	●	34.3	36.3
33	yoshio.46.n13.len13	14.5	20.7	17.7	15.1	11.1	11	15.3	12.9	15.3	11.8	14.5	●	●	48.3	30.7
34	yoshio.52.n12.len11	10.8	13.2	19	23	11.5	12.6	11.3	11.8	11.3	12.7	10.8	180.4	181.5	29.8	32.4
35	yoshio.52.n12.len12	10.3	10.7	13.6	20.5	11.9	9.5	8.9	8.3	8.9	11.6	11.2	●	●	24.4	40.3
36	gryzzles.0	38.2	24.1	44.2	37.5	22.7	12.4	30.4	2.8	210.3	42.3	90.8	●	181	117.1	26.1
37	gryzzles.3	0.3	0.3	0.3	0.3	0.7	0.3	1.2	0.3	1.6	0.2	0.3	21.7	0.5	0.6	1
38	gryzzles.7	0.3	0.5	0.5	0.4	0.8	0.3	3.7	0.4	81.3	0.5	0.3	180.5	3	2	1.1
39	gryzzles.18	0.6	0.7	0.7	0.6	1	0.6	6.3	0.6	34.8	1	0.6	4.8	1.8	2	2.5
40	gryzzles.47	1.3	1.4	1.9	1.7	1.9	1.5	7.6	1.5	116	1.4	1.1	●	18.7	8.5	11.3
41	dp_10.formula1-i-O2-b12	6.9	9.7	14.1	47.5	10.5	11.8	18.3	6.4	15.9	8.7	6.9	165.6	273.3	10.8	38
42	dp_12.formula1-i-O2-b14	44.8	73.6	102.6	200.1	43.6	70.6	77.9	88.1	234.8	38.4	64.6	●	●	75.4	150.5
43	dp_12.formula1-s-O2-b10	3.8	5.6	9.8	10.8	5.4	2.7	3.6	4.3	3.6	3.2	2.9	●	●	6.4	16.6
44	dp_10.fsa-D-i-O2-b10	2.4	0.4	0.3	10.4	4.1	1.5	6.8	0.9	39.7	0.8	3.5	294	6.5	33.9	1.2
45	dp_12.fsa-D-i-O2-b9	●	●	●	●	●	●	●	●	287.6	●	●	●	●	●	●
46	elevator_2-D-i-O2-b12	10.1	8.7	7.8	26.7	10	11.7	8.3	10.2	8.4	10.1	10.1	4.4	22.4	14.4	7.4
47	elevator_4-D-s-O2-b10	3.4	5.2	3.2	7.8	4.5	3.3	4.4	4	4.4	3.5	3.4	97	7.4	21	5.1
48	key_2-D-i-O2-b29	25.2	33.5	39.5	140.2	22.7	31	36.3	30.4	35.1	25.6	25.2	●	●	83.6	50.2
49	key_2-D-s-O2-b29	33.1	32.7	39	91	28	30.6	30.1	30.7	28.7	29.3	35.2	●	●	65	40.8
50	mmgt_3.fsa-D-i-O2-b10	9.6	15.9	23.8	35	6.5	9	10.6	8.7	10.6	8.6	9.6	40.8	16.9	13	9.5
51	mmgt_4.fsa-D-i-O2-b12	180.2	95.4	120.8	217	126.8	76.7	139.5	96	144.6	115.9	180.6	●	274.9	28.3	36.4
52	q_1.fsa-D-i-O2-b17	221.7	187	278.9	293.1	132.3	274.9	176.1	290	161.6	87.8	292.3	●	●	201.1	281.6
53	csp010-SocialGolfer_w3_g3_s6	35	35.7	50.3	103.3	57.9	50.7	34.8	33.4	38.9	84.3	34	●	●	57.2	23.8
54	csp010-SocialGolfer_w4_g3_s6	34.9	41.2	61.7	102.7	60.9	44.3	35.9	38.7	44.2	76	38.9	●	●	64.2	31
55	csp010-SocialGolfer_w6_g3_s4	14.8	24.2	9.5	277.9	97.1	14.8	9.2	20.6	9.3	17.2	16.9	189.2	187.7	40	38.5
56	csp010-SocialGolfer_w6_g3_s5	40.9	60.5	13.7	234.4	141.6	59.3	52.5	49.4	42.9	70.5	55.7	●	●	62.3	40
57	csp010-SocialGolfer_w7_g3_s6	40.1	79.8	18.4	284.5	52.1	91.8	38.7	38.4	45.3	80.6	45.4	●	●	75.1	35.5
58	sequence2-ss2	14.9	15.7	14.7	17.3	14.4	16.1	14.2	14.2	14.2	14.9	14.9	83.5	136.8	115.5	38.2
59	sequence3-ss2	27	27	27.2	27.1	27	26.9	26.9	26.9	26.9	27	26.9	22.8	22.8	24.6	12.8
60	sequence3-ss3	170.7	177	128.5	175.5	128.9	209.8	183.9	141.9	185.7	168.4	170.5	●	●	●	289.5
61	sequence4-ss2	69.2	70.9	80.8	75.8	77.2	73.5	70.1	69.6	70	69.1	69.1	235.6	232.1	225.1	294.8
62	sequence4-ss4	292.5	297.7	293.5	●	●	●	●	●	●	292.7	292.6	●	●	●	●
	Timeouts	23	25	31	66	26	65	33	67	57	22	40	396	342	110	79
	Best	51	56	147	41	64	52	47	64	120	60	42	26	17	21	67
	Worst	25	25	39	77	28	66	34	67	63	22	40	445	384	115	94
	Better	461	379	360	238	431	333	408	378	427	423	386	48	46	103	163
	Worse	159	241	260	382	189	287	212	242	193	197	234	572	574	517	457
	Average	39.9	41.3	45.3	70.5	40.1	61.5	49.4	58.4	53.4	38.5	58.3	233.8	212.7	109	87.2
	Euclidian Distance	341.9	312.2	387.4	685.3	312.6	651.9	476.2	645.9	509	287.2	597.9	1860.5	1753.5	1048.5	825.9

more timeouts. As one might expect, the variant without restarts (R4) is less robust than the restarting variants N and R2. This is also confirmed by *smodels*, where the restart option (Sr) significantly reduces the number of timeouts in comparison to the default setting (S). On the *clasp* variants R1 and R3, we however see that very short restart intervals also degrade performance. Except for *smodels*, all solvers shown in Table 1 use learning and turn out to be more robust than *smodels*. But we also observe that keeping all recorded nogoods, as done by *clasp* variant D2, degrades performance. In contrast, making the dynamic nogood database smaller (D1) was useful on the benchmarked instances. Finally, initial randomized runs (I) tend to slightly increase the solving time when compared to the fastest non-randomized *clasp* variants. However, if the deterministic variants of *clasp* fail, then randomization might be useful. The last 6 rows in Table 1 count how often a solver was "Best", "Worst", and "Better" or "Worse" than the median solving time on a (shuffled) instance, provide its "Average" time over all runs, and finally, the "Euclidian Distance" to the virtual optimum solver (best on all instances) in a 62–dimensional space. Benchmark results for combinations of different options are beyond the scope of this paper. However, the fine-tuning of *clasp* is an ongoing process.

References

1. (`http://www.cs.uni-potsdam.de/clasp`)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. [20] 386–392.
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. This volume.
5. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. MSc thesis, Simon Fraser University (2004)
6. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC'01. (2001) 530–535
7. Ward, J., Schlipf, J.: Answer set programming with clause learning. In: Proc. LPNMR'04. Springer (2004) 302–313
8. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
9. 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) (2006) 499–562
10. Mitchell, D.: A SAT solver primer. Bulletin of the EATCS **85** (2005) 112–133
11. Syrjänen, T.: Lparse 1.0 user's manual. (http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz)
12. Simons, P.: Extending and Implementing the Stable Model Semantics. Dissertation, Helsinki University of Technology (2000)
13. Anger, C., Gebser, M., Schaub, T.: Approaching the core of unfounded sets. In: Proc. NMR'06. Clausthal University of Technology (2006) 58–66
14. Freeman, J.: Improvements to propositional satisfiability search algorithms. PhD thesis, University of Pennsylvania (1995)
15. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: Proc. DATE'02. (2002) 142–149
16. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4) (1993) 173–180
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT'03. (2003) 502–518
18. Huang, J.: The effect of restarts on the efficiency of clause learning. [20] 2318–2323.
19. (`http://asparagus.cs.uni-potsdam.de`)
20. Proc. IJCAI'07, AAAI Press/The MIT Press (2007).

GrinGo: A New Grounder for Answer Set Programming

Martin Gebser, Torsten Schaub[⋆], and Sven Thiele

Universität Potsdam, Institut für Informatik,
August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We describe a new grounder system for logic programs under answer set semantics, called *GrinGo*. Our approach combines and extends techniques from the two primary grounding approaches of *lparse* and *dlv*. A major emphasis lies on an extensible design that allows for an easy incorporation of new language features in an efficient system environment.

1 Motivation, Features, and System Architecture

A major advantage of Answer Set Programming (ASP; [1]) is its rich modeling language. Paired with high-performance solver technology, it has made ASP a popular tool for declarative problem solving. As a consequence, all ASP solvers rely on sophisticated preprocessing techniques for dealing with the rich input language. The primary purpose of preprocessing is to accomplish an effective variable substitution in the input program. This is why these preprocessors are often referred to as *grounders*.

Although there is meanwhile quite a variety of ASP solvers, there are merely two major grounders, namely *lparse* [2] and *dlv*'s grounding component [3]. We enrich this underrepresented area and present a new grounder, called *GrinGo*, that combines and extends techniques from both aforementioned systems. A salient design principle of *GrinGo* is its *extensibility* that aims at facilitating the incorporation of additional language constructs. In more detail, *GrinGo* combines the following features:

- its input language features normal logic program rules, cardinality constraints, and further *lparse* constructs,
- its parser is implemented by appeal to *flex* and *bison++* paving an easy way for language extensions,
- it offers the new class of λ*-restricted programs* (detailed in Section 2) that extends *lparse*'s ω-restricted programs [4],
- its instantiation procedure uses back-jumping and improves on the technique used in *dlv*'s grounder [5] by introducing *binder-splitting* (see Section 3),
- its primary output language currently is textual, as with *dlv*'s grounding component; *lparse* format will be supported soon.

We identify four phases in the grounding process and base the core components of *GrinGo* upon them. The primary *GrinGo* architecture is shown in Figure 1. First, the *parser* checks the syntactical correctness of an input program and creates an internal representation of it. Subsequently, the *checker* verifies that the input program is λ-restricted, so that the existence of a finite equivalent ground instantiation is guaranteed.

[⋆] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

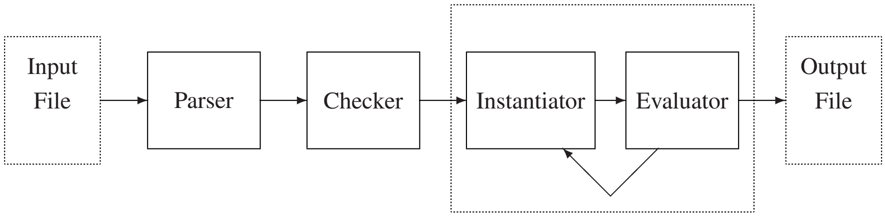


Fig. 1. The *GrinGo* architecture

From this analysis, the checker also schedules the grounding tasks. The *instantiator* computes ground instances of rules as scheduled. Note that the grounding procedure of the instantiator is based on an enhanced version of *dlv*'s back-jumping algorithm. The generated ground rules are then passed to the *evaluator* which identifies newly derived ground instances of predicates. The evaluator also checks for potential program simplifications and finally decides whether a ground rule is output or not.

2 λ-Restricted Programs

For simplicity, we confine ourselves to normal logic programs with function symbols and first-order variables. Let $\mathcal{F}$ and $\mathcal{V}$ be disjoint sets of *function* and *variable* symbols, respectively. As usual, a *term* is defined inductively: Each variable $v \in \mathcal{V}$ is a term, and $f(t_1, \ldots, t_k)$ is a term if $f/k \in \mathcal{F}$ and $t_1, \ldots, t_k$ are terms. Note that the arity k of f/k can be zero. For a term t, we let $V(t)$ denote the set of all variables occurring in t.

A *rule* r over $\mathcal{F}$ and $\mathcal{V}$ has the form

$$\begin{aligned} p_0(t_{1_0}, \ldots, t_{k_0}) \leftarrow p_1(t_{1_1}, \ldots, t_{k_1}), \ldots, p_m(t_{1_m}, \ldots, t_{k_m}), \\ \mathit{not}\ p_{m+1}(t_{1_{m+1}}, \ldots, t_{k_{m+1}}), \ldots, \mathit{not}\ p_n(t_{1_n}, \ldots, t_{k_n}) \ , \end{aligned} \tag{1}$$

where $p_0/k_0, \ldots, p_n/k_n$ are *predicate* symbols, $p_0(t_{1_0}, \ldots, t_{k_0}), \ldots, p_n(t_{1_n}, \ldots, t_{k_n})$ are *atoms*, and t_{j_i} is a term for $0 \leq i \leq n$ and $1 \leq j \leq k_i$. For an atom $p(t_1, \ldots, t_k)$, we let $P(p(t_1, \ldots, t_k)) = p/k$ be its predicate, and $V(p(t_1, \ldots, t_k)) = (V(t_1) \cup \cdots \cup V(t_k))$ be the set of its variables. For r as in (1), we define the head as $H(r) = p_0(t_{1_0}, \ldots, t_{k_0})$. The sets of atoms, positive body atoms, predicates, and variables, respectively, in r are denoted by $A(r) = \{p_i(t_{1_i}, \ldots, t_{k_i}) \mid 0 \leq i \leq n\}$, $B(r) = \{p_i(t_{1_i}, \ldots, t_{k_i}) \mid 1 \leq i \leq m\}$, $P(r) = \{P(a) \mid a \in A(r)\}$, and $V(r) = \bigcup_{a \in A(r)} V(a)$. For a rule r and a variable $v \in \mathcal{V}$, we let $B(v, r) = \{P(a) \mid a \in B(r), v \in V(a)\}$ be the set of *binders* for v in r. Note that the set of binders is empty if v does not occur in any positive body atom of r.

A *normal logic program* Π over $\mathcal{F}$ and $\mathcal{V}$ is a finite set of rules over $\mathcal{F}$ and $\mathcal{V}$. We let $P(\Pi) = \bigcup_{r \in \Pi} P(r)$ be the set of predicates in Π. For a predicate $p/k \in P(\Pi)$, we let $R(p/k) = \{r \in \Pi \mid P(H(r)) = p/k\}$ be the set of *defining* rules for p/k in Π. Program Π is *ground* if $V(r) = \emptyset$ for all $r \in \Pi$. The semantics of ground programs is given by their *answer sets* [1]. We denote by $AS(\Pi)$ the set of all answer sets of Π.

We now introduce the notion of λ-*restrictedness* for normal logic programs.

Definition 1. *A normal logic program Π over $\mathcal{F}$ and $\mathcal{V}$ is λ-restricted if there is a level mapping $\lambda : P(\Pi) \rightarrow \mathbb{N}$ such that, for every predicate $p/k \in P(\Pi)$, we have*

$$\overbrace{max\{\ max\{\overbrace{min\{\underbrace{\lambda(p'/k') \mid p'/k' \in B(v,r)}\} \mid v \in V(r)}\} \mid r \in R(p/k)\ \}} < \lambda(p/k)\,.$$

(We added over- and underbraces for the sake of easier readability.) Intuitively, λ-restrictedness means that all variables in rules defining p/k are bound by predicates p'/k' such that $\lambda(p'/k') < \lambda(p/k)$. If this is the case, then the domains of rules in $R(p/k)$, i.e., their feasible ground instances, are completely determined by predicates from lower levels than the one of p/k.

We now provide some properties of λ-restricted programs and compare them with the program classes handled by *lparse* and *dlv*. Recall that *lparse* deals with ω-restricted programs [2], while programs have to be *safe* with *dlv* [3].

Theorem 1. *If a normal logic program Π is ω-restricted, then Π is λ-restricted.*

Note that the converse of Theorem 1 does not hold. To see this, observe that the rules

$$a(1) \qquad\qquad b(X) \leftarrow a(X), c(X) \qquad\qquad \begin{array}{l} c(X) \leftarrow a(X) \\ c(X) \leftarrow b(X) \end{array}$$

constitute a λ-restricted program, but not an ω-restricted one. The cyclic definition of $b/1$ and $c/1$ denies both predicates the status of a domain predicate (cf. [2]). This deprives rule $c(X) \leftarrow b(X)$ from being ω-restricted. Unlike this, the λ-restrictedness of the above program is witnessed by the level mapping $\lambda = \{a \mapsto 0, b \mapsto 1, c \mapsto 2\}$.

On the one hand, the class of λ-restricted programs is more general than that of ω-restricted ones. On the other hand, there are safe programs (that is, all variables occurring in a rule are bound by positive body atoms) that are not λ-restricted. In contrast to safe programs, however, every λ-restricted program has a finite equivalent ground instantiation, even in the presence of functions with non-zero arity.

Theorem 2. *For every λ-restricted normal logic program Π, there is a finite ground program Π' such that $AS(\Pi') = AS(\Pi)$.*

To see the difference between safe and λ-restricted programs, consider the following program, which is safe, but not λ-restricted:

$$a(1) \qquad\qquad a(Y) \leftarrow a(X), Y = X + 1$$

This program has no finite equivalent ground instantiation, which is tolerated by the safeness criterion. To obtain a finite ground instantiation, *dlv* insists on the definition of a maximum integer value `maxint` (in the presence of arithmetic operations, like $+$) for restricting the possible constants to a finite number.

3 Back-Jumping Enhanced by Binder-Splitting

GrinGo's grounding procedure is based on *dlv*'s back-jumping algorithm [5,6]. To avoid the generation of redundant rules, this algorithm distinguishes atoms binding *relevant*

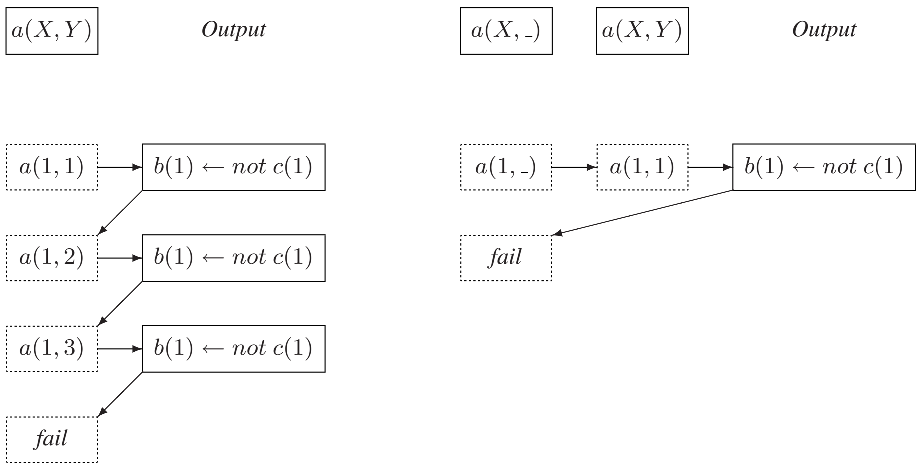


Fig. 2. Back-jumping in (a) *dlv* and (b) *GrinGo*

and *irrelevant* variables. A variable is relevant in a rule, if it occurs in a literal over an unsolved predicate; and a predicate is *solved*, if the truth value of each of its ground instances is known. *dlv*'s back-jumping algorithm avoids revisiting binders of irrelevant variables, whenever different substitutions for these variables result in rule instantiations that only differ in solved literals.

The back-jumping algorithm of *GrinGo* goes a step further and distinguishes between relevant and irrelevant variables within the same binder. The instantiator internally splits such binders into two new binders, the first one binding the relevant variables and the second one binding the irrelevant ones. While the original *dlv* algorithm necessitates that a binder is revisited whenever it contains some relevant variables to find all substitutions for these variables, the *GrinGo* approach allows us to jump over the binder of the irrelevant variables, directly to the binder of the relevant ones. This technique allows us to further reduce the generation of redundant rules.

To illustrate this, consider the rules

$$a(1, 1..3) \qquad b(X) \leftarrow a(X, Y), \mathit{not}\ c(X) \qquad c(X) \leftarrow b(X)\ .$$

The predicate $a/2$ is solved before the ground instantiations of the second rule are computed; the atom $a(X, Y)$ acts as binder for the relevant variable X and the irrelevant variable Y. Figure 2 illustrates on the left how *dlv*'s back-jumping algorithm works; it revisits the binder $a(X, Y)$ three times to create all possible substitutions and thus outputs three times the same rule. The scheme on the right in Figure 2 exemplifies *GrinGo*'s binder-splitting. The binder $a(X, Y)$ is replaced with a binder for the relevant variable $a(X, \_)$ plus a second binder $a(X, Y)$ accounting for the bindings of the irrelevant variable Y, depending on the substitution of X. Due to this binder-splitting, it is now possible to jump directly from a solution back to the binder of the relevant variable X, avoiding any further substitutions of Y. As no further substitutions of X are found, the algorithm terminates and does not generate redundant ground rules.

Table 1. *GrinGo*'s back-jumping versus *dlv*'s back-jumping and *lparse*'s backtracking

Sudoku

board	*lparse*	*GrinGo*
1	584.28	5.27
2	190.82	5.48
3	1878.91	5.44
4	1.29	5.40
5	42.13	5.14
6	94.78	5.40
7	10901.35	5.32
8	118.75	5.53
9	165.50	5.42
10	1.58	5.32
SUM	13979.39	53.72

Graph 3-Colorability

graph	*dlv*	*lparse*	*GrinGo*
g40_05_0	0.00	57.30	0.01
g40_05_1	0.00	62.27	0.01
g40_05_2	0.24	39.82	4.01
g40_05_3	0.07	3.49	0.04
g40_05_4	0.00	4.49	0.01
g40_05_5	0.01	21.24	0.03
g40_05_6	0.02	159.69	0.00
g40_05_7	0.62	0.81	57.50
g40_05_8	0.00	1.36	1.12
g40_05_9	4.70	71.38	3.48
SUM	5.66	421.85	66.21

4 Experiments

We tested *GrinGo* [7] (V 0.0.1) together with *dlv*'s grounder (build BEN/Jul 14 2006) and *lparse* (V 1.0.17) on benchmarks illustrating the computational impact of back-jumping and binder-splitting. All tests were run on an Athlon XP 2800+ with 1024 MB RAM; each result shows the average of 3 runs.

For demonstrating the effect of back-jumping, we use logic programs encoding *Sudoku* games. An encoding consists of a set of facts, representing the numbers in Sudoku board coordinates, viz. `number(1..9)`, and a single rule that encodes all constraints on a solution of the given Sudoku instance. All Sudoku instances are taken from newspapers and have a single solution, the corresponding logic programs are available at [7]. The major rule contains 81 variables, which exceeds the maximum number of variables that *dlv* allows in a single rule. We thus only compare our results with *lparse*,[1] the latter relying on systematic backtracking. However, given that *dlv* uses back-jumping as well, we would expect it to perform at least as good as *GrinGo* (if it would not restrict the number of allowed variables below the threshold of 81).

