WORKSHOP PROCEEDINGS

ICLP 2008 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2008)

Edited by Wolfgang Faber and Joohyung Lee

December 13, 2008

Preface

These are working notes of the workshop on *Answer Set Programming and Other Computing Paradigms (ASPOCP) 2008*, collocated with the *24th International Conference on Logic Programming (ICLP) 2008* in Udine, Italy.

Since its introduction in the 1990s, answer set programming (ASP) has been widely applied to various knowledge-intensive tasks and combinatorial search problems. ASP was found to be closely related to SAT, which has led to a new method of computing answer sets using SAT solvers and techniques adapted from SAT. While this has been the most studied relationship, identifying links between ASP and other computing paradigms, such as constraint satisfaction, quantified Boolean formulas (QBF), first-order logic (FOL), or databases, to name just a few, is the subject of active research.

The contributions brought about by these studies are manifold: New methods of computing answer sets are being developed, based on the relation between ASP and other paradigms, such as the use of pseudo-Boolean solvers, QBF solvers, and FOL theorem provers. New and improved languages are proposed, inspired by language constructs found in related paradigms. In a somewhat orthogonal way, languages or tasks in other research areas are reduced to ASP, one of the main benefits being that a computational engine is thereby automatically provided. Furthermore, language and solver integration is facilitated, allowing for multi-paradigm problem-solving; currently the integration of ASP with description logics (in the realm of the Semantic Web) and constraint satisfaction are the main focus of this type of activity.

This year, 2008, marks the 20th anniversary of the stable model semantics, a foundational event for answer set programming. We expect that during next decade work on the relation between answer set programming and other computing paradigms will intensify, bringing synergies to all of the involved areas.

This workshop aims at facilitating the discussion about crossing the boundaries of current ASP techniques, in combination with or inspired by other computing paradigms. We have received 12 submissions, of which 9 were accepted for presentation. We thank the contributors for their efforts to provide material of high quality, we thank the program committee members and reviewers for their valuable help to guarantee and improve the quality of the workshop, and last but not least the ICLP officials for making this workshop possible and for their smooth cooperation.

Wishing you all an informative and enjoyable workshop,

Wolfgang Faber, University of Calabria, Italy
Joohyung Lee, Arizona State University, USA

Programme Chairs

Wolfgang Faber, University of Calabria, Italy
Joohyung Lee, Arizona State University, USA

Programme Committee

Chitta Baral, Arizona State University, USA
Gerhard Brewka, University of Leipzig, Germany
Pedro Cabalar, University of A Coruña, Spain
Marc Denecker, Katholieke Universiteit Leuven, Belgium
Nicola Leone, University of Calabria, Italy
Vladimir Lifschitz, University of Texas at Austin, USA
Fangzhen Lin, Hong Kong University of Science and Technology, China
Thomas Lukasiewicz, University of Oxford, UK
Ilkka Niemelä, Helsinki University of Technology, Finland
Mirosław Truszczyński, University of Kentucky, USA
Dirk Vermeir, Vrije Universiteit Brussel, Belgium
Stefan Woltran, Vienna University of Technology, Austria
Yan Zhang, University of Western Sydney, Australia
Yuanlin Zhang, Texas Tech University, USA

Additional Reviewers

Giovambattista Ianni
Michael Jakl
Jianmin Ji
Simona Perri
Francesco Ricca
Yisong Wang
Johan Wittocx

Contents

Answer-Set Programming Encodings for Argumentation Frameworks

Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran

Institut für Informationssysteme, Technische Universität Wien,
Favoritenstraße 9–11, A–1040 Vienna, Austria

Abstract. We present reductions from Dung's argumentation framework (AF) and generalizations thereof to logic programs under the answer-set semantics. The reduction is based on a fixed disjunctive datalog program (the interpreter) and its input which is the only part depending on the AF to process. We discuss the reductions, which are the basis for the system ASPARTIX in detail and show their adequacy in terms of computational complexity.

1 Motivation

Dealing with arguments and counter-arguments in discussions is a daily life process. We usually employ this process to convince our opponent to our point of view. As everybody knows, this is sometimes a cumbersome activity because we miss a formal reasoning procedure for argumentation.

This problem is not new. Leibniz (1646–1716) was the first who tried to deal with arguments and their processing by reasoning in a more formal way. He proposed to use a *lingua characteristica* (a knowledge representation (KR) language) to formalize arguments and a *calculus ratiocinator* (a deduction system) to reason about them. Although Leibniz's dream of a complete formalization of science was destroyed in the thirties of the last century, restricted versions of Leibniz's dream survived.

In Artificial Intelligence (AI), the area of argumentation (see [1] for an excellent summary) has become one of the central issues within the last decade, providing a formal treatment for reasoning problems arising in a number of interesting applications fields, including Multi-Agent Systems and Law Research. In a nutshell, argumentation frameworks formalize statements together with a relation denoting rebuttals between them, such that the semantics gives an abstract handle to solve the inherent conflicts between statements by selecting admissible subsets of them. The reasoning underlying such argumentation frameworks turned out to be a very general principle capturing many other important formalisms from the areas of AI and Knowledge Representations.

The increasing interest in argumentation led to numerous proposals for formalizations of argumentation. These approaches differ in many aspects. First, there are several ways how "admissibility" of a subset of statements can be defined; second, the notion of rebuttal has different meanings (or even additional relationships between statements are taken into account); finally, statements are augmented with priorities, such that the semantics yields those admissible sets which contain statements of higher priority.

Argumentation problems are in general intractable, thus developing dedicated algorithms for the different reasoning problems is non-trivial. A promising approach to

implement such systems is to use a reduction method, where the given problem is translated into another language, for which sophisticated systems already exist. Earlier work [2, 3] proposed reductions for basic argumentation frameworks to (quantified) propositional logic. In this work, we present solutions for reasoning problems in different types of argumentation frameworks by means of computing the answer sets of a datalog program. To be more specific, the system is capable to compute the most important types of extensions (i.e., admissible, preferred, stable, complete, and grounded) in Dung's original framework [4], the preference-based argumentation framework [5], the value-based argumentation framework [6], and the bipolar argumentation framework [7, 8]. Hence our system can be used by researchers to compare different argumentation semantics on concrete examples within a uniform setting. In fact, investigations on the relationship between different argumentation semantics has received increasing interest lately [9].

The declarative programming paradigm of *Answer-Set Programming* (ASP) [10, 11] under the stable-models semantics [12] (which is our target formalism) is especially well suited for our purpose. First, advanced solvers such as Smodels, DLV, GnT, Cmodels, Clasp, or ASSAT which are able to deal with large problem instances (see [13]) are available. Thus, using the proposed reduction method delegates the burden of optimizations to these systems. Second, language extensions can be used to employ different extensions to AFs, which so far have not been studied (for instance, weak constraints or aggregates could yield interesting specially tailored problems for AFs). Finally, depending on the class of the program one uses for a given type of extension, one can show that, in general, the complexity of evaluation within the target formalism is of the same complexity as the original problem. Thus, our approach is adequate from a complexity-theoretic point of view.

With the fixed logic program (independent from the concrete AF to process), we are more in the tradition of a classical implementation, because we construct an interpreter in ASP which processes the AF given as input. This is in contrast to, e.g., the reductions to (quantified) propositional logic [2, 3], where one obtains a formula which completely depends on the AF to process. Although there is no advantage of the interpreter approach from a theoretical point of view (as long as the reductions are polynomial-time computable), there are several practical ones. The interpreter is easier to understand, easier to debug, and easier to extend. Additionally, proving properties like correspondence between answer sets and extensions is simpler. Moreover, the input AF can be changed easily and dynamically without translating the whole formula which simplifies the answering of questions like "What happens if I add this new argument?".

Our system makes use of the prominent answer-set solver DLV [10]. All necessary programs to run ASPARTIX and some illustrating examples are available at

http://www.kr.tuwien.ac.at/research/systems/argumentation/

2 Preliminaries

In this section, we first give a brief overview of the syntax and semantics of disjunctive datalog under the answer-sets semantics [12]; for further background, see [10, 14].

We fix a countable set $\mathcal{U}$ of *(domain) elements*, also called *constants*; and suppose a total order $<$ over the domain elements. An *atom* is an expression $p(t_1, \ldots, t_n)$, where

p is a *predicate* of arity $n \geq 0$ and each t_i is either a variable or an element from $\mathcal{U}$. An atom is *ground* if it is free of variables. By $B_{\mathcal{U}}$ we denote the set of all ground atoms over $\mathcal{U}$.

A *(disjunctive) rule* r is of the form

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \ldots, b_k, \ not\, b_{k+1}, \ldots, \ not\, b_m,$$

with $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$, and where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms, and "not" stands for *default negation*. The *head* of r is the set $H(r) = \{a_1, \ldots, a_n\}$ and the *body* of r is $B(r) = \{b_1, \ldots, b_k, \ not\, b_{k+1}, \ldots, \ not\, b_m\}$. Furthermore, $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. A rule r is *normal* if $n \leq 1$ and a *constraint* if $n = 0$. A rule r is *safe* if each variable in r occurs in $B^+(r)$. A rule r is *ground* if no variable occurs in r. A *fact* is a ground rule without disjunction and empty body. An *(input) database* is a set of facts. A program is a finite set of disjunctive rules. For a program $\mathcal{P}$ and an input database D, we often write $\mathcal{P}(D)$ instead of $D \cup \mathcal{P}$. If each rule in a program is normal (resp. ground), we call the program normal (resp. ground). A program $\mathcal{P}$ is called *stratified* if there exists an assignment $a(\cdot)$ of integers to the predicates in $\mathcal{P}$, such that for each $r \in \mathcal{P}$, the following holds: If predicate p occurs in the head of r and predicate q occurs (i) in the positive body of r, then $a(p) \geq a(q)$ holds; (ii) in the negative body of r, then $a(p) > a(q)$ holds.

For any program $\mathcal{P}$, let $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$ (if no constant appears in $\mathcal{P}$, an arbitrary constant is added to $U_{\mathcal{P}}$). $Gr(\mathcal{P})$ is the set of rules $r\sigma$ obtained by applying, to each rule $r \in \mathcal{P}$, all possible substitutions σ from the variables in $\mathcal{P}$ to elements of $U_{\mathcal{P}}$.

An *interpretation* $I \subseteq B_{\mathcal{U}}$ *satisfies* a ground rule r iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies a ground program $\mathcal{P}$, if each $r \in \mathcal{P}$ is satisfied by I. A non-ground rule r (resp., a program $\mathcal{P}$) is satisfied by an interpretation I iff I satisfies all groundings of r (resp., $Gr(\mathcal{P})$). $I \subseteq B_{\mathcal{U}}$ is an *answer set* of $\mathcal{P}$ iff it is a subset-minimal set satisfying the *Gelfond-Lifschitz reduct*

$$\mathcal{P}^I = \{H(r) \text{ :- } B^+(r) \mid I \cap B^-(r) = \emptyset, r \in Gr(\mathcal{P})\}.$$

For a program $\mathcal{P}$, we denote the set of its answer sets by $\mathcal{AS}(\mathcal{P})$.

Credulous and skeptical reasoning in terms of programs is defined as follows. Given a program $\mathcal{P}$ and a set of ground atoms A. Then, we write $\mathcal{P} \models_c A$ (credulous reasoning), if A is contained in some answer set of $\mathcal{P}$; we write $\mathcal{P} \models_s A$ (skeptical reasoning), if A is contained in each answer set of $\mathcal{P}$.

We briefly recall some complexity results for disjunctive logic programs. In fact, since we will deal with fixed programs we focus on results for data complexity. Recall that data complexity in our context is the complexity of checking whether $P(D) \models A$ when datalog programs P are fixed, while input databases D and ground atoms A are an input of the decision problem. Depending on the concrete definition of $\models$, we give the complexity results in Table 1 (cf. [15] and the references therein).

	stratified programs	normal programs	general case
$\models_c$	P	NP	Σ_2^P
$\models_s$	P	coNP	Π_2^P

Table 1. Data Complexity for datalog (all results are completeness results).

3 Encodings of Basic Argumentation Frameworks

In this section, we first introduce the most important semantics for basic argumentation frameworks in some detail. In a distinguished section, we then provide encodings for these semantics in terms of datalog programs.

3.1 Basic Argumentation Frameworks

In order to relate frameworks to programs, we use the universe $\mathcal{U}$ of domain elements also in the following basic definition.

Definition 1. *An* argumentation framework (AF) *is a pair $F = (A, R)$ where $A \subseteq \mathcal{U}$ is a set of arguments and $R \subseteq A \times A$. The pair $(a, b) \in R$ means that a attacks (or defeats) b. A set $S \subseteq A$ of arguments* defeats *b (in F), if there is an $a \in S$, such that $(a, b) \in R$. An argument $a \in A$ is* defended *by $S \subseteq A$ (in F) iff, for each $b \in A$, it holds that, if $(b, a) \in R$, then S defeats b (in F).*

An argumentation framework can be naturally represented as a directed graph.

Example 1. Let $F = (A, R)$ be an AF with $A = \{a, b, c, d, e\}$ and $R = \{(a, b), (c, b), (c, d), (d, c), (d, e), (e, e)\}$. The graph representation of F is the following.

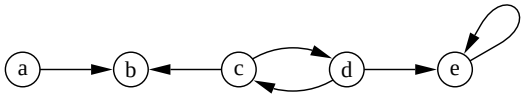


Fig. 1. Graph of Example 1.

In order to be able to reason about such frameworks, it is necessary to group arguments with special properties to *extensions*. One of the basic properties of such an extension is that the arguments are not in conflict with each other.

Definition 2. *Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is said to be* conflict-free (in F), *if there are no $a, b \in S$, such that $(a, b) \in R$. We denote the collection of sets which are conflict-free (in F) by $cf(F)$.*

The first concept of extension we present are the *stable extensions* which are based on the idea that an extension should not only be internally consistent but also able to reject the arguments that are outside the extension.

Definition 3. *Let $F = (A, R)$ be an AF. A set S is a* stable extension *of F, if $S \in cf(F)$ and each $a \in A \setminus S$ is defeated by S in F. We denote the collection all of stable extensions of F by $stable(F)$.*

The framework F from Example 1 has a single stable extension $\{a, d\}$. Indeed $\{a, d\}$ is conflict-free, since a and d are not adjacent. Moreover, each further element b, c, e is defeated by either a or d. In turn, $\{a, c\}$ for instance is not contained in $stable(F)$, although it is clearly conflict free. The obvious reason is that e is not defeated by $\{a, c\}$.

Stable semantics in terms of argumentation are considered as quite restricted. It is often sufficient to consider those arguments which are able to defend themselves from external attacks, like the admissible semantics proposed by Dung [4]:

Definition 4. *Let $F = (A, R)$ be an AF. A set S is an* admissible extension *of F, if $S \in cf(F)$ and each $a \in S$ is defended by S in F. We denote the collection of all admissible extensions of F by $adm(F)$.*

For the framework F from Example 1, we obtain, $adm(F) = \{\emptyset, \{a\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}\}$. By definition, the empty set is always an admissible extension, therefore reasoning over admissible extensions is also limited. In fact, some reasoning (for instance, given an AF $F = (A, R)$, and $a \in A$, is a contained in any extension of F) becomes trivial wrt admissible extensions. Thus, many researchers consider maximal (wrt set-inclusion) admissible sets, called preferred extensions, as more important.

Definition 5. *Let $F = (A, R)$ be an AF. A set S is a* preferred extension *of F, if $S \in adm(F)$ and for each $S' \in adm(F)$, $S \not\subset S'$. We denote the collection of all preferred extensions of F by $pref(F)$.*

Obviously, the preferred extensions of framework F from Example 1 are $\{a, c\}$ and $\{a, d\}$. We note that each stable extension is also preferred, but the converse does not hold, as witnessed by this example.

Finally, we introduce complete and grounded extensions which Dung considered as skeptical counterparts of admissible and preferred extensions, respectively.

Definition 6. *Let $F = (A, R)$ be an AF. A set S is a* complete extension *of F, if $S \in adm(F)$ and, for each $a \in A$ defended by S (in F), $a \in S$ holds. The least (wrt set inclusion) complete extension of F is called the* grounded extension *of F. We denote the collection of all complete (resp., grounded) extensions of F by $comp(F)$ (resp., $ground(F)$).*

The complete extensions of framework F from Example 1 are $\{a, c\}$, $\{a, d\}$, and $\{a\}$, with the last being also the grounded extensions of F.

We briefly review the complexity of reasoning in AFs. To this end, we define the following decision problems for $e \in \{stable, adm, pref, comp, ground\}$:

	stable	*adm*	*pref*	*comp*	*ground*
Cred_e	NP	NP	NP	NP	P
Skept_e	coNP	(trivial)	Π_2^P	P	P

Table 2. Complexity for decision problems in argumentation frameworks.

- Cred_e: Given AF $F = (A, R)$ and $a \in A$. Is a contained in some $S \in e(F)$?
- Skept_e: Given AF $F = (A, R)$ and $a \in A$. Is a contained in each $S \in e(F)$?

The complexity results are depicted in Table 2 (many of them follow implicitly from [16], for the remaining results and discussions see [17, 18]). All NP-entries as well as the coNP-entry and the Π_2^P-entry refer to completeness results. A few further comments are in order: We already mentioned that skeptical reasoning over admissible extensions always is trivially false. Moreover, we note that credulous reasoning over preferred extensions is easier than skeptical reasoning. This is due to the fact that the additional maximality criterion only comes into play for the latter task. Indeed for credulous reasoning the following simple observation makes clear why there is no increase in complexity compared to credulous reasoning over admissible extensions: a is contained in some $S \in adm(F)$ iff a is contained in some $S \in pref(F)$. A similar observation immediately shows why skeptical reasoning over complete extensions reduces to skeptical reasoning over the grounded extension. Finally, we recall that reasoning over the grounded extension is tractable, since the grounded extension of an AF $F = (A, R)$ is given by the least fix-point of the operator $\Gamma_F : 2^A \to 2^A$, defined as $\Gamma_F(S) = \{a \in A \mid a \text{ is defended by } S \text{ in } F\}$ (see [4]).

3.2 Encodings

We now provide a fixed encoding π_e for each extension of type e introduced so far, in such a way that the AF F is given as an input database $\widehat{F}$ and the answer sets of the combined program $\pi_e(\widehat{F})$ are in a certain one-to-one correspondence with the respective extensions. Note that having the fixed program π_e at hand, the only translation required for a given AF F is thus its reformulation as input $\widehat{F}$, which is very simple (see below). With some additions, we can of course combine the different encodings into a single program, where the user just has to specify which type of extensions she wants to compute.

In most cases, we have to guess candidates for the selected type of extensions and then check whether a guessed candidate satisfies the corresponding conditions. We use unary predicates $\mathrm{in}(\cdot)$ and $\mathrm{out}(\cdot)$ to make such a guess for a set $S \subseteq A$, where $\mathrm{in}(a)$ represents that $a \in S$. Thus the following notion of correspondence is relevant for our purposes.

Definition 7. *Let $\mathcal{S} \subseteq 2^{\mathcal{U}}$ be a collection of sets of domain elements and $\mathcal{I} \subseteq 2^{B_\mathcal{U}}$ a collection of sets of ground atoms. We say that $\mathcal{S}$ and $\mathcal{I}$ correspond to each other, in symbols $\mathcal{S} \cong \mathcal{I}$ iff $|\mathcal{S}| = |\mathcal{I}|$ and (i) for each $I \in \mathcal{I}$, there exists an $S \in \mathcal{S}$, such that $\{a \mid \mathrm{in}(a) \in I\} = S$; and (ii) for each $S \in \mathcal{S}$, there exists an $I \in \mathcal{I}$, such that $\{a \mid \mathrm{in}(a) \in I\} = S$.*

Let us first determine how an AF is presented to our programs as input. In fact, we encode a given AF $F = (A, R)$ as follows

$$\widehat{F} = \{\mathrm{arg}(a) \mid a \in A\} \cup \{\mathrm{defeat}(a, b) \mid (a, b) \in R\}.$$

The following program fragment guesses, when augmented by $\widehat{F}$ for a given AF $F = (A, R)$, any subset $S \subseteq A$ and then checks whether the guess is conflict-free in F:

$$\begin{aligned}\pi_{cf} = \{\ & \mathrm{in}(X) \text{ :- } \mathit{not}\ \mathrm{out}(X), \mathrm{arg}(X);\\ & \mathrm{out}(X) \text{ :- } \mathit{not}\ \mathrm{in}(X), \mathrm{arg}(X);\\ & \text{ :- } \mathrm{in}(X), \mathrm{in}(Y), \mathrm{defeat}(X, Y)\}.\end{aligned}$$

Proposition 1. *For any AF F, $cf(F) \cong \mathcal{AS}(\pi_{cf}(\widehat{F}))$.*

The additional rules for the stability test are as follows:

$$\begin{aligned}\pi_{stable} = \pi_{cf} \cup \{\ & \mathrm{defeated}(X) \text{ :- } \mathrm{in}(Y), \mathrm{defeat}(Y, X);\\ & \text{ :- } \mathrm{out}(X), \mathit{not}\ \mathrm{defeated}(X)\}.\end{aligned}$$

The first rule computes those arguments attacked by the current guess, while the constraint eliminates those guesses where some argument not contained in the guess remains undefeated. This brings us to an encoding for stable extensions, which satisfies the following correspondence result.

Proposition 2. *For any AF F, $stable(F) \cong \mathcal{AS}(\pi_{stable}(\widehat{F}))$.*

Next, we give the additional rules for the admissibility test:

$$\begin{aligned}\pi_{adm} = \pi_{cf} \cup \{\ & \mathrm{defeated}(X) \text{ :- } \mathrm{in}(Y), \mathrm{defeat}(Y, X);\\ & \mathrm{not\_defended}(X) \text{ :- } \mathrm{defeat}(Y, X), \mathit{not}\ \mathrm{defeated}(Y);\\ & \text{ :- } \mathrm{in}(X), \mathrm{not\_defended}(X)\}.\end{aligned}$$

The first rule is the same as in π_{stable}. The second rule derives those arguments which are not defended by the current guess, i.e., those arguments which are defeated by some other argument, which itself is not defeated by the current guess. If such a non-defended argument is contained in the guess, we have to eliminate that guess.

Proposition 3. *For any AF F, $adm(F) \cong \mathcal{AS}(\pi_{adm}(\widehat{F}))$.*

We proceed with the encoding for complete extensions, which is also quite straightforward. We define

$$\pi_{comp} = \pi_{adm} \cup \{\text{ :- } \mathrm{out}(X), \mathit{not}\ \mathrm{not\_defended}(X)\}.$$

Proposition 4. *For any AF F, $comp(F) \cong \mathcal{AS}(\pi_{comp}(\widehat{F}))$.*

We now turn to the grounded extension. Suitably encoding the operator Γ_F, we can come up with a stratified program which computes this extension. Note that here we are not able to first guess a candidate for the extension and then check whether the guess satisfies certain conditions. Instead, we "fill" the $\mathrm{in}(\cdot)$-predicate according to the definition of the operator Γ_F. To compute (without unstratified negation) the required predicate for being defended, we now make use of the order $<$ over the domain elements and we derive corresponding predicates for infimum, supremum, and successor.

$$\begin{aligned}\pi_< = \{\ & \mathrm{lt}(X,Y) \text{ :- } \mathrm{arg}(X), \mathrm{arg}(Y), X < Y;\\ & \mathrm{nsucc}(X,Z) \text{ :- } \mathrm{lt}(X,Y), \mathrm{lt}(Y,Z);\\ & \mathrm{succ}(X,Y) \text{ :- } \mathrm{lt}(X,Y), \mathit{not}\, \mathrm{nsucc}(X,Y);\\ & \mathrm{ninf}(Y) \text{ :- } \mathrm{lt}(X,Y);\\ & \mathrm{inf}(X) \text{ :- } \mathrm{arg}(X), \mathit{not}\, \mathrm{ninf}(X);\\ & \mathrm{nsup}(X) \text{ :- } \mathrm{lt}(X,Y);\\ & \mathrm{sup}(X) \text{ :- } \mathrm{arg}(X), \mathit{not}\, \mathrm{nsup}(X)\}.\end{aligned}$$

We now define the desired predicate $\mathrm{defended}(X)$ which itself is obtained via a predicate $\mathrm{defended\_upto}(X,Y)$ with the intended meaning that argument X is defended by the current assignment with respect to all arguments $U \leq Y$. In other words, we let range Y starting from the infimum and then using the defined successor predicate to derive $\mathrm{defended\_upto}(X,Y)$ for "increasing" Y. If we arrive at the supremum element in this way, we finally derive $\mathrm{defended}(X)$. We define

$$\begin{aligned}\pi_{defended} = \{\ & \mathrm{defended\_upto}(X,Y) \text{ :- } \mathrm{inf}(Y), arg(X), \mathit{not}\, \mathrm{defeat}(Y,X);\\ & \mathrm{defended\_upto}(X,Y) \text{ :- } \mathrm{inf}(Y), \mathrm{in}(Z), \mathrm{defeat}(Z,Y), \mathrm{defeat}(Y,X);\\ & \mathrm{defended\_upto}(X,Y) \text{ :- } \mathrm{succ}(Z,Y), \mathrm{defended\_upto}(X,Z),\\ & \qquad\qquad \mathit{not}\, \mathrm{defeat}(Y,X);\\ & \mathrm{defended\_upto}(X,Y) \text{ :- } \mathrm{succ}(Z,Y), \mathrm{defended\_upto}(X,Z),\\ & \qquad\qquad \mathrm{in}(V), \mathrm{defeat}(V,Y), \mathrm{defeat}(Y,X);\\ & \mathrm{defended}(X) \text{ :- } \mathrm{sup}(Y), \mathrm{defended\_upto}(X,Y)\},\ \text{and}\\ \pi_{ground} \ = \ & \pi_< \cup \pi_{defended} \cup \{\mathrm{in}(X) \text{ :- } \mathrm{defended}(X)\}.\end{aligned}$$

Note that π_{ground} is indeed stratified.