Further, we tested logic programs that encode *Graph 3-Colorability* as grounding problem on a set of random graphs such that a valid coloring corresponds to the ground instantiation of a program. We tested 10 randomly generated graphs, each having 40 nodes and a 5% probability that two nodes are connected by an edge.

Table 1 shows the run times of *lparse*, *dlv*, and *GrinGo* in seconds. Due to its back-jumping technique, *GrinGo*'s performance is almost constant on the Sudoku examples. In addition, *GrinGo* on most instances is faster than *lparse*, the latter showing a great variance in run times. Also on the Graph 3-Colorability examples the grounders using back-jumping techniques turn out to be more robust. The outlier of *GrinGo* on graph 'g40_05_7' however shows that also back-jumping needs a good heuristics for the instantiation order among binders; this is a subject to future improvement.

[1] Note that the grounding procedures of *lparse* and *GrinGo* actually solve the Sudokus, and could in principle be (ab)used for solving other constraint satisfaction problems as well.

Table 2. The effect of *GrinGo*'s binder-splitting

	dlv		*lparse*		*GrinGo*	
n	time	rules	time	rules	time	rules
50	4.13	252500	0.95	252500	0.09	7500
75	24.77	849375	3.14	849375	0.18	16875
100	72.91	1020000	7.80	1020000	0.37	30000
125	166.14	3921875	15.86	3921875	0.68	46875
150	332.40	6772500	26.29	6772500	0.99	67500
175	—	—	42.20	10749375	1.44	91875
200	—	—	65.89	16040000	1.87	120000

For showing the effect of *GrinGo*'s binder-splitting, we use a suite of examples that have a solved predicate with a large domain (viz. `b/2`) and rules in which this predicate is used as the binder of both relevant and irrelevant variables:

```
b(1..n, 1..n).
p(X,Z) :- b(X,Y), b(Y,Z), not q(X,Z).
q(X,Z) :- b(X,Y), b(Y,Z), not p(X,Z).
```

The programs in this suite mainly aim at comparing *dlv* and *GrinGo*, both using backjumping but differing in binder-splitting; *lparse* is included as a reference. The results for parameter `n` varying from 50 to 200 are provided in Table 2. It shows the run time in seconds and the number of generated rules for *dlv*, *lparse*, and *GrinGo*. In fact, all three systems output the same set of rules, differing only in the number of duplicates. Interestingly, both *dlv* and *lparse* even produce the same collection of $n^2 + 2n^3$ rules (ignoring `compute` statements in *lparse*'s output). A hyphen "—" indicates a (reproducible) system failure. The results clearly show that *dlv* and *lparse* generate many (duplicate) rules, avoided by *GrinGo*, and therefore perform poorly on these (artificial) examples. This gives an indication on the computational prospect of binder-splitting.

A more general evaluation of all three grounder systems is an ongoing yet difficult effort, given the small common fragment of the input languages.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Syrjänen, T.: Lparse 1.0 user's manual. (http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz)
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) (2006) 499–562
4. Syrjänen, T.: Omega-restricted logic programs. Proceedings of LPNMR'01. Springer (2001) 267–279
5. Leone, N., Perri, S., Scarcello, F.: Backjumping techniques for rules instantiation in the DLV system. Proceedings of NMR'04. (2004) 258–266
6. Perri, S., Scarcello, F.: Advanced backjumping techniques for rule instantiations. Proceedings of the Joint Conference on Declarative Programming. (2003) 238–251
7. (`http://www.cs.uni-potsdam.de/~sthiele/gringo`)

Using Answer Sets to Solve Belief Change Problems

Aaron Hunter, James P. Delgrande, and Joel Faber

School of Computing Science
Simon Fraser University
{amhunter,jim,jfaber}@cs.sfu.ca

Abstract. We describe BE, an implemented system for solving belief change problems in the presence of actions. We illustrate how we can use BE to compute the result of belief progression, belief revision, and belief evolution in a transition system framework. The basic idea is to identify belief change problems with path-finding problems in a transition system. First, BE translates belief change problems into logic programs with answer sets corresponding to an agent's evolving beliefs. Second, it uses existing smodels libraries to find the answer sets for the given logic programs. Conflicting observations are handled in a separate preprocessing phase, which is also based on finding answer sets.

1 Introduction

There has been a great deal of formal work on belief change, with comparatively few implemented systems. In this paper, we present a simple command-line tool for computing the result of some natural belief change operators. We are concerned with belief change in the presence of actions, where the effects of actions are given in the action language $\mathcal{A}$. In this context, our system is able to compute the belief change following iterated sequences of actions and observations.

This paper makes two contributions to existing research. The first contribution is the introduction of an implemented system for experimenting with iterated belief change due to action. The second contribution is methodological: we use answer set programming to compute the result of belief change. This application of answer sets has not been extensively explored.

We proceed as follows. Following a brief description of the action language $\mathcal{A}$, we define specific approaches to belief progression, belief revision, and belief evolution. We then illustrate how to identify belief change with path-finding in a transition system. Finally, we describe BE, an implemented system for solving belief change problems through answer set programming.

2 Preliminaries

2.1 The Action Language $\mathcal{A}$

We introduce some useful notation and definitions for describing action effects in a transition system framework, essentially following [1]. An *action signature* is a pair $\langle \mathbf{F}, \mathbf{A} \rangle$ where $\mathbf{F}$ is a non-empty set of *fluent symbols* and $\mathbf{A}$ is a non-empty set of of *action symbols*. A *formula* is a propositional combination of fluent symbols, using the usual set of logical connectives $\{\neg, \rightarrow, \wedge, \vee\}$. A *literal* is either an element of $\mathbf{F}$ or an element of $\mathbf{F}$ preceded by the negation symbol, and we let $Lits$ denote the set of all literals. A *state* is a propositional interpretation over $\mathbf{F}$. If ϕ is a formula, then $|\phi|$ denotes the set of states satisfying ϕ. We let S denote the set of all states for a fixed action signature.

A *proposition* of the language $\mathcal{A}$ is an expression of the form

$$A \textbf{ causes } f \textbf{ if } g_1 \wedge \cdots \wedge g_p$$

where $A \in \mathbf{A}$, $f \in Lits$ and each $g_i \in Lits$. A set of propositions is called an *action description*. The semantics of $\mathcal{A}$ is defined in terms of *transition systems*. Formally, a transition system T is a pair $\langle V, E \rangle$ where $V \subseteq S$ and $E \subseteq S \times \mathbf{A} \times S$.

Definition 1. *Let AD be an action description, let s, s' be states and let A be an action symbol. The transition system defined by AD is $\langle S, E \rangle$ where $(s, A, s') \in E$ iff*

$$E(A, s) \subseteq s' \subseteq E(A, s) \cup s$$

*where $E(A, s)$ is the set of literals such that $f \in E(A, s)$ iff (A **causes** f **if** $g_1 \wedge \cdots \wedge g_p) \in AD$ and $s \models g_1 \wedge \cdots \wedge g_p$.*

A *path* from s to s' is a finite sequence of edges $\langle (s, A_1, s_1), \ldots (s_{n-1}, A_n, s') \rangle$. We say that a transition system is *strongly connected* if there is a path between every pair of states.

2.2 Belief Progression

A *belief state* is a set of states. A *belief change operator* is a function mapping a belief state to a new belief state, in response to some event. Belief progression refers to the process where an agent's beliefs change due to the execution of a state-changing action (for a discussion, see [2]). For each belief state κ and each action symbol A,

$$\kappa \diamond A = \{s' \mid s \in \kappa \text{ and } (s, A, s') \in E\}.$$

Hence $\kappa \diamond A$ is the set of states obtained from κ by projecting each state to the outcome of A. If ϕ is a propositional formula, we will write $\phi \diamond A$ as a shorthand for $|\phi| \diamond A$.

2.3 Belief Revision

For our purposes, belief revision refers to the belief change that occurs when new information is obtained about an unchanged world. We assume the reader is familiar with AGM belief revision [3]. Our presentation differs from the standard AGM approach in that we represent beliefs as sets of states, rather than sets of formulas. We also represent new information by sets of states, which we call *observations*. Intuitively, an observation α represents evidence that the actual state is in α. Our implementation is defined for the *topological revision operator* associated with a transition system [4]. Let T be a strongly connected transition system, let κ be a belief state, and let α be an observation. The topological revision operator $*$ associated with T is defined by setting $\kappa * \alpha$ to be the subset of α that can be reached by a minimum length path in T. Translating the AGM postulates into conditions on sets of states, we have the following result.

Proposition 1. *For every strongly connected transition system, the associated topological revision operator is an AGM revision operator.*

We do not claim that topological revision is suitable for all action domains, but it is suitable for domains where erroneous beliefs can plausibly be explained by actions. Again, for formulas ϕ and ψ, we write $\phi * \psi$ as a shorthand for $|\phi| * |\psi|$.

2.4 Belief Evolution

Belief evolution operators were introduced to reason about iterated belief change due to action in a transition system framework. We briefly introduce the basic idea, and refer the reader to [5] for the details.

A *world view* W is a pair $\langle \bar{A}, \bar{\alpha} \rangle$, where $\bar{A} = A_1, \ldots, A_n$ is a sequence of action symbols and $\alpha = \alpha_1, \ldots, \alpha_n$ is a sequence of observations. A belief evolution operator $\circ$ is defined with respect to a given progression operator $\diamond$ and a given revision operator $*$. A belief evolution operator maps a belief state and a world view to a sequence of belief states representing the evolution of an agent's beliefs. Informally, the evolution $\kappa \circ \langle \bar{A}, \bar{\alpha} \rangle$ is intended to represent the iterated belief change

$$\kappa \diamond A_1 * \alpha_1 \cdots \diamond A_n * \alpha_n.$$

In examples such as Moore's litmus paper problem [6], it is clear that these operations can not be carried out sequentially.

Let α be an observation and let $A_1, \ldots, A_n$ be a sequence of action symbols. The *pre-image* $\alpha^{-1}(A_1, \ldots, A_n)$ consists of every state s such that $s \diamond A_1 \diamond \cdots \diamond A_n \in \alpha$. For consistent observations, belief evolution is defined as follows:

$$\kappa \circ \langle \bar{A}, \bar{\alpha} \rangle = \kappa * \bigcap_i \alpha_i^{-1}(A_1, \ldots, A_i) \diamond \bar{A}.$$

Hence, every observation is treated as information about the initial state, suitably translated. This is intuitively correct in examples like the litmus paper problem. In domains where there may be conflicting observations, we need to use some heuristic to find a maximally consistent subsequence of observations.

3 The Approach

Assume we are given an action description AD in the action language $\mathcal{A}$. This action description picks out a transition system, which in turn defines a belief progression operator $\diamond$, a topological revision operator $*$, and a belief evolution operator $\circ$. Suppose that λ denotes a null action that does not change the state of the world. The following equalities are immediate: $\kappa \diamond A = \kappa \circ \langle A, S \rangle$ and $\kappa * \alpha = \kappa \circ \langle \lambda, \alpha \rangle$. In other words, both belief progression and belief revision can be expressed as special cases of belief evolution. Hence, to implement all three operators, it is sufficient to consider only belief evolution.

To illustrate, consider the case involving a single action and a single observation:

$$\kappa \circ \langle A, \alpha \rangle = \kappa * \alpha^{-1}(A) \diamond A.$$

According to the definition of topological revision, $\kappa * \alpha^{-1}(A)$ consists of all states in $\alpha^{-1}(A)$ that are minimally distant from κ. This observation leads to the following simple procedure for computing $\kappa \circ \langle A, \alpha \rangle$.

1. Determine $\alpha^{-1}(A)$.
2. Let $PATH$ denote the set of shortest paths from κ to $\alpha^{-1}(A)$.
3. Let κ_0 be the set of terminal nodes on paths in $PATH$.
4. Return $\kappa_0 \diamond A$.

This procedure was originally presented in [4]. The only difficult computation in the procedure occurs at step 2, where we must find all paths from κ to $\alpha^{-1}(A)$.

4 The Implementation

4.1 A Logic Program Representing Action Effects

BE is a command-line tool that runs in a Linux environment, using the `smodels` API [7] to compute answer sets. We describe the implementation in this section.

In [4], a procedure is outlined for translating an action description AD into a logic program $\tau_n(AD)$ such that answer sets for $\tau_n(AD)$ correspond to plans of length n in the transition system described by AD. The translation is based on a well-known translation from $\mathcal{C}$ [8]. In the interest of space, we omit the details here.

Let K be a conjunction of literals representing the initial beliefs of an agent, and let $W = \langle \bar{A}, \bar{\alpha} \rangle$ be a world view. We can extend $\tau_n(AD)$ to a program $\tau_{n,m}(AD, K, W)$ where the answer sets correspond to plans in the action domain AD of length $n + m$ with the following properties.

1. m is the length of the sub-path between some state s that satisfies the initial conditions K, and some state $s' \in \alpha^{-1}(A_1, \ldots, A_n)$,
2. n is the length of the world view W,
3. K is satisfied at time 0,
4. A_i executes at time $m + i - 1$
5. α_i is satisfied at time $m + i - 1$.

The logic program $\tau_n(AD)$ representing the action description consists of time stamped atoms such as $F(i)$ and $notF(i)$, representing the fact that F holds or does not hold at time i. Hence, it is straightforward to add rules ensuring these conditions. For example, if $K = K_1 \wedge \cdots \wedge K_p$, then we can guarantee (3) by adding the rules $K_i(0)$ for each i.

The first step performed by BE is to generate the logic program $\tau_{n,m}(AD, K, W)$, given input AD, K and W. The input is entered at the command line, or it can be passed in from a prepared text file.

4.2 Conflicting Observations

If the observations in W are conflicting, then the logic program $\tau_{n,m}(AD, K, W)$ has no answer sets. Hence, we need a method for dealing with conflicting observations. In BE, our approach is to resolve conflicting observations by keeping the more recent information. Specifically, we handle conflict as follows. First, for $p \leq n$, we define a logic program $\tau_n(AD, W, p)$ where the answer sets correspond to paths of length n with the following properties.

1. A_i executes at time i, for all $i \leq n$.
2. α_p is satisfied at time p.
3. α_i is satisfied at time i for each i such that $p < i \leq n$.

We can detect conflict and remove conflicting observations using the following algorithm. The input is an action description AD and a world view W.

```
1. Set $p = n$, set $W' = \langle \bar{A}, \langle \alpha'_1, \ldots, \alpha'_n \rangle\rangle = W$.
2. Determine whether $\tau_n(AD, W, p)$ has any valid answer sets.
3. If there are no valid answer sets, set $\alpha'_p = \top$.
4. If $p > 1$, set $p = p - 1$ and goto 2.
5. If $p = 1$, return $W'$.
```

Note that W' is consistent in the sense that there is a path following $A_1, \ldots, A_n$ and satisfying each α'_i at time i. For any input (AD, K, W), BE removes inconsistencies by using this algorithm to implicitly change the input to (AD, K, W').

4.3 Belief Evolution

In order to solve belief evolution problems, we need to find the minimal m such that $\tau_{n,m}(AD, K, W')$ has a non-empty set of answer sets. We find the minimal m using the following algorithm.

```
1. Set $m = 0$
2. Determine all answer sets to $\tau_{n,m}(AD, K, W')$.
3. Let $PATH$ be the corresponding set of paths.
   (a) If $PATH = \emptyset$, set $m = m + 1$ and goto 2
   (b) If $PATH \neq \emptyset$, then continue.
4. Let $\kappa_i$ denote the set of states in $PATH$ at time $m + i$.
5. Return $\langle \kappa_0, \kappa_1, \ldots, \kappa_n \rangle$.
```

Each set κ_i represents the set of states believed possible at time i following belief evolution. A formula representing the final beliefs can be obtained in the obvious manner from κ_n. The current version of BE returns the complete history of beliefs, simply because it requires the same computational effort.

4.4 Example

We can represent the litmus paper problem in BE by creating a text file `litmus.txt` containing the following information.

```
FillAcid causes Acid
FillBase causes -Acid
Dip causes Red if Acid
Dip causes Blue if -Acid
|-Blue & -Red & Acid| o <<Dip>, <Blue>>
```

The first 4 lines give the action description that describes the domain, and the last line gives a belief evolution query. The meaning of each line is clear, since the `BE` representation is a simple translation of the usual formal syntax. In this problem, an agent initially believes that a particular beaker contains an acid. After dipping the paper, the agent observes that the paper is blue. We compute the belief change by typing `be < litmus.txt` at the command line. The output in this case is `K0`{{}} `K1`{{`Blue`}}. The ouput gives belief states representing an agent's beliefs at time 0 and time 1, respectively. A state is represented by the set of true fluent symbols. In this example, `K0` contains a single state where every fluent is false. Informally, this means that the agent now believes the beaker contained a base at time 0. This indicates that the agent has revised the initial belief state in response to the observed color of the paper. At time 1, the only difference is the color of the litmus paper. This is intuitively correct, and it is the actual ouput of the belief evolution operator defined by the given action description.

More detailed documentation, including command-line flags and complete source code, at `www.cs.sfu.ca/∼cl/software/BeliefEvolution`.

5 Discussion

We have presented an implemented system for solving belief change problems in an action domain. Our system defines belief progression and belief revision with respect to a transition system, and it can be used to to solve iterated belief change problems by using a recency-based consistency check. The key is that we solve belief change through path-finding; it is known that action languages and answer sets are well-suited for this task. In future versions of the software, we would like to allow more flexible initial belief states and we would like to consider alternative heuristics to deal with inconsistency. The current system provides a proof of concept, and represents a first step towards the development of a new application of the `smodels` API to explore non-monotonic belief change.

References

1. Gelfond, M., Lifschitz, V.: Action languages. Linköping Electronic Articles in Computer and Information Science **3**(16) (1998) 1–16
2. Lang, J.: About time, revision, and update. In: Proceedings of NMR 2006. (June 2006)
3. Alchourrón, C., G ardenfors, P., Makinson, D.: On the logic of theory change: Partial meet functions for contraction and revision. Journal of Symbolic Logic **50**(2) (1985) 510–530
4. Hunter, A., Delgrande, J.: An action description language for iterated belief change. In: Proceedings of IJCAI07. (January 2007)
5. Hunter, A., Delgrande, J.: Iterated belief change: A transition system approach. In: Proceedings of IJCAI05. (August 2005)
6. Moore, R.: A formal theory of knowledge and action. In Hobbs, J., Moore, R., eds.: Formal Theories of the Commonsense World. Ablex Publishing (1985) 319–358
7. Niemelä, I., Simons, P.: smodels - an implementation of stable model and well-founded semantics for normal logic programs. In: Proceedings of LPNMR 92. Volume 1265 of Lecture Notes in Artificial Intelligence. (1992) 141–151
8. Lifschitz, V., Turner, H.: Representing transition systems by logic programs. In: Proceedings of LPNMR 99. (1999) 92–106

An *Smodels* System with Limited Lookahead Computation

Gayathri Namasivayam and Mirosław Truszczyński

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA

Abstract. We describe an answer-set programming solver *smodels*$^-$, derived from *smodels* by eliminating some lookahead computations. We show that for some classes of programs *smodels*$^-$ outperforms *smodels* and demonstrate the computational potential of our approach.

1 Introduction

In this paper we describe an answer-set programming solver *smodels*$^-$. It is a modification of the *smodels* solver [8]. The main difference is that *smodels*$^-$ attempts to identify and eliminate unnecessary lookaheads.

A common step in many answer-set programming and satisfiability solvers consists of expanding partial truth assignments. That is, given a partial truth assignment P, the solver applies some efficient inference rules to derive additional truth assignments that are forced by P. In the case of satisfiability solvers this process is called *unit-propagation* or *boolean constraint propagation* (cf. [3] for a recent overview).

These rules generalize to logic programs. Together with some other inference rules, specific to logic programming (cf. [5] for examples), they imply an expansion method for logic programs that can be viewed as a computation of the Kripke-Kleene fixpoint [4]. This method can be implemented to run in linear time. A stronger expansion technique, giving in general more inferences, is obtained when we replace the *Kripke-Kleene* fixpoint computation with a computation of the *well-founded* fixpoint [9]. The greater inference power comes at a cost. The well-founded fixpoint computation can be implemented to run in polynomial time, but no linear-time implementation is known.

Any polynomial-time expansion technique can be strengthened to another polynomial-time expansion method by applying the *lookahead*. Given a partial assignment, we assume a truth value for an unassigned atom and apply the expansion method at hand to the resulting partial assignment. If a contradiction is derived, the opposite truth value can be inferred and the expansion procedure is invoked again. The *full lookahead* consists of applying this technique to every unassigned atom and to both ways atoms can be assigned truth values until no more truth values for atoms (no more literals) can be derived. The full lookahead and the well-founded fixpoint computation form the basis for the expansion method used by *smodels*.

When expansion terminates, a typical answer-set programming solver selects an atom for branching. This step has a major effect on the performance of the overall search. *Smodels* uses the results obtained by the lookahead computation to decide which atom to choose.

Thus, in at least two important ways the performance of *smodels* depends on the full lookahead: it strengthens the expansion method, and it provides a good method to select atoms for branching. However, the full lookahead is costly. Our goal in this paper is to propose and implement a method that aims to improve the performance of *smodels* by limiting its use of lookahead so that few essential lookaheads (possibly even none at all) are missed. We call *smodels*$^{-}$ the resulting modification of *smodels*.

2 Algorithm for Identification of Propagating Literals

An unassigned literal is *propagating* (with respect to a program, a partial assignment and a particular expansion method) if assuming it is true and running the expansion method infers another literal that has been unassigned so far.

We will now present a method to identify propagating literals. The programs we consider consist of rules of the following types (in particular, such programs are output by *lparse*[2]; a and a_i **stands** for atoms, l_i stands for literals, m, w and w_i are non-negative integers):

Basic rule: a :- $l_1, \ldots, l_k$
Choice rule: $\{a_1, \ldots, a_n\}$:- $l_1, \ldots, l_k$
Cardinality rule: a :- $m\{l_1, \ldots, l_k\}$
Weight rule: a :- $w\{l_1 = w_1, \ldots, l_k = w_k\}$.

A rule is *active* with respect to the current partial assignment if its body is neither implied to be **true** nor **false** by the assignment. Our algorithm identifies a literal l as propagating, if there is an active rule r which, if we assume l to be **true**, allows us to make an inference. To this end, we consider all active rules in which l or its dual, $\bar{l}$, appear. Let us assume that r is a rule currently under consideration. Let $hd(r)$ and $bd(r)$ denote the set of literals that appear in the head and body of the rule r, respectively. There are four main cases (the cases not listed below either cannot occur or do not allow additional derivations based just on r).

Case 1. The literal l appears in the $bd(r)$.
Case 1a: The rule r is basic. If l is the only unassigned literal in the $bd(r)$ and the $hd(r)$ is unassigned then, we identify l as propagating (assuming l is **true** allows us to derive the $hd(r)$).

If l and exactly one other literal, say l', in the $bd(r)$ are unassigned and, in addition, the $hd(r)$ is assigned **false**, then assuming l allows us to infer that l' must be **false**. Thus, we identify l as propagating.
Case 1b: The rule r is a weight rule (the case of the cardinality rule is a special case). If the $hd(r)$ is unassigned and the sum of the weights of literals in the weight atom of r that are assigned **true** in the current partial assignment plus the weight of the literal l exceeds the lower bound w, then we identify l as propagating (assuming l is **true** allows us to derive the $hd(r)$).

If the $hd(r)$ is assigned **false** and the sum of the weights of literals in the weight atom of r that are assigned **true** in the current partial assignment plus the weight of l plus the largest weight of an unassigned literal other than l (say this literal is l') exceeds the lower bound, then we identify l as propagating (assuming l is **true** would allow us to infer that l' is **false**).

Case 2. The literal $\bar{l}$ appears in the $bd(r)$. To handle this case, for each atom a we maintain a counter, $ctr(a)$, for the number of active rules with this atom in the head.
Case 2a: The rule r is a basic or choice rule. If the head of r is an unassigned atom, say h, with $ctr(h) = 1$, or contains an unassigned atom h with $ctr(h) = 1$, we identify l as propagating (assuming l to be **true** blocks r and allows us to establish that h is **false**).

If the $hd(r)$ is an atom h such that h is assigned **true** and $ctr(h) = 2$, or if the $hd(r)$ contains an atom h assigned **true** and such that the $ctr(h) = 2$, we identify l as propagating. Assuming l is **true** blocks r and leaves only one active rule, say r', to justify h. This allows us to infer that the body of r' is **true** and may allow new inferences of literals. We do not check whether knowing that the body of r' is **true** allows new inferences. Thus, we may identify l as propagating even though it actually is not. This may lead to some unnecessary lookaheads that *smodels*$^{-}$ will occasionally perform. We are currently developing an implementation which eliminates this possibility here (and in some other similar cases below).
Case 2b: The rule r is a weight rule (the cardinality rule is a special case). If the $hd(r)$ is unassigned, $ctr(hd(r)) = 1$, and if setting l to **true** makes the weight atom in the $bd(r)$ **false**[1], then we identify l as propagating (setting it to **true** blocks r and allows us to infer that h is **false**).

If h is assigned **true** and the $ctr(h) = 1$, then we identify l as propagating (assuming l to be **true** may force other literals in the weight atom to be **true**; in this case we again do not actually guarantee that l will lead to new inferences).

If h is assigned **true**, $ctr(h) = 2$, and assuming l to be true forces the weight atom in the body of r to be false (blocks r), then there is only one other rule, say r' that could be used to justify h. The body r' must be **true** and it may lead to new inferences of literals. Again, we identify l as propagating, even though there is actually no guarantee that it is.

Case 3. The literal l appears in the $hd(r)$. It follows that l is an atom. If the $ctr(l) = 1$, we identify l as propagating (assuming l **true** forces the $bd(r)$ to be true and will lead to new inferences in the case of basic and choice rules, and may lead to new inferences in the case of cardinality and weight rules).

Case 4. The literal $\bar{l}$ is the $hd(r)$ (that is, $l = \mathit{not}\ h$ for some atom h).
Case 4a: The rule r is a basic rule. If the $bd(r)$ contains a single unassigned literal, say t, we identify l as propagating (assuming l **true** forces t to be **false**).
Case 4b: The rule r is a weight rule (the cardinality rule is a special case). If

[1] It will be the case when the total weight of unassigned literals in the weight atom together with the total weight of literals assigned true in the weight atom is less than the lower bound of the weight atom.

the sum of the weights of all literals assigned true in the weight atom of r plus the largest weight of an unassigned literal (say t) exceeds the lower bound, we identify l as propagating (assuming l **true** allows us to infer t to be **false**).

3 Implementation and Usage

We implemented *smodels*$^-$ by modifying the source code of *smodels*. In *smodels*$^-$, we take each atom from the queue of atoms that *smodels* performs lookaheads on and check, using the approach described above, if any of the two literals of this atom is propagating. If so, then *smodels*$^-$ performs lookahead on this literal. Otherwise, *smodels*$^-$ skips this lookahead.

Our program[2] is used in exactly the same way as *smodels*. It requires that input programs consist of rules described above. *Lparse* can be used to produce programs in the appropriate input.

4 Experimental Results and Discussion

For tight logic programs our method to limit lookaheads does not miss any essential lookaheads. Therefore, on tight programs *smodels*$^-$ and *smodels* traverse the same search space. As concerns the time performance, *smodels*$^-$ performs (in general) fewer lookaheads. However, it incurs an overhead related to identifying atoms that do not propagate. For programs with many fewer lookaheads, the savings outweigh the costs and we expect *smodels*$^-$ to perform better than *smodels*. On programs where few lookaheads are saved, one might expect that *smodels*$^-$ would perform worse but not drastically worse, as our algorithm to eliminate lookaheads works in polynomial time (in the worst case).

Our experiments confirm this expected behavior. We considered three classes of programs: (1) programs obtained by encoding as logic programs instances used in the SAT 2006 competition of pseudo-boolean solvers [7] (154 instances); (2) randomly generated tight logic programs (39 instances); and (3) logic programs encoding instances of the weighted spanning-tree problem (27 instances) [1]. The results are presented in Figure 1. The graphs show how many times *smodels*$^-$ is slower or faster (whichever is the case) than *smodels*, based on the running times. The instances are arranged according to ascending running time of *smodels*$^-$. The dotted lines separate the instances into those for which *smodels*$^-$ is slower than, runs in the same time as, and is faster than *smodels*.

For the first category of programs, *smodels*$^-$ clearly outperforms *smodels*. It is due to two factors: *smodels*$^-$ performs on average 71% fewer lookahead computations; and the time needed by lookahead in *smodels* to discover that no inferences can be made is non-negligible.

For the next two classes of programs, the benefits of limiting lookahead are less obvious. Overall there is no significant difference in time in favor of any of the methods. For programs in the second group, calling lookahead for an atom and discovering no new inferences can be made does not take much computation due to a simple

[2] *Smodels*$^-$ can be obtained from http://www.cs.engr.uky.edu/ai/

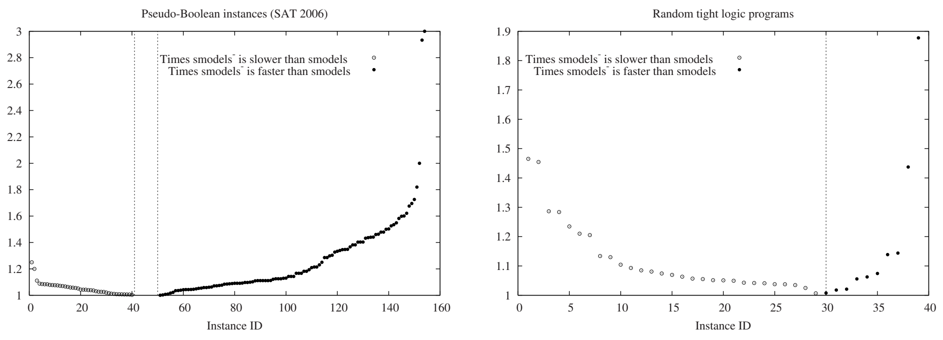


Fig. 1. PB Instances and random tight logic programs

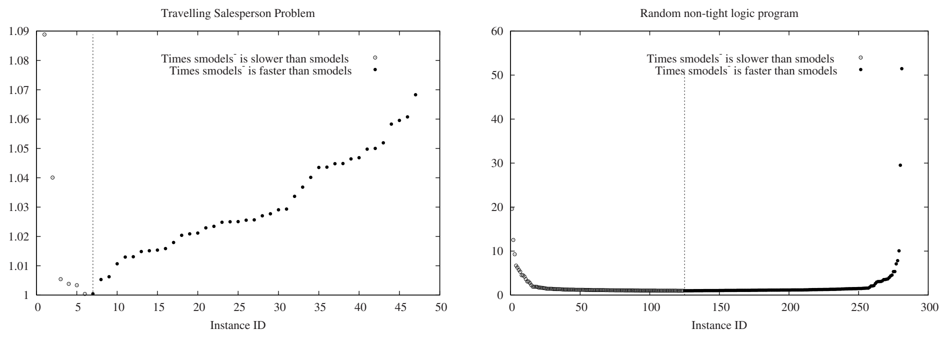


Fig. 2. Traveling Salesperson Problem, Random Logic Program (non-tight)

form of rules in programs (no weight or cardinality atoms, each rule consisting of three literals). Thus, even though fewer lookaheads were made by *smodels*$^-$ (16% fewer on average on programs in the second group), the savings often did not always compensate the overhead. For the programs in the third group, most atoms appear in 2-literal clauses and no such atom will be excluded from lookahead by our technique. Thus, both *smodels*$^-$ and *smodels* perform an identical number of lookahead computations and their time performance is within a small percentage from each other (in all but one case, within 1.5% from each other; 4% in the remaining case). As the times are essentially identical, we do not provide the graph.

For non-tight programs, our method may eliminate lookaheads yielding new inferences through the well-founded fixpoint computation (whether the computation yields new inferences cannot be determined by inspecting rules individually). Therefore, the expansion method of *smodels*$^-$ is in general weaker than that of *smodels*. Moreover, due to missing essential lookaheads, the search heuristics of *smodels*$^-$ may miss atoms that will be used for branching by *smodels*.

Thus, for non-tight programs the benefits of using *smodels*$^-$ may diminish or disappear entirely. However, in our experiments it was not so. We tested two classes of programs: the non-tight programs encoding the traveling salesperson problem (TSP) [6] (47 instances); and random logic programs [1] (281 instances, all turned out to be non-tight, even though it is not *a priori* guaranteed). The results are presented in Figure 2. For TSP problem, *smodels*$^-$ still shows an overall better performance than *smodels* (although the improvement does not exceed 7%). In the case of random programs, no program seems to have any discernible edge.

5 Conclusion

We presented an answer-set programming solver *smodels*$^-$, obtained by limiting lookaheads in *smodels*. The experiments show that on tight programs *smodels*$^-$ often outperforms *smodels*, especially on programs using many weight atoms. It is never much worse than *smodels* as it always searches through the same search space and extra computation it incurs runs in polynomial time. For non-tight programs, our experiments showed that *smodels*$^-$ performs comparably to *smodels* despite the fact it may miss essential lookaheads. However, we expect that there are classes of non-tight programs on which *smodels* would prove superior.

In our ongoing work we study additional techniques to limit lookahead and more efficient ways to implement them to decrease the overhead.

Acknowledgments

The authors thank Ilkka Niemelä for suggesting this research direction and Lengning Liu for help in preparing the graphs. The authors acknowledge the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. Asparagus, http://www.asparagus.cs.uni-potsdam.de/.
2. Lparse, http://www.tcs.hut.fi/Software/smodels/.
3. H.E. Dixon, M.L. Ginsberg, and A.J. Parkes, *Generalizing Boolean Satisfiability I: Background and Survey of Existing Work*, Journal of Artificial Intelligence Research **21** (2004), 193–243.
4. M. C. Fitting, *A Kripke-Kleene semantics for logic programs*, Journal of Logic Programming **2** (1985), no. 4, 295–312.
5. M. Gebser and T. Schaub, *Tableau calculi for answer set programming*, Proceedings of ICLP 2006 (S. Etalle and M. Truszczyński, eds.), LNCS, vol. 4079, Springer, 2006, pp. 11–25.
6. L. Liu and M. Truszczyński, http://www..cs.uky.edu/ai/pbmodels.
7. V. Manquinho and O. Roussel, *Pseudo boolean evaluation 2005*, 2006, http://www.cril.univ-artois.fr/PB06/.
8. P. Simons, I. Niemelä, and T. Soininen, *Extending and implementing the stable model semantics*, Artificial Intelligence **138** (2002), 181–234.
9. A. Van Gelder, K.A. Ross, and J.S. Schlipf, *The well-founded semantics for general logic programs.*, Journal of the ACM **38** (1991), no. 3, 620–650.

Programming Applications in CIFF

P. Mancarella[1], F. Sadri[2], G. Terreni[1], and F. Toni[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{paolo,terreni}@di.unipi.it
[2] Department of Computing, Imperial College London, UK
{fs,ft}@doc.ic.ac.uk

Abstract. We show how to deploy the CIFF System 4.0 for abductive logic programming with constraints in a number of applications, ranging from combinatorial applications to web management. We also compare the CIFF System 4.0 with a number of logic programming tools, namely the A-System, the DLV system and the SMODELS system.

1 Introduction

Abduction has broad applications as a tool for hypothetical reasoning with incomplete knowledge. It allows the labeling of some pieces of information as *abducibles*, i.e. as hypotheses, that can be assumed to hold, provided that they are "consistent" with the given knowledge base. Abductive Logic Programming (ALP) combines abduction with logic programming enriched by *integrity constraints* to further restrict the range of possible hypotheses. ALP has also been integrated with *Constraint Logic Programming* (CLP), for having an arithmetic tool for *constraint solving*. Important applications of ALP with constraints (ALPC) include agent programming [1,2] and web management tools [3].

Many proof procedures for ALPC have been proposed, including ACLP [4], the A-System [5] and CIFF [6], which extends the IFF procedure for ALP [7] by integrating a CLP solver and by relaxing some *allowedness* conditions given in [7]. In this paper we describe the CIFF System 4.0[1] implementation and we compare it empirically with other systems showing that (1) they have comparable performances and (2) the CIFF System 4.0 has some unique features, in particular its handling of variables taking values on unbound domains.

2 Background

Here we summarize some background concepts. For more details see [6,7,8].

An *abductive logic program with constraints* consists of three components: (i) A constraint logic program, referred to as the *theory*, namely a set of clauses of the form $A \leftarrow L_1 \wedge \ldots \wedge L_m$, where the L_is are literals (ordinary atoms, their negation, or constraint atoms) and A is an ordinary atom, whose variables are all implicitly universally quantified from the outside. (ii) A finite set of *abducible*

[1] The CIFF System 4.0 is available at `www.di.unipi.it/∼terreni/research.php`.

predicates, that do not occur in any conclusion A of any clauses in the theory. (iii) A finite set of *integrity constraints* (ICs), namely implications of the form $L_1 \wedge \cdots \wedge L_m \rightarrow A_1 \vee \cdots \vee A_n$ where the L_is are literals and the A_js are (ordinary or constraint) atoms or the special atom $false$. All the variables in an IC are implicitly universally quantified from the outside, except for variables occurring only in the A_js which are existentially quantified with scope $A_1 \vee \cdots \vee A_n$. The theory provides definitions for non-abducible, non-constraint, ordinary predicates; it can be extended by means of sets of atoms in the abducible predicates, subject to satisfying the integrity constraints. Constraint atoms are evaluated within an underlying structure, as in conventional CLP.

A *query* is a conjunction of literals (whose variables are free). An *answer* to a query specifies which instances of the abducible predicates have to be assumed to hold so that both (some instance of) the query is entailed by the constraint logic program extended with the abducibles and the ICs are satisfied, wrt a chosen semantics for (constraint) logic programming and a notion of satisfaction of ICs.

The CIFF procedure computes such answers, with respect to the notion of entailment given by the 3-valued completion. It operates with a presentation of the theory as a set of *iff-definitions*, which are obtained by the (selective) *completion* of all predicates defined in the theory except for the abducible and the constraint predicates. CIFF returns three possible outputs: (1) an abductive answer to the query, (2) a failure, indicating that there is no answer, and (3) an *undefined* answer, indicating that a critical part of the input is not *allowed* (i.e. does not satisfy certain restrictions on variable occurrences). (1) is in the form of a set of (non-ground) abducible atoms and a set of constraints on the variables of the query and of the abducible atoms.

The CIFF procedure operates on so-called *nodes* which are conjunctions of formulas called *goals*. Intuitively, sequences of nodes represent branches in the proof search tree. A proof (and the search tree) is initialised with a node containing the ICs and the original query. The iff-definitions are used to *unfold* defined predicates as they are encountered in goals. The proof procedure repeatedly replaces one node with another by applying the procedure's *rewrite rules* to the goals. If a disjunction of goals is encountered, then the *splitting* rule can be applied, giving rise to alternative branches in the search tree. Besides *unfolding* and *splitting*, CIFF uses other rewrite rules (see [6]) such as *propagation* with the ICs.

A node containing a goal $false$ is called a *failure node*. If all branches in a derivation terminate with failure nodes, then the procedure is said to fail (no answer to the query). A non-failure (allowed) node to which no more rewrite rules apply can be used to extract an (abductive) answer.

ICs can be used to specify *reactive rules* in many applications, e.g modelling agents. In some cases the classical treatment of negation in ICs in CIFF can lead to non-intuitive answers being computed. For example, ICs $A \wedge \neg B \rightarrow C$ and $A \rightarrow C \vee B$ are treated equivalently in CIFF. Hence, one way to satisfy these ICs is to ensure that C holds whenever A holds. However, the only "reactive meaning" of the original IC is to ensure that C holds when *both* A and $\neg B$ have been proven. We have therefore investigated a different way of treating negation within ICs, namely *negation as failure* (NAF) [8], and have integrated this treatment into CIFF [9], by allowing ICs to be either marked or unmarked depending upon the required treatment of negation in them.

3 The CIFF System 4.0

CIFF 4.0 is a Sicstus Prolog implementation of CIFF. It maintains the computational basis of version 3.0 [10,9], but the underlying engine has been almost completely rewritten in order to improve efficiency. The main predicate is `run_ciff( +ALP, +Query, -Answer)` where the first argument is a list of `.alp` files representing an abductive logic program with constraints, the `Query` is a query, represented as a list of literals, and `Answer` will be instantiated either to a triple with a list of abducible atoms and two lists of variable restrictions (i.e. disequalities and constraints on the variables in the `Answer` and/or in the `Query`) or to the special atom `undefined` if an allowedness condition is not met. In (any file in) `ALP`, abducible predicates `Pred`, e.g. with arity 2, are declared via `abducible(Pred(_,_))`, equality/disequality atoms are defined via `=, \==` and constraint atoms are defined via `#=, #\=, #<, #=<` and so on. Finally, negative literals are of the form `not(A)` where `A` is an ordinary atom. Clauses and ICs are represented (resp.) as

```
A :- L_1, ..., L_n.      [L_1, ..., L_m] implies [A_1, ..., A_n].
```

CIFF rewrite rules are implemented as Prolog clauses defining `sat(+State, -Answer)`, where `State` represents the current CIFF node and it is initialised, within the prolog clause defining `run_ciff( +ALP, +Query, -Answer)`, to `Query` plus all the ICs in the `ALP`. `State` is defined as:

`state(Diseqs,CLPStore,ICs,Atoms,Abds,Disjs).` The predicate `sat/2` calls itself recursively until no more rules can be applied to the current `State`.Then it instantiates the `Answer`.

Below we sketch the most important techniques used to render CIFF 4.0 efficient.

Managing variables. Variables play a fundamental role in CIFF nodes: they can be either universally quantified or existentially quantified or free. A universal variable can appear only in an IC (which defines its scope). An existential/free variable can appear anywhere in the node with scope the entire node. To distinguish at run-time free/existential and universal variables we associate with the former an `existential` attribute.

Determining variable quantification efficiently is very important as CIFF proof rules for variable operations such as *equality rewriting* and *substitution* are heavily used in a CIFF computation. In CIFF 4.0 these rules are not treated as separate rewrite rules, but have been incorporated within the main rewrite rules (*propagation*, *unfolding* etc) resulting in improvements of performances.

Constraint solving. Interfacing efficiently CIFF 4.0 with the underlying CLPFD solver in Sicstus Prolog is fundamental for performance purposes. However, the CLPFD solver binds variables to numbers when checking satisfiability of constraints in the `CLPstore`, while we want to be able to return non-ground answers. The solution adopted in CIFF 4.0 is an algorithm which allows, when needed, to check the satisfiability of the `CLPstore` as usual and then to restore the non-ground values via a forced backtracking.

Grounded integrity constraints. CIFF 4.0 adopts some specialised techniques for managing some classes of ICs, referred to as *grounded ICs*. Roughly speaking, grounded ICs are ICs whose variables will eventually be grounded during a computation, after unfolding and propagation. For example, if `p(1)` is the only clause for `p`, then the IC `[p(X)] implies [a(X)]` is grounded. If `p(1)` is replaced by `p(Y)` then the IC is not grounded anymore.

Grounded ICs are managed at a system level by exploiting both dynamic assertions/retractions of ground terms and the coroutining mechanisms of the underlying Prolog. This algorithm allows both to reduce the node size and to perform the operations on the grounded ICs efficiently because they are not physically in the node but they are dynamically maintained in the Prolog global state.

To declare an IC as grounded, the operator `implies_g` is used instead of `implies`.

4 Experimentation and Comparison

Experiments have been made on a Linux machine equipped with a 2.4 Ghz PENTIUM 4 - 1Gb DDR Ram, using SICStus Prolog version 3.11.2. Execution times are in seconds ("—" means "above 5 minutes"). Comparisons are with the A-system (`AS`) [5], the DLV system [11], and SMODELS (`SM`) [12]. In all examples, unless otherwise specified, the CIFF initial query is *true*. The adopted problem representations for the other systems are omitted due to lack of space but they can be found on the CIFF web site.

Problem 1: N-Queens. The CIFF formalisation of this problem is very simple:

```
abducible(q_pos(_,_)).              %%%ABDS

queen(X) :- q_domain(X).            %%%CLAUSES
q_domain(X) :- X in 1..n.           %%% in real code n is an integer!
exists_q(R) :- q_pos(R,C), q_domain(C).
safe(R1,C1,R2,C2) :- C1#\=C2, R1+C1#\=R2+C2, C1-R1#\=C2-R2.

[queen(X)] implies [exists_q(X)].  %%%ICS
[q_pos(R1,C1),q_pos(R2,C2),R1#\=R2] implies [safe(R1,C1,R2,C2)].
```

All systems return all the correct solutions. In the comparison below, we also include CIFF 3.0 to underline the performance improvements of CIFF 4.0.

N-Queens results (first solution)

Queens	CIFF 3	CIFF 4	AS	SM	DLV
n = 8	24.75	0.03	0.03	0.01	0.01
n = 28	—	0.29	0.27	55.32	35.17
n = 32	—	0.37	0.32	—	—
n = 64	—	1.62	1.52	—	—
n = 100	—	4.55	4.24	—	—

Problem 2: Hamiltonian cycles. The CIFF 4.0 encoding makes use of the NAF module for ICs in order to avoid loops and to collect all possible answers.

```
abducible(ham_edge(_,_,_)).   abducible(checked(_,_)).  %%%ABDS

ham_cycle(X) :- ham_cycle(X,X,0).                          %%%CLAUSES
ham_cycle(X,Y,N) :- ham_edge(X,Y,N),edge(X,Y),checked(X,N).
ham_cycle(X,Y,N) :- ham_edge(X,Z,N),edge(X,Z),checked(X,N),
                    ham_cycle(Z,Y,M),M#=N+1,Z\==Y.
is_checked(V2) :- checked(V2,M).

[checked(X,N),checked(X,M),M#\=N] implies [false].      %%%ICS
[vertex(V2),not(is_checked(V2))] implies [false].
```

The predicates edge/2 and vertex/1 represent any given graph and are given as (domain-dependent) additional clauses, and is_checked(V2) is introduced to guarantee allowedness. The query is [ham_cycle(V)] where V is any vertex of the graph. In the comparison below, CIFF_G stands for CIFF but replacing the first IC by a grounded IC. We omit here a comparison with the A-system as we were unable to specify the problem avoiding looping.