Proposition 5. *For any AF F, $ground(F) \cong \mathcal{AS}(\pi_{ground}(\widehat{F}))$.*

Obviously, we could have used the $\mathrm{defended}(\cdot)$ predicate in previous programs, especially π_{comp} could be defined as

$$\pi_{cf} \cup \pi_{defended} \cup \{\text{ :- } \mathrm{in}(X), \mathit{not}\, \mathrm{defended}(X);\ \text{ :- } \mathrm{out}(X), \mathrm{defended}(X)\}.$$

We now continue with the more involved encoding for preferred extensions. Compared to the one for admissible extensions, this encoding requires an additional maximality test. However, this is sometimes quite complicate to encode (see also [19] for a thorough discussion on this issue).

In fact, to compute the preferred extensions, we will use a saturation technique as follows: Having computed an admissible extension S, we make a second guess using new predicates, say $\mathrm{inN}(\cdot)$ and $\mathrm{outN}(\cdot)$, such that they represent a guess $S' \supset S$. For that guess, we will use disjunction (rather than default negation), which allows that *both* $\mathrm{inN}(a)$ and $\mathrm{outN}(a)$ are contained in a possible answer set (under certain conditions), for each a. In fact, exactly such answer sets will correspond to the preferred extension. The saturation is therefore performed in such a way that all predicates $\mathrm{inN}(a)$ and $\mathrm{outN}(a)$ are derived, for those S' which do *not* characterize an admissible extension. If this saturation succeeds for each $S' \supset S$, we want that saturated interpretation to become an answer set. This can be done by using a saturation predicate spoil, which is handled via a constraint $:\!\text{-}\ not\,\mathrm{spoil}$. This ensures that only saturated guesses survive.

Such saturation techniques always require a restricted use of negation. The predicates defined in $\pi_<$ will serve for this purpose. Two new predicates are needed: predicate eq which indicates whether a guess S' represented by atoms $\mathrm{inN}(\cdot)$ and $\mathrm{outN}(\cdot)$ is equal to the guess for S (represented by atoms $\mathrm{in}(\cdot)$ and $\mathrm{out}(\cdot)$). The second predicate we define is $\mathrm{undefeated}(X)$ which indicates that X is not defeated by any element from S'. Both predicates are computed in $\pi_{helpers}$ via predicates $\mathrm{eq\_upto}(\cdot)$ (resp. $\mathrm{undefeated\_upto}(\cdot,\cdot)$) in the same manner as we used $\mathrm{defended\_upto}(\cdot,\cdot)$ for $\mathrm{defended}(\cdot)$ in the module $\pi_{defended}$ above. To this end let

$$
\begin{aligned}
\pi_{helpers} = \pi_< \cup \{\ & \mathrm{eq\_upto}(Y) :\!\text{-}\ \mathrm{inf}(Y), \mathrm{in}(Y), \mathrm{inN}(Y);\\
& \mathrm{eq\_upto}(Y) :\!\text{-}\ \mathrm{inf}(Y), \mathrm{out}(Y), \mathrm{outN}(Y);\\
& \mathrm{eq\_upto}(Y) :\!\text{-}\ \mathrm{succ}(Z,Y), \mathrm{in}(Y), \mathrm{inN}(Y), \mathrm{eq\_upto}(Z);\\
& \mathrm{eq\_upto}(Y) :\!\text{-}\ \mathrm{succ}(Z,Y), \mathrm{out}(Y), \mathrm{outN}(Y), \mathrm{eq\_upto}(Z);\\
& \mathrm{eq} :\!\text{-}\ \mathrm{sup}(Y), \mathrm{eq\_upto}(Y);\\
& \mathrm{undefeated\_upto}(X,Y) :\!\text{-}\ \mathrm{inf}(Y), \mathrm{outN}(X), \mathrm{outN}(Y);\\
& \mathrm{undefeated\_upto}(X,Y) :\!\text{-}\ \mathrm{inf}(Y), \mathrm{outN}(X), \mathit{not}\,\mathrm{defeat}(Y,X);\\
& \mathrm{undefeated\_upto}(X,Y) :\!\text{-}\ \mathrm{succ}(Z,Y), \mathrm{undefeated\_upto}(X,Z),\\
& \qquad\qquad \mathrm{outN}(Y);\\
& \mathrm{undefeated\_upto}(X,Y) :\!\text{-}\ \mathrm{succ}(Z,Y), \mathrm{undefeated\_upto}(X,Z),\\
& \qquad\qquad \mathit{not}\,\mathrm{defeat}(Y,X);\\
& \mathrm{undefeated}(X) :\!\text{-}\ \mathrm{sup}(Y), \mathrm{undefeated\_upto}(X,Y)\}
\end{aligned}
$$

$$
\begin{aligned}
\pi_{spoil} = \{\ & \mathrm{inN}(X) \vee \mathrm{outN}(X) :\!\text{-}\ \mathrm{out}(X); && (1)\\
& \mathrm{inN}(X) :\!\text{-}\ \mathrm{in}(X); && (2)\\
& \mathrm{spoil} :\!\text{-}\ \mathrm{eq}; && (3)\\
& \mathrm{spoil} :\!\text{-}\ \mathrm{inN}(X), \mathrm{inN}(Y), \mathrm{defeat}(X,Y); && (4)\\
& \mathrm{spoil} :\!\text{-}\ \mathrm{inN}(X), \mathrm{outN}(Y), \mathrm{defeat}(Y,X), \mathrm{undefeated}(Y); && (5)\\
& \mathrm{inN}(X) :\!\text{-}\ \mathrm{spoil}, \mathrm{arg}(X); && (6)\\
& \mathrm{outN}(X) :\!\text{-}\ \mathrm{spoil}, \mathrm{arg}(X); && (7)\\
& :\!\text{-}\ \mathit{not}\,\mathrm{spoil}\}. && (8)
\end{aligned}
$$

	$stable$	adm	$pref$	$comp$	$ground$
Cred_e	$\pi_{stable}(\widehat{F}) \models_c a$	$\pi_{adm}(\widehat{F}) \models_c a$	$\pi_{adm}(\widehat{F}) \models_c a$	$\pi_{comp}(\widehat{F}) \models_c a$	$\pi_{ground}(\widehat{F}) \models a$
Skept_e	$\pi_{stable}(\widehat{F}) \models_s a$	(trivial)	$\pi_{pref}(\widehat{F}) \models_s a$	$\pi_{ground}(\widehat{F}) \models a$	$\pi_{ground}(\widehat{F}) \models a$

Table 3. Overview of the encodings of the reasoning tasks for AF $F = (A, R)$ and $a \in A$.

We define

$$\pi_{pref} = \pi_{adm} \cup \pi_{helpers} \cup \pi_{spoil}.$$

When joined with $\widehat{F}$ for some AF $F = (A, R)$, the rules of π_{spoil} work as follows: (1) and (2) guess a new set $S' \subseteq A$, which compares to the guess $S \subseteq A$ (characterized by predicates $\mathrm{in}(\cdot)$ and $\mathrm{out}(\cdot)$ as used in π_{adm}) as $S \subseteq S'$. In case $S' = S$, we obtain predicate eq and derive predicate spoil (rule (3)). The remaining guesses S' are now handled as follows. First, rule (4) derives predicate spoil if the new guess S' contains a conflict. Second, rule (5) derives spoil if the new guess S' contains an element which is attacked by an argument outside S' which itself is undefeated (by S'). Hence, we derived spoil for those $S \subseteq S'$ where either $S = S'$ or S' did not correspond to an admissible extension of F. We now finally spoil up the current guess and derive all $\mathrm{inN}(a)$ and $\mathrm{outN}(a)$ in rules (6) and (7). Recall that due to constraint (8) such spoiled interpretation are the only candidates for answer sets. To turn them into an answer set, it is however necessary that we spoiled for *each* S', such that $S \subseteq S'$; but by definition this is exactly the case if S is a preferred extension.

Proposition 6. *For any AF F, $pref(F) \cong \mathcal{AS}(\pi_{pref}(\widehat{F}))$.*

We summarize the results from this section.

Theorem 1. *For any AF F and $e \in \{stable, adm, comp, ground, pref\}$, it holds that $e(F) \cong \mathcal{AS}(\pi_e(\widehat{F}))$.*

We note that our encodings are *adequate* in the sense that the data complexity of the encodings mirrors the complexity of the encoded task. In fact, depending on the chosen reasoning task, the adequate encodings are depicted in Table 3. Recall that credulous reasoning over preferred extensions reduces to credulous reasoning over admissible extensions; and skeptical reasoning over complete extensions reduces to reasoning over the single grounded extension. The only proper disjunctive program involved is π_{pref}, all other are encodings are disjunction-free. Moreover, π_{ground} is stratified. Stratified programs have at most one answer set, hence there is no need to distinguish between $\models_c$ and $\models_s$. If one now assigns the complexity entries from Table 1 to the encodings as depicted in Table 3, one obtains Table 2.

However, we also can encode more involved decision problems using our programs. As an example consider the Π_2^P-complete problem of *coherence* [17], which decides whether for a given AF F, $pref(F) \subseteq stable(F)$ (recall that $pref(F) \supseteq stable(F)$ always holds). We can decide this problem by extending π_{pref} in such a way that an answer-set of π_{pref} survives only if it does not correspond to a stable extension. By

definition, the only possibility to do so, is if some undefeated argument is not contained in the extension.

Corollary 1. *The coherence problem for an AF F holds iff the program*

$$\pi_{pref}(\widehat{F}) \cup \{v \text{ :- } \mathrm{out}(X), \mathit{not}\, \mathrm{defeated}(X);\ \text{ :- } \mathit{not}\, v\}$$

has no answer set.

4 Encodings for Generalizations of Argumentation Frameworks

4.1 Value-Based Argumentation Frameworks

As a first example for generalizing basic AFs, we deal with value-based argumentation frameworks (VAFs) [6] which themselves generalize the preference-based argumentation frameworks [5]. Again we give the definition wrt the universe $\mathcal{U}$.

Definition 8. *A* value-based argumentation framework (VAF) *is a 5-tuple $F = (A, R, \Sigma, \sigma, <)$ where $A \subseteq \mathcal{U}$ are arguments, $R \subseteq A \times A$, $\Sigma \subseteq \mathcal{U}$ is a non-empty set of values disjoint from A, $\sigma : A \to \Sigma$ assigns a value to each argument from A , and $<$ is a preference relation (irreflexive, asymmetric) between values.*

Let $\ll$ be the transitive closure of $<$. An argument $a \in A$ defeats *an argument $b \in A$ in F if and only if $(a, b) \in R$ and $(b, a) \notin \ll$.*

Using this notion of defeat, we say in accordance to Definition 1 that a set $S \subseteq A$ of arguments *defeats* b (in F), if there is an $a \in S$ which defeats b. An argument $a \in A$ is *defended* by $S \subseteq A$ (in F) iff, for each $b \in A$, it holds that, if b defeats a in F, then S defeats b in F. Using these notions of defeat and defense, the definitions in [6] for conflict-free sets, admissible extensions, and preferred extensions are exactly along the lines of Definition 2, 4, and 5, respectively.

In order to compute these extensions for VAFs we thus only need to slightly adapt the modules introduced in Section 3.2. In fact, we just overwrite $\widehat{F}$ for a VAF F as

$$\begin{aligned}\widehat{F} = {} & \{\mathrm{arg}(a) \mid a \in A\} \cup \{\mathrm{attack}(a, b) \mid (a, b) \in R\} \cup \\ & \{\mathrm{val}(a, \sigma(a)) \mid a \in A\} \cup \{\mathrm{valpref}(w, v) \mid v < w\};\end{aligned}$$

and we require one further module, which now obtains the $\mathrm{defeat}(\cdot, \cdot)$ relation accordingly:

$$\begin{aligned}\pi_{vaf} = \{ & \mathrm{valpref}(X, Z) \text{ :- } \mathrm{valpref}(X, Y), \mathrm{valpref}(Y, Z); \\ & \mathrm{pref}(X, Y) \text{ :- } \mathrm{valpref}(U, V), \mathrm{val}(X, U), \mathrm{val}(Y, V); \\ & \mathrm{defeat}(X, Y) \text{ :- } \mathrm{attack}(X, Y), \mathit{not}\, \mathrm{pref}(Y, X)\}.\end{aligned}$$

We obtain the following theorem using the new concepts for $\widehat{F}$ and π_{vaf}, as well as re-using π_{adm} and π_{pref} from Section 3.2.

Theorem 2. *For any VAF F and $e \in \{adm, pref\}$, $e(F) \cong \mathcal{AS}(\pi_{vaf} \cup \pi_e(\widehat{F}))$.*

For the other notions of extensions, we can employ our encodings from Section 3.2 in a similar way. The concrete composition of the modules however depends on the exact definitions, and whether they make use of the notion of a defeat in a uniform way. In [20], for instance, stable extensions for a VAF F are defined as those conflict-free subsets S of arguments, such that each argument not in S is attacked (rather than defeated) by S. Still, we can obtain a suitable encoding quite easily using the following redefined module:

$$\begin{aligned}\pi_{stable} = \pi_{cf} \cup \{\ &\mathrm{attacked}(X) \text{ :- } \mathrm{in}(Y), \mathrm{attack}(Y,X);\\ &\text{ :- } \mathrm{out}(X), \mathit{not}\ \mathrm{attacked}(X)\}.\end{aligned}$$

Theorem 3. *For any VAF F, $stable(F) \cong \mathcal{AS}(\pi_{vaf} \cup \pi_{stable}(\widehat{F}))$.*

The coherence problem for VAFs thus can be decided as follows.

Corollary 2. *The coherence problem for a VAF F holds iff the program*

$$\begin{aligned}&\pi_{pref}(\widehat{F}) \cup \{\mathrm{attacked}(X) \text{ :- } \mathrm{in}(Y), \mathrm{attack}(Y,X);\\ &v \text{ :- } \mathrm{out}(X), \mathit{not}\ \mathrm{attacked}(X);\ \text{ :- } \mathit{not}\ v\}\end{aligned}$$

has no answer set.

4.2 Bipolar Argumentation Frameworks

Bipolar argumentation frameworks [7] augment basic AFs by a second relation between arguments which indicates supports independent from defeats.

Definition 9. *A bipolar argumentation framework (BAF) is a tuple $F = (A, R_d, R_s)$ where $A \subseteq \mathcal{U}$ is a set of arguments, and $R_d \subseteq A \times A$ and $R_s \subseteq A \times A$ are the defeat (resp., support) relation of F.*

An argument a defeats *an argument b in F if there exists a sequence $a_1, \ldots, a_{n+1}$ of arguments from A (for $n \geq 1$), such that $a_1 = a$, and $a_{n+1} = b$, and either*

- *$(a_i, a_{i+1}) \in R_s$ for each $1 \leq i \leq n-1$ and $(a_n, a_{n+1}) \in R_d$; or*
- *$(a_1, a_2) \in R_d$ and $(a_i, a_{i+1}) \in R_s$ for each $2 \leq i \leq n$.*

As before, we say that a set $S \subseteq A$ *defeats* an argument b in F if some $a \in S$ defeats b; an argument $a \in A$ is *defended* by $S \subseteq A$ (in F) iff, for each $b \in A$, it holds that, if b defeats a in F, then S defeats b in F.

Again, we just need to adapt the input database $\widehat{F}$ and incorporate the new defeat-relation. Other modules from Section 3.2 can then be reused. In fact, we define for a given BAF $F = (A, R_d, R_s)$,

$$\widehat{F} = \{\mathrm{arg}(a) \mid a \in A\} \cup \{\mathrm{attack}(a,b) \mid (a,b) \in R_d\} \cup \{\mathrm{support}(a,b) \mid (a,b) \in R_s\},$$

and for the defeat relation we first compute the transitive closure of the $\mathrm{support}(\cdot,\cdot)$-predicate and then define $\mathrm{defeat}(\cdot,\cdot)$ accordingly.

$$\begin{aligned}\pi_{baf} = \{\ &\mathrm{support}(X,Z) \text{ :- } \mathrm{support}(X,Y), \mathrm{support}(Y,Z);\\ &\mathrm{defeat}(X,Y) \text{ :- } \mathrm{attack}(X,Y);\\ &\mathrm{defeat}(X,Y) \text{ :- } \mathrm{attack}(Z,Y), \mathrm{support}(X,Z);\\ &\mathrm{defeat}(X,Y) \text{ :- } \mathrm{attack}(X,Z), \mathrm{support}(Z,Y)\}.\end{aligned}$$

Following [7], we can use this notion of defeat to define conflict-free sets, stable extensions, admissible extensions and preferred extensions[1] exactly along the lines of Definition 2, 3, 4, and 5, respectively.

Theorem 4. *For any BAF F and $e \in \{stable, adm, pref\}$, $e(F) \cong \mathcal{AS}(\pi_{baf} \cup \pi_e(\widehat{F}))$.*

More specific variants of admissible extensions from [7] are obtained by replacing the notion a conflict-free set by other concepts.

Definition 10. *Let $F = (A, R_d, R_s)$ be a BAF and $S \subseteq A$. Then S is called* safe *in F if for each $a \in A$, such that S defeats a, $a \notin S$ and there is no sequence $a_1, \ldots, a_n$ ($n \geq 2$), such that $a_1 \in S$, $a_n = a$, and $(a_i, a_{i+1}) \in R_s$, for each $1 \leq i \leq n-1$. A set S is* closed under R_s *if, for each $(a, b) \in R_s$, it holds that $a \in S$ if and only if $b \in S$.*

Note that for a BAF F, each safe set (in F) is conflict-free (in F). We also remark that a set S of arguments is closed under R_s iff S is closed under the transitive closure of R_s.

Definition 11. *Let $F = (A, R_d, R_s)$ be a BAF. A set $S \subseteq A$ is called an s-*admissible extension *of F if S is safe (in F) and each $a \in S$ is defended by S (in F). A set $S \subseteq A$ is called a c-*admissible extension *of F if S is* closed under R_s*, conflict-free (in F), and each $a \in S$ is defended by S (in F). We denote the collection of all s-admissible extensions (resp. of all c-admissible extensions) of F by $sadm(F)$ (resp. by $cadm(F)$).*

We define now further programs as follows

$$\begin{aligned}
\pi_{sadm} = \pi_{adm} \cup \{\ & \mathrm{supported}(X) \text{ :- } \mathrm{in}(Y), \mathrm{support}(Y, X); \\
& \text{ :- } \mathrm{supported}(X), \mathrm{defeated}(X)\ \} \\
\pi_{cadm} = \pi_{adm} \cup \{\ & \text{ :- } \mathrm{support}(X, Y), \mathrm{in}(X), \mathrm{out}(Y); \\
& \text{ :- } \mathrm{support}(X, Y), \mathrm{out}(X), \mathrm{in}(Y)\ \}.
\end{aligned}$$

Finally, one defines s-preferred (resp. c-preferred) extensions as maximal (wrt set-inclusion) s-admissible (resp. c-admissible) extensions.

Definition 12. *Let $F = (A, R_d, R_s)$ be a BAF. A set $S \subseteq A$ is called an s-*preferred extension *of F if $S \in sadm(F)$ and for each $S' \in sadm(F)$, $S \not\subseteq S'$. Likewise, a set $S \subseteq A$ is called a c-*preferred extension *of F if $S \in cadm(F)$ and for each $S' \in cadm(F)$, $S \not\subseteq S'$. By $spref(F)$ (resp. $cpref(F)$) we denote the collection of all s-preferred extensions (resp. of all c-preferred extensions) of F.*

Again, we can reuse parts of the π_{pref}-program from Section 3.2. The only additions necessary are to spoil in case the additional requirements are violated. We define

[1] These extensions are called d-admissible and resp. d-preferred in [7].

$$
\begin{aligned}
\pi_{spref} &= \pi_{sadm} \cup \pi_{helpers} \cup \pi_{\mathrm{spoil}} \cup \\
&\quad \{\ \mathrm{supported}(X) \text{ :- } \mathrm{inN}(Y), \mathrm{support}(Y, X); \\
&\quad\ \ \mathrm{spoil} \text{ :- } \mathrm{supported}(X), \mathrm{defeated}(X)\ \} \\
\pi_{cpref} &= \pi_{cadm} \cup \pi_{helpers} \cup \pi_{\mathrm{spoil}} \cup \\
&\quad \{\ \mathrm{spoil} \text{ :- } \mathrm{support}(X, Y), \mathrm{inN}(X), \mathrm{outN}(Y); \\
&\quad\ \ \mathrm{spoil} \text{ :- } \mathrm{support}(X, Y), \mathrm{outN}(X), \mathrm{inN}(Y)\ \}.
\end{aligned}
$$

Theorem 5. *For any BAF F and $e \in \{sadm, cadm, spref, cpref\}$, we have $e(F) \cong \mathcal{AS}(\pi_{baf} \cup \pi_e(\widehat{F}))$.*

Slightly different semantics for BAFs occur in [8], where the notion of defense is based on R_d, while the notion of conflict remains evaluated with respect to the more general concept of defeat as given in Definition 9. However, also such variants can be encoded within our system by a suitable composition of the concepts introduced so far.

Again, we note that we can put together encodings for complete and grounded extensions for BAFs, which have not been studied in the literature.

5 Discussion

In this work we provided logic-program encodings for computing different types of extensions in Dung's argumentation framework as well as in some recent extensions of it. To the best of our knowledge, so far no system is available which supports such a broad range of different semantics, although nowadays a number of implementations exists[2]. The encoding (together with some examples) is available on the web and can be run with the answer-set solver DLV [10]. We note that DLV also supplies the built-in predicate $<$ which we used in some of our encodings. Moreover, DLV provides further language-extensions which might lead to alternative encodings; for instance weak constraints could be employed to select the grounded extension from the admissible, or prioritization techniques could be used to compute the preferred extensions.

The work which is closest related to ours is by Nieves *et al.* [21] who also suggest to use answer-set programming for computing extensions of argumentation frameworks. The most important difference is that in their work the program has to be re-computed for each new instance, while our system relies on a *single fixed* program which just requires the actual instance as an input database. We believe that our approach thus is more reliable and easier extendible to further formalisms.

Future work includes a comparison of the efficiency of different implementations and an extension of our system by incorporating further recent notions of semantics, for instance, the semi-normal semantics [22] or the ideal semantics [23].

Acknowledgments The authors would like to thank Wolfgang Faber for comments on an earlier draft of this paper. This work was partially supported by the Austrian Science Fund (FWF) under grant P20704-N18.

[2] See http://www.csc.liv.ac.uk/~azwyner/software.html for an overview.

References

1. Bench-Capon, T.J.M., Dunne, P.E.: Argumentation in artificial intelligence. Artif. Intell. **171** (2007) 619–641
2. Besnard, P., Doutre, S.: Checking the acceptability of a set of arguments. In: Proceedings NMR'04. (2004) 59–64
3. Egly, U., Woltran, S.: Reasoning in argumentation frameworks using quantified boolean formulas. In: Proceedings COMMA'06, IOS Press (2006) 133–144
4. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. **77** (1995) 321–358
5. Amgoud, L., Cayrol, C.: A reasoning model based on the production of acceptable arguments. Ann. Math. Artif. Intell. **34** (2002) 197–215
6. Bench-Capon, T.J.M.: Persuasion in practical argument using value-based argumentation frameworks. J. Log. Comput. **13** (2003) 429–448
7. Cayrol, C., Lagasquie-Schiex, M.C.: On the acceptability of arguments in bipolar argumentation frameworks. In: Proceedings ECSQARU'05. Volume 3571 of LNCS., Springer (2005) 378–389
8. Amgoud, L., Cayrol, C., Lagasquie, M.C., Livet, P.: On bipolarity in argumentation frameworks. International Journal of Intelligent Systems **23** (2008) 1–32
9. Baroni, P., Giacomin, M.: A systematic classification of argumentation frameworks where semantics agree. In: Proceedings COMMA'08, IOS Press (2008) 37–48
10. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7** (2006) 499–562
11. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. **25** (1999) 241–273
12. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9** (1991) 365–386
13. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Proceedings LPNMR'07. Volume 4483 of LNCS., Springer (2007) 3–17
14. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22** (1997) 364–418
15. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys **33** (2001) 374–425
16. Dimopoulos, Y., Torres, A.: Graph theoretical structures in logic programs and default theories. Theor. Comput. Sci. **170** (1996) 209–244
17. Dunne, P.E., Bench-Capon, T.J.M.: Coherence in finite argument systems. Artif. Intell. **141** (2002) 187–203
18. Coste-Marquis, S., Devred, C., Marquis, P.: Symmetric argumentation frameworks. In: Proceedings ECSQARU'05. Volume 3571 of LNCS., Springer (2005) 317–328
19. Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. Theory and Practice of Logic Programming **6** (2006) 23–60
20. Bench-Capon, T.J.M.: Value-based argumentation frameworks. In: Proceedings NMR'02. (2002) 443–454
21. Nieves, J.C., Osorio, M., Cortés, U.: Preferred extensions as stable models. Theory and Practice of Logic Programming **8** (2008) 527–543
22. Caminada, M.: Semi-stable semantics. In: Proceedings COMMA'06, IOS Press (2006) 121–130
23. Dung, P.M., Mancarella, P., Toni, F.: Computing ideal sceptical argumentation. Artif. Intell. **171** (2007) 642–674

Efficient Parallel ASP Instantiation via Dynamic Rewriting*

Simona Perri, Francesco Ricca, and Saverio Vescio

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{perri,ricca,vescio}@mat.unical.it

Abstract. Answer Set Programming (ASP) is a powerful formalism for knowledge representation and reasoning. The computation of most ASP systems follows a two-phase approach: an instantiation (or grounding) phase generates a variable-free program which is then evaluated by propositional algorithms in the second phase. The instantiation process may be very expensive, especially for real-world problems, where huge input data are often to be dealt with.
A method that exploits the capabilities of multi-processor machines for improving instantiation performance has been recently proposed. This method, implemented in the grounding module of the ASP system DLV, proved to be effective especially when dealing with programs consisting of many rules.
In this paper, a dynamic rewriting of input rules is proposed that enhances the efficacy of the parallel evaluation also in the case of programs with very few rules. The effect of the technique is twofold: on the one hand, a kind of or-parallelism is induced by rewriting each rule at running time; on the other hand, the workload is dynamically distributed among processing units according to an heuristics.
Dynamic rewriting was implemented, and an experimental analysis was conducted that confirms the effectiveness of the technique. In particular, the new parallel implementation always outperforms the (sequential) DLV instantiator, and compared with the previous parallel method offers a very relevant gain especially in the case of programs with very few rules.

1 Introduction

In the last few years, multi-core/multi-processor architectures have become standard, thus making Symmetric MultiProcessing (SMP) [1] common also for entry-level systems and PCs. The principle behind SMP architectures is very simple: two or more identical processors connect to a single shared main memory, enabling simultaneous multithread execution. Such technology has been recently exploited with profit in the field of Answer Set Programming (ASP).