Hamiltonian cycles results (all solutions)

Nodes	**CIFF**	**CIFF_G**	**SM**	**DLV**
4	0.04	0.03	0.03	0.02
20	0.45	0.15	0.16	0.02
40	1.93	0.41	1.53	0.03
80	10.95	1.20	11.41	0.04
120	27.62	2.39	43.43	0.07

Problem 3: Web Sites repairing. The last example shows how abduction can be used for checking and repairing links in a web site, given the specification of the site via an abductive logic program with constraints. Here, a *node* represents a web page. [3]. As an example, consider a web site where a node is either a *book*, a *review* or a *library*, a *link* is a relation between two nodes and every book must have at least a link to both a review and a library. The CIFF System 4.0 formalisation of this problem (together with a simple web site instance) is:

```
abducible(add_node(_,_)).    abducible(add_link(_,_)).   %%% ABDS

is_node(N,T) :- node(N,T),node_type(T).            %%%CLAUSES
is_node(N,T) :- add_node(N,T),node_type(T).
node_type(lib).     node_type(book).    node_type(review).

is_link(N1,N2) :- link(N1,N2),link_check(N1,N2).
is_link(N1,N2) :- add_link(N1,N2),link_check(N1,N2).
link_check(N1,N2) :- is_node(N1,_), is_node(N2,_), N1 \== N2.
book_links(B) :- is_node(B,book), is_node(R,review),is_link(B,R),
                 is_node(L,lib),is_link(B,L).

[is_node(B,book)] implies [book_links(B)].     %%% ICS
[add_node(N,T),node(N,T)] implies [false].
[add_link(N1,N2),link(N1,N2)] implies [false].
[is_node(N,T1),is_node(N,T2),T1 \== T2] implies [false].

node(n1,book).  node(n3,review).  link(n1,n3). %%%WEB SITE INSTANCE
```

CIFF 4.0 returns the following answer representing correctly the need of a new link between the *book* `n1` and a new *library node* `L`:

```
Abds:   [add_link(n1,L), add_node(L,lib)].
Diseqs: [L\==n3,L\==n1]         CLP store: []
```

Notice that the variable `L` in the answer can neither be bounded to a finite domain nor grounded. This is the reason why the other systems seem unable to deal with these cases and thus no comparison is provided.

5 Conclusions

The experiments performed (including some for planning and graph-coloring omitted here for lack of space) show that CIFF 4.0 performances are comparable with other existing systems on classical problems, though allowing the exploitation of abduction on problems where non ground solutions are required. We plan to improve the treatment of (grounded) ICs, and to build a GUI for better usability. Finally, we are porting the system onto a free Prolog platform.

References

1. Kakas, A.C., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: The KGP model of agency. In: Proc. ECAI-2004. (2004) 340–367
2. Sadri, F., Toni, F., Torroni, P.: An abductive logic programming architecture for negotiating agents. In: Proc. JELIA02. (2002) 419–431
3. Toni, F.: Automated information management via abductive logic agents. Journal of Telematics and Informatics (2001) 89–104
4. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive constraint logic programming. Journal of Logic Programming **44** (2000) 129–177
5. Denecker, M., C.Kakas, A., Nuffelen, B.V.: A-system: Declarative problem solving through abduction. In: Proc. IJCAI 2001. (2001) 591–597
6. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The CIFF proof procedure for abductive logic programming with constraints. In: Proc. JELIA04. (2004) 31–43
7. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33**(2) (1997) 151–165
8. Sadri, F., Toni, F.: Abduction with negation as failure for active databases and agents. In: Proc. AI*IA 99. (1999) 49–60
9. Endriss, U., Hatzitaskos, M., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: Refinements of the CIFF procedure. In: Proc. ARW05. (2005)
10. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: Abductive logic programming with CIFF: system description. In: Proc. JELIA04. (2004) 680–684
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2006) 499–562
12. Niemela, I., Simons, P.: SMODELS - an implementation of the stable model and well-founded semantics for normal logic programs. In: Proc. LPNMR97. (1997) 420–429

CPP: A Constraint Logic Programming Based Planner with Preferences

Phan Huy Tu, Tran Cao Son, and Enrico Pontelli

Computer Science Department,
New Mexico State University, Las Cruces, New Mexico, USA
{tphan|tson|epontell}@cs.nmsu.edu

Abstract. We describe the development of a constraint logic programming based system, called CPP, which is capable of generating most preferred plans with respect to a user's preference and evaluate its performance.

1 Introduction

The problem of finding a plan satisfying the preferences of a user has been discussed in [13] and recently attracted the attention of researchers in planning [2,3,5]. Indeed, PDDL, the de-facto language of the planning community, has been recently extended [9] to include constructs for the specification of users' preferences.

The original proposal described in [13] describes a model to integrate the users' preferences into a planning system based on *Answer Set Programming (ASP)*. Due to the lack of a list operator and the inflexibility in dealing with function symbols, the encoding of preferences in this system is somewhat unnatural, and it requires the introduction of artificial constant and predicates symbols. Furthermore, the encoding proposed in [13] inherits the requirement of answer set solvers that all variables have to be instantiated before computing the answer sets, possibly leading to extremely large encodings. For these reasons, the encoding in [13] is not expected to scale well to handle complex preferences.

This paper describes a *Constraint Logic Programming (CLP)* [11] based system, called CPP, for computing most preferred plans of planning problems. The detailed implementation of CPP is presented in [14]. The choice of CLP is suggested by a number of factors. First, CLP is a logic programming based paradigm, very declarative, and it is expected to allow us to maintain the underlying model proposed in [13]. Second, CLP does not require program grounding, and it allows the use of lists and other function symbols, leading to a more compact encoding of problems. CLP (and, in particular, *CLP over Finite Domains (CLP(FD))* [15]) provides the ability to express and efficiently handle arithmetic constraints, and the paradigm offers methodologies for describing search and optimization strategies. These two features appear to be vital in the context of dealing with preferences. Finally, recent studies [7] have suggested that CLP can provide an effective alternative, in terms of performance, to ASP for many classes of problems.

The CPP system is implemented in GNU Prolog [6]—a Prolog compiler with constraint solving over finite domains capabilities. CPP accepts planning problems described in the action language $\mathcal{AL}$ [1] in terms of logic programs. Users' preferences are described in terms of logic programs as well, using the preference language $\mathcal{PP}$ from [13].

2 Background

Action Language $\mathcal{AL}$ and Planning Problems: An *action theory* of $\mathcal{AL}$ is a pair $\langle D, I\rangle$, where D is a set of propositions expressing the conditional effects of actions, the causal relationships between fluents and the executability conditions for the actions, and I is the set of propositions describing the initial state of the world. Semantically, an action theory $\langle D, I\rangle$ describes a transition diagram, whose nodes correspond to possible configurations of the world, and whose arcs are labeled with actions. A path in the transition diagram corresponds to a trajectory of a sequence of actions. We assume that domains are deterministic, i.e., there is at most one trajectory for a sequence of actions. A *planning problem* $\mathcal{P}$ in $\mathcal{AL}$ is a triple $\langle D, I, G\rangle$ where $\langle D, I\rangle$ is an action theory and G is a set of literals describing goal states. A *plan* of $\mathcal{P}$ is a sequence of actions that leads to a state satisfying G from the initial state, according to the transition diagram of $\langle D, I\rangle$.

Preference Language $\mathcal{PP}$: The language $\mathcal{PP}$ [13] supports three types of preferences: basic desires, atomic preferences, and general preferences. A *basic desire* is a temporal formula expressing desired constraints on trajectories of plans. E.g., in the transportation domain, to express the fact that a user prefers to stop by a post office, we can write

$$\mathbf{eventually}(\bigvee_{post\_office(X)} at(X))$$

or, if the user's preference is not to go by bus, we can use the formula

$$\mathbf{always}(\bigwedge_{bus\_line(X,Y)} \neg occ(bus(X,Y))).$$

In the above formulae, $occ(A)$ means that action A must occur at the current state; $\mathbf{eventually}(\varphi)$ (resp. $\mathbf{always}(\varphi)$) means that the formula φ must sometimes (resp. always) hold during the execution of the plan. An *atomic preference* provides users with a way to *rank* their basic desires, e.g., the fact that going by bus is preferred to going by subway can be expressed as $\bigwedge_{bus\_line(X,Y)\wedge subway(X,Y)} occ(bus(X,Y)) \lhd occ(subway(X,Y))$. A *general preference* is composed of atomic preferences using the connectives & (*and*), | (*or*), ! (*not*), and $\lhd$ (*preferable*). Given a preference ψ, $\mathcal{PP}$ defines an ordering relation $\prec_\psi$ between trajectories of plans: $\alpha \prec_\psi \beta$ means that the trajectory α is preferred to the trajectory β w.r.t. ψ. In this regard, a plan π is *most preferred* w.r.t. ψ if there is no plan π' s.t. the trajectory of π' is preferred to the trajectory of π.

3 System Description

We developed a system, called CPP[1], in GNU Prolog for finding most preferred plans. CPP takes as input a planning problem $\mathcal{P}$ and a set of preferences and

[1] CPP's source code is available at `http://www.cs.nmsu.edu/~tphan/software.htm`

returns as output most preferred plans of $\mathcal{P}$ w.r.t. a preference. It works by translating a planning problem into a constraint satisfaction problem, whose satisfying truth assignments correspond to plans of $\mathcal{P}$[2]. To handle preferences, CPP defines a weight function that maps plans to (integer) numbers in such a way that any plan with a maximal value of the weight function is a most preferred plan of $\mathcal{P}$. However, the other direction does not hold in general: some of the most preferred plans do not correspond to any optimal solution of the constraint satisfaction problem. If we wish to find them, we might have to use another weight function.

In our framework, a planning problem $\mathcal{P}$ is described in $\mathcal{AL}$ in terms of a Prolog program. Since GNU Prolog does not allow the symbol $\neg$, a literal $\neg F$ where F is a fluent is written as $neg(F)$. Listed below is an example of an input file for the transportation domain with three actions *walk*, *bus*, and *subway*, with obvious meaning. Initially the user is at home and his goal is to be at his office.

```
% locations, facilities & transportation media ---------------------------
loc(home). loc(office). loc(stA). loc(stB). loc(stC). loc(stD).
post_office(stA). post_office(stB). phone(stB). phone(stC). phone(stD).
bus_line(stA,stB). bus_line(stC,stD). sub_line(stA,stD). sub_line(stC,stB).
pedestrian(home,stA).  pedestrian(stB,office). pedestrian(home,stC).
pedestrian(stD,office).
% actions & fluents -----------------------------------------------------
action(bus(L1,L2)):- bus_line(L1,L2). action(subway(L1,L2)):- sub_line(L1,L2).
action(walk(L1,L2)) :- pedestrian(L1,L2). fluent(at(L)) :- loc(L).
% executable condition --------------------------------------------------
executable(bus(L1,L2), [at(L1)]) :- action(bus(L1,L2)).
executable(subway(L1,L2), [at(L1)]) :- action(subway(L1,L2)).
executable(walk(L1,L2), [at(L1)]) :- action(walk(L1,L2)).
% dynamic causal laws ---------------------------------------------------
causes(bus(L1,L2), at(L2), []) :- action(bus(L1,L2)).
causes(subway(L1,L2), at(L2), []) :- action(subway(L1,L2)).
causes(walk(L1,L2), at(L2), []) :- action(walk(L1,L2)).
% static causal laws: cannot be at L1 if at L2 --------------------------
caused([at(L2)],neg(at(L1))) :  loc(L1), loc(L2), L1 \== L2.
% initial state & goal --------------------------------------------------
initially(at(home)). goal(at(office)).
```

The set of preferences is also described in terms of Prolog clauses. A basic desire is expressed as a fact of the form `basic_desire(Name,Formula)`, where `Name` is the name of the basic desire and `Formula` is a temporal formula representing the basic desire. An atomic preference is expressed as `atomic_preference(Name,Desires)`, where `Name` is the name of the atomic preference and `Desires` is a list of basic desire names. Intuitively, `Desires = [D1,..,Dk]` corresponds to the atomic preference `D1` $\lhd$ .. $\lhd$ `Dk`. A general preference is `general_preference(Name,Formula)`, where `Name` is the name of the preference and `Formula` is the formula representing the preference.

[2] Our translation is similar to SAT-based approaches to planning such as [4,12,10].

In addition to basic desires, atomic preferences and general preferences, which are supported by $\mathcal{PP}$, CPP allows another type of preferences, called *metric preferences*, which are declared as `metric_preference([(D1,W1),..,(Dk,Wk)])` where `Di`'s are basic desires and `Wi`'s are weights associated with them. The *weight* of a plan π w.r.t. a metric preference ψ is the sum of the weights of the basic desires in ψ that are satisfied by π. A plan is most preferred if it has a maximal weight value. The following is a simple example of an input file for preferences.

```
basic_desire(ds1,eventually(or(L))) :- findall(at(X),post_office(X),L).
basic_desire(ds2,eventually(or(L))) :- findall(at(X),phone(X),L).
basic_desire(ds3,always(and(L)))    :-
                findall( neg(occ(bus(X,Y))), action(bus(X,Y)),  L ).
atomic_preference(ap1,[ds1,ds2]).     atomic_preference(ap2,[ds1,ds3]).
general_preference(gp1,and(ap1,ap2)). general_preference(gp2,or(ap1,ap2)).
metric_preference(mp1,[(ds1,2),(ds2,3),(ds4,5)]).
```

In this example, the first three lines define three different basic desires. The first basic desire `ds1` describes that the user wants to be at a post office during his plan (just because he wants to send a package). The second basic desire `ds2` says that he wants to be at a phone box (to make a call, for example). The last basic desire, `ds3`, states that he does not want to go by bus.

The fourth line defines two atomic preferences: `ap1` corresponds to the atomic preference `ds1` $\lhd$ `ds2`, and `ap2` corresponds to the atomic preference `ds1` $\lhd$ `ds3`, which means that `ds1` is preferred to `ds2`, and `ds1` is preferred to `ds3`, respectively. The next line describes two general preferences: `gp1` stands for `ap1` & `ap2`, and `gp2` stands for `ap1` | `ap2`. The last line describes a metric preference consisting of basic desires `ds1`, `ds2`, and `ds3` with weights 2, 3, and 5 respectively.

The input planning problem and the preference file are compiled to CLP, and execution is initiated by issuing the predicate `main/2` to compute the most preferred plans. The first argument of `main` is the name of the preference and the second argument is the length of the plan we wish to find. For example, to find a most preferred plan of length 3 w.r.t. the basic desire `ds1` (resp. `gp1`) we can submit the query `main(ds1,3)` (resp. `main(gp1,3)`). The following is the output of the queries `main(ds1,3)` and `main(gp1,3)`:

```
| ?- main(ds1,3).                      | ?- main(gp1,3).
A most preferred trajectory is:        A most preferred trajectory is:
    + walk(home,stA)                       + walk(home,stC)
    + bus(stA,stB)                         + subway(stC,stB)
    + walk(stB,office)                     + walk(stB,office)
```

4 Experiments

We compared CPP with the ASP planner in [13] on the blocks world domain (`Block`). In this domain, there are $m \times n$ blocks, numbered from $1 \ldots m \times n$. Initially, the blocks are organized in m piles, each having n blocks. The i-th pile consists of n blocks that are numbered $(1+(i-1)\times n) \ldots (i \times n)$, where the blocks with smaller numbers are on top of the blocks with larger numbers. The goal is to have a pile of blocks from 1 to $m \times n - 1$ where blocks with larger numbers are on top of ones with smaller numbers. The location of the remaining block (block number $m \times n$), can be anywhere (i.e., on the table, on block $m \times n - 1$, or below block 1). For this domain, we tested with preferences $\psi_1, \ldots, \psi_8$ defined as follows. The first four preferences are basic desires. ψ_1 is a goal preference of having the last block (block number $m \times n$) on the top of block $m \times n - 1$ in the final state. ψ_2 is also a goal preference but it prefers to have the last block under block 1. ψ_3 and ψ_4 are state desires. ψ_3 prefers to never place any block except block 1 on the table; ψ_4 states that eventually block 1 is on block $m \times n$. ψ_5 is the atomic preference $\psi_1 \lhd \psi_2$, and ψ_6 is the atomic preference $\psi_3 \lhd \psi_4$. ψ_7 and ψ_8 are the general preferences $\psi_4 \& \psi_5$ and $\psi_8 = \psi_4 | \psi_5$.

All the experiments have been conducted on a 2.4 GHz CPU, 768MB RAM machine. Time out is set to 10 minutes. The test results are shown in Table 1. In the table, N is the length of plans that we wish to find. In each cell of the

Table 1. Performance of CPP vs ASPlan

Domain	N	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	ψ_6	ψ_7	ψ_8
Block(1,4)	4	0.00 1.72	0.00 1.68	0.00 1.85	0.00 1.73	0.00 1.72	0.00 1.73	0.00 1.72	0.00 1.77
	5	0.02 1.79	0.00 1.75	0.00 1.78	0.00 1.78	0.02 1.81	0.02 1.83	0.02 1.85	0.02 1.81
	6	0.00 1.91	0.00 1.87	0.02 2	0.02 1.98	0.00 1.97	0.03 1.99	0.02 1.98	0.02 2.02
	7	0.02 2.11	0.00 2.07	0.02 2.29	0.03 2.36	0.00 2.91	0.03 2.16	0.03 2.1	0.03 2.18
Block(1,5)	5	0.00 5.65	0.02 5.51	0.02 5.63	0.02 5.73	0.00 5.53	0.02 5.65	0.02 5.73	0.02 5.74
	6	0.02 6.85	0.03 5.72	0.03 5.76	0.02 5.97	0.02 5.99	0.05 5.89	0.05 6.11	0.03 6.18
	7	0.03 6.15	0.03 6.27	0.05 6.19	0.06 6.13	0.03 6.18	0.09 6.32	0.05 6.53	0.06 6.45
	8	0.08 6.53	0.03 6.51	0.08 6.69	0.08 6.56	0.06 6.56	0.14 6.71	0.06 7.14	0.08 7.28
Block(1,6)	6	0.02 17.14	0.02 17.19	0.31 17.21	0.02 16.82	0.02 16.83	0.03 16.99	0.03 16.75	0.03 20.43
	7	0.06 17.6	0.08 17.64	0.16 17.7	0.09 17.56	0.09 17.72	0.17 17.59	0.14 17.62	0.14 17.51
	8	0.11 18.2	0.14 18.42	0.19 18.44	0.28 18.35	0.14 18.41	0.31 18.38	0.22 18.45	0.24 18.46
	9	0.42 19.26	0.39 19.87	0.28 19.17	0.56 19.3	0.47 19.37	0.52 19.28	0.66 19.56	0.67 19.48
	10	2.00 20.98	0.28 20.4	0.53 20.02	0.66 20.09	2.33 20.3	0.94 20.45	2.02 20.22	2.08 20.24
Block(1,7)	9	0.48 32.66	0.47 32.33	0.67 33.13	0.78 32.53	0.5 32.59	0.97 32.86	0.52 32.82	0.52 32.52
	10	1.94 34.84	1.5 34.82	0.86 34.72	1.81 34.97	2.28 34.5	1.55 34.4	1.72 38.86	1.72 35.18
	11	10.78 38.7	6.00 38.65	1.97 38.58	7.27 38.27	11.02 38.1	2.81 38.91	7.64 38.18	7.5 38.23
	12	66.28 42.47	3.34 42.12	2.61 42.66	5.3 42.36	76.24 42.66	5.08 45.64	40.67 42.71	41.31 43.62
Block(2,3)	5	0.2 16.45	0.23 16.84	0.36 16.85	0.38 16.74	0.24 16.73	0.55 17.19	0.56 16.69	0.66 17.85
	6	0.12 17.57	0.18 17.27	0.2 17.3	0.47 17.62	0.11 17.3	0.44 17.42	0.22 18.13	0.2 22.98
	7	0.25 19.51	0.31 19.24	0.23 19.24	0.75 19.01	0.27 19.19	0.38 19.25	0.34 19.02	0.33 19.37
	8	0.82 20.41	1.15 20.71	0.28 20.73	1.92 20.54	0.81 20.45	0.89 20.53	1.05 20.28	1.05 20.54
Block(2,4)	7	3.95 78.97	4.28 77.21	3.06 79.95	5.69 78.77	7.84 78.76	5.12 77.63	5.86 77.75	5.82 79.04
	8	1.26 82.53	1.72 80.87	1.08 138.38	5.51 86.54	2.15 80.8	2.65 81.05	1.4 82.08	1.4 82.56
	9	7.2 89.53	7.75 88.68	1.34 91.09	27.56 89.17	8.75 90.09	4.97 90.68	4.93 90.26	4.98 89.21
	10	44.58 99.02	33.75 96.81	1.58 98.24	43.42 96.64	38.79 96.08	8.95 98.7	27.15 98.46	27.12 98.34

table, the first row and the second row shows the solving times (in seconds) for CPP and ASPlan, respectively, to return a most preferred plan; TO indicates a timeout.

As can be seen from Table 1, CPP on the block world domain outperforms ASPlan on most of instances. When the number of blocks is less than or equal to 6 (problems $Block(1,4)$ and $Block(1,5)$), the solving time for CPP is negligible (less than $0.1s$), while that for ASPlan is in range from 1.7 to 7.2 seconds. However, when the number of blocks increases to more than 6 (instances $Block(1,6)$, $Block(1,7)$, $Block(2,3)$, and $Block(2,4)$), the solving time for CPP increases exponentially but is still much less than the solving time for ASPlan on most of instances.

5 Conclusion and Future Work

This paper describes a CLP based system, called CPP, for computing most preferred plans with respect to a user's preference. The preliminary results are encouraging and suggest a valid alternative for reasoning with actions and preferences. Our work is somewhat related to the work in [3] in the sense that planning problems with preferences are translated into constraint satisfaction problems. The main difference is that the work in [3] can handle preferences and constraints over *goals* only; they cannot handle preferences over *trajectories* of plans.

The system has been encoded in GNU Prolog. It is worth noting that there are other constraint programming systems, and the performance of a constraint program heavily depends on the encoding of the problem and on the underlying solver. Hence, as future work, we would like to try exploring encodings of CPP on different systems. We would also like to investigate the usefulness of heuristics and the applicability of *Constraint Handling Rules* [8] to improve the performance and extensibility of CPP. In addition, we would like to extend CPP to deal with non-deterministic and/or incomplete action theories. This involves extending the preference language $\mathcal{PP}$ so as to be able to compare plans in non-deterministic and/or incomplete action theories.

References

1. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer Academic Publishers (2000) 257–279
2. Bienvenu, M., Fritz, C., McIlraith, S.: Planning with qualitative temporal preferences. In KR, Lake District, UK (2006)
3. Brafman, R.I., Chernyavsky, Y.: Planning with goal preferences and constraints. In ICAPS, (2005) 182–191
4. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based Planning in Complex Domains: Concurrency, Constraints and Nondeterminism. AI **147**(1-2) (2003) 85–117
5. Delgrande, J.P., Schaub, T., Tompits, H.: Domain-specific preferences for causal reasoning and planning. In KR, (2004) 673–682

6. Diaz, D., Codognet, P.: Design and implementation of the GNU prolog system. Journal of Functional and Logic Programming **2001**(6) (2001)
7. Dovier, A., Formisano, A., Pontelli, E.: A comparison of CLP(FD) and ASP solutions to NP-complete problems. In ICLP, (2005)
8. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (1998) 95–138
9. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3. Technical Report RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia (2005)
10. Giunchiglia, E., Lifschitz, V.: An action language based on causal explanation: preliminary report. In: Proceedings of AAAI 98. (98) 623–630
11. Jaffar, J., Maher, M.: Constraint Logic Programming. JLP **19/20** (1994)
12. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92). (1992) 359–363
13. Son, T.C., Pontelli, E.: Planning with Preferences using Logic Programming. Theory and Practice of Logic Programming (2006) To Appear.
14. Tu, P.H., Son, T.C, Pontelli, E.: Planning with Preferences Using Constraint Logic Programming. Technical Report NMSU-CS-TR-2007-001, New Mexico State University (2007) http://www.cs.nmsu.edu/CSWS/techRpt/tr07-prefs.pdf.
15. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press (1989)

An Application of Defeasible Logic Programming to Decision Making in a Robotic Environment

Edgardo Ferretti[1], Marcelo Errecalde[1],
Alejandro J. García[2,3], and Guillermo R. Simari[3]

[1] Laboratorio de Investigación y Desarrollo en Inteligencia Computacional
Universidad Nacional de San Luis, Argentina
{ferretti,merreca}@unsl.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
[3] Department of Computer Science and Engineering
Universidad Nacional del Sur, Bahía Blanca, Argentina
{ajg,grs}@cs.uns.edu.ar

Abstract. Decision making models for autonomous agents have received increased attention, particularly in the field of intelligent robots. In this paper we will show how a Defeasible Logic Programming approach with an underlying argumentation based semantics, could be applied in a robotic domain for knowledge representation and reasoning about which task to perform next. At this end, we have selected a simple application domain, consisting of a micro-world environment using real and simulated robots for cleaning tasks.

1 Introduction

In this paper we will show how a Defeasible Logic Programming approach could be applied in a robotic domain for knowledge representation and reasoning about which task to perform next. At this end, we have selected a simple application domain, consisting of a micro-world environment using real and simulated robots for cleaning tasks. We use the *Khepera* 2 robot [1], a miniature mobile robot ideal for this kind of experimentation. We also use a professional simulator (see Fig. 1) called *Webots* [2], which allows behavior simulation prior to physical experimentation with the robot.

The experimental environment (see Fig. 1(a)) is a square arena of 100 units per side which is conceptually divided into square cells of 10 units per side each. There is a global camera which provides the necessary information to perform their activities. The *store* is a 30×30 units square on the top-right corner and represents the target area where boxes should be transported. There are boxes of three different sizes (*small*, *medium* and *big*) spread over the environment.

As the robot is not able of measuring the state of its battery, it cannot perform a globally optimized task. In this way, the robot will reason about which box is more convenient to select next trying to minimize the time spent in moving boxes. To reason, the robot will use perceptual information about the boxes and its preferences (represented with defeasible rules). For example, the robot could prefer the smallest box, or the nearest one, or the box that it is nearest to the store. As we will show below, arguments for and against selecting a box will be considered to select the more appropriate one.

A robot capable of solving this kind of problems must at least address the following issues: to perceive the surrounding world, to decide which goal to reach and to have the capabilities for reaching this goal. Several architectures providing the agents with these skills have been proposed [3,4,5]. In this work, we only consider the necessary reasoning processes to make decisions about which is the most suitable box to be transported by the robots. We will not address the low-level aspects related to sensorial perception and the implementation of low-level actions for the Khepera robots, because they have been presented elsewhere [6].

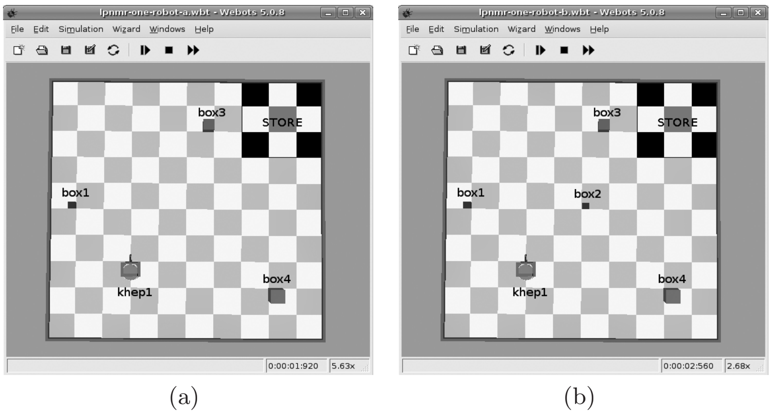

lpnmr-one-robot-a.wbt - Webots 5.0.8
File Edit Simulation Wizard Windows Help
box3
STORE
box1
khep1
box4
0:00:01:920
5.63x
(a)
lpnmr-one-robot-b.wbt - Webots 5.0.8
File Edit Simulation Wizard Windows Help
box3
STORE
box1
box2
khep1
box4
0:00:02:560
2.68x
(b)

Fig. 1. Two possible different environments

2 Knowledge Representation and Defeasible Reasoning

Figure 1(a) shows an example with one robot (*khep*1) and three boxes to be carried: a small one (*box*1) near to the robot, *box*3 that is medium size and it is near to the store, and *box*4 that is big and it is far from both, robot and store. Taking into account its preferences the robot will consider reasons for and against selecting each box. We will refer to these reasons as *arguments*.

For example, there is an argument *for* selecting *box*3 because *"it is near to the store"* but there is an argument *against* selecting *box*3 because *box*1 *"is near to the robot and it is smaller than box*3*."* As it will be shown below a dialectical analysis involving arguments and counter-arguments will be performed to decide which argument prevails. In this case *box*1 will be chosen, because it is the smallest box near to the robot. Since the environment is dynamic, when it changes new arguments could be generated and others could be invalidated. Thus, the robot might select different boxes in different circumstances. For instance, let us consider Fig. 1(b), that differs from Fig. 1(a) in that there is one more small box ($box2$) in the environment. Here, in the new situation, the robot $khep1$ will choose $box2$ because it has a new argument against selecting $box1$: *"there is another small box ($box2$) that it is nearer to the store than $box1$."*

The robot's knowledge about the environment and its preferences for selecting a box will be represented using Defeasible Logic Programming (DeLP) a formalism that combines logic programming and defeasible argumentation (for a detailed presentation see [7]). In DeLP, knowledge is represented using facts, strict rules or defeasible rules. *Facts* are ground literals representing atomic information or the negation of atomic information using the strong negation "$\sim$". *Strict Rules*, are denoted $L_0 \leftarrow L_1, \ldots, L_n$, where the *head* L_0 is a ground literal and the *body* $\{L_i\}_{i>0}$ is a set of ground literals. In the same way, *Defeasible Rules*, are denoted $L_0 \prec L_1, \ldots, L_n$, where the *head* L_0 is a ground literal and the *body* $\{L_i\}_{i>0}$ is a set of ground literals. In this work, facts will be used for representing perceptual information about the environment, (*e.g.*, $box(box2)$ or $near(box2, store)$), strict rules will be used for representing non-defeasible information (*e.g.*, $\sim far(box1, khep1) \leftarrow near(box1, khep1)$), and defeasible rules will be used for representing tentative reasons for (or against) selecting a box (*e.g.*, $choose(X) \prec small(X)$). The symbol "$\prec$" distinguishes a defeasible rule from a strict one with the pragmatic purpose of using a defeasible rule to represent defeasible knowledge, *i.e.*, tentative information.

A Defeasible Logic Program $\mathcal{P}$ is a set of facts, strict rules and defeasible rules. When required, $\mathcal{P}$ is denoted (Π, Δ) where $\Pi = \Pi_f \cup \Pi_r$, distinguishing the subset Π_f of facts, strict rules Π_r and the subset Δ of defeasible rules. Observe that strict and defeasible rules are ground following the common convention [9]. Some examples will use "schematic rules" with variables. As usual in Logic Programming, variables are denoted with an initial uppercase letter.

Strong negation is allowed in the head of program rules, and hence may be used to represent contradictory knowledge. From a program (Π, Δ) contradictory literals could be derived, however, the set Π (which is used to represent non-defeasible information) must possess certain internal coherence. Therefore, Π has to be non-contradictory, *i.e.*, no pair of contradictory literals can be derived from Π. Given a literal L the complement with respect to strong negation will be denoted $\overline{L}$ (*i.e.*, $\overline{a} = \sim a$ and $\overline{\sim a} = a$). To deal with contradictory and dynamic information, in DeLP, *arguments* for conflicting pieces of information are built and then compared to decide which one prevails. The prevailing argument provides a *warrant* for the information that it supports (A DeLP interpreter satisfying the semantics of [7] is accessible online at http://lidia.cs.uns.edu.ar/DeLP).

In DeLP a literal L is *warranted* from (Π, Δ) if a non-defeated argument $\mathcal{A}$ supporting L exists. An *argument* for a literal L, denoted $\langle \mathcal{A}, L \rangle$, is a minimal set of defeasible rules $\mathcal{A} \subseteq \Delta$, such that $\mathcal{A} \cup \Pi$ is non-contradictory and there is a derivation for L from $\mathcal{A} \cup \Pi$. To establish if $\langle \mathcal{A}, L \rangle$ is non-defeated, *argument rebuttals* or *counter-arguments* that could be *defeaters* for $\langle \mathcal{A}, L \rangle$ are considered, *i.e.*, counter-arguments that by some criterion are preferred to $\langle \mathcal{A}, L \rangle$. Since counter-arguments are arguments, defeaters for them may exist, and defeaters for these defeaters, and so on. Thus, a sequence of arguments called *argumentation* *line* is constructed, where each argument defeats its predecessor in the line. Given a query Q there are four possible answers: YES, if Q is warranted; NO, if the complement of Q is warranted; UNDECIDED, if neither Q nor its complement are warranted; and UNKNOWN, if Q is not in the language of the program.

3 Robot Decision Making: A Simple Example

In this section we describe the components used by the robot to decide which box to transport next. Consider the situation depicted in Fig. 1(b). The knowledge of the robot, referring to this particular scenario, will be represented with the defeasible logic program $\mathcal{P} = (\Pi,\Delta)$ shown in Fig. 2.

$robot(khep1)$ (1)
$self(khep1)$ (2)
$box(box1)$ (3)
$box(box2)$ (4)
$box(box3)$ (5)
$box(box4)$ (6)
$small(box1)$ (7)
$small(box2)$ (8)
$medium(box3)$ (9)
$big(box4)$ (10)
$near(box1, khep1)$ (11)
$near(box2, khep1)$ (12)
$near(box2, store)$ (13)
$near(box3, store)$ (14)
$far(box1, store)$ (15)
$far(box3, khep1)$ (16)
$far(box4, store)$ (17)
$far(box4, khep1)$ (18)

(a) Π_f

$\sim near(X, O) \leftarrow far(X, O)$ (19)
$smaller(X, Y) \leftarrow small(X), medium(Y)$ (20)
$smaller(X, Y) \leftarrow small(X), big(Y)$ (21)
$smaller(X, Y) \leftarrow medium(X), big(Y)$ (22)
$\sim smaller(X, Y) \leftarrow same\_size(X, Y)$ (23)
$same\_size(X, Y) \leftarrow small(X), small(Y)$ (24)
$same\_size(X, Y) \leftarrow medium(X), medium(Y)$ (25)
$same\_size(X, Y) \leftarrow big(X), big(Y)$ (26)

(b) Π_r

$choose(X) \mathrel{-\!\!\prec} near(X, store)$ (27)
$choose(X) \mathrel{-\!\!\prec} self(Z), near(X, Z)$ (28)
$\sim choose(X) \mathrel{-\!\!\prec} self(Z), near(Y, Z), near(X, store), diff(X, Y)$ (29)
$\sim choose(X) \mathrel{-\!\!\prec} big(X)$ (30)
$choose(X) \mathrel{-\!\!\prec} big(X), self(Z), near(X, Z)$ (31)
$choose(X) \mathrel{-\!\!\prec} big(X), near(X, store)$ (32)
$choose(X) \mathrel{-\!\!\prec} small(X)$ (33)
$\sim choose(X) \mathrel{-\!\!\prec} small(X), far(X, store), self(Z), far(X, Z)$ (34)
$\sim choose(X) \mathrel{-\!\!\prec} self(Z), near(X, Z), near(Y, Z), near(Y, store),$
$diff(X, Y), same\_size(X, Y)$ (35)
$\sim choose(X) \mathrel{-\!\!\prec} choose(Y), smaller(Y, X)$ (36)

(c) Δ

Fig. 2. Defeasible Logic program $\mathcal{P} = (\Pi,\Delta)$

The defeasible rules of Δ describe the robot's preferences about which box to choose in different situations. In this case, the defeasible rules model preference criteria with respect to the size and location of the boxes. For instance, rules (27) and (28) represent the robot's preferences on those boxes near to the store or near to itself. Moreover, rule (29) states that boxes near to the robot are more desirable than those near to the store. Furthermore, rules (30)-(34) represent the preferences of the robot with respect to the boxes' size as well as in which situations, boxes of a determined size are eligible. In addition, rule (35) states that when two boxes of the same size, X and Y, are near to the robot, but Y is also near to the store, then Y is preferred over X. Finally, rule (36) provides defeasible reasons not to choose X if a smaller box Y was already chosen.

In the scenario depicted in Fig. 1(b), the robot should choose *box*2 because it is near to itself and it is also near to the store. Observe that although from $\mathcal{P}$, there are two arguments ($\mathcal{A}_1$ and $\mathcal{A}_2$) supporting *choose*(*box*1), these arguments are defeated by $\mathcal{A}_3$. Thus, the answer for *choose*(*box*1) is NO.

$$\mathcal{A}_1 = \{\, choose(box1) \mathrel{-\!\!\prec} small(box1) \,\} \quad \mathcal{A}_2 = \{\, choose(box1) \mathrel{-\!\!\prec} self(khep1), near(box1, khep1) \,\}$$

$$\mathcal{A}_3 = \left\{ \begin{array}{c} \sim choose(box1) \mathrel{-\!\!\prec} self(khep1), near(box1, khep1), near(box2, khep1), near(box2, store), \\ diff(box1, box2), same\_size(box1, box2) \end{array} \right\}$$

The interaction among these arguments is shown below (where white triangles represent defeated arguments, black triangles non-defeated ones, and arrows the defeat relation). From $\mathcal{P}$ three arguments *for choose*(*box*2) can be obtained ($\mathcal{A}_4$, $\mathcal{A}_5$ and $\mathcal{A}_6$) and one argument *against* it ($\mathcal{A}_7$). Here, $\mathcal{A}_7$ is a blocking defeater of $\mathcal{A}_4$ and $\mathcal{A}_5$ and it is a proper defeater of $\mathcal{A}_6$, but $\mathcal{A}_5$ is also a blocking defeater of $\mathcal{A}_7$, therefore, the answer to *choose*(*box*2) is YES because $\mathcal{A}_6$ is defeated by $\mathcal{A}_7$ which is in turn defeated by $\mathcal{A}_5$, reinstating $\mathcal{A}_6$. Finally, the answers for *choose*(*box*3) and *choose*(*box*4) are NO because arguments that use rule (36) are built to support ∼*choose*(*box*3) and ∼*choose*(*box*4).

$$\mathcal{A}_4 = \{\, choose(box2) \mathrel{-\!\!\prec} small(box2) \,\} \quad \mathcal{A}_5 = \{\, choose(box2) \mathrel{-\!\!\prec} self(khep1), near(box2, khep1) \,\}$$

$$\mathcal{A}_6 = \{\, choose(box2) \mathrel{-\!\!\prec} near(box2, store) \,\}$$

$$\mathcal{A}_7 = \{\, \sim choose(box2) \mathrel{-\!\!\prec} self(khep1), near(box1, khep1), near(box2, store), diff(box2, box1) \,\}$$

$$\begin{tikzcd}
A_1\ \triangle & A_2\ \triangle & A_7\ \triangle & A_4\ \triangle & A_5\ \triangle & A_6\ \blacktriangle \\
A_3\ \blacktriangle \arrow[u] & A_3\ \blacktriangle \arrow[u] & A_5\ \blacktriangle \arrow[u] & A_7\ \blacktriangle \arrow[u] & A_7\ \blacktriangle \arrow[u] & A_7\ \triangle \arrow[u] \\
 & & & & & A_5\ \blacktriangle \arrow[u]
\end{tikzcd}$$

4 Related Work

Our proposal is closely related to the approach adopted by Parsons *et al.* [10]. In particular, in our work we follow some ideas exposed in [10] about the integration of high-level reasoning facilities with low-level robust robot control. We share the approach of seeing the low-level module as a black box which receives goals to be achieved from the high-level component, and plans to reach goals are internally generated. However, our work differs from the proposal of [10] in that we do not use a BDI deliberator as high-level reasoning layer, instead we use a non-monotonic reasoning module based on a defeasible argumentation system.

With respect to this last issue, our approach to decision making is related to other works which use argumentative processes as a fundamental component in the decision making of an agent [11,12,13]. It is important to note that these argumentation systems have been usually integrated in *software* agents. On the other hand, in our approach, defeasible argumentation is applied in a robotic domain where the uncertainty generated by noisy sensors and effectors, changes in the physical environment and incomplete information about it, make this kind of problems a more challenging test-bed for the decision processes of an agent.

5 Conclusions and Future Work

In this paper we have shown how a Logic Programming approach could be applied in a robotic domain for knowledge representation and reasoning about which task to perform next. Our approach considers the ability of Defeasible Logic Programming to reason with incomplete and potentially inconsistent information. The simple application domain described consists of a micro-world environment using real and simulated robots for cleaning tasks. We have presented a problem and its solution when there is only one robot in the environment. Future work includes considering more complex environments, such as more than one robot with different abilities working in the same environment with the inclusion of obstacles.

Acknowledgment

We thank the Universidad Nacional de San Luis and the Universidad Nacional del Sur for their unstinting support. This work is partially supported by, CONICET (PIP 5050), and ANPCyT (PICT 2002, Nro.13096 and Nro.12600).

References

1. K-Team: Khepera 2. http://www.k-team.com (2002)
2. Michel, O.: Webots: Professional mobile robot simulation. Journal of Advanced Robotics Systems **1**(1) (2004) 39–42
3. Gat, E.: On three-layer architectures. In: Artificial Intelligence and Mobile Robots. (1998)
4. Estlin, T., Volpe, R., Nesnas, I., Muts, D., Fisher, F., Engelhardt, B., Chien, S.: Decision-making in a robotic architecture for autonomy. In: International Symposium, on AI, Robotics and Automation for Space. (2001)
5. Rotstein, N.D., García, A.J.: Defeasible reasoning about beliefs and desires. In: Proc. of the 11th Int. Workshop on Non-Monotonic Reasoning. (2006) 429–436
6. Ferretti, E., Errecalde, M., García, A., Simari, G.: Khedelp: A framework to support defeasible logic programming for the khepera robots. In: ISRA06. (2006)
7. García, A.J., Simari, G.R.: Defeasible logic programming: an argumentative approach. Theory and Practice of Logic Programming (2004)
8. Simari, G.R., Loui, R.P.: A mathematical treatment of defeasible reasoning and its implementation. Artificial Intelligence (1992)
9. Lifschitz, V.: Foundations of logic programming. In: Principles of Knowledge Representation. CSLI (1996)
10. Parsons, S., Pettersson, O., Saffiotti, A., Wooldridge, M.: Robots with the Best of Intentions. In: Artificial Intelligence Today: Recent Trends and Developments. Springer (1999)
11. Atkinson, K., Bench-Capon, T.J.M., Modgil, S.: Argumentation for decision support. In: DEXA. (2006) 822–831
12. Kakas, A., Moraitis, P.: Argumentation based decision making for autonomous agents. In: AAMAS. (2003)
13. Parsons, S., Fox, J.: Argumentation and decision making: A position paper. In: FAPR. (1996)

On the Effectiveness of Looking Ahead in Search for Answer Sets

Guohua Liu and Jia-Huai You

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

Abstract. Most complete SAT/ASP solvers are based on DPLL. One of the constraint propagation methods is the so-called *lookahead*, which has been somewhat controversial, due to its high overhead. In this paper, we show characterizations of the problems for which lookahead is ineffective, and demonstrate, experimentally, that for problems that lie in the phase transition regions, search efficiency can be improved significantly by lookahead. This understanding leads to the proposal of a mechanism called *adaptive lookahead*, which decides when lookahead should be invoked dynamically upon learned information. Our experiments show that adaptive lookahead adapts well to different situations where lookahead may or may not be beneficial.

1 Introduction

Most complete SAT/ASP solvers are based on DPLL [2]. *Unit propagation*, also called *boolean constraint propagation* (BCP) [3], is considered the most important constraint propagation technique in a DPLL search engine [11]. In answer set solver Smodels [8], the component that corresponds to BCP is called the *Expand* function.

On top of BCP/*Expand*, other deductive mechanisms may be adopted. One of these is called *lookahead* [3] - before a decision on a choice point is made, for each atom, if fixing the atom's truth value leads to a contradiction, the atom gets the opposite truth value. In this way, such an atom gets a truth value from the truth value propagation of the already assigned atoms without going through a search process.

Lookahead, however, incurs high overhead [10] and has been shown ineffective in some SAT solvers [6]. The high pruning power, along with non-ignorable overhead, has made lookahead a controversial technique. There are two camps of constraint solvers. In one of them lookahead is employed during the search and in the other it is not.

This paper investigates the effectiveness of lookahead, namely how to exploit its pruning power and avert the unnecessary overhead. We choose the well-known answer set solver Smodels as our experimental system. We report the following findings. First, we show some characterizations of the programs for which lookahead is ineffective, and identify representative benchmarks in which the situation arises. Second, we show experimentally that lookahead can significantly increase search efficiency in solving some hard problem instances especially those in their phase transition regions. Third, we propose a mechanism called *adaptive lookahead*, which turns lookahead on and off dynamically upon learned information. We implemented adaptive lookahead in Smodels, which adapts well to different search environments it is going through.

Next section provides the background. In Section 3 we identify problems that run much slower with lookahead and discuss the reasons. Section 4 introduces the algorithm of adaptive lookahead. Section 5 provides experimental results, with final remarks given in Section 6.

2 Constraint Propagation in Smodels

The class of logic programs considered here is that of normal programs, which are collections of program rules of the form $h \leftarrow a_1, ..., a_m, \textbf{not } b_1, ..., \textbf{not } b_n$, where h, a_i, $1 \leq i \leq m$ and b_i, $1 \leq i \leq n$ are ground atoms, or *positive literals* and **not** b_i are *negative literals*. For an atom a, $not(\textbf{not } a)$ yields a. The *answer sets* of a program are defined in [5].

A set of literals is *consistent* if there is no atom a such that a and **not** a are both in the set, otherwise there is a *conflict*. A *partial assignment* is a consistent set of literals.