ASP is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [2–7] which features a high declarative nature combined with a relatively high expressive power [8, 9]. There are nowadays a number of systems that support ASP and its variants [8, 10–17]. The kernel modules of ASP

* 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."

systems work on a ground instantiation of the input program. Thus, an input program $\mathcal{P}$ first undergoes the so-called instantiation process, which produces a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$, but not containing any variable. This phase is computationally very expensive (see [7, 9]); thus, having an efficient instantiation procedure is, in general, crucial for the performance of the entire ASP systems. Indeed, recent applications of ASP in different emerging areas (see e.g., [18–21]), have evidenced the practical need for faster and scalable ASP instantiators.

In [22] a technique for the parallel instantiation of ASP programs was proposed, allowing for the performance of instantiators to be improved by exploiting the power of multiprocessor computers. The technique takes advantage of some structural properties of input programs in order to reduce the usage of concurrency control mechanisms [1], and, thus, the so-called parallel overhead. The strategy focuses on two different aspects of the instantiation process: on the one hand, it examines the structure of the input program $\mathcal{P}$, splits it into modules (or sub-programs) and, according to the interdependencies between the modules, decides which of them can be processed in parallel; on the other hand, it parallelizes the evaluation of rules within each module. This strategy has been implemented into the instantiator module of the ASP system DLV [8], thus obtaining a parallel ASP instantiator.

This parallel system proved to be effective especially in the instantiation of programs consisting of several rules with a large amount of input data [22]. However, it is not fully exploitable in case of programs with few rules. The reason for this behavior can be easily understood by considering the following disjunctive encoding for the well-known 3-Colorability problem:

$$(r) \quad col(X, red) \vee col(X, yellow) \vee col(X, green) \text{ :- } node(X).$$
$$(c) \quad \text{:- } col(X, C),\ col(Y, C),\ edge(X, Y).$$

Predicates $node$ and $edge$ represent the input graph; rule (r) guesses the possible colorings of the graph, and the constraint (c) imposes that two adjacent nodes cannot have the same color.

In this case, the technique proceeds by first instantiating (r), thus computing the extension of col, and then, only once this is done, by processing the constraint (c). Thus, such encoding does not allow the existing technique to make the evaluation parallel at all. However, one may provide different encodings (with more rules) for the same problem, which are more amenable for the technique. In general, this would require the user to know *how* the evaluation process works, while writing a program: clearly, such a requirement is not desirable for a fully declarative system. Nevertheless, an automatic rewriting of the input program for an equivalent one, whose evaluation can be made more parallel, could make this optimization process transparent to the user.

In this paper, a dynamic rewriting of input rules is proposed that enhances the efficacy of the existing parallel evaluation technique, especially in the case of programs with very few rules. The basic idea is to rewrite input rules at execution time in order to induce a form of Or-parallelism [23–26]. This can be obtained, given a rule r, by "splitting" the extension of one single body predicate p of r in several parts. Each part is associated with a different temporary predicate; and, for each of those predicates, say p_i, a new rule, obtained by replacing p with p_i, is produced. The so-created rules will

be instantiated in parallel in place of r; when they are done, a realign step gets rid of the new names in order to obtain the same output of the original algorithm.

However, the choice of the most convenient predicate to split is not trivial; indeed, a "bad" split might reduce or neutralize the benefits of parallelism, thus making the overall time consumed by the parallel evaluation not optimal (and, in some corner case, even worse than the time required to instantiate the original encoding). Thus, an heuristic has also been proposed to select the "best" predicate to split in order to minimize the parallel execution time. Summarizing, the contributions of this paper are the following:

- A technique is presented for rewriting rules of ASP programs in such a way that they can be evaluated in parallel; rules are rewritten at execution time, thus dynamically distributing the workload among processing units.
- An heuristic for selecting the most convenient way for rewriting a rule is proposed aiming at minimizing the parallel time.
- An implementation of the dynamic rewriting was done into the parallel version of the DLV instantiator.
- An experimental analysis was conducted for assessing the technique.

The results of the experiments show that the new parallel implementation always outperforms the (sequential) DLV instantiator, and, compared with the previous parallel method, offers a very relevant gain especially in case of programs with few rules.

2 Answer Set Programming

In this section, we briefly recall syntax and semantics of Answer Set Programming.

Syntax. A variable or a constant is a *term*. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where p is an atom. A *disjunctive rule* (*rule*, for short) r is a formula $a_1 \vee \cdots \vee a_n$:- $b_1, \cdots, b_k$, `not` $b_{k+1}, \cdots,$ `not` b_m. where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are 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 the conjunction $b_1, ..., b_k$, `not` $b_{k+1}, ...,$ `not` b_m is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*.

$H(r)$ denotes the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, ..., b_k,$ `not` $b_{k+1}, \ldots,$ `not` $b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. An atom, a literal, a rule, or a program is *ground* if no variables appear in it. Accordingly with the database terminology, a predicate occurring only in *facts* is referred to as an *EDB* predicate, all others as *IDB* predicates; the set of facts of $\mathcal{P}$ is denoted by $EDB(\mathcal{P})$.

Semantics. Let $\mathcal{P}$ be a program. The *Herbrand Universe* and the *Herbrand Base* of $\mathcal{P}$ are defined in the standard way and denoted by $U_\mathcal{P}$ and $B_\mathcal{P}$, respectively.

Given a rule r occurring in $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the

variables occurring in r to constants in $U_{\mathcal{P}}$; $ground(\mathcal{P})$ denotes the set of all the ground instances of the rules occurring in $\mathcal{P}$.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal `not` A is *true* w.r.t. I if A is false w.r.t. I; otherwise `not` A is false w.r.t. I. Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$.

Given a ground program $\mathcal{P}$ and an interpretation I, the *reduct* of $\mathcal{P}$ w.r.t. I is the subset $\mathcal{P}^I$ of $\mathcal{P}$, which is obtained from $\mathcal{P}$ by deleting rules in which a body literal is false w.r.t. I. Note that the above definition of reduct, proposed in [27], simplifies the original definition of Gelfond-Lifschitz (GL) transform [2], but is fully equivalent to the GL transform for the definition of answer sets [27].

Let I be an interpretation for a program $\mathcal{P}$. I is an *answer set* (or stable model) for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the program $\mathcal{P}^I$) [28, 2]. The set of all answer sets for $\mathcal{P}$ is denoted by $ANS(\mathcal{P})$.

3 Parallel Instantiation of ASP programs

In this Section a sketchy description of the parallel instantiation algorithm of [22] is provided. A detailed discussion about this technique is out of the scope of this paper; for further insights we refer the reader to [22].

Given an input program $\mathcal{P}$, the algorithm efficiently generates a ground instantiation of the input program that has the same answer sets as the full one, but is much smaller in general. In order to generate a small ground program equivalent to $\mathcal{P}$, the parallel instantiator computes ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$, and thus avoiding the combinatorial explosion which can be obtained by naively considering all the atoms in the Herbrand Base. This is done by taking into account some structural information of the input program, concerning the dependencies among IDB predicates.

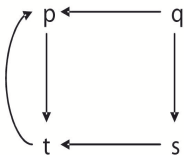
In particular, each program $\mathcal{P}$ is associated with a graph, called the *Dependency Graph* of $\mathcal{P}$, which, intuitively, describes how predicates depend on each other. More in detail, given a program $\mathcal{P}$, the *Dependency Graph* of $\mathcal{P}$ is a directed graph $G_{\mathcal{P}} = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of $\mathcal{P}$, and E contains an arc $e = (p, q)$ if there is a rule r in $\mathcal{P}$ such that q occurs in the head of r and p occurs in a positive literal of the body of r.

The graph $G_{\mathcal{P}}$ induces a subdivision of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate p if p appears in the head of r. For each strongly connected component (SCC)[1] C of $G_{\mathcal{P}}$, the

[1] A strongly connected component of a directed graph is a maximal subset of the vertices, such that every vertex is reachable from every other vertex.

set of rules defining all the predicates in C is called *module* of C and is denoted by $\mathcal{P}_c$. A rule r occurring in a module $\mathcal{P}_c$ (i.e., defining some predicate $q \in C$) is said to be *recursive* if there is a predicate $p \in C$ occurring in the positive body of r; otherwise, r is said to be an *exit rule*. As an example, consider the following program $\mathcal{P}$, where a is an EDB predicate, and its dependency graph $G_{\mathcal{P}}$:

$$
\begin{array}{l}
p(X,Y) \vee s(Y) \text{ :- } q(X), q(Y), not\ t(X,Y). \\
p(X,Y) \text{ :- } q(X), t(X,Y). \\
q(X) \text{ :- } a(X). \\
t(X,Y) \text{ :- } p(X,Y), s(Y).
\end{array}
$$



the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{q\}$ and $\{p,t\}$. They correspond to the three following modules: $\{\ p(X,Y) \vee s(Y) \text{:-} q(X), q(Y), not\ t(X,Y).\ \}$, $\{\ q(X) \text{:-} a(X).\ \}$, and $\{\ p(X,Y) \text{:-} q(X), t(X,Y).\ \ p(X,Y) \vee s(Y) \text{:-} q(X), q(Y), not\ t(X,Y).\ \ t(X,Y) \text{:-} p(X,Y), s(Y).\}$ Note that the first and second module do not contain recursive rules, while the third one contains one exit rule, namely $p(X,Y) \vee s(Y) \text{:-} q(X), q(Y), not\ t(X,Y)$, and two recursive rules.

The dependency graph induces a partial ordering among its SCCs, defined as follows: for any pair of SCCs A, B of $G_{\mathcal{P}}$, we say that B *directly depends on* A if there is an arc from a predicate of A to a predicate of B; and, B *depends* on A if there is a path in the Dependency Graph from A to B.

Intuitively, this partial ordering guarantees that a node A precedes a node B if the program module corresponding to A has to be evaluated before the one of B. Moreover, if two components do not depend on each other, they can be evaluated in parallel.

The parallel instantiation algorithm exploits this partial ordering in order to both produce a small instantiation and identify modules that can be evaluated in parallel. It follows a pattern similar to the classical producer-consumers problem. A *manager* thread (acting as a producer) identifies the components of the dependency graph of the input program $\mathcal{P}$ that can run in parallel, and delegates their instantiation to a number of *instantiator* threads (acting as consumers).

The *Parallel_Instantiate* procedure, shown in Figure 1, acts as a manager. It receives as input both a program $\mathcal{P}$ to be instantiated and its Dependency Graph $G_{\mathcal{P}}$; and it outputs a set of ground rules Π, such that $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. First of all, the algorithm creates a new set of atoms S that will contain the subset of the Herbrand Base significant for the instantiation; more in detail, S will contain, for each predicate p in the program, the extension of p, that is, the set of all the ground atoms having the predicate name of p (significant for the instantiation).

Initially, $S = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, the manager checks, whether some SCC C can be instantiated; in particular, it checks if there is some other component C' such that C depends on C' and C' has not been evaluated yet. As soon as a component C is processable, a new *ComponentInstantiator* thread is spawned for instantiating C.

Procedure *ComponentInstantiator*, in turn, takes as input, among the others, the component C to be instantiated and the set S; for each atom a belonging to C, and for each rule r defining a, it computes the ground instances of r containing only atoms which can possibly be derived from $\mathcal{P}$. At the same time, it updates the set S with the

```
Procedure Parallel_Instantiate (P: Program; G_P: DependencyGraph; var Π: GroundProgram)
begin
    var S: SetOfAtoms; var C: SetOfPredicates;
    S = EDB(P); Π := ∅;
    while G_P ≠ ∅ do
        take a SCC C from G_P that can run in parallel
        Spawn(ComponentInstantiator, P, C, S, Π, G_P)
    end while
end;
Procedure ComponentInstantiator (P: Program; C: Component; var S: SetOfAtoms;
                                 var Π: GroundProgram; var G_P: DependencyGraph)
begin
    var ΔNS: SetOfAtoms; var ΔS: SetOfAtoms;
    ΔS := ∅; ΔNS := ∅ ;
    for each r ∈ Exit(C, P); do
        I_r = Spawn (InstantiateRule, r, S, ΔS, ΔNS, Π);
    for each r ∈ Exit(C, P); do
        join_with_thread(I_r);
    do
        ΔS := ΔNS; ΔNS := ∅ ;
        for each r ∈ Recursive(C, P); do
            I_r = Spawn (InstantiateRule, r, S, ΔS, ΔNS, Π);
        for each r ∈ Recursive(C, P); do
            join_with_thread(I_r);
        S := S ∪ ΔS;
    while ΔNS ≠ ∅
    Remove C from G_P;
end Procedure;
Procedure InstantiateRule (r: rule; S: SetOfAtoms; ΔS: SetOfAtoms
                           var ΔNS: SetOfAtoms var Π: GroundProgram)
/* Given S and ΔS, builds all the ground instances of r, adds them to Π, and add to ΔNS
   the head atoms of the newly generated ground rules. */
```

Fig. 1. The Parallel Instantiation Procedures.

atoms occurring in the heads of the rules of Π. To this end, each rule r in the program module of C is processed by calling procedure *InstantiateRule*. This, given the set of atoms which are known to be significant up to now, builds all the ground instances of r, adds them to Π, and marks as significant the head atoms of the newly generated rules.

It is worth noting that, exit rules are instantiated by a single call to *InstantiateRule*, whereas recursive ones are processed several times according to a semi-naïve evaluation technique [29], where at each iteration n only the significant information derived during iteration $n-1$ has to be used. This is implemented by partitioning significant atoms into three sets: ΔS, S, and ΔNS. ΔNS is filled with atoms computed during current iteration (say n); ΔS contains atoms computed during previous iteration (say $n-1$); and, S contains the ones previously computed (up to iteration $n-2$).

Initially, ΔS and ΔNS are empty; the exit rules contained in the program module of C are evaluated and, in particular, one new thread for each exit rule, running procedure *InstantiateRule*, is spawned. Only once all the threads are done, recursive rules are processed (do-while loop). At the beginning of each iteration, ΔNS is assigned to ΔS, i.e. the new information derived during iteration n is considered as significant information for iteration $n+1$. Then, for each recursive rule, a new thread is spawned,

running procedure *InstantiateRule*, which receives as input S and ΔS; when all threads terminate, ΔS is added to S (since it has already been exploited). The evaluation stops whenever no new information has been derived (i.e. $\mathcal{N}S = \emptyset$). Eventually, component C is removed from Π.

Proposition 1. [22] Let $\mathcal{P}$ be an ASP program, and Π be the ground program generated by the algorithm *Parallel_Instantiate*. Then $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$ (i.e. $\mathcal{P}$ and $\Pi \cup EDB(\mathcal{P})$ have the same answer sets). □

4 Parallel Instantiation via Dynamic Rewriting

In this section, a rewriting technique is described that enhances the parallel instantiation algorithm of the previous Section.

Some Motivation. As already pointed out, there are problem encodings that do not allow the instantiation technique described in Section 3, to make the evaluation parallel at all; the following encoding of the 3-Colorability problem, is an example for that:

$$(r) \quad col(X, red) \vee col(X, yellow) \vee col(X, green) \text{ :- } node(X).$$
$$(c) \quad \text{:- } col(X, C),\ col(Y, C),\ edge(X, Y).$$

However, one may provide different encodings for the same problem, which are more amenable for the application of the technique. Oversimplifying, since each rule of the input program is processed by one processing unit, one may think of rewriting it into an equivalent program containing several rules. For instance, the following is a possible rewriting for the constraint (c), of the 3-Colorability encoding reported above:

$$(c_1) \quad \text{:- } col(X, C),\ col(Y, C),\ edge1(X, Y).$$
$$(c_2) \quad \text{:- } col(X, C),\ col(Y, C),\ edge2(X, Y).$$

The set of edges is *split up* into two subsets, represented by predicates $edge1$ and $edge2$. The evaluation of constraints (c_1) and (c_2) is equivalent to the evaluation of the original constraint c, modulo renaming, but the computation now can be carried out in parallel by two different processing units (instantiators).

This rewriting strategy can be straightforwardly extended for allowing more than two instantiators to work in parallel, and it can be generalized in order to deal with any program. However, there are different, sometimes many, ways to apply it. For instance, another possible encoding for 3-Colorability could be obtained by working on a literal whose predicate name is col, and by introducing new predicates $col_1 \ldots col_n$ (obtained by distributing the extension of col). Hereafter, with a small abuse of notation we indicate as extension of a literal l the extension of the predicate p corresponding to l (having the same name as l). Note that, differently from $edge$, col is not an EDB predicate (it occurs in the head of rule (r)); thus, in this case, a rewriting preserving the original semantics, would require to further modify the original program. Indeed, the extension of col is not known a-priori; thus, the split of col has to be induced by the split of the predicates it depends on by means of other rules. This may lead to an intricate rewriting of the entire program (not only rules to be split) and a possibly slower instantiation.

However, the extension of the predicate to be split is known during the instantiation when the rule is taken for evaluating it. Thus, if the rewriting is performed at execution

time, a rule can be split without involving the entire program. Moreover, it can be easily automatized in order to be transparently applied. Note that, this strategy somewhat induces a form of *Or-parallelism* [23–26], which is here simulated via rewriting.

Dynamic Rewriting. The enhanced parallel instantiation algorithms are now described in detail that are based on the idea described above.

Procedure *ComponentInstantiator* used in the algorithm of Section 3 is replaced by a new one, called *ComponentInstantiator_Rew*, reported in Figure 2. It takes as input the component C to be instantiated and the set of significant atoms S; and for each atom a belonging to C, and for each rule r defining a, it computes the ground instances of r.

At the beginning, the new set of atoms ΔS and $\mathcal{N}S$ (which initially are empty) are created; then, exit rules are evaluated. More in detail, each of them is rewritten into a set of new exit rules which are added to $\mathcal{P}$. This is done by calling the procedure *Rewrite* which is detailed in the following. At this point, a thread running *InstantiateRule* is spawned for each exit rule. Only when all the threads are done, function *Realign* (i) restores S and ΔS by removing all the ground atoms inserted by procedure *Rewrite*, (ii) properly formats the output ground rules in such a way that "split predicates" do not appear; (iii) deletes from the program $\mathcal{P}$ all the rules containing split predicates and reintroduces the original ones. Now, as in *ComponentInstantiator*, recursive rules are evaluated according to a semi-naïve schema. Also in this case, rules are first rewritten and the output is "realigned". However, since recursive rules are processed several times, this strategy is applied at each iteration of the do-while loop.

One may think that, recursive rules could be "split" only once, but this choice is not correct in the general case. Indeed, if the split predicate is recursive, its extension may change at each iteration; hence, the distribution made during the rewriting step could not be sufficient to compute all the ground instances of the original rule. In addition, this choice has a relevant side-effect: at each iteration the workload is dynamically redistributed among instantiators, thus inducing a dynamic load balancing. Note that this feature intervenes just in case of the evaluation of recursive rules, which are often the most time consuming part of the computation.

Procedure *Rewrite* is now described in detail. It receives as input: the rule r to be "split", the sets S and ΔS containing the extensions of the body predicates, and the program $\mathcal{P}$. *Rewrite* first selects, according to an heuristics, a positive literal to split, say l, in the body of r; then, it replaces r in $\mathcal{P}$ by a set of rules r_i ($i = 1, \ldots, k$). Each r_i is obtained from r by substituting l with a new literal l_i having a fresh[2] new predicate name built by concatenating i to the name of l. This is done by function *SplitRules*, whereas procedure *Distribute* creates the extension of the new literals l_i ($i = 1, \ldots, k$) by uniformly distributing the extension of l (both S and ΔS are affected).

Concerning the selection of the literal to split, the choice has to be carefully made, since it may strongly affect the cost of the instantiation of rules; a good heuristics should minimize it. It is well-known that this cost strictly depends on the order of evaluation of body literals, since computing all the possible instantiations of a rule is equivalent to computing all the answers of a conjunctive query joining the extensions of literals of the rule body. However, the choice of the split literal may influence the time spent

[2] If a predicate with this name already exists, another string which does not appear elsewhere in the program is appended to the name.

on instantiating each split rule, whatever the join order. In order to help the intuition, suppose that the rule $r : h(X) :- a(X), b(X), c(X).$ has to be split in ten parts, and that the size of the extensions of a, b, and c are 10, 20, and 30, respectively. The following table reports the number of operations (i.e. comparisons) needed to instantiate a single split rule of r in the worst case, by varying the literal to split (on the columns) and by considering three different body orders (on the rows). Note that, any other order is equivalent to one of the table w.r.t. the number of operations.

order/split	split a	split b	split c
ABC	**620**	**620**	800
ACB	**630**	900	**630**
CBA	1200	**660**	**660**

Looking at the table, some considerations can be made. First of all, the order ABC is the most efficient; moreover, in each order, the number of operations is minimum when one of the first two literals is split. Note also that, the size of the extension alone is not a good discriminant for choosing the literal to split. Indeed, the table shows that splitting on c (which has the largest extension) is always a bad choice; whereas there is an order (CBA) in which, even if the split literal is the smallest (a) the number of operations is the highest. Similar considerations still hold if more precise estimations of costs are made. According to these considerations, the heuristics proposed here consists of selecting an optimal ordering and splitting the first literal in this order. Such heuristics tries, on the one hand, to minimize the overall (sequential) execution time; and, on the other hand, to distribute the workload in order to minimize the parallel execution time.

Since the ordering problem has already been investigated and an effective strategy [31] has already been successfully implemented in DLV, it was decided to adopt it.

This choice has also another important consequence: since all the factors the heuristics is based on are always already computed during the computation, its implementation does not introduce any overhead.

Proposition Let $\mathcal{P}$ be an ASP program, and Π be the ground program generated by the algorithm *Parallel_Instantiate* where procedure *ComponentInstantiator_Rew* is used instead of *ComponentInstantiator*; then $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$.

Proof. (sketch) This follows from Proposition 1 if *ComponentInstantiator_Rew* produces the same output of *ComponentInstantiator* when invoked on the same input. First, observe that the ground instances of exit rules produced by the two procedures are the same modulo a renaming, that is performed by the realign step. Concerning recursive rules, their evaluation is obtained by applying several times the same algorithm used for the exit ones, where the output of each iteration is used as input for the next one. Thus, since the realign step is performed at the end of each iteration the thesis follows. □

5 Experiments

In order to check the validity of the dynamic rewriting, it was implemented into the parallel grounding engine of [22]. The resulting system was compared with the previous one on a collection of benchmark programs taken from different domains.

Both problems whose encodings cannot be evaluated in parallel with the existing technique were considered, and problems where the old technique applies. All of them have already been used for assessing ASP instantiators performance ([8, 32, 33]).

```
Procedure ComponentInstantiator_Rew ( P: Program; C: Component; var S: SetOfAtoms;
                                      var Π: GroundProgram, var G_P: DependencyGraph)
begin
    var ΔS, NS: SetOfAtoms;
    ΔS := ∅; NS := ∅ ;
    for each r ∈ Exit(C, P); do
        Rewrite (r, S, ΔS,P );   // this will add new exit rules
    for each r ∈ Exit(C, P); do
        I_r = Spawn (InstantiateRule, r, S, ΔS, NS, Π);
    for each r ∈ Exit(C, P); do
        join_with_thread(I_r);
    Realign(Π,S,ΔS);
    do
        ΔS := NS; NS := ∅ ;
        for each r ∈ Recursive(C, P); do
            Rewrite (r, S, ΔS,P );   // this will add new recursive rules
        for each r ∈ Recursive(C, P); do
            I_r = Spawn (InstantiateRule, r, S, ΔS, NS, Π);
        for each r ∈ Recursive(C, P); do
            join_with_thread(I_r);
        Realign(Π,S,ΔS,P);
        S := S ∪ ΔS;
    while  NS ≠ ∅
    Remove C from G_P;
end Procedure;

Procedure Rewrite (r: Rule; var S: SetOfAtoms; var ΔS: SetOfAtoms;var P: Program)
begin
    Select l ∈ B(r);   //according to an heuristics
    P = P ∪ SplitRules(r, l);
    P = P \ {r};
    Distribute(l,S,ΔS);
end Procedure;

Program Function SplitRules (r: Rule; l: Literal)
/*
   Given rule r, returns a program containing rules r_i (i = 1, . . . , k) obtained from r
   by replacing l with a new literal l_i having a fresh new name built by concatenating i
   to the name of l.
*/

Procedure Distribute (l: Literal; var S: SetOfAtoms; var ΔS: SetOfAtoms)
begin
    for each a ∈ S; do
        if a has the same name of l
            index s_id = DetectSplit(a);
            // create atom a_{s_id} whose name is built by concatenating s_id
            // to the name of a and add it to S.;
            S = S ∪ {a_{s_id}};
end Procedure;

Procedure Realign (Π: GroundProgram; var S: SetOfAtoms; var ΔS: SetOfAtoms;
                   var P: Program)
/*
   For each ground rule r ∈ Π, if B(r) contains a split literal l_i replace its name with
   the original one; restore S and ΔS by removing all the ground atoms inserted
   by function Distribute; replace rules originated by SplitRules with the original ones.
*/
```

Fig. 2. The Parallel Instantiation Procedures enhanced by Dynamic Rewriting.

The system was built with GCC 4.1.2, dynamically linking the Posix Thread Library. Experiments were performed on a machine equipped with two Intel Xeon "Woodcrest" (quad core) processors clocked at 3.00GHz with 4 MB of Level 2 Cache and 4GB of RAM, running Debian GNU Linux 4.0.

The implementation allows the user for setting the number of splits as an input argument. For our experiments, this number was set to 8 which coincides with the number of available processors (if the extension of the split literal contains x instances where $x < 8$ then exactly x split rules are geenrated). Actually, an experimental analysis (not reported here for space reasons) confirmed that this fixed setting is optimal. The total time needed to instantiate the inputs was measured. In order to obtain more trustworthy results, each single experiment was repeated three times, and both the average and standard deviation of the results are reported. In the following, the benchmark problems are described, and finally, the results of the experiments are reported and discussed.

5.1 Benchmark Problems and Data

A brief description of the problems considered for the experiments follows. In order to meet space constraints, encodings are not presented but they are available at *http://www.mat.unical.it/parallel/parallel_bench_08.tar.gzip*. To help the understanding of the results some information is given on the number of rules of each program. About data, we considered for each problem three instances of increasing size.

3-Colorability. This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors. The encoding of this problem consists of one rule and one constraint. Three simplex graphs were generated with the Stanford GraphBase library [34], by using the function $simplex(n, n, -2, 0, 0, 0, 0)$, ($n \in \{140, 150, 170\}$).

Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. The encoding of this problem consists of one exit rule and a recursive one. Tree graphs were generated [35] having pair (number of levels, number of siblings): (12,2), (14,2), and (10,3), respectively.

Hamiltonian Path. A classical NP-complete problem in graph theory, and can be expressed as follows: given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once? The encoding of this problem consists of several rules, one of these is recursive. Instances were generated, by using a tool by Patrik Simons (cf. [36]), having 5800, 6500 and 7200 nodes, respectively.

Player. A data integration problem [18]. Given some tables containing discording data, find a repair where some key constraints are satisfied. The encoding of this problem consists of several rules, and one constraint. The considered randomly generated databases have 32000, 39000, 45500 tuples.

n-Queens. The 8-queens puzzle is the problem of putting eight chess queens on an 8x8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. The n-queens puzzle is the more general problem of placing n queens on an nxn chessboard ($n \geq 4$). The encoding consists of one rule and four constraints. Instances were considered having $n \in \{37, 39, 41\}$.

Problem	serial	no_split	split	split vs no_split
$3col_1$	60.76 (0.58)	60.79 (0.10)	9.24 (0.09)	657%
$3col_2$	92.00 (0.55)	91.97 (0.06)	13.28(0.10)	692%
$3col_3$	171.30 (0.29)	171.36 (0.20)	24.80 (0.29)	690%
$reach_1$	14.20 (0.03)	14.26 (0.02)	2.24 (0.08)	634%
$reach_2$	268.13 (0.31)	267.98 (0.60)	35.12(0.14)	763%
$reach_3$	802.35 (10.74)	805.47 (0.40)	101.51(0.62)	790%
$hampath_1$	229.43 (0.13)	141.02 (0.29)	33.35 (0.87)	423%
$hampath_2$	303.66 (0.92)	185.83 (0.56)	45.49 (0.35)	409%
$hampath_3$	377.85 (0.48)	231.89 (1.07)	57.73 (0.47)	402%
$queens_1$	4.79 (0.01)	2.29 (0.04)	0.76 (0.03)	301%
$queens_2$	5.89 (0.01)	2.79 (0.02)	0.93 (0.01)	300%
$queens_3$	7.16 (0.01)	3.38 (0.03)	1.08 (0.02)	312%
$player_1$	141.94 (2.02)	61.16 (0.08)	20.78 (0.26)	296%
$player_2$	289.72 (0.62)	126.48 (2.19)	41.95 (0.15)	301%
$player_3$	481.65 (11.23)	209.88 (1.15)	69.84 (0.16)	300%

Table 1. Effect of the enhanced technique - Average Times and Standard Deviation.

5.2 Experimental Results and Discussion

The performance of the compared systems is summarized in Table 1. In particular, the first three columns report the average instantiation times (and standard deviation) for the serial instantiation, the old parallel instantiator and the new one, respectively; the last column shows percentage gains given by the new technique w.r.t. the old one.

First of all, notice that for 3-Colorability and Reachability the old technique does not apply, thus the time reported in column "no_split" coincides with the serial execution time for those problems; conversely, the time required by the instances of Hamiltonian Path, n-Queens and Player already benefits of the presence of more than one processor.

It is worth noting that, where the old technique has no effect, the best performance is near to the theoretical maximum obtainable with eight processors. For example, in $reach_3$ the execution time changes from 805 seconds to 101 (i.e. gain about 790%).

The better performance obtained in the case of Reachability (w.r.t 3-Colorability) is due to the dynamic workload distribution made in case of recursive rules.

Looking at the remaining problems the picture is still very good: the introduction of the dynamic rewriting always allows to significantly improve performance. Indeed, for these problems, the old technique already allows for some interesting improvements w.r.t. the serial execution. But the combination of the two techniques is always the best performer, and reaches performance gains up to 700% w.r.t. the serial version.

In particular, when comparing the old parallel instantiator with the new one, gains range from 296% of $player_1$ to 423% of $hampath_1$. Note that, the better performance obtained for Hamiltonian Path is due (as for Reachability) to the dynamic workload distribution made in the presence of recursive rules. Indeed, Hamiltonian Path and Reachability are the only ones exploiting recursion among problems in Table 1.

Such good results may be further improved by applying more sophisticated technique for the distribution of the extension of the literal to split.

6 Related Work and Conclusions

In this paper a new strategy is proposed for increasing parallelism into the instantiation process of ASP programs. This strategy allows performance to be improved by performing a dynamic rewriting of the input program that, when combined with existing paral-

lel instantiation techniques, naturally induces both a form of of *Or*-parallelism [23–26] and a dynamic load-balancing technique. The technique was implemented into the parallel DLV instantiator and an experimental analysis was conducted that confirmed the effectiveness of the technique. In particular, the new parallel implementation always outperforms the (sequential) DLV instantiator, and compared with the previous parallel method offers a very relevant gain especially in case of programs with very few rules.

Concerning related work, there are several studies about parallel techniques for the evaluation of ASP programs that focus on both the propositional (model search) phase [37–39], and the instantiation phase [40, 22]. About the latter group of proposals (which, evidently, is the only one strictly-related to this work), there are two distinct approaches: in [40] a parallelization technique was designed for the ASP instantiator Lparse [41]; whereas, in [22] the instantiator of the ASP system DLV was parallelized. Although the two approaches have several differences concerning both the input language and the exploited technology (clusters vs shared memory), both of them proceed by delegating the instantiation of rules of the program to different processing units. Thus, the method proposed in this paper, which was successfully implemented as an extension of the parallel DLV instantiator, may also be adapted to increase parallelism in case of the Lparse-based one (e.g. one might rewrite the input program, by splitting a domain predicate, just before launching the parallel computation). Regarding load-balancing, it is worth pointing out that, in [40] the distribution of work to processing units is statically determined at the beginning of the computation while, in the approach described in this paper, the work is distributed at running time.

Our work is also related to the efforts of parallelizing the evaluation of Datalog [42–45], dating back to 90's. In many of them, only restricted classes of Datalog programs are parallelized; whereas, the most general ones (reported in [43, 45]) are applicable to normal Datalog programs. Clearly, none of them consider the peculiarities of disjunctive programs and unstratified negation. More in detail, [43] provides the theoretical foundations for the so-called *copy and constrain* technique, whereas [45] enhances it in such a way that the communication overhead in distributed systems can be minimized.[3] The copy and constrain technique works as follows: rules are replicated with additional constraints attached to each copy; such constraints are generated by exploiting an hash function and allow for selecting a subset of the tuples. The obtained restricted rules are evaluated in parallel. Our technique shares the idea of splitting the instantiation of each rule, but has several differences that allow for obtaining an effective implementation. Indeed, in [43, 45] copied rules are generated and statically associated to instantiators according to an hash function which is independent from the current instance in input. Conversely, in our technique, the distribution of predicate extensions is performed dynamically, before assigning the rules to instantiators, by taking into account the "actual" predicate extensions. In this way, the non-trivial problem [45] of choosing an hash function that properly distributes the load is completely avoided in our approach. Moreover, the evaluation of conditions attached to the rule bodies during the instantiation phase would require to either modify the standard instantiation procedure (for efficiently selecting the tuples from the predicate extensions according to added constraints) or to

[3] Since the enhancements introduced in [45] are not relevant in our setting (i.e. SMP machines with *shared memory*), in the following we focus on [43]).

incur in a possible non negligible overhead due to their evaluation. As far as future work is concerned, it is planned to further study both load-balancing techniques and heuristics. A possibility is to extend to our framework dynamic load redistribution techniques like the one in [46].

References

1. Stallings, W.: Operating systems (3rd ed.): internals and design principles. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1998)
2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
3. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99) 23–37
4. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm-A 25-Year Perspective. (1999) 375–398
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
6. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence **138**(1–2) (2002) 3–38
7. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
8. 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
9. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
10. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: LPNMR-7. LNCS 2923, (2004) 331–335
11. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451
12. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
13. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proc. of IJCAI 2007, 386–392
14. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1–2) (2004) 115–137
15. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS 2923, (2004) 346–350
16. Anger, C., Konczak, K., Linke, T.: `NoMoRe`: A System for Non-Monotonic Reasoning. In: LPNMR'01. LNCS 2173, (2001) 406–410
17. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. Proc. of LPAR 2005, 95–109
18. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. Proc. of ACM SIGMOD 2005, 915–917
19. Curia, R., Ettorre, M., Iiritano, S., Rullo, P.: Textual Document Per-Processing and Feature Extraction in OLEX. In: Proceedings of Data Mining 2005, Skiathos, Greece (2005)
20. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Nonmonotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar (2005)
21. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. Proc. of ASP05, Bath, UK (2005) 248–262

22. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. J. of Algorithms in Cognition, Informatics and Logic (2008) **63**(1-3) 34 - 54.
23. Leone, N., Restuccia, P., Romeo, M., Rullo, P.: Expliciting Parallelism in the Semi-Naive Algorithm for the Bottom-up Evaluation of Datalog Programs. Database Technology **4**(4) (1993) 245–158
24. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Transactions on Programming Language Systems **23**(4) (2001) 472–602
25. de Kergommeaux, J.C., Codognet, P.: Parallel Logic Programming Systems. ACM Comput. Surv. **26**(3) (1994) 295–336
26. Gupta, G., Jayaraman, B.: Analysis of Or-Parallel Execution Models. ACM Transactions on Programming Language Systems **15**(4) (1993) 659–680
27. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. LNCS 3229, (2004) 200–212
28. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424
29. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
30. Garey, M.R., Johnson, D.S.: Computers and Intractability, A Guide to the Theory of NP-Completeness. W.H. Freeman and Company (1979)
31. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. Proc of LPNMR 2001, LNCS 2173, (2001) 280–294
32. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR 2007, LNCS 4483, 3–17
33. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. AMAI **51**(2–4) (2007) 195–228.
34. Knuth, D.E.: The Stanford GraphBase : A Platform for Combinatorial Computing. ACM Press, New York (1994)
35. Giorgio, T., Leone, N., Vincenzino, L., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP **7**, Cambridge University Press (2007) 1–37
36. Simons, P.: Extending and Implementing the Stable Model Semantics. PhD thesis, Helsinki University of Technology, Finland (2000)
37. Finkel, R.A., Marek, V.W., Moore, N., Truszczynski, M.: Computing stable models in parallel. In: Proc. of ASP'01 Workshop, Stanford (2001) 72–76
38. Gressmann, J., Janhunen, T., Mercer, R.E., Schaub, T., Thiele, S., Tichy, R.: Platypus: A Platform for Distributed Answer Set Solving. Proc. of LPNMR 2005, 227–239
39. Pontelli, E., El-Khatib, O.: Exploiting Vertical Parallelism from Answer Set Programs. In: Proc. of ASP'01 Workshop, Stanford (2001) 174–180
40. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing **31**(6) (2005) 608–647
41. Niemelä, I., Simons, P.: Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In: LPNMR'97. LNCS 1265, 420–429
42. Wolfson, O., Silberschatz, A.: Distributed Processing of Logic Programs. Proc. of ACM SIGMOD 1998, 329–336
43. Wolfson, O., Ozeri, A.: A new paradigm for parallel and distributed rule-processing. In: ACM SIGMOD 1990, 133–142
44. Ganguly, S., Silberschatz, A., Tsur, S.: A Framework for the Parallel Processing of Datalog Queries. In: SIGMOD Conference 1990, Atlantic City, NJ, 23-25, 1990. (1990) 143–152
45. Zhang, W., Wang, K., Chau, S.C.: Data Partition and Parallel Evaluation of Datalog Programs. IEEE TKDE **7**(1) (1995) 163–176
46. Dewan, H.M., Stolfo, S.J., Hernández, M., Hwang, J.J.: Predictive dynamic load balancing of parallel and distributed rule and query processing. Proc. of ACM SIGMOD 1994, 277–288

Modeling preferences on resource consumption and production in ASP

Stefania Costantini* and Andrea Formisano**

Abstract. Recently we have proposed RASP, an extension of Answer Set Programming that permits declarative specification and reasoning on consumption and production of resources. In this paper, we extend this framework to allow the declarative specification of preferences among alternative use of different resources. We provide syntax and semantics for the resulting formalism, where preferences expressed on resource usage induce a preference order on answer sets.
Key words: Answer set programming, quantitative reasoning, preferences, non-monotonic logic programming, language extensions.

Introduction

As it is well-known, Answer Set Programming (ASP) is a form of logic programming based on the answer set semantics [13], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [19]. ASP is nowadays applied in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see, among others, [5, 2, 18, 22, 14] and the references therein).

However, the possibility was lacking of performing some kind of quantitative reasoning which is instead possible in non-classical logics such as, for instance, Linear Logics [15] and Description Logics [3]. In recent work [8], an extension of ASP, called RASP (standing for Resourced ASP), has been proposed so to support declarative reasoning on consumption and production of resources.

In this paper, we go further in this extension, by adding declarative *preferences* to the specification of production/consumption processes. In particular, in realizing the same process (modeled through the firing of rules), one may prefer to produce and/or consume certain resources rather than another ones. This extension can be particularly useful in configuration applications where one can, for instance, prefer to save money while spending more time or vice versa or may prefer to employ a certain amount of cheap components rather than a little amount of expensive parts.

Let us briefly recall syntax and intended semantics of RASP programs through a simple example. We will then modify such example to informally introduce preferences. A RASP program is composed of r-facts and r-rules, where

* Università di L'Aquila. Email: stefcost@di.univaq.it
** Università di Perugia. Email: formis@dipmat.unipg.it

numbers associated with the heads of r-facts and rules indicate which amount of a certain resource is respectively: available, in case of r-facts; produced, in case of r-rules, where production can take place if the body holds (this implies that required resources are either available or produced). Available or produced resources can in turn be consumed: quantities are associated to atoms occurring in the bodies of r-rules as well. The example concerns the preparation of desserts:

$cake{:}1 \leftarrow egg{:}3,\ flour{:}4,\ sugar{:}3.$ $\quad flour{:}8.$ $\quad egg{:}3.$
$ice\_cream{:}1 \leftarrow egg{:}2,\ sugar{:}2,\ milk{:}2.$ $\quad sugar{:}6.$ $\quad milk{:}3.$

Atoms of the form $q{:}a$ are called *amount-atoms*, where the *amount-symbol* a denotes the quantity of that resource which is either produced (if the amount-atom is in the head of a rule), or consumed (if it is in the body), or available (if it is a fact), respectively. We may notice that different solutions stem, in this case, from the fact that, with the available ingredients, one may prepare either a cake or an ice-cream, but not both.

Usual ASP literals (possibly involving negation-as-failure) may freely occur in RASP rules. Semantics of a RASP program is determined by interpreting usual literals as in ASP (i.e., by exploiting stable model semantics) and amount-atoms in an auxiliary algebraic structure (that supports operations and comparisons). For instance, we could modify the above rule by requiring that ice-cream can be made only if there is a fridge and there is someone who is a good cook:

$ice\_cream{:}1 \leftarrow egg{:}2,\ sugar{:}2,\ milk{:}2,\ fridge,\ a\_cook\_is\_here.$
$a\_cook\_is\_here \leftarrow is\_here(remy),\ is\_here(linguini).$
$is\_here(remy).$ $\quad is\_here(linguini).$

Intuitively, the first rule of this program is applicable only in correspondence of models that satisfy the literals *fridge* and $a\_cook\_is\_here$.

RASP offers some constructs to express limited forms of preferences on resource consumption/production. For instance, a number of budget policies are exploitable to control rule firings and, consequently, to influence what resources to produce and in which quantity, and whether the firing of r-rules is optional or mandatory. The various policies can be combined in a mixed strategy by choosing one of them for each single rule of the program. These features, among others, are fully dealt with in [8]. In what follows we develop a general and more expressive form of preferences on resource usage.

Recall the initial example and suppose you might prepare a cake either with corn flour or with potato flour. The following rules express the two possibilities, but do not say which one you would prefer, assuming both of them to be feasible:

$$cake{:}1 \leftarrow egg{:}3, flour{:}4, sugar{:}3.$$
$$cake{:}1 \leftarrow egg{:}3, potato\_flour{:}3, sugar{:}3.$$

We propose in this paper P-RASP (RASP with preferences), to allow one to explicitly state which resource (s)he would prefer to use, e.g., the formulation

$$cake{:}1 \leftarrow potato\_flour{:}3{>}flour{:}4,\ egg{:}3,\ sugar{:}3.$$

indicates that consuming potato flour is preferred onto consuming corn flour. Or also, if the recipe includes milk, one might prefer to use skim milk if available:

$$cake{:}1 \leftarrow potato\_flour{:}3 > flour{:}4,\ skim\_milk{:}2 > whole\_milk{:}2,\ egg{:}3,\ sugar{:}3.$$

In this reformulation, we have two *preference lists* (or for short *p-lists*). Actually, p-lists may involve any number of amount-atoms. The intuitive reading is that leftmost elements of a p-list have higher priority. P-lists may occur in the head of r-rules, as shown in the example below, where one prefers to employ available ingredients to make an ice-cream instead of two cups of zabaglione:

$$ice\_cream{:}1 > zabaglione{:}2 \leftarrow skim\_milk{:}2 > whole\_milk{:}2,\ egg{:}2,\ sugar{:}3.$$

The introduction of p-lists requires a concept of *preferred answer set*. In case several p-lists occur either in one rule or in different r-rules, it is necessary to establish which answer set better satisfies the preferences. Intuitively, if we choose to consider as "better" the answer sets which satisfy the higher number of leftmost elements, in the last example we would have: producing an ice-cream with skim milk is the best solution. Producing: (a) an ice-cream with whole milk or (b) two zabagliones with skim milk would be equally good (but worse than the previous solution) as each of them employs the leftmost element of one p-list. Producing two zabagliones with whole milk is the less preferred solution. Clearly, one has to choose the best *possible* solution, given the available resources. One might choose other strategies, e.g. one might give higher priorities to p-lists in rule heads, where consequently solution (a) above would become better than (b). One may also imagine to introduce a choice among different strategies to be employed in different contexts. Finally, preferences may be conditional. One may for instance prefer skim milk when on a diet, i.e. the last rule may become:

$$ice\_cream{:}1 > zabaglione{:}2 \leftarrow (skim\_milk{:}2 > whole\_milk{:}2\ IF\ diet),\ egg{:}2,\ sugar{:}3.$$

If *diet* does not hold, then the preference list reduces to a disjunction, i.e. either skim milk or whole milk can be indistinctly used. This generalizes to several p-lists, like in the example below:

$$(ice\_cream{:}1 > zabaglione{:}2\ IF\ summer) \leftarrow (skim\_milk{:}2 > whole\_milk{:}2\ IF\ diet),\ egg{:}2,\ sugar{:}3.$$

In this paper, we propose a definition of P-RASP and its semantics, we briefly addresses the complexity and the implementation issues, and outline a comparison with related work. We notice that P-RASP has been fully implemented but for the sake of space, a description of the implementation is out of the scope of this paper. The interested reader can refer to [8, 9].

1 From RASP to P-RASP: Syntax

In order to formally introduce syntax and semantics of P-RASP, we need to briefly summarize the basic notions about RASP syntax as presented in [8, 10].

To accommodate the new language expressions that involve resources and their quantities, the underlying language of RASP is partitioned into *P*rogram symbols and *R*esource symbols. Precisely, let $\langle \Pi, \mathcal{C}, \mathcal{V} \rangle$ be an alphabet where $\Pi = \Pi_P \cup \Pi_R$ is a set of predicate symbols such that $\Pi_P \cap \Pi_R = \emptyset$, $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_R$ is a set of symbols of constant such that $\mathcal{C}_P \cap \mathcal{C}_R = \emptyset$, and $\mathcal{V}$ is a set of variable symbols. The elements of $\mathcal{C}_R$ are said *amount-symbols*, while the elements of Π_R are said *resource-predicates*. A *program-term* is either a variable or a constant symbol. An *amount-term* is either a variable or an amount-symbol.

Amount-atoms are introduced in addition to plain ASP atoms, here called program atoms. Let $\mathcal{A}(X, Y)$ denote the collection of all expressions of the form $p(t_1, \ldots, t_n)$, with $p \in X$ and $\{t_1, \ldots, t_n\} \subseteq Y$. Then, a *program atom* is an element of $\mathcal{A}(\Pi_P, \mathcal{C} \cup \mathcal{V})$. An *amount-atom* is an expression of the form $q{:}a$ where $q \in \Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C} \cup \mathcal{V})$ and a is an amount-term. Let $\tau_R = \Pi_R \cup \mathcal{A}(\Pi_R, \mathcal{C})$. We call elements of τ_R *resource-symbols*. E.g., in the two expressions $p{:}3$ and $q(2){:}b$, p and $q(2)$ are resource-symbols (with $p, q \in \Pi_R$ and $2 \in \mathcal{C}$) aimed at defining two resources which are available in quantity 3 and b, resp., (with $3, b \in \mathcal{C}_R$ amount-symbols). Expressions such as $p(X){:}V$ where V, X are variable symbols are also allowed, as resources can be either directly specified as constants or derived. Notice that the set of variables is not partitioned, as the same variable may occur both as a program term and as an amount-term. *Ground* amount- or program-atoms contain no variables. As usual, a *program-literal* L is a program-atom A or the negation *not* A of a program-atom (intended as negation-as-failure).[1] If $L = A$ (resp., $L = \mathit{not}\ A$) then $\overline{L}$ denotes *not* A (resp., A).

Definition 1. *A* resource-literal *(r-literal) is either a program-literal or an amount-atom.*

Therefore, we do not allow negation of amount-atoms. (See [8] for a discussion about this point.) Finally, we distinguish between plain rules and rules that involve amount-atoms. In particular, a *program-rule* is defined as a regular ASP rule, including the case of ASP *constraints*, i.e., rules with empty head. Beside program-rules we introduce resource-rules which differ from program rules in that they may contain amount-atoms.[2]

Definition 2. *A* resource-proper-rule *has the form*

$$H \leftarrow B_1, \ldots, B_k \tag{1}$$

where $B_1, \ldots, B_k$, $k > 0$ are r-literals and H is either a program-atom or a (non-empty) list of amount-atoms.

Resource-facts are intended to model the fixed amount of resources that are available "from the beginning". They are defined as follows:

[1] We will only deal with negation-as-failure. Though, classical negation of program literals could be used in (P-)RASP programs and treated as usually done in ASP.

[2] A more general definition of r-rule is given in [8] that offers the possibility of expressing bounds on the (finite) number of times each r-rule is fired. Here, for simplicity, we restrict ourselves to the simpler case in which each r-rule may be fired at most once. The treatment of the general case does not offer significant differences.

Definition 3. *A* resource-fact *(*r-fact*, for short) has the form* $H \leftarrow .$ *, where* H *is an amount-atom* $q{:}a$ *and* a *is an amount-symbol.*

According to the definition, the amount of an initially available resource has to be explicitly stated. Thus, in an r-fact the amount-term a cannot be a variable.

Definition 4. *A* resource-rule *(*r-rule*, for short) can be either a resource-proper-rule or a resource-fact. A* RASP-rule *(*rule*, for short)* γ *is either a program-rule or a resource-rule. An* r-program *is a finite set of RASP-rules.*

Remark 1. Notice that we admit several amount-atoms in the head of a resource-proper-rule, while the case in which a rule γ has an empty head is admitted only if γ is a program-rule (i.e., γ is an ASP *constraint*).

The list of amount-atoms composing the head of an r-rule has to be intended conjunctively, i.e., as a collection of those resources that are all contemporaneously produced by firing the rule.

P-RASP programs are obtained from RASP programs by introducing alternatives in using resources expressed by preference lists:

Definition 5. *A* preference-list *of amount-atoms (*p-list*, for short) is a writing of the form* $q_1{:}a_1 > \cdots > q_h{:}a_h$*, where* $h \geqslant 2$ *and* $q_1, \ldots, q_h$ *are pairwise distinct resource-symbols. We say that the amount-atom* $q_i{:}a_i$ *has* grade of preference i *in the p-list.*

We have now to extend the definition of an r-rule accordingly. This is done by including p-lists in r-literals:

Definition 6. *A P-RASP* resource-literal *(r-literal) is either a program-literal or an amount-atom or a p-list.*

In practice, P-RASP rules differ from RASP rules in that p-lists are admitted in place of amount-atoms. More precisely, the syntax of an r-rule in P-RASP is defined as in Def. 2 where in (1) some of the $B_1, \ldots, B_k, H$ may be p-lists.

Intuitively speaking, a p-list plays a role similar to an exclusive disjunction of amount-atoms. If a p-list occurs in the body (resp. head) of a rule, it encodes the requirement that one (and only one) resource among $q_1, \ldots, q_h$ has to be consumed (resp. produced), in the indicated amount, if the rule is fired. Moreover, q_i is preferred to q_j, for $i < j$.

Remark 2. The kind of preference among alternative uses of resources expressed by p-lists has a *local scope*: each p-list is seen in the context of a particular rule (which models a specific process in manipulating some amounts of resources). Clearly, such a local aspect is strictly correlated with the constraints on global resource balance and resource availability. Consequently, preferences locally stated for different rules might/should be expected to interact "over distance" with those expressed in other rules. Nevertheless, different preference orders on the same amount-atoms can be expressed in different p-lists.

Example 1. Assembling different PCs requires different sets of components (motherboard, processor(s), ram modules, etc.) and preference might be imposed depending on the kind of PC. For instance, in case of servers one might prefer SCSI disks rather than EIDE disks and vice versa for normal PCs:

cpu:5. $scsihd$:5. $eidehd$:9. $motherboard$:7. ram_module:20.
$pc(server)$:1 $\leftarrow$ cpu:2, ($scsihd$:2>$eidehd$:2), $motherboard$:1, ram_module:4.
$pc(desk)$:1 $\leftarrow$ cpu:1, ($eidehd$:2>$scsihd$:2), $motherboard$:1, ram_module:2.

Notice that completely antithetic orders are expressed in the two r-rules. Nevertheless, both r-rules might be fired at the same time, since enough resources are available.