A *choice point* is defined as a point during search where the branching heuristic picks a literal. In the literature, this is also referred to as *making a decision*. Smodels performs constraint propagation before a decision is made. When lookahead is not involved, constraint propagation is carried out by a function called $expand(P, A)$, where P is a program and A a partial assignment. When lookahead is employed, constraint propagation is carried out as follows: for each unassigned atom a, assumes a truth value for it. If this leads to a conflict, then a gets the opposite truth value. This process continues, repeatedly, until no atom can be fixed a truth value by lookahead. Truth values are propagated in lookahead by $expand(P, A)$, which returns a superset of A, representing the process of propagating the values of the atoms in A to some additional atoms. For more details, the reader is referred to [8].

3 Situations Where Overhead Dominates

Easy sub-program. Given a program, when searching for a solution it could happen that no pruning is ever generated by lookahead in the course of solving some parts of the problem. This could be caused by easy sub-programs described as follows.

Let P' and P be two ground programs, P' is called a *sub-program* of P if $P' \subseteq P$. A program P is said to be *easy* if any partial assignment can be extended to a solution. In the process of solving an easy program lookahead is totally wasted in that, if $expand(P, A) = A$ then $lookahead(P, A) = A$, for any partial assignment A generated during the search.

A hard program may contain many easy sub-programs. An example is the pigeonhole problem, which is known to be hard for resolution-based solvers. When the number of holes is smaller than the number of pigeons by a small number, e.g. by one, the corresponding program has lots of easy sub-program, which can be obtained by removing some facts about pigeons so that the resulting number of pigeons equals to or less than the number of holes. The process of lookahead for solving such a sub-program is entirely useless.

Spurious Pruning. When lookahead finds a conflict, some search space is pruned. But some pruning may be immaterial to the rest of the search. Suppose, by an invocation of lookahead, a literal, say l, is added to the current partial assignment A. The addition of l may not contribute to further constraint propagations. This can be described by the following equation: $expand(P, A) \cup \{l\} = expand(P, A \cup \{l\})$. In this case, lookahead is unnecessary since the decision on l can be delayed to any later choice point.

We can use the number of calls to the *Expand* function during search to measure the effectiveness of lookahead. Let us use N_{lh} and N_{nlh} to denote the number of calls to *Expand* in Smodels with and without lookahead, respectively. It is easy to see, for a given program P, if all the pruning by lookahead are spurious, then $N_{nlh} \leq N_{lh}$.

4 Adaptive Lookahead

Adaptive lookahead is designed to avoid lookahead when its use tends to be ineffective. Two pieces of information are useful for this purpose. One is the number of failed literals (literals whose addition to the current partial assignment cause a conflict by the expand function). Note that a failed literal does not necessarily cause backtracking, since it could be the case that the negation of the failed literal is consistent with current partial assignment. Another is the number of dead-ends (where backtracking is needed) detected during the search. The idea is that if after some runs failed literals have been rare, it is likely that the search is in a space where pruning is insignificant, and likely to remain so for sometime to come, so lookahead is turned off; if dead-ends have been frequently encountered after some runs, it is likely that the search has entered into a space where pruning can be significant, so lookahead is turned on.

We call the resulting system *adaptive smodels* The control of lookahead is realized by manipulating two scores, $look\_score$ and $dead\_end\_counter$. The $look\_score$ is initialized to be some positive number, then deducted each time when lookahead does not detect any conflict. When it becomes zero, lookahead will be turned off. The $dead\_end\_counter$ is initialized to be zero and increased each time when a dead-end is encountered. Lookahead will be turned on if $dead\_end\_counter$ reaches some threshold. The counters will be reset after each turn.

In addition to the on/off control, lookahead will never be used in late search processes if it cannot detect any conflict after a number of atoms have been assigned. This is because the search efficiency cannot be improved much by lookahead if the conflicts it detects only happen in late stages of the search. The lateness is measured by a ratio of the number of assigned literals to the number of literals in the program.

The initial value for $look\_score$, the amount of increase/decrease, and the thresholds are all determined empirically. We set the amount of increase/decrease to 1, $look\_score$ to 10, the thresholds of $dead\_end\_counter$ and $ratio$ to 1 and 0.8, respectively.

5 Experiments

The experiments serve three purposes. First, they confirm our findings of the problems where the performance is significantly deteriorated by the use of lookahead; second, they show that lookahead tends to be very effective for hard programs, especially for those that lie in the known regions of phase transition; and third, adaptive lookahead behaves as if it “knows” when to employ lookahead and when not to.

We run Smodels 2.32 with lookahead, without lookahead, and with adaptive lookahead, respectively. All of the experiments are run on Red Hat Linux AS release 4 with 2GHz CPU and 2GB RAM.

5.1 Cases Where Lookahead Improves Search Efficiency

Graph coloring. We use Culberson’s flat graph generator [1] to generate graph instances. In these graphs, each pair of vertices is assigned an edge with an independent identity probability p. We use the suggested value of p to sample across the “phase transition”. The number of colors is 3 and the number of vertices of the graph is 400. The cutoff time is 3 hours. For each measure point, we generate 100 instances and the average running time is reported.

The experiments show that lookahead drastically speeds up the search in the hard region. In the easier regions, the effectiveness of lookahead is insignificant (Fig. 1). The savings by lookahead can also be measured by the number of calls to *Expand* (Fig. 2.[1])

Random 3-SAT. Another problem with well-known phase transition is random 3-SAT. We test instances around its known phase transition region [7]. The results are similar to graph coloring. The data are omitted for lack of space.

Blocks-world. The blocks-world problem is a typical planning problem. The instances we use are generated as follows. For n blocks, b_1, ..., b_n, the initial configuration is b_1 on the table and b_{i+1} on b_i for $1 \leq i \leq n-1$. The goal is b_n on the table, b_1 on b_n and b_{i+1} on b_i for $1 \leq i \leq n-2$. Under this setting, each block in the initial state has to be moved to get to the goal state, so the problem turns out to be nontrivial. The minimum steps needed is $2n-2$.

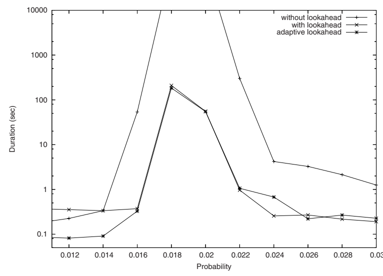


Fig. 1. Running time for graph coloring

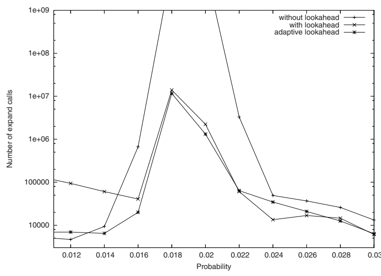


Fig. 2. Number of $expand$ calls

Grippers. The grippers problem is another intensively studied planning problem. It’s interesting partly because the inherent symmetry in the problem causes a domain independent planner to waste a lot of time on exploring symmetric search subtrees.

[1] In comparison, the number of choice points is usually not a good indicator, as a reduction may be achieved in the expense of a huge overhead.

Table 1. Blocks-world Problem

b	s	No Lookahead	Lookahead	A-Lookahead
14	26	165.88	30.49	7.34
	25	359.65	5.45	5.46
15	28	335.11	53.07	10.54
	27	4673.35	7.35	7.35
16	30	375.20	6276.28	14.66
	29	8585.08	10.15	10.50
17	32	1197.65	145.59	21.24
	31	701.86	16.48	16.40
18	34	–	145.48	29.28
	33	–	21.42	21.39

Table 2. Gripper problem

R1	R2	s	No Lookahead	Lookahead	A-Lookahead
4	0	7	0.1	0.43	0.41
		6	0.09	0.17	0.17
5	0	11	8.31	77.35	64.94
		10	1824.55	189.40	97.12
6	0	11	263.75	1117.02	1084.31
		10	2345.49	490.93	487.64
3	3	11	0.70	15.80	15.98
		10	77.93	15.64	15.52
4	4	12	1749.35	419.42	412.64
		11	5463.57	175.61	175.73

Table 3. Pigeon hole

p	h	No Lookahead	Lookahead	A-Lookahead
5	4	0.00	0.01	0.01
6	5	0.00	0.02	0.01
7	6	0.02	0.06	0.02
8	7	0.13	0.49	0.11
9	8	1.18	4.38	0.94
10	9	12.10	43.66	9.78
11	10	137.06	480.19	112.47
12	11	1608.41	5439.09	1343.93

Table 4. Hamiltonian cycle

n	No Lookahead	Lookahead	A-Lookahead
50	2.25	64.92	8.57
60	3.91	183.59	19.74
70	6.31	467.28	40.18
80	9.58	986.85	74.02
90	13.77	1862.86	123.66
100	19.12	3522.24	194.24
110	25.56	6129.59	288.53
120	33.36	7255.97	412.58

The goal of the problem is to transport all the balls from room $R1$ to room $R2$. To accomplish this, a robot is allowed to move from one room to the other, pick up and put down a ball. Each gripper of the robot can hold one ball at a time.

In our experiments two kinds of settings are used. In the first, all of the balls are in $R1$ initially and $R2$ in the goal state. In the second, there are equal number of balls in $R1$ and $R2$ initially and in the goal state, the balls initially in $R1$ are in $R2$ and initially in $R2$ are in $R1$.

The use of lookahead is very helpful in both of these two planning problems, especially when the instances are hard (Tables 1 and 2).

5.2 Cases Where Lookahead Reduces Search Efficiency

We choose the pigeon-hole problem where the number of pigeons is greater than holes by 1 and Hamiltonian cycle problem over complete graphs as the representatives of programs with easy sub-programs and spurious pruning respectively. The experiments show lookahead hugely degrades the search(Table 3, 4).

5.3 Adaptive Smodels

The adaptiveness of adaptive smodels is clearly shown by the experiments. For the problems where lookahead helps, adaptive smodels performs as well as or sometimes even better than smodels (Figures 1, 2 and Tables 1, 2). For the problems where the overhead of lookahead dominates, adaptive smodels works largely as smodels without lookahead (Tables 3, 4).

6 Summary and Future Directions

In this paper, we show that lookahead could be a burden as well as an accelerator to the DPLL search. We analyze why lookahead sometimes slows down the search and characterize the reasons as embedded easy sub-programs and spurious pruning. Based on this analysis, we propose an adaptive lookahead mechanism, which takes the advantage of lookahead while avoiding the unnecessary overhead caused by it.

The effectiveness of lookahead in SAT solvers were studied in [6]. The main conclusion is that lookahead does not pay off when integrated with look-back techniques. This paper focuses on the effect of lookahead on an ASP solver without the mechanism of look-back. Our results are applicable to a SAT solver with lookahead but without look-back in comparison with the same SAT solver with only BCP (e.g., with lookahead turned off). Under this setting, our conclusion is somewhat different from [6]: lookahead can indeed significantly improve search efficiency for some of the extremely hard problem instances.

Some recent answer set solvers [4,9] adopt look-back techniques in ASP solvers. The effect of lookahead in these solvers and the better way to integrate lookahead into them require additional studies.

References

1. J. Culberson. http://web.cs.ualberta.ca/joe/coloring/index.html.
2. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
3. Jon Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
4. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proc. IJCAI'07*, pages 386–392, 2007.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080, 1988.
6. E. Ginuchiglia, M. Maratea, and A. Tacchella. (In)Effectiveness of look-ahead techniques in a modern SAT solver. In *Proc. CP'03*, pages 842–846, 2003.
7. D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proc. AAAI'92*, pages 459–465, 1992.
8. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2), pages 181–234, 2002.
9. J. Ward and J. Schlipf. Answer set programming with clause learning. In *Proc. ICLP'04*, pages 302–313, 2004.
10. J. You and G. Hou. Arc consistency + unit propagation = lookahead. In *Proc. ICLP'04*, pages 314–328, 2004.
11. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. CAV'02*, pages 17–36, 2002.

Enhancing ASP Systems for Planning with Temporal Constraints

Veena S. Mellarkod and Michael Gelfond

Texas Tech University
{veena.s.mellarkod,michael.gelfond}@ttu.edu

Abstract. The paper introduces a collection of knowledge representation languages, $\mathcal{V}(\mathcal{C})$, parametrised over a class $\mathcal{C}$ of constraints. $\mathcal{V}(\mathcal{C})$ is an extension of both CR-Prolog and CASP allowing the separation of a program into two parts: a regular program of CR-Prolog and a collection of denials[1] whose bodies contain constraints from $\mathcal{C}$ with variables ranging over large domains. We study an instance $\mathcal{V}_0$ from this family where $\mathcal{C}$ is a collection of constraints of the form $X - Y > K$. We give some implementation details of an algorithm computing the answer sets of programs of $\mathcal{V}_0$ which does not ground constraint variables and tightly couples the "classical" ASP algorithm with an algorithm checking consistency of difference constraints. This makes it possible to solve problems which could not be solved by pure ASP or constraint solvers.

1 Introduction

The KR language CR-Prolog [4] is an extension of Answer Set Prolog (ASP) [3] which allows natural encoding of "rare events". These events are normally ignored by a reasoner associated with the program and only used to restore consistency of the reasoner's beliefs. CR-Prolog has been shown to be a useful tool for knowledge representation and reasoning [4,6]. Simple CR-Prolog solvers [5] built on top of the ASP solvers: Smodels and Surya[], proved to be sufficiently efficient for building industrial size applications [6]. Neither ASP nor CR-Prolog however, can deal with applications which require a combination of, say, planning and scheduling. This happens because reasoning with time normally requires programs which include variables with rather large numerical domains. ASP and CR-Prolog solvers compute answer sets of a ground instance of the input program. If a program contains variables with large domains such an instance can be too large, which renders the program unmanageable for the solver, despite the use of multiple optimization procedures. A step toward resolving this problem was made in [1], where the authors introduced a language $CASP$, which splits a given program into two modules: regular rules of ASP, not containing variables with large domains, and denials containing such variables which must satisfy constraints from some given class $\mathcal{C}$. In this paper we expand this idea to CR-Prolog. In particular, we introduce a collection, $\mathcal{V}(\mathcal{C})$, of languages parametrised

[1] By a denial we mean a logic programming rule with an empty head.

over a class $\mathcal{C}$ of constraints. $\mathcal{V}(\mathcal{C})$ is an extension of both, CR-Prolog and CASP. We study an instance $\mathcal{V}_0$ of the resulting language where $\mathcal{C}$ is a collection of constraints of the form $X - Y > K$ where X and Y range over integers or reals and K is an integer. We design and implement an algorithm computing the answer sets of programs of $\mathcal{V}_0$ which does not ground constraint variables and tightly couples the classical ASP algorithm with an algorithm checking the consistency of difference constraints. This makes it possible to declaratively solve problems which could not be solved by pure ASP or by pure constraint solvers.

2 Syntax and Semantics of $\mathcal{V}(\mathcal{C})$

2.1 Syntax

The language $\mathcal{V}(\mathcal{C})$ contains a sorted signature Σ, with sorts partitioned into two classes: *regular*, s_r, and *constrained*, s_c. Intuitively, the former are comparatively small but the latter are too large for the ASP grounders. Functions defined on regular (constrained) classes are called r-functions (c-functions). Terms are built as in first-order languages. Predicate symbols are divided into three disjoint sets called *regular*, *constrained* and *mixed* and denoted by P_r, P_c and P_m respectively. Constrained predicate symbols are determined by $\mathcal{C}$. Parameters of regular and constrained predicates are of sorts s_r and s_c respectively. Mixed predicates have parameters from both classes. Atoms are defined as usual. A literal is an atom a or its negation $\neg a$. An extended literal is a literal l or *not* l, where *not* stands for *negation as failure*. Atoms formed from regular, constrained, and mixed predicates are called r-atoms, c-atoms and m-atoms respectively. Similarly for literals. We assume that predicates of P_c have a predefined interpretation, represented by the set M_c of all true ground c-atoms. For instance, if $' >' \in P_c$, and ranges over integers, M_c consists of $\{...0 > -1, 1 > 0, ...\}$. The c-literals allowed in $\mathcal{V}(\mathcal{C})$ depend on the class $\mathcal{C}$. The $\mathcal{V}(\mathcal{C})$ rules over Σ are defined as follows.

Definition 1. [rules]

1. *A regular rule (r-rule) ρ is a statement of the form:*

$$h_1 \text{ or } \cdots \text{ or } h_k \leftarrow l_1, \cdots, l_m, \text{ not } l_{m+1}, \cdots, \text{not } l_n$$

where $k >= 0$; h_i's and l_i's are r-literals.

2. *A constraint rule (c-rule) is a statement of the form:*

$$\leftarrow l_1, \cdots, l_m, \text{ not } l_{m+1}, \cdots, \text{not } l_n$$

where at least one l_i is non-regular.

3. *A consistency restoring rule (cr-rule) is a statement of the form:*

$$r: h_1 \text{ or } \cdots \text{ or } h_k \overset{+}{\leftarrow} l_1, \cdots, l_m, \text{ not } l_{m+1}, \cdots, \text{not } l_n$$

where $k > 0$, r is a term which uniquely denotes the name of the rule and h_i's and l_i's are r-literals.

A regular rule and constraint rule have the same intuitive reading as standard rules of ASP. The intuitive reading of a cr-rule is: *if one believes in* $l_1, \ldots l_m$ *and has no reason to believe* $l_{m+1}, \ldots, l_n$*, then one may possibly believe one of* $h_1, \ldots, h_k$*.* The implicit assumption is that this possibility is used as little as possible, and only to restore consistency of the agent's beliefs.

Definition 2. [program] *A* $\mathcal{V}(\mathcal{C})$ *program is a pair* (Σ,Π)*, where* Σ *is a sorted signature and* Π *is a set of* $\mathcal{V}(\mathcal{C})$ *rules over* Σ*.*

Example 1. To represent conditions: "John goes to work either by car which takes 30 to 40 minutes, or by bus which takes at least 60 minutes", we start by defining the signature $\Sigma = \{C_r = \{start, end\},\ P_r = \{by\_car, by\_bus\},\ C_c = \{D_c = [0..1439], R_c = [-1439..1439]\},\ V_c = \{T_s, T_e\},\ F_c = \{-\}, P_c = \{>\},\ P_m = \{at\}\}$. The sets C_r and P_r contain regular constants and predicates; elements of C_c, V_c, F_c, and P_c are constrained constants, variables, functions and predicate symbols. P_m is the set of mixed predicates. Values in D_c represent time in minutes. Consider one whole day from 12:00am to 11:59pm mapped to the interval [0..1439]. Regular atom "*by_car*" says that "John travels by car"; mixed atom $at(start, T)$ says that "John starts from home at time T". Similarly for "*by_bus*" and "$at(end, T)$". Function "$-$" has the domain D_c and range R_c; T_s, T_e are variables for D_c. The rules below represent the information from the story.
% 'John travels either by car or bus' is represented by an r-rule
$r_a:\ by\_car\ or\ \ by\_bus.$
% Travelling by car takes between 30 to 40 minutes. This information is encoded by two c-rules
$r_b:\ \leftarrow by\_car,\ at(start, T_s),\ at(end, T_e), T_e - T_s > 40.$
$r_c:\ \leftarrow by\_car,\ at(start, T_s),\ at(end, T_e),\ T_s - T_e > -30.$
% Travelling by bus takes at least 60 minutes
$r_d:\ \leftarrow by\_bus,\ at(start, T_s),\ at(end, T_e),\ T_s - T_e > -60.$
$T_e - T_s > 40$, $T_s - T_e > -30$, and $T_s - T_e > -60$ are c-atoms.

Example 2. Let us expand the story from example 1 by new information: *'John prefers to come to work before 9am'.* We add new constant $'time0'$ to C_r of Σ which denotes the start time of the day, regular atom 'late' which is true when John is late and constrained variable T_t for D_c. Time 9am in our representation is mapped to 540th minute. We expand example 1 by the following rules:
% Unless John is late, he comes to work before 9am
$r_e:\ \leftarrow at(time0, T_t),\ at(end, T_e),\ \neg late,\ T_e - T_t > 540$
% Normally, John is not late
$r_f:\ \ \neg late\ \leftarrow\ not\ late$
% On some rare occasions he might be late, which is encoded by a cr-rule
$r_g:\ \ late \overset{+}{\leftarrow}$

2.2 Semantics

We denote the sets of r-rules, cr-rules and c-rules in Π by Π^r, Π^{cr} and Π^c respectively. A rule r of (Π, Σ) will be called *r-ground* if regular terms in r are ground. A program is called *r-ground* if all its rules are r-ground. A rule r^g is called a *ground instance* of a rule r if it is obtained from r by: (1). replacing variables by ground terms of respective sorts; (2). replacing the remaining terms by their values. For example, 3+4 will be replaced by 7. The program $ground(\Pi)$ with all ground instances of all rules in Π is called the *ground instance* of Π. Obviously ground(Π) is an r-ground program.

We first define semantics for programs without cr-rules. We believe that this definition is slightly simpler than the equivalent definition from [1].

Definition 3. [answer set 1] *Given a program (Σ, Π), where Π contains no cr-rules, let X be a set of ground m-atoms such that for every predicate $p \in P_m$ and every ground r-term t_r, there is exactly one c-term t_c such that $p(\bar{t}_r, \bar{t}_c) \in X$. A set S of ground atoms over Σ is an answer set of Π if S is an answer set of $ground(\Pi) \cup X \cup M_c$.*

Example 3. Consider example 1 and let $X = \{at(start, 430), at(end, 465)\}$. The set $S = \{by\_car, at(start, 430), at(end, 465)\} \cup M_c$ is an answer set of $ground(\Pi) \cup X \cup M_c$ and therefore is an answer set of Π. According to S, John starts to travel by car at 7:10am and reaches work at 7:45am. Of course there are other answer sets where John travels by car and his start and end times differ but satisfy given constraints. There are also answer sets where John travels by bus.

Now we give the semantics for programs with cr-rules. By $\alpha(r)$, we denote a regular rule obtained from a cr-rule r by replacing $\overset{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to a set R of cr-rules. Recall that according to [6], a minimal (with respect to set theoretic inclusion) collection R of cr-rules of Π such that $\Pi^r \cup \Pi^c \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of Π. Definition 4 is a simplification of the original definition from [4], which includes special preference rules.

Definition 4. [answer set 2] *A set S is called an answer set of Π if it is an answer set of program $\Pi^r \cup \Pi^c \cup \alpha(R)$ for some abductive support R of Π.*

The next two examples illustrate the above definition.

Example 4. Consider example 2 and let $X = \{at(start, 430), at(end, 465)\}$. The set $S = \{by\_car, \neg late,\ at(time0, 0),\ at(start, 430),\ at(end, 465)\} \cup M_c$ is an answer set of $ground(\Pi) \cup X \cup M_c$ and therefore is an answer set of Π. According to S, John starts by car at 7:10am and reaches work at 7:45am and is not late. The cr-rule was not applied and $\alpha(\emptyset) = \emptyset$.

Example 5. Let us consider example 2, and add new information that 'John's car is not available and he could not start from home until 8:15am'.
% 'John's cannot use his car' is encoded by an r-rule with empty head.
$r_h :\ \leftarrow by\_car.$
% 'John cannot start before 8:15am' is encoded as a c-rule:
$r_i :\ \leftarrow at(time0, T_t),\ at(start, T_s),\ T_t - T_s > -495.$
Let $X = \{at(time0, 0), at(start, 495), at(end, 560)\}$. $S = \{\ by\_bus, late, at(time0, 0), at(start, 495), at(end, 560)\ \} \cup M_c$ is an answer set of the program, where John arrives late by bus at 9:20am. The cr-rule r_g was used and $\alpha(\{r_g\}) = \{late \leftarrow\}$.