2 Semantics of P-RASP

In this section, we first define the semantics of ground P-RASP programs. The general case is then easily dealt with by considering the grounding of a program P to be the set of all ground instances of rules of P that are obtainable through ground substitutions using constants occurring in P.

Semantics of a (ground) P-RASP program is determined by interpreting program-literals as in ASP and amount-atoms in an auxiliary algebraic structure that supports operations and comparisons. The rationale behind the proposed semantic definition is the following. On the one hand, we translate each r-rule into a fragment of a plain ASP program, so that we do not have to modify the definition of stability which remains the same: this is of some importance in order to make the several theoretical and practical advances in ASP still available for RASP and P-RASP. However, an answer set of a P-RASP program will support the firing of an r-rule only if: the rule is satisfied (in the usual way) as concerns its program-literals; and the requested amounts are allocated for all the resource-atoms. Hence, an interpretation (and consequently an answer set) for an r-program has two components: a set of program atoms and an allocation of actual quantities to amount-atoms.

In describing the semantics of an r-program P we will proceed as follows. First we fix an algebraic structure to represent quantities and support operations on them. Then, we develop a representation for collections of amounts with positive balance. Each of these collections will be a potential allocation of quantities to all the amount-atoms relative to a single resource symbol in P. Then, we introduce the notion of r-interpretation of P by selecting an allocation of amounts for each resource symbol in P.

Modeling Amounts. Amounts are modeled by choosing a collection Q of *quantities*, the operations to combine and compare quantities, and a mapping $\kappa : \mathcal{C}_R \rightarrow Q$ that associates quantities to amount-symbols. A natural choice is $Q = \mathbb{Z}$. In this case, positive (resp. negative) integers model produced (resp. consumed) amounts of resources. Alternative options for Q are obviously viable. (For instance, one could choose Q to be the set of rational numbers.) For the sake of simplicity, in the rest of the presentation, we will adopt a simplification by

identifying $\mathcal{C}_R$ with $\mathbb{Z}$ (and κ being the identity). This will not cause loss in the generality of the treatment.

Notation. Before going on, we introduce some useful notation. Given two sets X, Y, let $\mathcal{FM}(X)$ denote the collection of all finite multisets of elements of X, and let Y^X denote the collection of all (total) functions having X and Y as domain and codomain, respectively. For any (multi)set Z of integers, $\sum(Z)$ denotes their sum (e.g., $\sum(\{\!\{ 2, 5, 3, 3, 5 \}\!\}) = 18$).

Given a collection S of (non-empty) sets, a *choice function* $c(\cdot)$ for S is a function having S as domain and such that for each s in S, $c(s)$ is an element of s. In other words, $c(\cdot)$ chooses exactly one element from each set in S.

In order to deal with the preference order syntactically expressed by a p-list, we consider each amount-atom in a p-list as marked with an integer index. Such indexes are intended to represent the grade of preference of the amount-atoms (cf., Def. 5). Operationally, for each p-list, its composing amount-atoms will be associated, from left to right, with successive indexes starting from 1; for simple amount-atoms, the index will always be 0.

As mentioned, the elements of $Q = \mathbb{Z}$ provide the interpretations for amount symbols. To deal with the preference orders expressed by p-lists, we need a structure slightly richer that Q. In fact, to take into account of the preference grades, we will interpret amount-atoms in $\mathbb{N} \times Q$. We call *amount couples* the elements of $\mathbb{N} \times Q$. For instance: an interpretation for a p-list such as *skim_milk*:2>*whole_milk*:2, occurring in the head of an r-rule, will involve one of the couples $\langle 1, 2\rangle$ and $\langle 2, 2\rangle$, where the first components of the couples reflect the grades of preference and the second elements are the quantities.[3] For single amount-atoms (in a head of an r-rule), such as *egg*:2, no preference is involved and a potential interpretation is the amount couple $\langle 0, 2\rangle$.

Given an amount couple $r = \langle n, x\rangle$, let $grade(r) = n$ and $amount(r) = x$. We extend such a notation to sets and multisets, as one expects: namely, if X is a multiset then $grade(X)$ is defined as the multiset $\{\!\{ n \mid \langle n, x\rangle \text{ is in } X \}\!\}$, and similarly for $amount(X)$. E.g., if $X = \{\!\{ \langle 1, 2\rangle, \langle 2, 4\rangle, \langle 3, 1\rangle, \langle 1, 2\rangle \}\!\}$ then $grade(X)$ is $\{\!\{ 1, 2, 3, 1 \}\!\}$ and $amount(X)$ is $\{\!\{ 2, 4, 1, 2 \}\!\}$.

Interpretation of P-RASP Programs. An interpretation for P must determine an allocation of amounts for all occurrences of such amount symbols in P. We represent produced quantities (corresponding to amount-atoms in heads) by positive values, while negative values model consumed amounts (corresponding to amount-atoms in bodies). Since amounts and resource-symbols are used to model production and consumption of "real world" objects, we must take into account the obvious constraint that we cannot consume more than what is produced. In other words, for each resource symbol q, the overall sum of quantities allocated to amount-atoms of the form $q{:}a$ must not be negative. The collection $\mathbb{S}_P$ of all potential allocations (i.e., those having a non-negative global balance)—for any single resource-symbol occurring in P (considered as a set of rules)—is

[3] Recall that we are identifying the set of amount-symbols $\mathcal{C}_R$ with the domain of quantities $Q = \mathbb{Z}$. Consequently, the symbol 2 in the amount couple $\langle 1, 2\rangle$ is an element of Q, whereas the symbol 2 in *skim_milk*:2 is an element of $\mathcal{C}_R$.

the following collection of mappings:

$$\mathbb{S}_P = \Big\{ F \in (\mathcal{FM}(\mathbb{N} \times Q))^P \mid 0 \leqslant \sum \Big(\bigcup_{\gamma \in P} amount\big(F(\gamma)\big) \Big) \Big\} \qquad (2)$$

The rationale behind the definition of $\mathbb{S}_P$ is as follows. Let q be a fixed resource-symbol. Each element $F \in \mathbb{S}_P$ is a function that associates to every rule $\gamma \in P$ a (possibly empty) multiset $F(\gamma)$ of amount couples, assigning certain quantities to each occurrence of amount-atoms of the form $q{:}a$ in γ. All such Fs satisfy, by definition of $\mathbb{S}_P$, the requirement that, considering the entire P, the global sum of all the quantities F assigns must be non-negative. As we will see later, only some of these allocations will actually be acceptable as a basis for a model.

An r-interpretation of the amount symbols in a ground r-program P is defined by providing a mapping $\mu : \tau_R \to \mathbb{S}_P$. Such a function determines, for each resource-symbol $q \in \tau_R$, a mapping $\mu(q) \in \mathbb{S}_P$. In turn, this mapping $\mu(q)$ assigns to each rule $\gamma \in P$ a multiset $\mu(q)(\gamma)$ of quantities, as explained above. The use of multisets allows us to handle multiple copies of the same amount-atom. Each of them corresponds to a different amount of resource to be taken into account.

Let $\mathcal{B}(X, Y)$ denote the collection of all ground atoms built up from predicate symbols in X and terms in Y. We have the following

Definition 7. *An* r-interpretation *for a (ground) r-program P is a pair $\mathcal{I} = \langle I, \mu \rangle$, with $I \subseteq \mathcal{B}(\Pi_P, \mathcal{C})$ and $\mu : \tau_R \to \mathbb{S}_P$.*

Intuitively: I plays the role of a usual answer set assigning truth values to program-literals; μ describes an allocation of resources.

Example 2. Let $\Pi_P = \{have\_black\_powder, have\_gun\_powder\}$ and $\Pi_R = \{saltpeter, charcoal, sulfur\}$, and consider the following program P_1:

(γ_1) *have_black_powder* $\leftarrow$ *saltpeter*:15, *charcoal*:3, *sulfur*:2.
(γ_2) *have_gun_powder* $\leftarrow$ *saltpeter*:7, *charcoal*:3.
(γ_3) *sulfur*:4. (γ_4) *saltpeter*:18. (γ_5) *charcoal*:5.

An r-interpretation for P_1 is $\langle I, \mu \rangle$ with $I = \{have\_gun\_powder\}$ and μ such that $\mu(saltpeter)(\gamma_2) = \{\!\!\{ \langle 0, -7 \rangle \}\!\!\}$, $\mu(saltpeter)(\gamma_4) = \{\!\!\{ \langle 0, 18 \rangle \}\!\!\}$, $\mu(charcoal)(\gamma_2) = \{\!\!\{ \langle 0, -3 \rangle \}\!\!\}$, $\mu(charcoal)(\gamma_5) = \{\!\!\{ \langle 0, 5 \rangle \}\!\!\}$, $\mu(sulfur)(\gamma_3) = \{\!\!\{ \langle 0, 4 \rangle \}\!\!\}$, and $\mu(q)(\gamma_i) = \{\!\!\{ \}\!\!\}$ otherwise.

The firing of an r-rule (which involves consumption/production of resources) can happen only if the truth values of the program-literals satisfy the rule. We reflect the fact that the satisfaction of an r-rule γ depends on the truth of its program-literals by introducing a suitable fragment of ASP program $\widehat{\gamma}$. Let the r-rule γ have $L_1, \ldots, L_k$ as program-literals and $R_1, \ldots, R_h$ as amount-atoms (or p-lists). The ASP-program $\widehat{\gamma}$ is so defined:

$$\widehat{\gamma} = \begin{cases} \{ \leftarrow \overline{L_1}, \ldots, \leftarrow \overline{L_k} \} & \text{if the head of } \gamma \text{ consists of amount-atoms or p-lists} \\ \{ \leftarrow \overline{L_1}, \ldots, \leftarrow \overline{L_k}, \\ \quad H \leftarrow L_1, \ldots, L_k \} & \text{if } \gamma \text{ has the program-atom } H \text{ as head} \\ & \quad \text{and } h > 0 \\ \{ \gamma \} & \text{otherwise (e.g., } \gamma \text{ is a program-rule).} \end{cases}$$

Def. 8, to be seen, states that in order to be a model, an r-interpretation $\mathcal{I}$ that allocates non-void amounts to some amount-atoms of γ (i.e., γ is fired), has to model the ASP-rules in $\widehat{\gamma}$. (Notice that if γ is a program rule then $\widehat{\gamma} = \{\gamma\}$.)

So far we have developed a semantic structure in which r-rules are interpretable by singling-out suitable collections of amount couples.

Different ways of allocating amount of resources to an r-program are possible. In order to be acceptable, an allocation has to reflect, for each p-list r in P, one of the admissible choices that r implicitly represents. In order to extract from P the information about such admissible choices we need some further notation.

Let ℓ be either an amount-atom or a p-list in a resource-rule γ. Let

$$\mathit{setify}(\ell) = \begin{cases} \{\langle 0, q, a\rangle\} & \text{if } \ell \text{ is } q{:}a \\ \{\langle 1, q_1, a_1\rangle, \ldots, \langle h, q_h, a_h\rangle\} & \text{if } \ell \text{ is } \ q_1{:}a_1{>}\cdots{>}q_h{:}a_h \end{cases}$$

We will use *setify* to represent the amount-atoms of rules as triples denoting: the position in each preference list where they occur; the resource-symbol they contain; the amount that is required for this resource-symbol in that preference list. We generalize the notion to any multiset X of amount-atoms and p-lists: $\mathit{setify}(X) = \{\!\!\{\, \mathit{setify}(\ell) \mid \ell \text{ in } X \,\}\!\!\}$.

Let $r\text{-}\mathit{head}(\gamma)$ and $r\text{-}\mathit{body}(\gamma)$ denote the multiset of amount-atoms or p-lists occurring in the head and in the body of γ, respectively. In order to distinguish, in the representation, between amount-atoms occurring in heads and in bodies, we define $\mathit{setify}_b(\gamma)$ and $\mathit{setify}_h(\gamma)$ as the multisets $\{\!\!\{\, \mathit{setify}(x) \mid x \in r\text{-}\mathit{body}(\gamma) \,\}\!\!\}$ and $\{\!\!\{\, \mathit{setify}(x) \mid x \in r\text{-}\mathit{head}(\gamma) \,\}\!\!\}$, respectively.

At this point we can associate to each r-rule γ, a set $\mathcal{R}(\gamma)$ of multisets, intended to represent the collection of admissible choices we mentioned above:

$$\mathcal{R}(\gamma) = \Big\{ \{\!\!\{\, \langle i, q, a\rangle \mid \langle i, q, a\rangle = c_1(S_1) \text{ and } S_1 \text{ in } \mathit{setify}_h(\gamma) \,\}\!\!\} \cup \{\!\!\{\, \langle i, q, -a\rangle \mid \langle i, q, a\rangle = c_2(S_2) \text{ and } S_2 \text{ in } \mathit{setify}_b(\gamma) \,\}\!\!\} \mid \text{for } c_1 \text{ and } c_2 \text{ choice functions for } \mathit{setify}_h(\gamma) \text{ and } \mathit{setify}_b(\gamma), \text{ resp.} \Big\}$$

where c_1 (resp. c_2) ranges on all possible choice functions for $\mathit{setify}_h(\gamma)$ (resp. for $\mathit{setify}_b(\gamma)$). Each element of $\mathcal{R}(\gamma)$ represents a possible admissible selection of one amount-atom from each of the p-lists in γ and an actual allocation of an amount to it. Negative quantities are associated to amount-atoms of the body of γ, as these resources are *consumed*.[4] Vice versa, the quantities associated to amount-atoms occurring in the head are positive, as these resources are *produced*.

Definition 8. *Let $\mathcal{I} = \langle I, \mu\rangle$ be an r-interpretation for a (ground) r-program P. $\mathcal{I}$ is an* answer set *for P if the following conditions hold:*

- *for all rules $\gamma \in P$*

$$\Big(\forall\, q \in \tau_R \,\big(\mu(q)(\gamma) = \emptyset\big)\Big) \;\vee\; \Big(\bigcup_{q\in\tau_R} \Big\{\{\!\!\{\, \langle i, q, v\rangle \mid \langle i, v\rangle \textit{ is in } \mu(q)(\gamma) \,\}\!\!\}\Big\} \in \mathcal{R}(\gamma)\Big)$$

[4] To be precise, the admissible quantity corresponds to the negation of the amount occurring in an amount-atom of the body. One may also specify negative *byproducts* directly in the body, as in the amount-atom $q{:}{-}2$, for instance. In this case, amounts of q are produced and not consumed (cf., [8]).

- *I is a stable model for the ASP-program $\widehat{P}$, so defined*

$$\widehat{P} = \bigcup \left\{ \widehat{\gamma} \;\middle|\; \begin{array}{l} \gamma \text{ is a program-rule in } P, \text{ or} \\ \gamma \text{ is a resource-rule in } P \text{ and } \exists\, q \in \tau_R \left(\mu(q)(\gamma) \neq \emptyset\right) \end{array} \right\}$$

The two disjuncts in the formula in Def. 8 correspond to the two cases: a) the rule γ is not fired, so null amounts are allocated to all its amount-atoms; b) the rule γ is actually fired and all needed amounts are allocated (by definition this happens if and only if $\exists\, q \in \tau_R \left(\mu(q)(\gamma) \neq \emptyset\right)$ holds). (Again, notice that case b) imposes that the amount couples assigned by μ to a resource q in a rule γ reflect one of the possible choices in $\mathcal{R}(\gamma)$.)

We now formally introduce the notion *resource balance*:

Definition 9. *Let $\mathcal{I} = \langle I, \mu \rangle$ be an answer set for a (ground) r-program P. The* resource balance *for P, w.r.t. $\langle I, \mu \rangle$, is the mapping $\varphi : \tau_R \to Q$ defined as:*

$$\varphi(q) = \sum \left(\left\{\!\!\left\{ \sum \left(amount\big(\mu(q)(\gamma)\big) \right) \mid \gamma \in P \right\}\!\!\right\} \right)$$

which summarizes consumptions and productions of all resources.

Finally, we say that an r-interpretation $\mathcal{I}$ is an answer set of an r-program P if it is an answer set for the grounding of P.

Note that the above definition however does not in general fulfill the preferences expressed through p-lists. In order to impose a preference order on the answer sets of an r-program, we need to provide a *preference criterion* $\mathcal{PC}$ to compare answer sets. Such a criterion should impose an order on the collection of answer sets by reflecting the (preference grades in the) p-lists. Any criterion has to take into account that each rule determines a (partial) preference ordering on answer sets. In a sense, $\mathcal{PC}$ should aggregate/combine all "local" partial order to obtain a global one.

Fundamental techniques for combining preferences (seen as generic binary relations) can be found for instance in [1]. Regarding combination of preferences in Logic Programming, criteria are also given, for instance, in [4, 7, 6, 21].

Here we will just consider for P-RASP two of the simpler criteria among the variety of alternative possible choices. As a first example, we directly exploit the ordering of amount-atoms in the p-lists (i.e., their relative position). For any multiset m in $\mathcal{FM}(\mathbb{N} \times Q)$ and $i \in \mathbb{N}$, let be $\beta_i(m) = |\{\!\{ \langle i, v \rangle \mid \langle i, v \rangle \text{ is in } m \}\!\}|$. A partial order on answer sets can be defined as follows. Given two answer sets $\mathcal{I}_1 = \langle I_1, \mu_1 \rangle$ and $\mathcal{I}_2 = \langle I_2, \mu_2 \rangle$ for an r-program P, with $\mu_1 \neq \mu_2$, let m_i be the multiset

$$m_i = \bigcup_{\gamma \in P,\, q \in \tau_R} \mu_i(q)(\gamma),$$

for $i \in \{1, 2\}$, and let j be the minimum natural number such that $\beta_j(m_1) \neq \beta_j(m_2)$. We put $\mathcal{I}_1 \prec_1 \mathcal{I}_2$ if and only if $\beta_j(m_1) > \beta_j(m_2)$.

Our first preference criterion $\mathcal{PC}_1$ states that $\mathcal{I}_1$ is preferred to $\mathcal{I}_2$ if it holds that $\mathcal{I}_1 \prec_1 \mathcal{I}_2$. The *preferred answer sets* with respect to $\mathcal{PC}_1$ are those answer sets that are $\prec_1$-minimal. In a sense, the criterion $\mathcal{PC}_1$ has a "positional flavor": the answer sets that selects the highest possible number of leftmost elements (in

the p-lists) are preferred. Our second criterion brings into play the magnitude of the preference grades. This can be done by considering the grades as weights and by optimizing with respect to the global weight expressed by the entire answer set. (Clearly, more complex assignments of weights are viable.) For any answer set $\mathcal{I} = \langle I, \mu \rangle$ let

$$\omega(\mathcal{I}) = \sum_{\gamma \in P,\, q \in \tau_R} grade\big(\mu_i(q)(\gamma)\big).$$

Given $\mathcal{I}_1$ and $\mathcal{I}_2$ as before, we put $\mathcal{I}_1 \prec_2 \mathcal{I}_2$ if and only if $\omega_j(m_1) < \omega_j(m_2)$. Consequently, our second preference criterion $\mathcal{PC}_2$ states that $\mathcal{I}_1$ is preferred to $\mathcal{I}_2$ if it holds that $\mathcal{I}_1 \prec_2 \mathcal{I}_2$. As before, the preferred answer sets, with respect to $\mathcal{PC}_2$, are those that are $\prec_2$-minimal.

3 Conditional preferences on resources.

Let us extend the syntax of r-rules by admitting p-lists (or amount-atoms) whose activation is subject to the truth of a conjunctive condition. A *conditional p-list* (cp-list, for short) is a writing of the form

$$(r \ \ \textbf{\textit{if}} \ L_1, \ldots, L_m)$$

where r is a p-list $q_1{:}a_1 > \cdots > q_h{:}a_h$ (or simply an amount-atom), and $L_1, \ldots, L_m$ are program-literals. The intended meaning of a cp-list occurring in the body of a r-rule γ (the case of the head is analogous) is that whenever γ is fired the rule has to consume one of the resources occurring in r. If the firing occurs in correspondence of an answer set that satisfies the literals $L_1, \ldots, L_m$, then the choice of which resource to consume is determined by the preference expressed by the p-list. Otherwise, if any of the L_i is not satisfied, a non-deterministic choice is performed. (Hence the conjunction $L_1, \ldots, L_m$ need not to be satisfied in order to fire γ.) More precisely, the r-rule containing the cp-list becomes, if $L_1, \ldots, L_m$ does not hold, equivalent to h r-rules, each containing exactly one of the amount-atoms $q_j{:}a_j$, in place of the cp-list.

Such an extension of P-RASP can be treated by translating the rules involving cp-lists into regular r-rules. For instance, the rule

$$H \leftarrow B_1, \ldots, B_k, (r \ \ \textbf{\textit{if}} \ L_1, \ldots, L_m)$$

is translated into this fragment of r-program:

$p \leftarrow not\ np.$	$np \leftarrow not\ p.$	
$\leftarrow np, L_1, \ldots, L_m.$	$\leftarrow p, \overline{L_i}.$	for $i \in \{1, \ldots, m\}$
$H \leftarrow B_1, \ldots, B_k, r, p.$		
$H \leftarrow B_1, \ldots, B_k, q_j{:}a_j, pq_j, np.$		for $j \in \{1, \ldots, h\}$
$npq_i \leftarrow pq_j.$	$pq_j \leftarrow not\ npq_j.$	for $i, j \in \{1, \ldots, h\}, i \neq j$

where p, np, npq_j and pq_j (for each $j \in \{1, \ldots, h\}$) are fresh program atoms. Consequently, the semantics of cp-lists is given in terms of that of p-lists.

Similarly, one can introduce cp-lists with different semantics. For example, one might imagine a cp-list that, differently from the previous case, when some L_i does not hold the firing does not require any consumption of resources in r.

4 On Complexity and Implementation of P-RASP

The analysis of the complexity of PRASP can be made by establishing a relationship with LPOD [7]. In this approach, one can define rules of the form:

$$A_1 \times A_2 \ldots \times A_n \leftarrow Body.$$

meaning that one or more of the A_i's can be derived provided that $Body$ holds, where A_1 is the best preferred option, A_2 the second best, and so on. These preferences can be expressed only in the head of rules and have a global flavor, i.e., their scope is the entire program. Apart from the notation, there is a clear similarity with P-RASP p-lists when occurring in the head of r-rules. Then, by considering resource atoms as plain atoms and adapting the syntax, a PRASP r-rule with a p-list in the head and no p-lists in the body can be considered to be an LPOD rule (and, vice versa, an LPOD program can be considered to be a RASP program by substituting, e.g., each A_i by A_i:1).

As concerns an r-rule, say γ_i, with p-lists in the body, i.e., of the form:

$$H \leftarrow \ldots, B_1 > \ldots > B_k, \ldots$$

we can rephrase it in LPOD terms as the set of rules (where q_i is a fresh symbol)

$$\begin{array}{l} H \leftarrow \ldots, q_i \ldots \\ q_i \leftarrow B_1. \\ \ldots \\ q_i \leftarrow B_k. \\ B_1 \times \ldots \times B_k. \end{array}$$

We may notice that the k atoms in the p-list have been replaced in γ_i by the single atom q_i, but then they appear in the LPOD fact. In addition, we have introduced k new rules composed of two atoms each: therefore, we have substituted k atoms with $k + 2k + 1$ atoms, which ensures that the program resulting from this transformation is just linearly larger than the original one. For the transformed program, as discussed in [7], credulous reasoning is either Σ_P^2 complete or stays in Δ_P^2, according to the chosen preference criterion for selecting preferred answer sets (all this considering that credulous reasoning for plain RASP it has been proved in [10] to be NP-complete).

The approach of [20] has the same complexity, which means that each of the three formalisms (LPOD, [20] and P-RASP) can in principle be translated into each other (but only as far as preferences are concerned, as neither [20] nor [7] deal with resources). However, if one considers the programming style, then P-RASP is significantly different from the mentioned approaches as they provide global preferences, i.e., imposed all over the program, while in P-RASP preferences are local to rules: i.e., the same amount-atoms might be ordered differently in different p-lists, cf., Example 1. Reflecting such a "locality" character by means of global preferences would originate as seen above an unnatural representation, also making it harder to design an efficient implementation and to prove its correctness. Therefore, we have chosen to provide an "autonomous"

semantics which better reflects, in our opinion, the intuitive meaning that a programmer assigns to resources and quantities.

Remark 3. The above discussion suggests that the approach to preferences that we have presented in this paper can be extended from amount atoms to any kind of atoms, as shown in the following example, where when looking for a path one prefers green edges to red edges (whenever both are available). This might, e.g., model the configuration of a car navigator when one prefers a kind of road upon another one. However, the semantic account of such usage of p-lists remains to be assessed.

$$path(X,Y) \leftarrow (green\_edge(X,Z) > red\_edge(X,Z))$$
$$path(X,Y) \leftarrow (green\_edge(X,Z) > red\_edge(X,Z)), path(Z,Y)$$

As regards the implementation, a solver for (P-)RASP built on top of an existing ASP-solver is described in [9, 10]. Basically, a preliminary translation converts an r-program into ASP, by rendering the semantics presented in Section 2. This ASP program is then joined to an ASP specification of an inference engine which performs the real reasoning on resources allocation and that remains independent from the particular r-program at hand. Preference criteria (as well as some cost-based features and budget policies) are encoded in the inference engine by exploiting optimization statement commonly supported by ASP-solvers such as `smodels` or `clasp`.

5 Related Work

In RASP, we adopt the original intuition of linear logic, i.e., "give me as many As as I might need and I will give you one B's" in the context of the ASP semantics. Despite of the limitations (e.g., finite domain) we stay within a decidable setting. For a comparison between RASP and the various approaches to resource-based reasoning, the interested reader can refer to [8, 10]. Concerning preferences, we are not aware of approaches to preferences in linear logic. Thus, in order to understand whether P-RASP might be rephrased as a fragment of linear logic, a direct comparison would be needed, that can be a subject of future work.

Concerning preferences in logic programming and non-monotonic reasoning, here we briefly mention some of the existing approaches. (See [12] for a comprehensive treatment of preferences in non-monotonic reasoning.) As interesting attempts to introduce preference in (constraint) prolog-like logic programming, we mention [17, 16, 11]

Various forms of preferences have also been introduced in ASP (see [12]). Most of the proposed approaches are based on establishing priorities/preferences among rules. In [4], A-Prolog is enriched with ordered disjunction and preferences among rules are handled by means of a rule-naming mechanism. In the case of *ordered logic programs* [23], preferences are expressed through a partial order imposed on the set of rules. The order is used to implement defeating

of less-preferred rules. Other approaches express priorities among answer sets. Intuitively, this is done by declaring those atoms whose truth is "preferred" (typically, in these cases some forms of disjunction in the heads of rules is introduced). In *prioritized logic programs* [20], a set of *priorities* determines preferences on literals: from priorities, a preference relation on answer sets is drawn. In [7] preferences on atoms are modeled by *ordered disjunction* in the head of rules. Considering a given answer set of a program, for each rule a *degree of satisfaction* is determined depending on which atom of the head is satisfied. Satisfaction degrees of all rules are then combined, according to some criterion, to rank the answer sets. Through similar ideas, a *Preference Description Language* is defined in [6] to formalize penalty-based preference handling in ASO. A comparison of these approaches can be found in [23].

Notice that in almost all the above mentioned cases, preferences are expressed globally, e.g., by providing an order relation that applies on all the rules (or atoms) of the program. In P-RASP, as shown, preferences are imposed, by using p-lists, on some of the atoms of a rule. In this sense preference in P-RASP has a local character, cf., Remark 2 and Example 1.

Conclusions

In this paper, we have presented a refinement of the RASP approach (that allows for production/consumption of resources in ASP) to include preferences on which resources to exploit/produce. Preferences are expressed by means of p-lists of amount-atoms, where leftmost ones are assumed to have higher priority in consumption/production. P-lists, that can be conditional, can in fact occur both in the body and in the head of r-rules. We have extended both syntax and semantics of RASP to account for this kind of preferences and we have introduced a concept of preferred answer set as a partial ordering among possible solution according to a certain strategy.

In future work, we intend to further generalize P-RASP by introducing preferences among sets of amount-atoms (i.e., one might e.g. prefer to use resources a and b instead of resources x, y and z), as well as (explicit) preferences on rules. We intend to apply P-RASP to practical problems, e.g., of configuration, so as to have the ground for defining and experimenting different strategies for choosing preferred answer sets.

References

[1] H. Andréka, M. Ryan, and P.-Y. Schobbens. Operators and laws for combining preference relations. *J. Log. Comput.*, 12(1):13–53, 2002.

[2] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See `http://asparagus.cs.uni-potsdam.de`.

[3] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[4] M. Balduccini and V. S. Mellarkod. CR-Prolog$_2$ with ordered disjunction. In *Proc. of ASP'03*, 2003.

[5] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[6] G. Brewka. Complex preferences for answer set optimization. In *Proc. of KR'04*, 2004.

[7] G. Brewka, I. Niemelä, and T. Syrjänen. Logic programs with ordered disjunction. *Comput. Intell.*, 20(2):335–357, 2004.

[8] S. Costantini and A. Formisano. Modeling resource production and consumption in answer set programming. In *Proc. of ASP07*, 2007. Extended version in `www.dipmat.unipg.it/~formis/papers/report2008_04.ps.gz`.

[9] S. Costantini and A. Formisano. Modeling preferences on resource consumption and production in ASP. Rep. 9/08, Dip. di Matematica e Informatica, Univ. di Perugia, 2008. In `www.dipmat.unipg.it/~formis/papers/report2008_09.ps.gz`.

[10] S. Costantini and A. Formisano. Ground RASP: complexity and implementation. Rep. 16/08, Dip. di Matematica e Informatica, Univ. di Perugia, 2008. In `www.dipmat.unipg.it/~formis/papers/report2008_16.ps.gz`.

[11] B. Cui and T. Swift. Preference logic grammars: Fixed point semantics and application to data standardization. *Artif. Intell.*, 138(1-2):117–147, 2002.

[12] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Comput. Intell.*, 20(12):308–334, 2004.

[13] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of ICLP'88*, pp. 1070–1080. The MIT Press, 1988.

[14] M. Gelfond. Answer sets. In *Handbook of Knowledge Representation, chapter 7*. Elsevier, 2007.

[15] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[16] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference queries in deductive databases. *New Generation Comput.*, 19(1):57–86, 2000.

[17] H.-F. Guo and B. Jayaraman. Mode-directed preferences for logic programs. In *Proc. of ACM-SAC'05*, pp. 1414–1418, 2005.

[18] N. Leone. Logic programming and nonmonotonic reasoning: From theory to systems and applications. In C. Baral, G. Brewka, and J. S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007*, page 1, 2007.

[19] V. W. Marek and M. Truszczyński. *Stable logic programming - an alternative logic programming paradigm*, pp. 375–398. Springer, 1999.

[20] C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artif. Intell.*, 123(1-2):185–222, 2000.

[21] T. C. Son and E. Pontelli. Planning with preferences using logic programming. *TPLP*, 6(5):559–607, 2006.

[22] M. Truszczyński. Logic programming for knowledge representation. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007*.

[23] D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. *TPLP*, 6(1-2):107–167, 2006.

Towards Logic Programs with Ordered and Unordered Disjunction*

Philipp Kärger[1], Nuno Lopes[2], Daniel Olmedilla[1], and Axel Polleres[2]

[1] L3S Research Center & Leibniz University of Hannover, Germany
[2] DERI Galway, National University of Ireland

Abstract. Logic Programming paradigms that allow for expressing preferences have drawn a lot of research interest over the last years. Among them, the principle of ordered disjunction was developed to express totally ordered preferences for alternatives in rule heads. In this paper we introduce an extension of this approach called Disjunctive Logic Programs with Ordered Disjunction (DLPOD) that combines ordered disjunction with common disjunction in rule heads. By this extension, we enhance the preference notions expressible with totally ordered disjunctions to *partially* ordered preferences. Furthermore, we show that computing optimal stable models for DLPODs still stays in Σ_2^p for head-cycle free programs and establish Σ_3^p upper bounds for the general case.

1 Introduction

Expressing preferences in logic programs has been a research issue in the community for quite some time now. One can distinguish two directions: preferences between rules of a logic program and preferences among literals. In both cases, typically the semantics of the approaches require a total order preference relation to be imposed in the preference expressions. But requiring total order preferences is a restriction that does not fit the world of subjective expressions: total order preferences do not allow for cases where for several options it is not known which one is preferred. But such cases, called indifferences, are common, be it due to incomplete information about the world or due to the lack of decision of a user between options.

In this paper we present an approach for preferences in logic programming that allows to specify *partially* ordered preferences among literals. We achieve this by combining two things which were handled separately until now: first, the usual disjunction common in disjunctive logic programs (DLP) [1–3]; second, the preference approach of Brewka et al. [4] called Logic Programming with Ordered Disjunction (LPOD). LPOD is an extension to logic programming that introduces a special disjunction denoted by the operator $\times$ that exploits the order of literals in a disjunction in order to express preferences among these literals. We argue that allowing either ordered or unordered disjunctions alone in the head of a

* This work has been supported by Science Foundation Ireland under the Lion project (SFI/02/CE1/I131) and by the European FP6 project inContext (IST-034718)

program's rules is not sufficient whenever it comes to statements of indifference in the preferences. Typically, one may be indifferent between some options but still prefer some others rather than defining a total order between all the options. In our approach, we propose to use the semantics of the ordered disjunction to express preferences and the disjunction to express indifferences. For example, the preference concerning the activities for a night may then look as follows (inspired by the example given in [4]):

$$pub \times (cinema \vee tv).$$

The intuition behind this expression is that *pub* is the most preferred option and, in case *pub* is not possible, both *cinema* and *tv* are equally preferred.

The remainder of the paper is structured as follows. In Section 2 we recall definitions about DLP and LPODs used later in the paper. In Section 3, we detail our new language including syntax and semantics definition. An encoding of partial order preferences in DLPODs is given in Section 4. Section 5 provides an implementation of our approach and in Section 6 general complexity results for computing optimal answer sets of a DLPOD are given. We want to point out that this section marks preliminary results in the sense that we have not yet nailed down exact complexity bounds for the newly defined language and that some proofs are admittedly sketchy due to space restrictions.

2 Preliminaries

The stable model semantics extends the typical least model semantics for logic programs (where all rules are definite Horn clauses) to so-called normal logic programs, i.e. programms allowing negation as failure in rule bodies. Logic programming under the answer set semantics, often referred to as "Answer Set Programming", further extends the stable model semantics by features such as various forms of disjunction. In the following, we review the definitions of two such forms of disjunction, which we will refer to later in the paper. Namely, we will introduce Disjunctive Logic Programs (DLPs) and Logic Programs with Ordered Disjunction (LPOD).

In this paper we will mostly restrict our elaborations and examples to propositional programs. As usual in answer set programming rules containing variables—also called rule schemata—are considered as representations of their instantiations where variables are replaced by the constants occurring in the program.

2.1 Disjunctive Logic Programming

Given disjoint sets of predicate, constant and variable symbols, σ^{pred}, σ^{con} and σ^{var} respectively, an atom can be defined as $p(t_1, \ldots, t_n)$ where $p \in \sigma^{pred}$, $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var}$ and n is called the arity of p. Atoms such that $n = 0$ are called *propositional*. A literal is an atom a or its negation $\neg a$ ('$\neg$' represents classical negation).

Definition 1 (DLP). *A disjunctive logic program (DLP) P is defined as a set of rules r of the form* $h_1 \vee \ldots \vee h_l \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n$ *where each h_i (b_j) is a literal and* not *represents negation as failure. We further define $Head(r) = \{h_1, \ldots, h_l\}$, $Body^+(r) = \{b_1, \ldots, b_m\}$, $Body^-(r) = \{b_{m+1}, \ldots, b_n\}$, and $Lit(P)$ as the set of all literals occurring in P.*

Variables present in a program P are assumed to be a shorthand notation representing each element of the Herbrand Universe of program P, HU_P, which corresponds to the set of all constants $c \in \sigma^{con}$ present in P. The semantics of DLPs is defined as usual by its disjunctive stable models, or answer sets, i.e., a set of literals S is an answer set of P if and only if it is a minimal Herbrand model of the Gelfond-Lifschitz reduct P^S, see [5, 1] for details.

Head-Cycle Free Logic Programs [6] are a special kind of disjunctive logic programs which will be of interest later in the paper. They are defined based on the notion of a program's dependency graph:

Definition 2 (Dependency graph). *The dependency graph of a Logic Program P is defined as a directed graph where every literal that occurs in P is represented as node l and there is an edge from l' to l if there is a rule in P such that $l \in Head(r)$ and $l' \in Body^+(r)$.*

Definition 3 (Head-Cycle Free). *P is head-cycle free if its dependency graph does not contain directed cycles that go through two literals occurring in the same rule head.*

2.2 Logic Programming with Ordered Disjunction

In [4], Brewka et al. describe so-called Logic Programs with Ordered Disjunction (LPOD) for expressing preferences in logic programming based on a special kind of disjunction called *ordered disjunction* and denoted by $\times$. It expresses a disjunction while at the same time building up a preference order between the single disjuncts. This ordered disjunction is—similarly to DLP—only allowed to appear in a rule's head. A typical example rule is "$pub \times cinema \times tv$." stating that *pub* is preferred to be true. If for some reason *pub* can not hold, *cinema* would be the second option, and so on. Due to space restrictions we ommit a formal presentation of LPODs and their semantics and refer the reader to [4].

2.3 Other Related Work

There is a lot of work about modeling and exploiting preferences in logic programs, we refer the reader to [7, 8] for a complete overview. To the best of our knowledge, none of the existing approaches to preference handling in logic programs allow for *partial* order preferences expressions. For the sake of completeness, we want to mention the work presented in [9]. There, LPODs are used as a basis for the policy language PPDL describing the behaviour of a network node and allowing for preference definitions between possible actions a node can perform. This approach models partial order preferences by assigning levels to elemets of distinct branches

of the partial order. However, this leveling approach does not work with all partial order preferences. For instance, the one described in our Example 5 later in the paper: there is no unique level assignment that keeps B and D as well as C and D incomparable but the semantics of partial orders defines both pairs as incomparable.

3 DLPOD—Disjunctive Logic Programs with Ordered Disjunction

In this section we will detail our approach to combine ordered and unordered disjunctions. In a few words: we allow both, ordered disjunctions indicated by the operator $\times$ and normal disjunctions indicated by the operator $\vee$ in a rule's head. Based on this we can extend the example given in [4] and define the rule

$$pub \times (cinema \vee tv) \leftarrow not\ sunny.$$

Here we allow an indifference between the two options *cinema* and *tv*. Intuitively, in case the body of the rule is true, a user prefers *pub* to be true, that is, to be contained in the answer set. If *pub* can not be satisfied (e.g., another rule remedies the possibility of visiting a pub), it is considered equal if either of the options *cinema* and *tv* are true. In the following we will first provide a detailed definition of how such rules look like and second, we define their exact semantics.

3.1 Syntax

Our syntax simply extends Logic Programs with Ordered Disjunction from [4] with the common disjunction '$\vee$' used in Disjunctive Logic Programming.

Definition 4 (Ordered Disjunctive Term). *An ordered disjunctive term is a (possibly nested) term of literals $C_1, \ldots, C_n$ connected by $\vee$ or $\times$. We define such terms recursively as follows.*

- *Any literal L is an Ordered Disjunctive Term.*
- *If t_1 and t_2 are Ordered Disjunctive Terms, then $(t_1 \times t_2)$ and $(t_1 \vee t_2)$ are Ordered Disjunctive Terms as well.*

We define a DLPOD as an extended logic program with an Ordered Disjunctive Term in the head:

Definition 5 (Disj. Log. Program with Ordered Disjunction). *A Disjunctive Logic Program with Ordered Disjunction (DLPOD) P is a set of rules of the form $r = Head_r \leftarrow Body_r$. where $Body_r = B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_k$ such that all B_i $(1 \leq i \leq k)$ are literals and $Head_r$ is an Ordered Disjunctive Term. We further define $Body^+(r) = \{B_1, \ldots, B_m\}$, $Body^-(r) = \{B_{m+1}, \ldots, B_k\}$.*

In the following we define the semantics of a DLPOD by first introducing answer sets of a DLPOD and subsequently defining a preference relation among those answer sets.

3.2 Answer Sets of a DLPOD

The definition of the answer sets of a DLPOD is based on an extended notion of split programs as they are introduced in [4]. For defining split programs of a DLPOD we first define what an *Ordered Disjunctive Normal Form (ODNF)* of a rule is. Then, we show how to transform each rule's head into this normal form. Based on rules given in this normal form and on the definition of the *option* of such a rule, we can define the split programs of a DLPOD.

Definition 6 (Ordered Disjunctive Normal Form (ODNF)). *The Ordered Disjunctive Normal Form of an Ordered Disjunctive Term is*

$$\bigvee_{i=1}^{n} \bigtimes_{j=1}^{m_i} C_{i,j} = (C_{1,1} \times \ldots \times C_{1,m_1}) \vee \ldots \vee (C_{n,1} \times \ldots \times C_{n,m_n})$$

We call $(C_{i,1} \times \ldots \times C_{i,k_i})$ *the i-th* Ordered Disjunct *of the ODNF. We say that a rule r is in ODNF if* $Head(r)$ *is in ODNF.*

We treat arbitrarily nested DLPOD rules as shorthand for DLPOD rules in ODNF. I.e., given an ordered disjunctive term S and subterms a, b and c of S the following rewriting rules can be used to expand S to ODNF:

$$a \times (b \vee c) \Rightarrow (a \times b) \vee (a \times c) \tag{1}$$

$$(a \vee b) \times c \Rightarrow (a \times c) \vee (b \times c) \tag{2}$$

$$(a \times b) \times c \Rightarrow a \times b \times c \tag{3}$$

$$a \times (b \times c) \Rightarrow a \times b \times c \tag{4}$$

Example 1. By exhaustive application of these rules, we can transform any rule in a program into ODNF. For instance

$$pub \times (cinema \vee tv) \leftarrow not\ sunny.$$

yields the following rule in ODNF:

$$(pub \times cinema) \vee (pub \times tv) \leftarrow not\ sunny.$$

◇

Using the rewriting rules (1)–(4), hereafter we will define the semantics of a DLPOD P in terms of rules in ODNF only. We begin with the definition of the split programs of P which—intuitively—denote combinations of all *options* of each rule:

Definition 7 (Option of a rule). *Let r be a DLPOD rule in ODNF:*

$$\bigvee_{i=1}^{n} \bigtimes_{j=1}^{m_i} C_{i,j} \leftarrow body.$$

where m_i *is the number of literals in the i-th Ordered Disjunct of r. An option of r is any rule of the form* $(j_i \leq m_i)$*:*

$$
\begin{aligned}
C_{1,j_1} \vee C_{2,j_2} \vee \ldots \vee C_{n,j_n} \leftarrow\ & body, \\
& not\ C_{1,1}, not\ C_{1,2}, \ldots, not\ C_{1,j_1-1}, \\
& not\ C_{2,1}, not\ C_{2,2}, \ldots, not\ C_{2,j_2-1}, \\
& \ldots \\
& not\ C_{n,1}, not\ C_{i,2}, \ldots, not\ C_{n,j_n-1}.
\end{aligned}
$$

Example 2. The ODNF rule $(pub \times cinema) \vee (pub \times tv) \leftarrow not\ sunny.$ has the following four options (for example purposes repeated atoms are not removed):

$$
\begin{aligned}
pub \vee pub &\leftarrow not\ sunny. \\
pub \vee tv &\leftarrow not\ sunny, not\ pub. \\
cinema \vee pub &\leftarrow not\ sunny, not\ pub. \\
cinema \vee tv &\leftarrow not\ sunny, not\ pub, not\ pub.
\end{aligned}
$$

◇

Definition 8 (Split program of a DLPOD). *A split program P' of a DLPOD P is obtained by replacing each rule in P by one of its options.*

It is important to note that—in contrast to [4]—the split programs of DLPODs are *disjunctive* logic programs.

Example 3. Given the following DLPOD P:

$$
\begin{aligned}
pub \times (cinema \vee tv) &\leftarrow not\ sunny. \\
beach \vee hiking &\leftarrow sunny.
\end{aligned}
$$

We obtain the following four split programs:

1. $pub \leftarrow not\ sunny.$
   $beach \vee hiking \leftarrow sunny.$
2. $pub \vee tv \leftarrow not\ sunny, not\ pub.$
   $beach \vee hiking \leftarrow sunny.$
3. $cinema \vee pub \leftarrow not\ sunny, not\ pub.$
   $beach \vee hiking \leftarrow sunny.$
4. $cinema \vee tv \leftarrow not\ sunny, not\ pub.$
   $beach \vee hiking \leftarrow sunny.$

◇

Analogously to disjunctive logic programs, we define head-cycle-freeness [6] for DLPODs as follows:

Definition 9 (Dependency graph). *The dependency graph of a DLPOD P is the directed graph containing all literals in P as nodes such that there is an edge from l' to l iff there is a rule r in P such that $l \in Head(r)$ and $l' \in Body^+(r)$.*

Definition 10 (Head-Cycle Free). *A DLPOD P is head-cycle free if its dependency graph does not contain directed cycles that go through two literals occurring in two ordered disjuncts C_i and C_j $(i \neq j)$ of the same rule head.*

The following observation can be easily verified:

Proposition 1. *Split programs of head-cycle free DLPODs are head-cycle free.*

The possible optimal answer sets of a DLPOD are the answer sets of all split programs. In the following section we will explain in detail which answer set we call optimal according to the original DLPOD.

3.3 Optimal Answer Sets of a DLPOD

For the definition of the semantics of a DLPOD we still miss the notion of preferred answer sets of a DLPOD. So far, we have shown how the possible answer sets of a DLPOD are defined. In this section we will detail how to compare these possible answer sets in order to find the optimal ones (i.e., the most preferred answer sets according to the ordered and unordered disjunctions in the rules' heads). First, we define the Satisfaction Degree Vector as a measurement of how much an answer set satisfies a DLPOD rule:

Definition 11 (Satisfaction Degree Vector). *Let r be a DLPOD rule of the form*

$$r = \bigvee_{i=1}^{n} \bigtimes_{j=1}^{m_i} C_{i,j} \leftarrow A_1, ..., A_l, not\ B_1, ..., not\ B_k$$

and let S be a set of literals. The satisfaction degree vector *D of r in S is a vector of the form $D = (d_1, \ldots, d_n)$ representing degrees of satisfaction for each disjunct in r's head where each d_i is either a natural number or the constant ϵ. We define the dimensions of the Satisfaction Degree Vector as follows:*

1. *$D = (1, \ldots, 1)$ if (a) $Body_r^+ \not\subseteq S$, or (b) $Body_r^- \cap S \neq \emptyset$, or otherwise*
2. *$d_i = \epsilon$ if $C_{i,j} \notin S$ for all $1 \leq j \leq m_i$,*
3. *$d_i = min\{t | C_{i,t} \in S\}$.*

We denote the Satisfaction Degree Vector of r in S by $Deg_S(r)$.

Intuitively, in this definition, we assign to each Ordered Disjunct a penalty representing how much the answer set satisfies the disjunct. For each rule, these penalties build up a vector of degree values—one dimension for each disjunct. We choose degree ϵ for head disjuncts which do not overlap with S (cf. Condition *2*). With ϵ we denote that a particular disjunct does not tell anything about how much an answer set is preferred. Further, like in [4], we assign the best satisfaction degree (i.e., the vector $(1, \ldots, 1)$) in case a rule's body is not satisfied (cf. Conditions *1*): there is no reason to be dissatisfied if a rule does not apply for a particular answer set.

Example 4. Let us again consider the rule

$$r = (pub \times cinema) \vee (pub \times tv) \leftarrow not\ sunny.$$

Since this rule has two Ordered Disjuncts, any Satisfaction Degree Vector has two dimensions. The set of literals $\{pub\}$ as well as $\{sunny\}$ satisfies this rule to degree $(1, 1)$ (applied condition *3.* and condition *1.(b)*, respectively). The set $\{cinema\}$ satisfies r to degree $(2, \epsilon)$ and the set $\{tv\}$ satisfies r to degree $(\epsilon, 2)$. $\diamond$

Definition 12 (Preference acc. to a rule). *A set of literals S_1 is preferred to another S_2 according to a rule r (denoted as $S_1 \succ_r S_2$) iff $Deg_{S_1}(r) = (d_1^1, \ldots, d_n^1)$ Pareto-dominates $Deg_{S_2}(r) = (d_1^2, \ldots, d_n^2)$. That is, the following two conditions hold:*

1. *$\forall i\ (\ d_i^1 \leq d_i^2\ \vee\ d_i^1 = \epsilon\ \vee\ d_i^2 = \epsilon\ )$*
2. *$\exists i\ d_i^1 < d_i^2\ \ (d_i^1 \neq \epsilon \wedge d_i^2 \neq \epsilon)$.*

Intuitively, we require all dimensions in $Deg_{S_1}(r)$ to show a smaller or equal number than in $Deg_{S_2}(r)$ and in at least one dimension $Deg_{S_1}(r)$ has to show a strictly smaller number than $Deg_{S_2}(r)$. The constant ϵ plays the role of a placeholder which is, roughly speaking, equal to any number (cf. Condition *1*). As we will see in Section 4, this ϵ provides us with the "incomparability" needed to capture partial orders.

Now, we finally extend the preference notion to a relation comparing sets of literals according to a whole DLPOD:

Definition 13 (Preference acc. to a program). *A set of literals S_1 is preferred to another S_2 according to a set of rules $R = \{r_1, \ldots, r_n\}$ (denoted as $S_1 \succ S_2$) iff $\exists i(S_1 \succ_{r_i} S_2) \wedge \neg\exists j(S_2 \succ_{r_j} S_1)$.*

The conditions in both definitions follow the fair principle of Pareto optimality: an object is preferred if it is better or equal to another in all attributes (in our case in all Ordered Disjuncts or in all rules, respectively) and strictly better in at least one attribute. Finally, we provide the Definition of a preferred answer set of a program P:

Definition 14 (Preferred Answer Set). *Given a DLPOD P, one of its split programs P', and an answer set S of P'. S is called a preferred answer set (of P) if there is no answer set S' of P' for which $S' \succ S$ holds.*

4 Encoding Partial Order Preferences into DLPODs

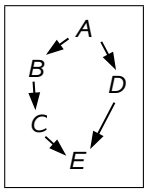
As hinted already in the introduction part, DLPOD-programs extend the approach of preferences in logic programming towards *partial* order preference relations. In this section we detail how to actually model partial order preference relations with Ordered Disjunctive Terms. For this, we specify a transformation of a partial order of literals into a Disjunctive Normal Form yielding a partial order preference statement in a rule's head.

Definition 15 (Transformation of a Partial Order). *Given a Partial Order $<$ over a set of literals S and its corresponding covering relation $<_*$ (that is, $<_*$ contains the transitive reflexive reduction of $<$), the transformation P of $<$ into an Ordered Disjunctive Term is defined as: $P(<, S) = \bigvee_{j=1}^{n}(C_1 \times \ldots \times C_{k_j})$ such that $(\forall C_i : C_i \in S) \ \wedge (\neg\exists C : C <_* C_1) \wedge (\neg\exists C : C_k <_* C) \wedge (\forall i : C_i <_* C_{i+1})$.*

Intuitively speaking, given a partial order preference relation represented by its Hasse-diagram [10], for each possible path from an element with no incoming edges to an element with no outgoing edges, we create an Ordered Disjunct $(C_1 \times \ldots \times C_k)$ where C_1 is a node with no incoming edge, C_k is a node with no outgoing edge, and there is an edge between any pair C_i, C_{i+1}.

Example 5.
Given the preference relation $<$ over the set of literals $S = \{A, B, C, D, E\}$ as depicted in the Hasse diagram on the right hand side, the transformation $P(<, S)$ yields the following Ordered Disjunctive Term:

$$(A \times B \times C \times E) \vee (A \times D \times E).$$



◇

This transformation provides us with the means for modelling partial order preferences in DLPODs: now every partial order preference expressed for literals can be formulated as the head of a rule in a DLPOD.

5 Implementation

As for a possible implementation, we extend the implementation of LPOD by Brewka et al. [11] towards DLPODs. As we shall see, this is not entirely straightforward. Concretely, in [11] the LPOD semantics is implemented on top of a standard solver for non-disjunctive logic programs based on the observation that each split program corresponds to guessing exactly one degree for each rule with ordered disjunction. Our approach and [11] basically share the following procedure to compute a preferred answer set given a program P:

1. Guess a particular satisfaction degree vector for each rule (i.e., a split program) and compute the answer sets for this guess. This is encoded in a in program called *generator* $G(P)$.
2. For each answer set S, check whether there is no split program which yields a better answer set than S. This is encoded in a program $T(P, S)$ called *tester*, which is called in an interleaved fashion for each answer set generated by $G(P)$. Whenever the tester does not find a better answer set, S is a preferred answer set.