3 Implementation

In this section we describe the algorithm which takes a program (Σ, Π) of $\mathcal{V}_0$ as an input and returns sets A and X of regular and mixed atoms such that $M = A \cup X \cup M_c$ is an answer set of Π. We *assume* that c-rules of the program contain exactly one c-literal. Since the negation of constraint $X - Y > K$ is a difference constraint $X - Y \leq K$, this restriction makes it possible to find answer sets of Π by solving the constraints with a difference constraint solver. The algorithms consists of the *partial grounder* and the $\mathcal{V}_0$ *solver*. The grounder outputs the r-ground program obtained from program Π of $\mathcal{V}_0$ by replacing regular variables by their values and ground terms by their values. The implementation of partial grounder uses *lparse* [8]. To allow for partial grounding by *lparse*, we need intermediate transformations before and after grounding by lparse. The transformations ensure that c-variables are not ground and rules containing m-atoms are not removed by *lparse*. The transformations remove and store c-variables, m-atoms and c-atoms from Π before grounding and then restore them back after grounding. The $\mathcal{V}_0$ solver integrates the standard CR-Prolog solver and a difference constraint solver. The input to the constraint solver is a conjunction of difference constraints. The output is a consistent solution, which is defined as the assignment of values to the variables in the constraints such that the constraints are satisfied. If there is no consistent solution then the constraint solver returns $false$. The constraint solver is interleaved with asp solver. The set of difference constraints in constraint store (input to the constraint solver) depends on the partial model of Π as computed by asp solver. With changes in the partial model, new constraints are added or old constraints are removed from the constraint store. To allow re-use of information on solutions of the current store, we implemented an incremental difference constraint solver [7]. When a new constraint is added, the incremental solver does not compute solution from scratch but makes use of the solution of the previous store and computes a new solution much faster. When a constraint is added to store, the complexity of finding a solution is $O(m + nlogn)$ where m and n are the number of constraints and variables respectively. Finding consistent solutions when a constraint is deleted takes constant time.

4 Related Work and Conclusion

Semantically programs in $\mathcal{V}_0$ and CR-Prolog are fairly similar to each other. In particular, $\mathcal{V}_0$ programs can be translated to CR-Prolog programs with answer sets of $\mathcal{V}_0$ having a one to one correspondence with answer sets of CR-Prolog. $\mathcal{V}_0$ programs *not containing cr-rules*, can be transformed to ASP programs with one to one correspondence between their answer sets. The main difference between $\mathcal{V}_0$ and CR-Prolog/ASP is the efficiency of the solvers to compute answer sets. There are some loosely coupled solvers, in particular CASP solver [1]. The CASP solver uses off-the-shelf ASP and CLP solvers and couples them to compute answer sets for programs in CASP language. ASP and CLP solvers are not tightly integrated as in our solver, but they allow general constraint atoms. We believe that if substantial backtracking is required then a tightly coupled solver would be faster than a loosely coupled solver. If no backtracking is needed, then loosely coupled solvers can be faster because tightly coupled solvers do more book-keeping. It is not entirely obvious how much time the book-keeping takes. A full investigation needs to be done on the advantages and disadvantages of tight coupling. Recently, SAT solvers have been integrated with constraint solvers [2] towards developing solvers for satisfiability modulo theories. The knowledge represented by the input language of these solvers is all propositional. The language of $\mathcal{V}_0$ allows variables, and can represent more sophisticated knowledge using recursive rules and cr-rules.

Finally to recap, we introduced a collection, $\mathcal{V}(\mathcal{C})$, of languages parametri-sed over a class $\mathcal{C}$ of constraints. We studied an instance $\mathcal{V}_0$ where $\mathcal{C}$ is a collection of constraints of the form $X - Y > K$. We designed and implemented a system for computing the answer sets of programs of $\mathcal{V}_0$. This implementation does not ground constraint variables and tightly couples the classical ASP solver with a difference constraint solver. The implementation integrates the use of A-Prolog / CR-Prolog methodology of knowledge representation for planning; and computational machinery for reasoning with difference constraints. We solved some problems which could not be solved either by ASP or constraint solvers alone. We are investigating the extent of tight-coupling that would maximize the solver efficiency.

Acknowledgments

This work was partially supported by ARDA grant ASU06C-0143 and NASA grant NASA-NNG05GP48G. The authors wish to thank Marcello Balduccini, and Yuan-Lin Zhang for useful discussions on this subject.

References

1. S. Baselice, P. A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *In Proceedings of ICLP*, pages 52–66, 2005.
2. Marco Bozzano et. al. The mathsat 3 system. *In proc. CADE-20, Int. Conference on Automated Deduction*, Jul 2005.
3. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
4. Balduccini M. *Answer Set Based Design of Highly Autonomous, Rational Agents.* PhD thesis, Texas Tech University, Dec 2005.
5. Balduccini M. CR-models: An inference engine for CR-prolog. In *Logic Programming and Nonmonotonic Reasoning*, May 2007.
6. Balduccini M., Gelfond M., and Nogueira M. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 2006.
7. G. Ramalingam, J. Song, L. Joscovicz, and R. Miller. Solving difference constraints incrementally. *Algorithmica*, 23:261–275, 1999.
8. Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Technical Report 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.

Semantics for Possibilistic Disjunctive Programs

Juan Carlos Nieves[1], Mauricio Osorio[2], and Ulises Cortés[1]

[1] Universitat Politècnica de Catalunya
Software Department (LSI)
c/Jordi Girona 1-3, E08034, Barcelona, Spain
{jcnieves,ia}@lsi.upc.edu
[2] Universidad de las Américas - Puebla
Centia
Sta. Catarina Mártir, Cholula, Puebla, 72820 México
osoriomauri@googlemail.com

Abstract. In this paper by considering answer set programming approach and some basic ideas from possibilistic logic, we introduce a possibilistic disjunctive logic programming approach able to deal with reasoning under uncertain and incomplete information. Our approach permits to use explicitly labels like possible, probable, plausible, *etc.*, for capturing the incomplete state of a belief in a disjunctive logic program.

1 Introduction

As Tversky and Kahneman observed in [12], many decisions that we make in our common life are based on beliefs concerning the likelihood of uncertain events. In fact, we commonly use statements such as "I think that . . . ", "chances are . . . ", "it is *probable* that . . . ", "it is *plausible* that . . . ", *etc.*, for supporting our decisions. In this kind of statements usually we appeal to our experience or our commonsense. It is not surprising to think that a reasoning based on these kind of statements could reach *bias conclusions*. However these conclusions could reflect the experience or commonsense of an expert. Pelletier and Elio pointed out in [8] that people simply have tendencies to ignore certain information because of the (evolutionary) necessity to make decisions quickly. This gives rise to "biases" in judgments concerning what they "really" want to do.

In view of the fact that we know that a reasoning based on statements which are quantified by relative likelihoods could capture our experience or our commonsense, the question is: how could these statements be captured by real application systems like Multi Agent Systems? For those steeped in probability, Halpern has remarked in [6] that probability has its problems. For one thing, the numbers are not always available. For another, the commitment to numbers means that any two events must be comparable in terms of their probabilities: either one event is more probable than the other, or they have equal probability. Now, the question is why not to use explicitly labels like *possible*, *probable*, *plausible*, *etc.*, for capturing the incomplete state of a belief in a logic program when the numerical representations are not available or difficult to get.

In [7], it was proposed a possibilistic framework for reasoning under uncertainty. It is a combination between Answer Set Programming (ASP) [1] and Possibilistic Logic [3].

This framework is able to deal with reasoning that is at the same time non-monotonic and uncertain. Nicolas *et al.*'s approach is based on the concept of *possibilistic stable model* which defines a semantics for possibilistic normal logic programs. One weak point of this approach is that it relies on the expressiveness of normal logic programs and it always depends of a numerical representation for capturing the incomplete state of a belief.

In this paper, we introduce the use of *possibilistic disjunctive clauses* which are able to capture *incomplete information* and *incomplete states of a knowledge base* at the same time. It is important to point out that our approach is not exactly a generalization of Nicolas *et al.*'s approach since our semantics is based on an operator $\mathcal{T}$ which is inspired in partial evaluation [2] and an inference rule of possibilistic logic [3]. Also whereas Nicolas *et al.*'s approach only permits to express the states of a belief by totally ordered sets, our approach permits to consider partially ordered sets for expressing the states of a belief. Moreover we does not adopt to use *strict α-cuts* for handling an inconsistent possibilistic logic program. However our approach in the class of possibilistic normal logic programs coincides with Nicolas *et al.*'s approach when it considers totally ordered sets for capturing the incomplete state of a belief and the possibilistic program is consistent.

By considering partially ordered sets, it is possible to capture the confidence of a claim by using quantifies like the Toulmin's famous "quantifies"[11]. For instances, in [4] Fox and Modgil discuss the expressiveness of these quantifiers for capturing the uncertainty of medical claims. We use relative likelihoods for modeling different quantifiers *e.g., certain*, *confirmed*, *probable*, *plausible*, *supported* and *open*[1], where each quantifier is a possible world/class of beliefs. The user can provide a likelihood ordering for the worlds/classes of beliefs as it is shown in Fig. 1.

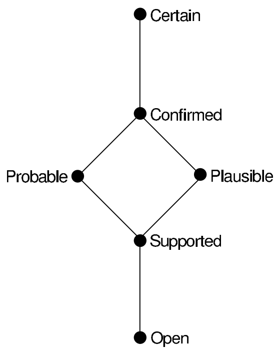


Fig. 1. A lattice where the following relations hold: $Open \preceq Supported$, $Supported \preceq Plausible$, $Supported \preceq Probable$, $Probable \preceq Confirmed$, $Plausible \preceq Confirmed$, and $Confirmed \preceq Certain$

The rest of the paper is divided as follows: In the next section, it is presented the syntax and semantics of our possibilistic framework. In the last section, we present our conclusions.

[1] This set of labels was taken from [4].

2 Possibilistic Disjunctive Logic Programs

In this section, we introduce our possibilistic logic programming framework. We shall start by defining the syntax of a valid program and some relevant concepts, after that we shall define the semantics for the possibilistic disjunctive logic program.

In whole paper, we will consider finite lattices. This convention was taken based on the assumption that in real applications rarely we will have an infinite set of labels for expressing the incomplete state of a knowledge base. Also we will assume some background on ASP. Mainly we will assume knowledge on the syntax and semantics of *extended disjunctive logic programs* (see [5] for definitions).

2.1 Syntax

First of all, we start defining some relevant concepts[2]. A *possibilistic literal* is a pair $l = (a, q) \in L \times Q$, where L is a finite set of literals and $(Q, \leq)$ is a lattice. We apply the projection $*$ over l as follows: $l^* = a$. Given a set of possibilistic literals S, we define the generalization of $*$ over S as follows: $S^* = \{l^* | l \in S\}$. Given a lattice $(Q, \leq)$ and $S \subseteq Q$, $LUB(S)$ denotes the least upper bound of S and $GLB(S)$ denotes the greatest lower bound of S. A possibilistic disjunctive logic program is a finite set of possibilistic disjunctive clauses of the form:

$$r = (\alpha : \; l_1 \vee \ldots \vee l_m \leftarrow l_1, \ldots, l_j, not \; l_{j+1}, \ldots, not \; l_n)$$

where $\alpha \in Q$. The projection $*$ over r is the *extended disjunctive clause* $r^* = l_1 \vee \ldots \vee l_m \leftarrow l_1, \ldots, l_j, not \; l_{j+1}, \ldots, not \; l_n$. $n(r) = \alpha$ is a necessity degree representing the certainty level of the information described by r (see [3] for a formal definition of n). If P is a possibilistic disjunctive logic program, then $P^* = \{r^* | r \in P\}$ is *an extended disjunctive program*. Given an extended disjunctive clause C, we denote C by $\mathcal{A} \leftarrow \mathcal{B}^+, \; not \; \mathcal{B}^-$, where $\mathcal{A}$ contains all the head literals, $\mathcal{B}^+$ contains all the positive body literals and $\mathcal{B}^-$ contains all the negative body literals.

2.2 Semantics

The semantics of the possibilistic disjunctive logic programs is defined in terms of a syntactic reduction which is defined as follows:

Definition 1 (Reduction P^M). *Let P be a possibilistic disjunctive logic program, M be a set of literals. P reduced by M is the positive possibilistic disjunctive program:*
$P^M := \{(n(r) : \mathcal{A} \cap M \leftarrow \mathcal{B}^+) | r \in P, \mathcal{A} \cap M \neq \emptyset, \mathcal{B}^- \cap M = \emptyset, \mathcal{B}^+ \subseteq M\}$
where r^* is of the form $\mathcal{A} \leftarrow \mathcal{B}^+, \; not \; \mathcal{B}^-$.

Notice that $(P^*)^M$ is not exactly the *Gelfond-Lifschitz reduction*. In fact, our reduction is stronger that Gelfond-Lifschitz reduction when P^* is a disjunctive program [5]. One of the main differences is the condition $\mathcal{A} \cap M \neq \emptyset$ which suggests that any clause that does not have a true head literal is false.

[2] Some concepts presented in this subsection extend some terms presented in [7].

Once a possibilistic logic program P has been reduced by a set of literals M^*, it is possible to test whether M is a possibilistic answer set of the program P. In order to define a possibilistic answer set, we introduce an operator which is inspired in partial evaluation for disjunctive logic programs [2] and an inference rule of Possibilistic Logic [3].

Definition 2. *Let P be a possibilistic logic program. The operator $\mathcal{T}(P)$ is defined as follows:*

$$\mathcal{T}(P) := P \cup \left\{ \begin{array}{ll} r' & \textit{if } (\alpha : \mathcal{A} \leftarrow (\mathcal{B}^+ \cup \{B\}),\ not\ \mathcal{B}^-) \in P \textit{ and} \\ & (\alpha_1 : \mathcal{A}_1 \leftarrow \top) \in P \textit{ such that } B \in \mathcal{A}_1 \end{array} \right\}$$

where $r' := GLB(\{\alpha, \alpha_1\}) : \mathcal{A} \cup (\mathcal{A}_1 \setminus \{B\}) \leftarrow \mathcal{B}^+,\ not\ \mathcal{B}^-$.

Intuitively, the operator $\mathcal{T}$ is an inference rule for possibilistic disjunctive logic programs. For instance, let us consider the lattice of Fig. 1 and the possibilistic clauses: $probable : a \vee b \leftarrow \top$ and $confirmed : e \leftarrow b$. Then by applying the operator $\mathcal{T}$, one can get the new possbilistic clause $supported : e \vee a \leftarrow \top$. Also if we consider the possibilistic clauses: $probable : a \vee b \leftarrow \top$ and $plausible : a \leftarrow b$, one can get the new possibilistic clause $supported : a \leftarrow \top$. An important property of the operator $\mathcal{T}$ is that it always reaches a fix-point.

Proposition 1. *Let P be a possibilistic disjunctive logic program. If $\Gamma_0 := \mathcal{T}(P)$ and $\Gamma_i := \mathcal{T}(\Gamma_{i-1})$ such that $i \in \mathcal{N}$, then $\exists\ n \in \mathcal{N}$ such that $\Gamma_n = \Gamma_{n-1}$. We denote Γ_n by $\Pi(P)$.*

From any possibilistic program, it is possible to identify a set of possibilistic literals which we call Sem_{min}.

Definition 3. *Let P be a possibilistic logic program and $Facts(P, A) := \{(\alpha : A \leftarrow \top) | (\alpha : A \leftarrow \top) \in P\}$. $Sem_{min}(P) := \{(x, \alpha) | Facts(P, x) \neq \emptyset$ and $\alpha := LUB_{r \in Facts(P,x)}(n(r))\}$ where $x \in \mathcal{L}_P$.*

Notice that if a possibilistic literal is obtained by different possibilistic clauses, then the possibilistic part of the literal will be obtained by LUB. Now by considering the operator $\mathcal{T}$ and Sem_{min}, we define a posibilistic answer set of a possibilistic program as follows:

Definition 4 (Possibilistic answer set). *Let P be a possibilistic disjunctive logic program and M be a set of possibilistic literals such that M^* is an answer set of P^*. M is a possibilistic answer set of P if and only if $M = Sem_{min}(\Pi(P^{M^*}))$.*

We have to notice that there is an important condition *w.r.t.* the definition of the *possibilistic answer sets*. This is that a possibilistic set S is not a possibilistic answer set of a possibilistic logic program P if S^* is not an answer set of the extended logic program P^*. This condition guarantees that any clause of P^* is satisfied by M^*. In fact, when all the possibilistic clauses of a possibilistic program P have as certainty level the top of the lattice that was considered in P, the answer sets of P^* can be directly generalized to the possibilistic answer sets of P.

In the class of possibilistic normal logic programs[3], our definition of possibilistic answer set is closely related to the definition of possibilistic stable model presented in [7]. In fact, both semantics coincide.

Proposition 2. *Let P be a possibilistic normal logic program. M is a possibilistic answer set of P if and only if M is a possibilistic stable model.*

In terms of computability, we can observe that $\Pi(P)$ is computable.

Proposition 3. *Let P be a finite possibilistic disjunctive logic program. Suppose also that $(Q, \leq)$ is a finite lattice. Then $\Pi(P)$ is computable.*

The main implication of Proposition 3 is that the possibilistic answer sets of a possibilistis logic program are computable.

Proposition 4. *Given a possibilistic program P there exists an algorithm that compute the set of possibilistic answer sets of P.*

3 Conclusions

We have been working in the decision making process for deciding if a human organ is viable or not for being transplanted [10,9]. Our experience suggests that in our medical domain, we require a *qualitative* theory of default reasoning like ASP for modeling incomplete information and a *quantitative* theory like possibilistic logic for modeling uncertain events which always exist in the medical domain.

This paper describes a possibilistic disjunctive logic programming approach which considers some basic ideas from ASP and possibilistic logic. This approach introduces the use of possibilistic disjunctive clauses which are able to capture *incomplete information* and *incomplete states of a knowledge base* at the same time. In fact, one of main motivations of our approach is to define a description languages and a reasoning process where the user could consider relative likelihoods for modeling different levels of uncertainty *e.g., possible*, *probable*, *plausible*, *supported* and *open*, where each likelihood is a possible world/class of beliefs. We know that this kind of representation of uncertainty could reach *bias conclusions*. However, we have to accept that this is a common form that ordinary people perform a reasoning. In fact, these kind of bias are many times well-accepted since they could reflect the experience or commonsense of an expert in a field [12].

In general terms, we are proposing a possibilistic disjunctive logic programming framework able to deal with reasoning under uncertainty and incomplete information. This framework permits to use explicitly labels like *possible*, *probable*, *plausible*, *etc.*, for capturing the incomplete state of a belief in a disjunctive logic program when the numerical representations are not available or difficult to get. In terms of computability, we observe that our approach is computable.

In conclusion, the possibilistic disjunctive logic programs define a possibilistic approach able to capture *incomplete information* and *incomplete states of a knowledge*

[3] A possibilistic logic program P is called possibilistic normal logic program if P^* is a normal program.

base. To the best of our knowledge this approach is the first one that deals with disjunctive programs and partially ordered sets in order to define a possibilistic disjunctive semantics.

Acknowledgement. We are grateful to anonymous referees for their useful comments. J.C. Nieves thanks to CONACyT for his PhD Grant. J.C. Nieves and U. Cortés were partially supported by the grant FP6-IST-002307 (ASPIC). The views expressed in this paper are not necessarily those of ASPIC consortium.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, 2003.
2. S. Brass and J. Dix. Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, 38(3):167–213, 1999.
3. D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pages 439–513. Oxford University Press, Oxford, 1994.
4. J. Fox and S. Modgil. From arguments to decisions: extending the toulmin view. In *Arguing on the Toulmin model: New essays on argument analysis and evaluation*. Argumentation Library series published by Kluwer Academic, Currently in press.
5. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
6. J. Y. Halpern. *Reasoning about uncertainty*. The MIT Press, 2005.
7. P. Nicolas, L. Garcia, I. Stéphan, and C. Lafèvre. Possibilistic Undertainty Handling for Answer Set Programming. *Annal of Mathematics and Artificial Intelligence*, 47(1-2):139–181, June 2006.
8. F. J. Pelletier and R. Elio. *Scope of Logic, Methodology and Philosophy of Science*, volume 1 of *Synthese Library*, chapter Logic and Computation, pages 137–156. Dordrecht: Kluwer Academic Press, 2002.
9. P. Tolchinsky, U. Cortés, S. Modgil, F. Caballero, and A. López-Navidad. Increasing Human-Organ Transplant Availability: Argumentation-Based Agent Deliberation. *IEEE Intelligent Systems: Special Issue on Intelligent Agents in Healthcare*, 21(5):30–37, November/December 2006.
10. P. Tolchinsky, U. Cortés, J. C. Nieves, A. López-Navidad, and F. Caballero. Using arguing agents to increase the human organ pool for transplantation. In *Proc. of the Third Workshop on Agents Applied in Health Care (IJCAI 2005)*, 2005.
11. S. E. Toulmin. *The Uses of Argument*. Cambridge University Press, 1958.
12. A. Tversky and D. Kahneman. *Judgment under uncertainty:Heuristics and biases*, chapter Judgment under uncertainty:Heuristics and biases, pages 3–20. Cambridge Univertisy Press, 1982.

Modularity in SMODELS Programs

Emilia Oikarinen

Laboratory for Theoretical Computer Science
P.O.Box 5400, FI-02015 Helsinki University of Technology, Finland
Emilia.Oikarinen@tkk.fi

Abstract. A recently proposed module system for answer set programming is generalized for the input language of the SMODELS system. To show that the stable model semantics is compositional and modular equivalence is a congruence for composition of SMODELS program modules, a general translation-based scheme for introducing syntactic extensions of the module system is presented. A characterization of the compositionality of the semantics is used as an alternative condition for module composition, which allows compositions of modules even in certain cases with positive recursion between the modules to be composed.

1 Introduction

There is a number of approaches within answer set programming (ASP) [1] involving modularity in some sense, based on e.g. *generalized quantifiers* [2], *templates* [3], *import rules* [4], the *splitting set theorem* [5] or its variants [6,7]. However, only few of these approaches describe a flexible module system with a clearly defined interface for module interaction, and a very typical restriction is that *no recursion between modules is allowed*. In [8] we accommodate Gaifman and Shapiro's program modules [9] to the context of ASP resulting in a simple and intuitive notion for *normal logic program modules* under the *stable model semantics* [10]. A module interacts through an *input/output interface*, and full compatibility of the module system and the stable model semantics is achieved by allowing *positive recursion inside modules only*. However, the use of negative recursion is not limited in any way, and positive recursion is allowed inside modules. One of the main results is a *module theorem* showing that module-level stability implies program-level stability, and vice versa, as long as the stable models of the submodules are *compatible*. We also introduce a *notion of modular equivalence* which is a proper *congruence relation for composition of modules*, i.e., modular equivalence is preserved if a submodule is substituted with a modularly equivalent one.