This is analog to [11] except for the following three modifications. First, in order to generate all possible splits we need to guess a satisfaction degree *vector* per rule (instead of a single degree value). Second, we need to generate the answer sets for each split, which is—as opposed to LPODs—a *disjunctive* logic program. Third, we need to modify the *tester* program which establishes whether a better answer set can be found.

Before adapting the formal definitions of Brewka et al.'s generator and tester we need to prove two lemmata. The first Lemma states that one can replace a head symbol h in a disjunctive rule of a program P with a new symbol h' by adding some extra rules without changing the semantics of P:

Lemma 1 (Ground head atom replacement). *Let* $r = h_1 \vee \ldots \vee h_i \vee \ldots \vee h_n \leftarrow Body_r.$ *be a rule in a disjunctive logic program* P *such that* h_i *is ground, and let further* $P' = P \setminus r \cup \{h_1 \vee \ldots \vee h'_i \vee \ldots \vee h_n \leftarrow Body.\ h'_i \leftarrow h_i.\ h_i \leftarrow h'_i.\}$ *such that* h'_i *does not occur in* P*. Then* S *is an answer set of* P *if and only if* $S' = S \cup \{h'_i \mid h_i \in S\}$ *is an answer set of* P'*.*

Similarly, we note that a part of the body of r can essentially be "outsourced" to an external rule by the following Lemma:

Lemma 2 (Body replacement). *Let $r = Head \leftarrow Body_1, Body_2.$ be a rule in a disjunctive logic program P, and let further*

$$P' = P \setminus r \cup \{Head \leftarrow b', Body_2.b' \leftarrow Body_1.\}$$

such that b' does not occur in P. Then S is an answer set of P if and only if $S' = S \cup \{b' \mid Body_1 \text{ true in } S\}$ is an answer set of P'.

Using these lemmata, we are almost ready to go ahead to define the *generator* program. For this definition we make use of the cardinality constraint notation $L\{l_1, \ldots l_n\}U$ [12] in the head of a rule. Here, $l_1, \ldots, l_n$ are literals and L (lower bound) and U (upper bound) are natural numbers. The intuition is that this statement holds if at least L and at most U of the literals $l_1, \ldots, l_n$ are satisfied.

Definition 16 (Generator Program, adapts [11, Def. 10]). *Let r be the rule of a DLPOD of the form*

$$\begin{array}{ll} H_{1,1} \times \ldots \times H_{1,m_1} \vee & \\ \vdots & \\ H_{i,1} \times \ldots \times H_{i,m_i} \vee & \leftarrow Body_r. \\ \vdots & \\ H_{n,1} \times \ldots \times H_{n,m_n} & \end{array}$$

Then the transformation $G(r)$ is defined as the following set of rules:

(a) $\{\ 1\{c_{r,i}(1), \ldots, c_{r,i}(m_i)\}1 \leftarrow\ Body_r. \mid 1 \leq i \leq n\}$
(b) $\cup\ \{\ h_{r,1} \vee \ldots \vee h_{r,n} \leftarrow b_{r,1}, \ldots, b_{r,n}, Body_r.\}$
(c) $\cup\ \{\ h_{r,i} \leftarrow H_{i,j}, c_{r,i}(j).\ H_{i,j} \leftarrow h_{r,i}, c_{r,i}(j). \mid 1 \leq i \leq n, 1 \leq j \leq m_i\}$
(d) $\cup\ \{\ b_{r,i} \leftarrow c_{r,i}(j), not\ H_{i,1}, \ldots, not\ H_{i,j-1}. \mid 1 \leq i \leq n, 1 \leq j \leq m_i\}$
(e) $\cup\ \{ \leftarrow not\ H_{1,1}, \ldots, not\ H_{1,m_1}, \ldots,$
$not\ H_{i-1,1}, \ldots, not\ H_{i-1,m_{i-1}},$
$not\ H_{i,1}, \ldots, not\ H_{i,j-1},$
$H_{i,j}, not\ c_{r,i}(j),$
$not\ H_{i+1,1}, \ldots, not\ H_{i+1,m_{i+1}}, \ldots$
$not\ H_{n,1}, \ldots, not\ H_{n,m_n}. \mid\ 1 \leq i \leq n, 1 \leq j \leq m_i\}$

Finally, the transformation $G(P)$ of a complete DLPOD is the union of all its transformed rules:

$$G(P) = \bigcup \{G(r) | r \in P\}.$$

Here, the newly introduced predicates $c_{r,i}$, $h_{r,i}$, $b_{r,i}$ $(1 \leq i \leq n)$ stand for "choice", "head", and "body" auxiliary symbols. Whereas the $c_{r,i}$ plays the role of modeling the choice of an actual degree vector, the $h_{r,i}$ and $b_{r,i}$ predicates are auxiliary symbols used according to Lemmas 1 and 2 for a particular choice. Rules (a) are guessing a particular choice option forming a split. Using this choice, rules

(b) to (e) represent the actual rules in the split program for the particular choice, by using Lemma 1 in rules (c) and Lemma 2 in rules (d). Finally, rules (e) ensure that – in case all other ordered disjuncts $k \neq i$ are false – we must choose to add $H_{i,j}$ if no better literal $H_{i,l}$ in disjunct i with $l < j$ is already in the model. [3]

Example 6. Let us consider the following rule $r = (A \times B) \vee (C \times D) \leftarrow Body$. Then, the transformation $G(r)$ looks as follows:

(a) $1\{c_{r,1}(1), c_{r,1}(2)\}1 \leftarrow Body.$
$1\{c_{r,2}(1), c_{r,2}(2)\}1 \leftarrow Body.$
(b) $h_{r,1} \vee h_{r,2} \leftarrow b_{r,1}, b_{r,2}, Body.$
(c) $h_{r,1} \leftarrow A, c_{r,1}(1).\ A \leftarrow h_{r,1}, c_{r,1}(1).$
$h_{r,1} \leftarrow B, c_{r,1}(2).\ B \leftarrow h_{r,1}, c_{r,1}(2).$
$h_{r,2} \leftarrow C, c_{r,2}(1).\ C \leftarrow h_{r,2}, c_{r,2}(1).$
$h_{r,2} \leftarrow D, c_{r,2}(2).\ D \leftarrow h_{r,2}, c_{r,2}(2).$
(d) $b_{r,1} \leftarrow c_{r,1}(1), not\ A.$
$b_{r,1} \leftarrow c_{r,1}(2), not\ A, not\ B.$
$b_{r,2} \leftarrow c_{r,2}(1), not\ C.$
$b_{r,2} \leftarrow c_{r,2}(2), not\ C, not\ D.$
(e) $\leftarrow A, not\ c_{r,1}(1), not\ B, not\ C, not\ D.$
$\leftarrow notA, B, not\ c_{r,1}(2), not\ C, not\ D.$
$\leftarrow notA, notB, C, not\ c_{r,2}(1), not\ D.$
$\leftarrow notA, notB, notC, D, not\ c_{r,2}(2).$ ◇

Proposition 2. *Let P be a DLPOD. Then (i) $G(P)$ is polynomial in the size of P and (ii) S is an answer set of $G(P)$ if and only if $S \cap Lit(P)$ is an answer set of P.*

Proof. [sketch] (i) is easy to see by looking at the rules (a) to (e). The idea for (ii) is similar to the analogous Proposition 2 in [11] where we additionally need to apply Lemma 1 and 2. Intuitively, each "guess" of the $c_{r,i}(j)$ in rules (a) yields a split program in the sense that each rule not belonging to that particular guess is "projected" away by putting $c_{r,i}(j)$ in the bodies of rules (c) and (d). By lemmas 1 and 2 now, rule (b) exactly corresponds to the guess rule in the split program corresponding to the guess modeled in (a). □

Each answer set S of $G(P)$ is subsequently tested by a tester program $T(P, S)$ for whether it is a preferred answer set.

Definition 17 (Tester Program). *Let P be a DLPOD and S be a set of literals. The tester program checking whether there is a better answer set than S is defined as follows:*

$$\begin{aligned} T(P,S) = G(P) \\ \cup \{O_{i,j}. \mid H_{i,j} \in S\} \cup \{rule(r). \mid r \in P\} \\ \cup \{better(r) \leftarrow rule(r), O_{i,j}, H_{i,k}. \mid r \in P, 1 \leq i \leq n, 1 \leq k \leq m_i, 1 \leq j < k\} \end{aligned}$$

[3] For the interested reader, rules (a) roughly correspond to the rule in equation (8) in [11], rules (b)–(d) to rule (4) in [11], and finally rules (e) to rule (5) in [11].

$$\cup \{worse(r) \leftarrow rule(r), O_{i,k}, H_{i,j}. \mid r \in P, 1 \leq i \leq n, 1 \leq k \leq m_i, 1 \leq j < k\}$$
$$\cup \{betterRule(R) \leftarrow better(R), not\ worse(R).$$
$$worseRule(R) \leftarrow worse(R), not\ better(R).$$
$$worseSet \leftarrow worseRule(R).$$
$$betterSet \leftarrow betterRule(R), not\ worseSet.$$
$$\leftarrow not\ betterSet.$$

Intuitively, the predicate $better(r)$ fires if there is a dimension i in r's satisfaction degree vector according to S such that $T(P, S)$ found an answer set S' with a satisfaction degree vector that is better in position i. Conversely, $worse(r)$ fires if a dimension can be found where S' is worse. Note that we do not need to encode ϵ in the Tester, since the rules defining $better(r)$ and $worse(r)$, respectively, are only constructed for comparable options, i.e., pairs of literals occurring in the same disjunct of the same rule. Next, $S' \succ_r S$ (expressed by $betterRule(r)$) holds if there is a dimension where S' is better least but there is no dimension where S' is worse. Analogously, $worseRule(r)$ determines rules such that $S \succ_r S'$. By the remaining two rules and the final constraint, answer set S' only "survives", if it is better in some rule and not worse in any rule. Thus, only those answer sets $S' \succ S$ "pass", (cf. Definition 13).

Proposition 3. *Let S be an answer set of $G(P)$. If $T(P, S)$ does not have any consistent answer sets, then S is an optimal answer set of P.*

By this result, we can implement DLPOD using a standard solver for disjunctive logic programming such as GnT [13]. We further note that LPODs are just a special case of DLPODs:

Proposition 4. *LPODs are a special case of DLPODs and the preferred answer set of an LPOD computed by $G(P)$ and $T(P, S)$ correspond 1-to-1 to the preferred answer sets computed by the generator and tester presented in [11].*

Proof. [sketch] This is easy to see by the correspondence of $G(P)$ and $T(P, S)$ modulo application of lemmas 1 and 2, i.e., the answer sets of the generator and tester programs outlined in [11] only differ by the auxiliary symbols $h_{r,i}$ and $b_{r,i}$ which are introduced according to both lemmata. □

6 Complexity

In the following, we sketch some complexity results for DLPODs which mainly derive from lifting respective results from normal LPODS to the disjunctive case. At this point, we focus on establishing membership results and leave hardness proofs for future work.

Considering the complexity of finding an optimal answer set for LPODs we observe the following. Firstly, it is easy to see that determining whether an optimal answer set exists is not more difficult than determining whether "any" answer set exists, i.e. we can straightforwardly lift Theorem 1 from [11], by the Σ_2^p-completeness of disjunctive logic programs [3].

Theorem 1. *Deciding whether a DLPOD P has an optimal answer set is Σ_2^p-complete.*

The same "lifting" to the second level of the polynomial hierarchy also works for checking whether S is optimal.

Theorem 2. *Deciding whether an answer set S of a DLPOD is an optimal answer set is in Π_2^p.*

Proof. [sketch] *Membership:* analogously to [11]. □

We also conjecture hardness, but leave the proof to future work at this point. The idea would be that in variation of the proof for co-NP-hardness for the non-disjunctive LPOD case—see [11, Proof of Theorem 2]—we should be able to use, instead of a reduction of SAT, a variation of the "standard" disjunctive encoding of QSAT with two quantifier alternations into ASP, see e.g. [14].

Theorem 3. *Given a DLPOD P and a literal $l \in Lit(P)$, deciding whether there exists an optimal answer set S such that $l \in S$ is in Σ_3^p.*

Again, we conjecture hardness, but leave the in-depth investigation to future work. For the moment, let us just focus on membership, which we show by arguing that the algorithm sketched in the previous section indeed can be brought down to Σ_3^p.

Proof. Membership: First note that the algorithm from [11] can, with slight modifications, be used to solve exactly this decision problem. Namely, we need to simply add to the "outer" $G(P)$ computation the constraint "$\leftarrow$ *not* l." invalidating answer sets that do not contain l in the initial guess to G(P). Obviously, this modification yields an algorithm which is in the complexity class $\Sigma_2^{p\,\Sigma_2^p}$. It remains to be shown that this indeed boils down to $\Sigma_3^p = \mathsf{NP}^{\Sigma_2^p}$. Here the idea is the following: As $\Sigma_2^{p\,\Sigma_2^p} = (\mathsf{NP}^{\mathsf{NP}})^{\Sigma_2^p}$ we should be able to use the "outer" Σ_2^p oracle also to compute the "inner" NP oracle calls. In the following, we will sketch how the algorithm of Section 5 can be modified accordingly.

Note that, since G(P) is a *disjunctive* logic program, it can—following the same approach as GnT [13]—be rewritten to two *normal* logic programs: first, $Gen(G(P))$ which takes care of computing the supported models of $G(P)$ and second, $Test(G(P), M)$ which tests for each supported model M whether it is indeed a stable model. Again, the test succeeds by non-existence of an answer set for $Test(G(P), M)$.

After disambiguating symbols occurring in $T(P, M)$ and $Test(G(P), M)$ by replacing symbols within $Test(G(P), M)$ we obtain $Test'(G(P), M)$. This guarantees no "interferences" between the two test modules, which then can simply be combined into a joint tester: $T(P, M) \cup Test'(G(P), M)$. We end up in a modified algorithm for computing optimal answer sets which proceeds as follows:

- Compute an answer set of $Gen(G(P))$
- Determine whether $T(P, M) \cup Test'(G(P), M)$ has no answer set

where $Gen(\cdot)$ and $Test(\cdot,\cdot)$ are the transformations as defined in [13]. Clearly, since $Gen(G(P))$ is solvable in NP, and $T(P,M) \cup Test'(G(P),M)$ is solvable in Σ_2^p, we have shown membership of optimal answer set computation of a DLPOD in Σ_3^p. □

We note that DLPODs preserve the better computational properties when only head-cycle free programs are considered. Actually, all examples in this paper fall in this class of programs.

Theorem 4. *Given a head-cycle-free DLPOD P and a literal $l \in Lit(P)$, deciding whether there exists an optimal answer set S such that $l \in S$ is Σ_2^p-complete.*

Proof. Hardness follows immediately from hardness of this problem for non-disjunctive LPODs. As for membership, we have stated already in Proposition 1 that each split program of a head-cycle-free program is head-cycle-free again. Thus, we can observe that guessing a split and checking whether an answer set S exists such that $l \in S$ is doable in NP and likewise checking non-existence of a better answer set is in co-NP which brings the overall problem down to Σ_2^p. □

In fact, we note that using the methodology in [14] we could even obtain an algorithm encoding optimal answer set computation for head-cycle-free DLPODs into a single disjunctive logic program, instead of interleaved computations.

As next steps, we plan to experimentally compare all three possible implementations, (i) the interleaved computation from Section 5, (ii) its refinement from the proof of Theorem 3, as well as (iii) the integrated encodings for head-cycle-free programs following [14]. We note that many Σ_2^p-complete problems have more concise encodings than the metainterpreter-based encoding in [14] and plan to explore such more concise encodings for the head-cycle-free case.

7 Conclusions and Future Work

In this paper, we have presented a new approach for modelling preferences in logic programs. By extending the approach of Logic Programs with Ordered Disjunction with normal disjunction in the head of rules, we introduce partial order preference expressions for non-monotonic reasoning. We show how to transform a DLPOD into an interleaved disjunctive logic program which allows normal ASP solvers to compute preferred answer sets. Furthermore, we show that computing optimal stable models for our extension still stays in Σ_2^p for head-cycle free programs and establish Σ_3^p upper bounds for the general case.

For future work we plan to experimentally evaluate variants of the generator and tester programs provided in Section 5 with dierent ASP solvers. We remark that our considerations have so far been restricted to DLPODs in (ordered) disjunctive normal form and that the naive transformation to this normal form by applying "distributivity" rewriting rules potentially leads to exponential blowup. A generalization of the definition of the semantics to arbitrarily nested ordered disjunctive terms along with the investigation of the applicability of cheaper, structure-preserving normal form transformations is on our agenda.

In this work we focused on a Pareto-semantic based preference notion. We are aware that in [4] two other preference notions (namely cardinality-preferred and inclusion-preferred) are introduced. However, at the same time they are proven to be not general enough (see the motivation for Def. 11 in [4]). We argue that these two preferences are based on counting and hence do not reflect the qualitative nature of partial order preferences. However, we leave to future work considerations of integrating these preferences into our approach.

For basing our language extensions on solid ground, we are planning to add the hardness proofs for Theorem 2 and Theorem 3. As already stated at the end of Section 6, we plan to compare and evaluate the different implementation strategies outlined in Sections 5 and 6.

References

1. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. New Generation Computing **9** (1991) 401–424
2. Minker, J., Rajasekar, A., Lobo, J.: Foundations of Disjunctive Logic Programming. MIT Press (1992)
3. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (September 1997) 364–418
4. Brewka, G., Niemelä, I., Syrjänen, T.: Logic programs with ordered disjunction. Computational Intelligence **20** (May 2004) 335–357(23)
5. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
6. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. Annals of Mathematics and Artificial Intelligence **12** (1994) 53–87
7. Delgrande, J.P., Schaub, T., Tompits, H., Wang, K.: Towards a classification of preference handling approaches in nonmonotonic reasoning. Computational Intelligence **20** (2003) 308–334
8. Niemelä, I.: Language extensions and software engineering for ASP. Technical report, European Working group on Answer Set Programming (2005)
9. Bertino, E., Mileo, A., Provetti, A.: PDL with preferences. In: Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05), Los Alamitos, CA, USA, IEEE Computer Society (2005) 213–222
10. Skiena, S.: 5.4.2 Hasse Diagrams. In: Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Addison-Wesley (1990) 163, 169–170, and 206–208
11. Brewka, G., Niemelä, I., Syrjänen, T.: Implementing ordered disjunction using answer set solvers for normal programs. In: JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence, London, UK, Springer-Verlag (2002) 444–455
12. Simons, P.: Extending the smodels system with cardinality and weight constraints. In: Logic-Based Artificial Intelligence, Kluwer Academic Publishers (2000) 491–521
13. Janhunen, T., Niemelä, I.: GnT – A Solver for Disjunctive Logic Programs. In: LPNMR 2004. (2004) 331–335
14. Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. Theory and Practice of Logic Programming (TPLP) **6**(1-2) (2006) 23–60

Quantified Logic Programs, Revisited

Rachel Ben-Eliyahu - Zohary

Ben-Gurion University and Jerusalem College of Engineering
Israel
rachel@bgu.ac.il

Abstract. We consider again quantified logic programs (QLP). QLP is a logic program where a positive number not greater than 1 called *dominance* is associated with each rule in the program. The intuition is that rules with higher dominance are more plausible or more reliable. Literals in the answer sets of QLPs are also annotated with weights, with the intuition that a literal with a higher weight is more likely to be true. We present three different applications of QLPs: Ontology Matching, Ranking of search results, and Inheritance Networks. We also address the problem of computing answer sets of QLPs. We show that we can compute the answer sets by first using existing state of the art tools for answer set generation, and then applying a polynomial time forward-chaining like algorithm, called the *Proof-Rank* Algorithm in order to compute the weight of each literal in the answer set.

1 Introduction

Answer set semantics, as defined by Gelfond and Lifschitz [6, 7], and all other known classical semantics for general logic programs like the well founded semantics [5] treat all rules equally. That is, all the rules in the program are considered to be of the same importance, or reliability, or dominance. In this paper, we suggest that sometimes it is desirable to rank the rules in the program, and argue that an answer set of a program with ranked rules should be a set of ranked literals. We call logic programs with rank on rules *quantified* logic programs.

Many forms of quantified logic program exist in the literature (e.g. [3, 8–10]). To distinguish the programs presented here from the rest we will call them nonmonotonic quantified logic programs, or QLPs, in short. QLPs have the following five features. As far as we can tell, none of the quantified logic programs suggested in the past have all of the characteristics listed below:

1. The language of the program is the language of extended logic programs (that is, the programs have classical negation in addition to negation by failure).
2. Each rule in the program has a rank associated with it. Rules with higher rank are considered more reliable.
3. Each literal in an answer set of the program has a rank associated with it. Literals with higher ranks are considered more plausible.

4. The programs are a generalization of extended logic programs under answer set semantics.
5. The answer sets of the programs can be computed very efficiently using the state of the art systems for computing answer sets of extended logic programs.

In the sequel, we will provide a detailed comparison to existing work.

The need for QLPs as presented here has emerged in an ontology matching project. We wanted to develop a tool for semiautomatic interactive matching. The tool should have worked as follows. It should suggest to the user certain possible matches, based on some preliminary information on the similarity of the concepts. Then, the user should accept or decline some of the matches presented to her. The user's feedback on the right matching should be feed into a reasoning engine that will calculate what are the next plausible matches based on the user's input.

The problem was that if we adopt logic programs with stable model semantics as a reasoning engine we would get the result that some pairs of nodes are the next candidates for a match and some are not, but all the pairs that are a possible match have the same stand. We have no way to distinguish between the possible pairs and order them somehow according to the likelihood of them being matched. Using QLPs instead of standard logic programs, we were able to develop a reasoning tool that provides ordering among the possible matches. This motivating example will be explained in detail in the following sections.

Once we have develop QLPs for ontology matching, we have found two additional application domains for QLPs: page rank and Inheritance Networks. We will elaborate on this in the sequel as well.

2 Preliminary Definitions

In this section we briefly review preliminary definitions used in extended logic programs. Note that only the propositional fragment of these logics is considered here. The extension of this work to first-order logic programs will be discussed in future work.Thus, whenever a a logic program with variables is used, it is referred to as an abbreviation of its grounded version.

A literal is an expression of the form ℓ or $\neg\ell$ where ℓ is a propositional letter and the symbol '$\neg$' denotes classical negation. A propositional ELP (Extended Logic Program) is a collection of rules of the form

$$L_1 \leftarrow L_2, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$

where $n, m \geq 0$, the symbol $'not'$ denotes negation by default and each L_i is a literal. L_1 is the head of the rule and the literals to the right of the arrow are called the body of the rule. If the head is empty then the rule is called an *integrity clause*. The literals $L_2, \ldots, L_m$ are said to *appear positive* in the rule, while the literals $L_{m+1}, \ldots, L_n$ are said to *appear negative* in the rule. When $m + n = 0$, the rule is called "body-free". Sometimes we will call body-free rules

observations. Sometimes we will write body-free rule $L_1 \longleftarrow$ simply as L_1 (that is, without the arrow).

An ELP Π is given semantics using *answer sets* [7], which are defined as follows: Let $lits(\Pi)$ denote the set of literals obtained using the propositional letters occurring in Π. By a *context* [1] we mean any subset of $lits(\Pi)$.

Let P be a *negation-by-default-free* ELP. A context S is *closed under* P if for each rule $L_1 \leftarrow L_2, \ldots, L_m$, if $L_2, \ldots L_m \in S$ then $L_1 \in S$. An answer set of P is any minimal context S such that (1) S is closed under P and (2) if S is inconsistent then $S = lits(\Pi)$.

For general ELPs answer sets are defined as follows: Let the *reduct of P w.r.t. the context S*, denoted by $red(P, S)$, be the ELP obtained from P by deleting (i) each rule that has *not* L in its body, for some $L \in S$, and (ii) all remaining subformulas of the form *not* L from rule bodies. Then, any context S which is an answer set of $red(P, S)$ is an *answer set* of P.

Definition 21 *A set of literals S satisfies the body of a rule δ if all the literals that appear positive in the body of δ are in S and all the literals that appear negative in δ are not in S.*

According to [1], a proof of a literal is a sequence of rules that can be used to derive the literal from the program. The authors of [1] provide a definition of a proof of a disjunctive LP. Here we take the non-disjunctive version of the definition:

Definition 22 (Proof) *[1] A literal L has a* proof *w.r.t. a set of literals S and a logic program Π if and only if there is a sequence of rules $\delta_1, ..., \delta_n$ from Π such that:*

1. *for all $1 \leq i \leq n$ the literal in the head of δ_i belongs to S.*
2. *there exists $1 \leq i \leq n$ such that L is the head of δ_i.*
3. *for all $1 \leq i \leq n$, the body of δ_i is satisfied by S.*
4. *δ_1 has no literals that appear positive in its body, and for each $1 < i \leq n$, each literal that appears positive in the body of δ_i is in the head of some δ_j for some $1 \leq j < i$*

The following is Lemma B.5 from [1] restricted for the case of nondisjunctive LP:

Lemma 1 ([1]). *Let Π be a logic program and let S be an answer set of Π. Then each literal in S has a proof with respect to Π and S.*

3 Definition of QLP

In this section we formally define QLPs and their semantics.

Definition 31 (QLP) Quantified Logic Program*, or in short,* QLP *is a logic program where a positive number p_δ which is less or equal to 1 is associated with each rule δ. We will call p_δ the* dominance *of δ. Given a QLP Π, we will denote by $\hat{\Pi}$ the set of rules in Π (that is, $\hat{\Pi}$ is Π where we ignore the dominance of each rule).*

Intuitively, in QLPs the rules are annotated with numbers that are supposed to amount to their reliability. The answer sets of a QLP Π are the same as the answer sets of its corresponding ELP $\hat{\Pi}$, except that each literal in an answer set has also a number attached to it certifying to its plausibility. The basic idea is as follows. According to Lemma B.5 of [1] cited in the previous section, each literal in an answer set has a proof. If the strongest proof of a literal L_1 in an answer set is based on rules that are more reliable than the rules used in the strongest proof of a different literal L_2, then L_1 should have a higher rank attached to it. We next define the strength, or the *dominance* of a proof.