In this article we extend the module system in [8] for SMODELS *programs* [11] by proposing a general translation-based scheme for introducing syntactical extensions of the module system. Furthermore, we present a semantical reformulation of module composition and modular equivalence, which occasionally allows compositions of modules even if there is positive recursion between modules to be composed.

2 SMODELS Programs and Equivalence Relations

We consider the class of programs in the input language of the SMODELS system [11] excluding *optimization statements*. An SMODELS program P is a finite set of *basic* *constraint rules* and *compute statements*, combined with a *Herbrand base* $\mathrm{Hb}(P)$, which is a fixed finite set of atoms containing all atoms appearing in P. A basic constraint rule is either a *weight rule* of the form $h \leftarrow w \leq \{B = W_B, {\sim}C = W_C\}$ or a *choice rule* of the form $\{H\} \leftarrow B, {\sim}C$ and compute statements are of the form compute$\{B, {\sim}C\}$, where h is an atom, B, C, and H are sets of atoms, $H \neq \emptyset$, $W_B, W_C \subseteq \mathbb{N}$, and $w \in \mathbb{N}$. In a weight rule each $b \in B$ ($c \in C$) is associated with a weight $w_b \in W_B$ ($w_c \in W_C$). A *basic rule*, denoted by $h \leftarrow B, {\sim}C$, is a special case of a weight rule with all weights equal to 1 and $w = |B| + |C|$. An SMODELS program consisting only of basic rules is called a *normal logic program* (NLP). Basic constraint rules consist of two parts: h or H is the *head* of the rule, and the rest is called the *body*. $\mathrm{Head}(P)$ denotes the set of atoms appearing in the heads of basic constraint rules in an SMODELS program P. An *interpretation* M of a program P is a subset of $\mathrm{Hb}(P)$ defining which atoms $a \in \mathrm{Hb}(P)$ are *true* ($a \in M$) and which are *false* ($a \notin M$). A choice rule in P is satisfied in all interpretations $M \subseteq \mathrm{Hb}(P)$; a weight rule in P is satisfied in M iff $w \leq \sum_{b \in B \cap M} w_b + \sum_{c \in C \setminus M} w_c$ implies $h \in M$; and a compute statement in P is satisfied in M iff $B \subseteq M$ and $M \cap C = \emptyset$. An interpretation M is a *model* of a program P, denoted by $M \models P$, iff all the rules in P are satisfied in M.

Definition 1. *The reduct P^M of an* SMODELS *program P w.r.t. $M \subseteq \mathrm{Hb}(P)$ contains*

1. *rule $h \leftarrow B$ iff there is a choice rule $\{H\} \leftarrow B, {\sim}C$ in P such that $h \in H \cap M$, and $M \cap C = \emptyset$;*
2. *rule $h \leftarrow w' \leq \{B = W_B\}$ iff there is a weight rule $h \leftarrow w \leq \{B = W_B, {\sim}C = W_C\}$ in P and $w' = \max(0, \sum_{c \in C \setminus M} w_c)$.*

An SMODELS program P is *positive* if each rule in P is a weight rule with $C = \emptyset$. Given the *least model semantics* for positive programs the stable model semantics [10] straightforwardly generalizes for SMODELS programs [11,16]. In analogy to the case of NLPs the reduct from Definition 1 is used, but the effect of compute statements must also be taken into account. Let $\mathrm{CompS}(P)$ denote the union of literals appearing in the compute statements of P.

Definition 2. *An interpretation $M \subseteq \mathrm{Hb}(P)$ is a stable model of an* SMODELS *program P, denoted by $M \in \mathrm{SM}(P)$, iff $M = \mathrm{LM}(P^M)$ and $M \models \mathrm{CompS}(P)$.*

Given $a, b \in \mathrm{Hb}(P)$, we say that b *depends directly* on a, denoted by $a \leq_1 b$, iff there is a basic constraint rule in P such that b is in the head of the rule and a appears in the positive body B of the rule. The *positive dependency graph* of P, denoted by $\mathrm{Dep}^+(P)$, is a graph with $\mathrm{Hb}(P)$ and $\{\langle b, a\rangle \mid a \leq_1 b\}$ as the sets of vertices and edges, respectively. A *strongly connected component* (SCC) of a graph is a maximal subset D of vertices such that there is a path between a and b for all $a, b \in D$.

There are several notions of equivalence proposed for logic programs. Given SMODELS programs P and Q, they are *weakly equivalent* [12], denoted by $P \equiv Q$, iff $\mathrm{SM}(P) = \mathrm{SM}(Q)$, and *strongly equivalent* [12], denoted by $P \equiv_\mathrm{s} Q$, iff $P \cup R \equiv Q \cup R$ for all SMODELS programs R. *Visible equivalence relation* [13] takes the interfaces of programs into account; the Herbrand base of P is partitioned into two parts, $\mathrm{Hb}_\mathrm{v}(P)$ and $\mathrm{Hb}_\mathrm{h}(P)$ determining the *visible* and the *hidden* parts of $\mathrm{Hb}(P)$, respectively. Programs P and Q are visibly equivalent, denoted by $P \equiv_\mathrm{v} Q$, iff $\mathrm{Hb}_\mathrm{v}(P) = \mathrm{Hb}_\mathrm{v}(Q)$

and there is a bijection $f : \mathrm{SM}(P) \rightarrow \mathrm{SM}(Q)$ such that for all $M \in \mathrm{SM}(P)$, it holds $M \cap \mathrm{Hb_v}(P) = f(M) \cap \mathrm{Hb_v}(Q)$. The verification of $\equiv/\equiv_\mathrm{s}$ is a **coNP**-complete decision problem for SMODELS programs [14,15]. Deciding $\equiv_\mathrm{v}$ can be hard in general, but the computational complexity can be governed by limiting the use of hidden atoms by the property of having *enough visible atoms*, i.e. the EVA property. Intuitively, if a program has the EVA property, then its stable models can be distinguished on the basis of their visible parts. For SMODELS programs with the EVA property, the verification of visible equivalence is a **coNP**-complete decision problem [16].

3 Modular SMODELS Programs

We define SMODELS *program modules* in analogy to *normal logic program modules* (NLP modules) in [8] and *program modules* by Gaifman and Shapiro [9].

Definition 3. *A triple $\mathbb{P} = (P, I, O)$ is a* SMODELS *program module, if (i) P is a finite set of basic constraint rules and compute statements, and I and O are sets of atoms; (ii) $I \cap O = \emptyset$; and (iii)* $\mathrm{Head}(P) \cap I = \emptyset$.

The Herbrand base of an SMODELS program module $\mathbb{P}$ is the set of atoms appearing in P combined with $I \cup O$, $\mathrm{Hb_v}(\mathbb{P}) = I \cup O$, and $\mathrm{Hb_h}(\mathbb{P}) = \mathrm{Hb}(\mathbb{P}) \setminus \mathrm{Hb_v}(\mathbb{P})$. As noted in [9,8] *module composition* needs to be restricted in order to achieve *compositionality* for the semantics. In [9] module composition is restricted to cases in which the output sets of the modules are disjoint and the hidden part of each module remains local.

Definition 4. *Let $\mathbb{P}_1 = (P_1, I_1, O_1)$ and $\mathbb{P}_2 = (P_2, I_2, O_2)$ be* SMODELS *program modules such that (i) $O_1 \cap O_2 = \emptyset$, and (ii)* $\mathrm{Hb_h}(\mathbb{P}_1) \cap \mathrm{Hb}(\mathbb{P}_2) = \mathrm{Hb_h}(\mathbb{P}_2) \cap \mathrm{Hb}(\mathbb{P}_1) = \emptyset$*. Then the* GS-composition *of $\mathbb{P}_1$ and $\mathbb{P}_2$ is defined as*

$$\mathbb{P}_1 \oplus \mathbb{P}_2 = (P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2).$$

As shown in [8, Example 3] the conditions for $\oplus$ are not enough to guarantee compositionality under the stable model semantics. We say that there is a *positive recursion* between $\mathbb{P}_1 = (P_1, I_1, O_1)$ and $\mathbb{P}_2 = (P_2, I_2, O_2)$, if there is a SCC in $\mathrm{Dep}^+(P_1 \cup P_2)$ containing atoms from both O_1 and O_2. We deny positive recursion between modules as a further restriction for module composition.

Definition 5. *Let $\mathbb{P}_1 = (P_1, I_1, O_1)$ and $\mathbb{P}_2 = (P_2, I_2, O_2)$ be* SMODELS *program modules. If $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined and there is no positive recursion between $\mathbb{P}_1$ and $\mathbb{P}_2$, the* join *of $\mathbb{P}_1$ and $\mathbb{P}_2$, denoted by $\mathbb{P}_1 \sqcup \mathbb{P}_2$, is defined as $\mathbb{P}_1 \oplus \mathbb{P}_2$.*

The stable model semantics of an SMODELS program module is defined with respect to a given input, i.e., a subset of the input atoms of the module. Input is seen as a set of facts (or a database) to be combined with the module. The *instantiation of a module* $\mathbb{P} = (P, I, O)$ with respect to an input $A \subseteq I$ is $\mathbb{P}(A) = \mathbb{P} \sqcup \mathrm{F}_A = (P \cup \mathrm{F}_A, \emptyset, I \cup O)$, where $\mathrm{F}_A = \{a. \mid a \in A\}$. In the sequel $\mathbb{P}(A)$ is identified with the program $P \cup \mathrm{F}_A$.

Definition 6. *An interpretation $M \subseteq \mathrm{Hb}(\mathbb{P})$ is a stable model of an* SMODELS *program module $\mathbb{P} = (P, I, O)$, denoted by $M \in \mathrm{SM}(\mathbb{P})$, iff $M = \mathrm{LM}(P^M \cup \mathrm{F}_{M \cap I})$ and $M \models \mathrm{CompS}(P)$.*

We generalize the notions of *visible and modular equivalence*, denoted by $\equiv_{\mathrm{v}}$ and $\equiv_{\mathrm{m}}$, respectively, for SMODELS program modules as follows.

Definition 7. *For* SMODELS *program modules* $\mathbb{P}=(P,I_P,O_P)$ *and* $\mathbb{Q}=(Q,I_Q,O_Q)$*,*

- $\mathbb{P}\equiv_{\mathrm{v}}\mathbb{Q}$ *iff* $\mathrm{Hb_v}(\mathbb{P})=\mathrm{Hb_v}(\mathbb{Q})$ *and there is a bijection* $f:\mathrm{SM}(\mathbb{P})\rightarrow\mathrm{SM}(\mathbb{Q})$ *such that for all* $M\in\mathrm{SM}(\mathbb{P})$*,* $M\cap\mathrm{Hb_v}(\mathbb{P})=f(M)\cap\mathrm{Hb_v}(\mathbb{Q})$*; and*
- $\mathbb{P}\equiv_{\mathrm{m}}\mathbb{Q}$ *iff* $I_P=I_Q$ *and* $\mathbb{P}\equiv_{\mathrm{v}}\mathbb{Q}$*.*

The definition of $\equiv_{\mathrm{m}}$ above is a reformulation of the one given in [8] and results in exactly the same relation. Note that the condition $\mathrm{Hb_v}(\mathbb{P})=\mathrm{Hb_v}(\mathbb{Q})$ insisted by $\equiv_{\mathrm{v}}$ together with $I_P=I_Q$ implies $O_P=O_Q$ for $\equiv_{\mathrm{m}}$. Visible equivalence relates more modules than modular equivalence, e.g., consider $\mathbb{P}=(\{a\leftarrow b.\},\{b\},\{a\})$ and $\mathbb{Q}=(\{b\leftarrow a.\},\{a\},\{b\})$. Now, $\mathbb{P}\equiv_{\mathrm{v}}\mathbb{Q}$ as $\mathrm{SM}(\mathbb{P})=\mathrm{SM}(\mathbb{Q})=\{\emptyset,\{a,b\}\}$, but $\mathbb{P}\not\equiv_{\mathrm{m}}\mathbb{Q}$ because input interfaces of $\mathbb{P}$ and $\mathbb{Q}$ differ. This indicates that visibly equivalent modules cannot necessarily act as substitutes for each other.

A concept of *compatibility* is used to describe when interpretations of modules can be combined together. Given modules $\mathbb{P}_1$ and $\mathbb{P}_2$ we say that $M_1\subseteq\mathrm{Hb}(\mathbb{P}_1)$ and $M_2\subseteq\mathrm{Hb}(\mathbb{P}_2)$ are *compatible* iff $M_1\cap\mathrm{Hb_v}(\mathbb{P}_2)=M_2\cap\mathrm{Hb_v}(\mathbb{P}_1)$. Furthermore, given sets of interpretations $A_1\subseteq 2^{\mathrm{Hb}(\mathbb{P}_1)}$ and $A_2\subseteq 2^{\mathrm{Hb}(\mathbb{P}_2)}$ for modules $\mathbb{P}_1$ and $\mathbb{P}_2$, the *natural join* of A_1 and A_2, denoted by $A_1\bowtie A_2$, is defined as

$$\{M_1\cup M_2\mid M_1\in A_1, M_2\in A_2 \text{ and } M_1\cap\mathrm{Hb_v}(\mathbb{P}_2)=M_2\cap\mathrm{Hb_v}(\mathbb{P}_1)\}.$$

If a program (module) consists of several submodules, its stable models are locally stable for the respective submodules; and on the other hand, local stability implies global stability for compatible stable models of the submodules.

Theorem 1. *(Module theorem) Let* $\mathbb{P}_1$ *and* $\mathbb{P}_2$ *be* SMODELS *program modules such that* $\mathbb{P}_1\sqcup\mathbb{P}_2$ *is defined. Then* $\mathrm{SM}(\mathbb{P}_1\sqcup\mathbb{P}_2)=\mathrm{SM}(\mathbb{P}_1)\bowtie\mathrm{SM}(\mathbb{P}_2)$.

Theorem 1 is a generalization of [8, Theorem 1] for SMODELS program modules. Instead of proving Theorem 1 directly from scratch we propose a general translation-based scheme for introducing syntactical extensions for the module theorem.

Proposition 1. *Let* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *be two classes of logic program modules such that* $\mathcal{C}_2\subseteq\mathcal{C}_1$*, and consider a translation function* $\mathrm{Tr}:\mathcal{C}_1\rightarrow\mathcal{C}_2$ *such that for any program modules* $\mathbb{P}=(P,I,O),\mathbb{Q}\in\mathcal{C}_1$*,*

1. *if* $\mathbb{P}\sqcup\mathbb{Q}$ *is defined, then* $\mathrm{Tr}(\mathbb{P})\sqcup\mathrm{Tr}(\mathbb{Q})$ *is defined,*
2. $\mathrm{Tr}(\mathbb{P})\sqcup\mathrm{Tr}(\mathbb{Q})\equiv_{\mathrm{m}}\mathrm{Tr}(\mathbb{P}\sqcup\mathbb{Q})$*, and*
3. $(P,I,O\cup\mathrm{Hb_h}(\mathbb{P}))\equiv_{\mathrm{m}}(\mathrm{Tr}(P),I,O\cup\mathrm{Hb_h}(\mathbb{P}))$.

Now, if the module theorem holds for modules in $\mathcal{C}_2$*, then it holds for modules in* $\mathcal{C}_1$*.*

The proof is omitted due to space limitations. Intuitively, the conditions for the translation function serve the following purposes: first, possible compositions of modules are not limited by the translation; second, the translation is *modular*; and third, it has to be *faithful* in the sense that it preserves the roles to the atoms in the original module.

Now, to prove Theorem 1 it suffices to provide a translation from SMODELS program modules to NLP modules satisfying the conditions of Proposition 1. For instance, it suffices to take a (possibly exponential) translation similarly to [11]. Furthermore, the *congruence property* of $\equiv_{\mathrm{m}}$ directly generalizes for SMODELS program modules, as Theorem 1 can be used instead of [8, Theorem 1] in the proof of [8, Theorem 2].

Corollary 1. *Let* $\mathbb{P}, \mathbb{Q}$ *and* $\mathbb{R}$ *be* SMODELS *program modules such that* $\mathbb{P} \sqcup \mathbb{R}$ *and* $\mathbb{Q} \sqcup \mathbb{R}$ *are defined. If* $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$*, then* $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{R}$.

To analyze the computational complexity of verifying $\equiv_{\mathrm{m}}$ for SMODELS program modules, note first that $P \equiv_{\mathrm{v}} Q$ iff $(P, \emptyset, \mathrm{Hb}_{\mathrm{v}}(P)) \equiv_{\mathrm{m}} (Q, \emptyset, \mathrm{Hb}_{\mathrm{v}}(Q))$ for any SMODELS programs P and Q. Furthermore, given SMODELS program modules $\mathbb{P} = (P, I, O)$ and $\mathbb{Q} = (Q, I, O)$, $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ iff $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$, where $\mathbb{G}_I = (\{\{I\}.\}, \emptyset, I)$ a context generator module for I. Thus the problem of verifying $\equiv_{\mathrm{m}}$ has the same computational complexity as verification of $\equiv_{\mathrm{v}}$. We say that an SMODELS program module $\mathbb{P} = (P, I, O)$ has the EVA property iff P has the EVA property for $\mathrm{Hb}_{\mathrm{v}}(P) = I \cup O$ [16]. Verification of $\equiv_{\mathrm{m}}$ is a **coNP**-complete decision problem for SMODELS program modules with the EVA property, since $\mathbb{G}_I$ has the EVA property trivially.

4 Semantical Reformulation of Modular Equivalence

Even though [8, Example 3] shows that conditions for $\oplus$ are not enough to guarantee that the module theorem holds, there are cases where $\mathbb{P} \sqcup \mathbb{Q}$ is not defined and $\mathrm{SM}(\mathbb{P} \oplus \mathbb{Q}) = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{Q})$. Consider, e.g., $\mathbb{P} = (\{a \leftarrow b.\ a \leftarrow {\sim}c.\}, \{b\}, \{a, c\})$ and $\mathbb{Q} = (\{b \leftarrow a.\}, \{a\}, \{b\})$. Now, $\mathbb{P} \oplus \mathbb{Q} = (\{a \leftarrow b.\ a \leftarrow {\sim}c.\ b \leftarrow a.\}, \emptyset, \{a, b, c\})$ is defined as outputs and hidden atoms differ. Since $\mathrm{SM}(\mathbb{P}) = \{\{a\}, \{a, b\}\}$ and $\mathrm{SM}(\mathbb{Q}) = \{\emptyset, \{a, b\}\}$, we get $\mathrm{SM}(\mathbb{P} \oplus \mathbb{Q}) = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{Q}) = \{\{a, b\}\}$. This suggests that the denial of positive recursion between modules can be relaxed in certain cases.

We define a semantical characterization for module composition that maintains the compositionality of the semantics.

Definition 8. *Let* $\mathbb{P}_1 = (P_1, I_1, O_1)$ *and* $\mathbb{P}_2 = (P_2, I_2, O_2)$ *be* SMODELS *program modules such that* $\mathbb{P}_1 \oplus \mathbb{P}_2$ *is defined and* $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$*. Then the semantical join of* $\mathbb{P}_1$ *and* $\mathbb{P}_2$*, denoted by* $\mathbb{P}_1 \underline{\sqcup} \mathbb{P}_2$*, is defined as* $\mathbb{P}_1 \oplus \mathbb{P}_2$.

The module theorem holds by definition for SMODELS program modules composed with $\underline{\sqcup}$. We present an alternative formulation for modular equivalence taking features from strong equivalence [12]: $\mathbb{P} = (P, I_P, O_P)$ and $\mathbb{Q} = (Q, I_Q, O_Q)$ are *semantically modularly equivalent*, denoted by $\mathbb{P} \equiv_{\mathrm{sem}} \mathbb{Q}$, iff $I_P = I_Q$ and $\mathbb{P} \underline{\sqcup} \mathbb{R} \equiv_{\mathrm{v}} \mathbb{Q} \underline{\sqcup} \mathbb{R}$ for all modules $\mathbb{R}$ such that $\mathbb{P} \underline{\sqcup} \mathbb{R}$ and $\mathbb{Q} \underline{\sqcup} \mathbb{R}$ are defined. It is straightforward to see that $\equiv_{\mathrm{sem}}$ is a congruence for $\underline{\sqcup}$ and reduces to $\equiv_{\mathrm{v}}$ for modules with a completely specified input.

Theorem 2. $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ *iff* $\mathbb{P} \equiv_{\mathrm{sem}} \mathbb{Q}$ *for any* SMODELS *program modules* $\mathbb{P}$ *and* $\mathbb{Q}$.

Theorem 2 implies that $\equiv_{\mathrm{m}}$ is a congruence for $\underline{\sqcup}$, too, and it is possible to replace $\mathbb{P}$ with modularly equivalent $\mathbb{Q}$ in contexts allowed by $\underline{\sqcup}$. The syntactical restriction denying positive recursion between modules is easy to check, since SCCs can be found in a linear time with respect to the size of the dependency graph [17]. Checking whether $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$ holds can be a computationally much harder.

Theorem 3. *For* SMODELS *program modules* $\mathbb{P}_1$ *and* $\mathbb{P}_2$ *such that* $\mathbb{P}_1 \oplus \mathbb{P}_2$ *is defined, deciding whether* $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$ *holds is a* $\mathbf{coNP}$*-complete decision problem.*

Theorem 3 shows that there is a tradeoff for allowing positive recursion between modules, as more effort is needed to check that composition of such modules does not compromise the compositionality of the semantics.

Acknowledgements. The author is grateful to Tomi Janhunen for his helpful comments and suggestions. The research is partially funded by Academy of Finland (project #211025). The financial support from Helsinki Graduate School in Computer Science and Engineering, Nokia Foundation, Finnish Foundation of Technology, and Finnish Cultural Foundation is gratefully acknowledged.

References

1. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. **25**(3-4) (1999) 241–273
2. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: LPNMR, Volume 1265 of LNCS, Springer (1997) 290–309
3. Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C., Calimeri, F.: Enhancing answer set programming with templates. In: NMR (2004) 233–239
4. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to ACC tournament scheduling. In: ASP, CEUR-WS.org (2005) 277–292
5. Lifschitz, V., Turner, H.: Splitting a logic program. In: ICLP, MIT Press (1994) 23–37
6. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM TODS **22**(3) (1997) 364–418
7. Faber, W., Greco, G., Leone, N.: Magic sets and their application to data integration. In: ICDT, Volume 3363 of LNCS, Springer (2005) 306–320
8. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: ECAI, IOS Press (2006) 412–416
9. Gaifman, H., Shapiro, E.Y.: Fully abstract compositional semantics for logic programs. In: POPL, ACM Press (1989) 134–142
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP, MIT Press (1988) 1070–1080
11. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
12. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM TOCL **2**(4) (2001) 526–541
13. Janhunen, T.: Some (In)translatability Results for Normal Logic Programs and Propositional Theories. JANCL **16**(1-2) (2006) 35–86
14. Marek, V.W., Truszczyński, M.: Autoepistemic logic. J. ACM **38**(3) (1991) 588–619
15. Pearce, D., Tompits, H., Woltran, S.: Encodings for equilibrium logic and logic programs with nested expressions. In: EPIA, Volume 2258 of LNCS, Springer (2001) 306–320
16. Janhunen, T., Oikarinen, E.: Automated verification of weak equivalence within the SMODELS system. TPLP **7**(4) (2007) 1–48
17. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comp. **1**(2) (1972) 146–160

Author Index