Definition 32 (dominance of a proof) *Let Π be a QLP, S an answer set of $\hat{\Pi}$, $L \in S$ and $\delta_1, ..., \delta_n$ a proof of L w.r.t. $\hat{\Pi}$ and S. The* dominance *of the proof $\delta_1, ..., \delta_n$ is $p_{\delta_1} * ... * p_{\delta_n}$, where p_{δ_i} is the dominance of δ_i.*

So the dominance of a proof is the product of the dominance of the rules used in the proof. Now we can define the answer set of a QLP.

Definition 33 (QAS) *A* Quantified Answer Set*, or in short, QAS, is an answer set where a positive number which is less or equal to 1 is associated with each literal in the answer set. Given a QAS S, we will denote by $\hat{S}$ the set of literals in S (that is, $\hat{S}$ is S where we ignore the number associated with each literal).*

The dominance of a literal is the highest dominance of a proof of the literal.

Definition 34 (dominance of a literal in an answer set) *Let Π be a QLP, S an answer set of $\hat{\Pi}$ and $L \in S$. The dominance of L is the maximum dominance of any proof of L w.r.t. S and Π.*

Definition 35 (QAS of a QLP) *A QAS S is a QAS of a QLP Π iff the following conditions hold:*

1. *$\hat{S}$ is an answer set of $\hat{\Pi}$;*
2. *The number associated with each literal in S is its dominance w.r.t. Π and $\hat{S}$.*

Note that if we assign dominance 1 to all the rules in the programs the QLP will be actually a standard ELP.

4 Applications

4.1 Ontology matching

Ontology typically provides a vocabulary that describes a domain of interest and a specification of the meaning of terms used in the vocabulary. For example, the United Nations Development Program and Dun & Bradstreet combined their efforts to develop the UNSPSC ontology, which provides terminology for products and services (www.unspsc.org). Ontology matching is a promising solution

to the semantic heterogeneity problem, as it finds correspondences between semantically related entities of ontologies. These correspondences can be used for various tasks, such as ontology merging, query answering, data translation, or for navigation on the semantic web.

Matching ontologies enables the knowledge and data expressed in the matched ontologies to interoperate. The distributed nature of ontology development has led to a large number of different ontologies covering the same or overlapping domains. In order for two parties to understand each other, they should use the same formal representation for the shared conceptualization (namely, use the same ontology). Unfortunately, it is not easy to convince everybody to agree on the same ontology for a domain, and when you have different ontologies for the same domain, the problem shows up: parties with different ontologies do not understand each other.

In a research that we have conducted for a consortium of imaging machines companies, we had to develop a semi-automatic interactive tool for ontology matching. We were looking for an efficient algorithm that when given two ontologies, will output a table of matching candidates between concepts of the two ontologies. The algorithm should have offered the user initial matching candidates, and the user should decide which ones to accept, and which ones to decline. The algorithm should recalculate new matching candidates, given the user's feedback.

Suppose we would like to match the two ontologies depicted in Figure 1, where the edges represent an Is-A relation. The concepts are shortened as follows: V-Vehicles, LV - Land Vehicles, C- Car, Cs - Cars, T-Trucks, A- Airplanes, Tn - Train, D-Diesel, E-Electronic, P-Pickup, H-Hybrid. Assume we want to use logic programming as a reasoning tool for deciding which pair to match at each stage. We can use the logic program Π having the following rules.

Ontology 1 Concept(V), Concept (Cs), Concept(T), Concept(A), Concept(D), Concept(E), Concept(P). IsA(C,V), IsA(T,V), IsA(A,V), IsA(D,C), IsA(E,C), IsA(P,T).

Ontology 2 Concept(LV), Concept (C), Concept(T), Concept(H). IsA(C,LV), IsA(T,LV), IsA(H,C), IsA(D,C).

Child to Father matching default

$$Match(x,y) \longleftarrow Match(x',y'), IsA(x',x), IsA(y',y), not\ \neg Match(x,y).$$

The "child to father" matching defaults asserts that if two concepts match, then by default, unless the contrary is known, their corresponding super classes match. Suppose now that the user gives us the information that concept *Diesel* in Ontology 1 matches concept *Diesel* in Ontology 2, and we add the rule r : $Match(D,D) \longleftarrow$ to the program. In the answer set of $\Pi \bigcup \{r\}$ we have the following literals (in addition to the literals that describe the otologies): $Match(Cs,C)$, $Match(V,VL)$. And of course, if D and D had more ancestors in the ontolgy hierrachy we would have even more "Match" literals.

Intuitively, if it is given that two concepts match, it is more likely that their immediate predecessors match than the other predecessors. We would like the

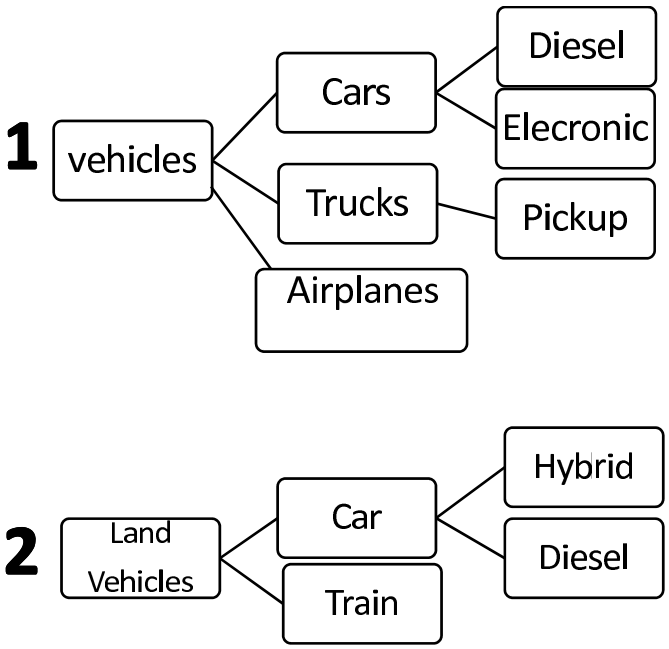


Fig. 1. Two ontologies to be matched.

system to be able to suggest that $Cars$ and Car are the next best possible match.

The problem is that, using the standard ELP, we have no way to distinguish between the "Match" literals, and therefore we cannot suggest to the user which one is probably the next match. Another problem with using the standard logic programming paradigm, is that it does not allow us to state that some rules are more reliable than others. For example, we would like to use another rule, that states the following:

Father to Child matching default

$$Match(x', y') \longleftarrow Match(x, y), IsA(x', x), IsA(y', y), not\ \neg Match(x', y').$$

However, the father-to child matching rule and the child to father matching rule are not of the same strength. If two concepts match, it is more likely that their immediate superclass match than two of their subclasses. Moreover, we would like to state in advance, before getting the initial feedback from the user, that some matches are more plausible than the other, or in other words, that, for example, $Match(Diesel, Diesel)$ is more likely than $Match(Electronic, Diesel)$. We can guess this using wordnet::similarity [11] (see also `http://wn-similarity.sourceforge.net`).

Given the limitations of standard ELP illustrated above and of other approaches developed in the past (see for example [4]), we have decided to develop the semantics of QLPs. Using QLP, the interactive ontology matching could work as follows. First, based on concept-similarity tools like wordnet::similarity, we suggest to the user the match between *Diesel* of Ontology 1 and *Diesel* of Ontology 2. Suppose the user confirms this match. We continue with the QLP Π_1 that includes the description of the two ontologies (the literals in the ontology description each has dominance 1) and the following rules. The numbers in brackets are the dominance.

$$Match(x, y) \longleftarrow Match(x', y'), IsA(x', x), IsA(y', y), not\ \neg Match(x, y)\ (0.8)$$
$$Match(x', y') \longleftarrow Match(x, y), IsA(x', x), IsA(y', y), not\ \neg Match(x', y')\ (0.3)$$
$$Match(Diesel, Diesel)\qquad (1)$$

The answer set S_1 of $\hat{\Pi_1}$ is computed, and once the dominance of the literals in S_1 is computed, it is easy to see that $Match(Car, Cars)$ is the literal with the highest dominance (0.8). We now suggest to the user the match between *Cars* of Ontology 1 and *Car* of Ontology 2. Suppose the user confirms this match. So we continue with the QLP $\Pi_2 = \Pi_1 \bigcup \{Match(Cars, Car)\}$, where $\{Match(Cars, Car)\}$ has dominance 1. The answer set S_2 of $\hat{\Pi_2}$ is computed, and once the dominance of the literals in S_2 is computed, it is easy to see that $Match(Vehicles, Land_Vehicles)$ is the literal with the highest dominance (0.8). We now suggest to the user the match between *Vehicles* of Ontology 1 and *Land Vehicles* of Ontology 2. Suppose the user denies this match. So we continue with the QLP $\Pi_3 = \Pi_2 \bigcup \{\neg Match(Vehicles, Land\_Vehicles)\}$

($\neg Match(Vehicles, Land\_Vehicles)$ having weight 1) and so on. This interactive process continues until all possible matches are suggested or alternatively, until the user decides to stop it. It seems to us that the tool described above could not be developed using the standard ELP.

4.2 Dynamic Page Rank

Search engines apply sophisticated algorithms in order to rank the pages that match the query posed by the user. The pages are then displayed in order according to their rank. Suppose we would like to use dynamic ranking. That is, instead of ordering the pages in advance, re-order them according to the user actions. If a user clicks on a link and stays there for more than 3 seconds, it means that the page is relevant. If the user clicks on a link and does not stay their more than half a second, it means that it is not relevant. We then want to move pages that are similar to the relevant page higher in the ordering.

Suppose that we have already identified categories by which to judge whether two pages are similar: e.g., the IP address, the language they are written in, the type of the page, etc. Moreover, suppose that we have a similarity algorithm that decides how much two pages are similar based on these criteria. Based on the information provided by the similarity algorithm, we can classify any two pages as "somewhat similar","similar", "very similar", "almost identical", etc . In general, we can classify any two pages as $similar_1, similar_2, ..., similar_n$ according to their level of similarity. Then we can have the following rules, for $1 \leq i \leq n$:

$$Relevant(y) \longleftarrow Similar_i(x, y), Relevant(x), not\ \neg Relevant(y)$$

The rule asserts that if a page x is relevant and page y is similar to x and if we do not know that page y is not relevant, then page y is relevant as well. Since we have several levels of similarity, the dominance of each such rule should be in proportion to the level of similarity that the predicate similar used in the rule represents. For this purpose we can use QLPs.

Dynamic ordering can also be useful in other search episodes such as searching in an electronic mailbox. We can find criteria according to which we can classify messages: name of sender, IP address, does the message has attachment,the date of the message, etc.

4.3 Inheritance Networks

A considerable effort was invested in the past in order to provide formal semantics to multiple inheritance networks (see for example [2, 13, 12]). We will provide here two examples that demonstrate how QLPs can help in representing inheritance in an intuitive way.

4.3.1 Penguins and Birds Suppose we formalize the famous defaults "birds fly", "penguins do not fly", and the observation "Pit is a penguin" in ELP:

$$Bird(x) \longleftarrow Penguin(x) \quad (1)$$
$$\neg Fly(x) \longleftarrow Penguin(x), not\ Fly(x) \quad (2)$$
$$Fly(x) \longleftarrow Bird(x), not\ \neg Fly(x) \quad (3)$$
$$penguin(Pit) \longleftarrow \quad (4)$$

The above program has two answer sets: 1. $\{Penguin(Pit), Bird(Pit), Fly(Pit)\}$, and 2. $\{Penguin(Pit), Bird(Pit), \neg Fly(Pit)\}$. Intuitively, we prefer the 2nd answer set, but the standard ELP is not capable of conveying this. We argue that with QLPs we can obtain the desired result. We need first the defintion of preferred answer set:

Definition 41 (Preferred Answer Set) *Let Π be a QLP and let S and S' be two QASs of Π. We say that S is* preferred *over S' is the average of the weight of the literals in S is greater than the average of the weight of the literals in S'.*

The intuition behind this definition is that the higher the average of the weights of the literals in the answer set is, the more sense the answer set makes because the literals were inferred using rules that are more reliable.

Back to QLP- if we represent the knowledge in QLP, we will assign dominance 1 to rules 1 and 4, and assign to Rule 3 dominance which is smaller than that of Rule 2 (e.g. Rule 3 will have dominance 0.8 and Rule 4 -0.95). This is because the rule "birds fly" has many more exceptions than the rule "Penguins do not fly". Then we will get the result that answer set 2 is the preferred answer set.

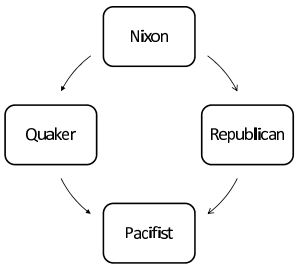


Fig. 2. The Nixon Diamond

4.3.2 Nixon Diamond Consider the famous Nixon Diamond Paradox:

- Nixon is a Quaker
- Nixon is Republican
- Republicans are not pacifists.
- Quakers are pacifists.

This knowledge can be encoded in the following LP:

$$\begin{aligned} Republican(Nixon).& \\ Quaker(Nixon).& \\ \neg Pacifist(x) &\longleftarrow Republican(x), not\ Pacifist(x) \\ Pacifist(x) &\longleftarrow Quaker(x), not\ \neg Pacifist(x) \end{aligned}$$

This program has two answer sets:

1. $\{Republican(Nixon), Quaker(Nixon), \neg Pacifist(Nixon)\}$, and
2. $\{Republican(Nixon), Quaker(Nixon), Pacifist(Nixon)\}$.

It seems like one answer set makes more sense the other. This depends on how do we evaluate the strength of each of the two rules in the program. Suppose we believe that the rule "Republicans are not pacifists" is more dominant than the default "quakers are pacifist". In addition, there is no doubt about the fact that Nixon was Republican, but we are not sure to what extent he was indeed a quaker. By assigning to the rules numbers that reflect these believes we can get the result that the answer set that is more intuitive to us is preferred.

5 Related Work

The research that is the most relevant to the work presented here seems to be the work on prioritized defaults (e.g. [3]) and the work on probabilistic logic programs (e.g. [9]). As we have mentioned in the introduction, each of these frameworks lacks some of the features that QLPs presented here have. Prioritized defaults enable specifying priorities among defaults so that one answer set is preferred over the other, but it is not possible to rank literals in one answer set according to their likelihood. Probabilistic logic programs, on the other hand, are not rich enough to allow negation by default. Moreover, as far as we can tell, none of the above paradigms offer an efficient procedure for computing the answer sets. Computing answer sets of QLPs is relatively easy, as we explain in the next section.

6 Computational Issues

Given a QLP Π, , we can find all its QASs very easily using existing methods for computing answer sets. First, we will compute all answer sets of $\hat{\Pi}$. Then, for each answer set of $\hat{\Pi}$ we will apply the Proof-Rank Algorithm described below to compute the weight of each literal in the answer set.

The data structures that the proof-rank algorithm handles are the following:

1. For each rule r in Π: proof[r] is the best proof that can be found so far for the literal in the head of r provided that the last rule in the proof is r, and weight[r] is the dominance of that proof. body[r] is the number of literals in the body of r for which the best proof was not found yet.
2. For each literal P in the answer set, dominance[P] will hold the dominance of P once we know it and proof[P] is the best proof for P (that is, the proof of P with the highest dominance).
3. A heap where all the rules r are kept according to the value of the variable $weight[r]$.

Proof-Rank Algorithm can work as follows:

1. For each rule r, let weight[r] = 0, body[r] = number of literals in the body of r, proof[r] = $\emptyset$.
2. For each rule r such that body[r] = 0, let weight[r] = the dominance of r, $Proof[r] = \{r\}$.
3. Sort all rules in a heap by their weight, such that a heaviest one is on top. Let top[*heap*] denote the rule at the top of the heap.
4. While the heap is not empty and the weight of r =top[heap] is greater than 0 do
   (a) Let P be the literal in the head of r. let $dominance[P] = weight[r]$, $Proof[P] = Proof[r]$.
   (b) For each rule r in the heap such that P is in the head of r, remove r from the heap.
   (c) For each rule r in the heap such that P is in the body of r do the following
      i. Decrease body[r] by 1.
      ii. If body[r] = 0 do the following, where $Q_1, ..., Q_m$ are all the literals in the body of r.
         A. Proof[r]=Proof[Q_1] $\bigcup$ Proof[Q_2] $\bigcup$... $\bigcup$ Proof[Q_m] $\bigcup$ $\{r\}$.
         B. Let weight[r] be the product of the dominance of all the rules in proof[r].
         C. Move r to its correct place in the heap.

It is not difficult to see that the complexity of the Proof-Rank Algorithm is $O(nl^2)$ where n be the number of literals in Π and l the number of rules in Π. The proof of the correctness of the algorithm is left for the full paper. We next illustrate how it works.

Example 61 *Suppose we are given the following QLP Π:*

$$1.\neg D \longleftarrow not\ C(0.4)$$
$$2.C \longleftarrow \neg D(0.9)$$
$$3.C \longleftarrow (0.8)$$
$$4.A \longleftarrow not\ B(0.5)$$
$$5.\neg D \longleftarrow A, C(0.8)$$

In order to compute the quantified answer set of Π, we first compute the answer set $\hat{S}$ of $\hat{\Pi}$, which is $\{C, A, \neg D\}$. We then take $red(\Pi, S)$ which is:

$$2.C \longleftarrow \neg D(0.9)$$
$$3.C \longleftarrow (0.8)$$
$$4.A \longleftarrow (0.5)$$
$$5.\neg D \longleftarrow A, C(0.8)$$

We now sort the rules in a heap, with the following weight, "body", and "proof" values:

$$3.C \longleftarrow (0.8)\ body = 0\ proof = \{3\}$$
$$4.A \longleftarrow (0.5)\ body = 0\ proof = \{4\}$$
$$2.C \longleftarrow \neg D(0)\ body = 1\ proof = \emptyset$$
$$5.\neg D \longleftarrow A, C(0)\ body = 2\ proof = \emptyset$$

Rule number 3 is the heaviest, we remove it from the heap and set $dominance[C] = 0.8$ and $proof[C] = \{3\}$. We remove Rule 2 from the heap because it has the head C, and we decrease the variable body of Rule 5 by 1 because Rule 5 has C in the body. So the heap now looks like this:

$$4.A \longleftarrow (0.5)\ body = 0\ proof = \{4\}$$
$$5.\neg D \longleftarrow A, C(0)\ body = 1\ proof = \emptyset$$

Rule number 4 is now the heaviest, we remove it from the heap, set $dominance[A] = 0.5$ and $proof[A] = \{4\}$. We decrease the variable body of Rule 5 by 1 because Rule 5 has A in the body. Rule 5 has now body=0 so we set $proof[5]$ to be $\{3, 4, 5\}$ and $weight[5]$ to be $0.8 * 0.5 * 0.8$, which is the product of the dominance of rules $3, 4$ and 5.

$$5.\neg D \longleftarrow A, C(0.32) body = 0 proof = \{3, 4, 5\}$$

Rule number 5 is now the heaviest, we remove it from the heap, set $dominance[\neg D] = 0.32$ and $proof[\neg D] = \{3, 4, 5\}$. Now the heap is empty and therefore the algorithm terminates.

7 Conclusions

Stable model semantics is becoming more and more practical with the development of advanced tools for computing answers sets of extended logic programs. In this paper we show that still there are applications for which the standard ELPs cannot provide a solution, and we present QLPs. QLPs can still benefit from all the progress done in stable model computation. This is because answer sets of QLPs are computed by first computing an answer set of a standard ELP and then applying a polynomial time algorithm.

The major disadvantage of QLPs, as we see it, is how to decide what would be the exact dominance of each rule in the program. As of now, we leave this decision to the intuition of the knowledge engineer who writes the program.

This work can be generalized to head-cycle free (HDF) *disjunctive* extended logic programs in a straightforward way. This is because the definition of a proof and the lemma that asserts that every literal in an answer set should have a proof extend very naturally to the disjunctive case (for details see [1].

Acknowledgement

This research was supported by a grant from the Israeli Ministry of Industry, Trade, and Labor.

References

1. Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
2. Craig Boutilier. A semantical approach to stable inheritance reasoning. In *IJCAI-89: Proceedings of the 11th international joint conference on AI*, pages 1134–1139, Detroit, Michigan, USA, August 1989.
3. Gerhard Brewka and Thomas Eiter. Prioritizing default logic: Abridged report. In *In Festschrift on the occasion of Prof.Dr. W. Bibel's 60th birthday*. Kluwer, 1999.
4. Andrea Calì, Thomas Lukasiewicz, Livia Predoiu, and Heiner Stuckenschmidt. Tightly integrated probabilistic description logic programs for representing ontology mappings. In *FoIKS*, pages 178–198, 2008.
5. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38:620–650, 1991.
6. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the 5th international conference*, pages 1070–1080. MIT Press, 1988.
7. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
8. Thomas Lukasiewicz. Probabilistic logic programming. In *In Proc. of the 13th European Conf. on Artificial Intelligence (ECAI-98*, pages 388–392. J. Wiley & Sons, 1998.
9. Raymond Ng and V. S. Subrahmanian. Probabilistic logic programming. *Inf. Comput.*, 101(2):150–201, 1992.
10. Pascal Nicolas, Laurent Garcia, Igor Stéphan, and Claire Lefèvre. Possibilistic uncertainty handling for answer set programming. *Ann. Math. Artif. Intell.*, 47(1-2):139–181, 2006.
11. Ted Pedersen and Siddharth Patwardhan. Wordnet::similarity - measuring the relatedness of concepts. In *In Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04*, pages 1024–1025, 2004.
12. E. Sandwell. nonmonotonic inference rules for multiple inheritance with exceptions. In *Proc. of the IEEE*, pages 1345–1353, 1986.

13. David S. Touretzky, John F. Horty, and Richmond H. Thomason. A clash of intuitions: The current state of nonmonotonic multiple inheritance systems. In *IJCAI-87: Proceedings of the 10th international joint conference on AI*, pages 476–482, Milan, Italy, August 1987.

On Demand Indexing for the DLV Instantiator

Gelsomina Catalano, Nicola Leone, and Simona Perri

Department of Mathematics, University of Calabria
I-87030 Rende (CS), Italy
{catalano,leone,perri}@mat.unical.it

Abstract. In Answer Set Programming (ASP) systems, the computation consists of two main phases: (1) the input program is first instantiated and simplified, generating a ground (i.e., variable free) program, (2) propositional algorithms are then applied on the ground program to generate the answer sets. The instantiation process may be computationally expensive and the instantiator is crucial for the efficiency of the entire ASP system, especially in real-world applications.
In this paper, we propose to employ main-memory indexing techniques for enhancing the performance of the instantiation procedure of the ASP system **DLV**. In particular, we adapt a classical first argument indexing schema to our context, and propose an on demand indexing strategy where indexes are computed during the evaluation (and only if exploitable). Moreover, we define two heuristics which can be used for determining the most appropriate argument to be indexed, when more than one possibility exists.
We have implemented such techniques in the **DLV** instantiator, and we have carried out an experimentation activity on a collection of benchmark problems, including also a number of real-world instances. The results of experiments are very positive and confirm the intuition that indexing allows for notably improving the efficiency of the instantiation process. Moreover, the on demand indexing strategy always outperforms the classical first argument schema, especially when the argument to be indexed is chosen according to a better heuristic.

1 Introduction

Answer set programming (ASP) – a declarative approach to programming proposed in the area of logic programming and nonmonotonic reasoning – has gained popularity in the last years also thanks to the availability of a number of effective implementations. Indeed, there are nowadays a number of systems that support Answer Set Programming and its variants, including [1–9], that can be utilized as advanced tools for solving real-world problems in a highly declarative way.

The computation of the answer sets in ASP systems consists of two main phases: the input program is first instantiated and simplified, generating a ground (i.e., variable free) program, and then propositional algorithms are applied on the ground program to generate the answer sets. The instantiation phase may be computationally expensive. Thus having a good instantiator is crucial for the efficiency of the entire ASP system. Some emerging application areas of ASP, like knowledge management and information integration, [1] where large amount of data are to be processed, make very evident the need of improving ASP instantiators significantly.

[1] The application of ASP in these areas has been investigated also in the EU projects INFOMIX IST-2001-33570, and ICONS IST-2001-32429, and is profitably exploited by Exeura s.r.l., a spin-off of University of Calabria having precisely this mission.

This paper focuses on the instantiator of **DLV** which is widely recognized to be a very strong point of the **DLV** system. Indeed, instantiation in **DLV** is much more than a simple variables-elimination; it allows to evaluate relevant programs fragments, and produces a ground program which has precisely the same answer sets as the theoretical instantiation, but it is sensibly smaller in size. For instance, if the input program is disjunction-free and stratified, then its evaluation is completely done by the instantiator which computes the single answer set without producing any instantiation.

The **DLV** instantiator already incorporates a number of optimization techniques [10–12] but, since ASP applications grow in size, there is the need to efficiently handle larger and larger amount of data.

A critical issue for the efficiency of the instantiator is the retrieval of ground instances from the extensions of the predicates. Indeed, rule instantiation is essentially performed by evaluating the relational join of the positive body literals [2], and, as for join computation, in the absence of techniques for speeding-up the retrieval, the time spent in identifying candidate instances can dramatically affect the performances.

In this paper, we face this issue and, in this respect, we propose the use of indexing techniques, that is techniques for the design and the implementation of data structures that allow to efficiently access to large datasets.

Indexing methods have been originally introduced in the database field for improving the speed of the operations in a table [3], and are now profitably used also in the logic programming area [14, 15]. Indeed, effective indexing has become an integral component of high performance declarative programming systems. Almost all the Prolog implementations support indexing on the main functor symbol of the first argument of predicates and some of them, like XSB [16], SWI-Prolog [17], support more sophisticated indexing schemata. Recently, a smart dynamic indexing method for Prolog systems has also been proposed [18] which resulted to be very effective.

The reason for developing several different indexing techniques is that the conditions under which data have to be retrieved differ from context to context. In addition, if on the one hand indexing allows to significantly speedup the retrieval, on the other hand it could lead to a considerable memory consumption and hence a compromise between these two factors has to be made. Thus, there is not an optimum indexing technique for all the applications, rather each application may require to develop its own specialized technique.

In this work we investigate the use of indexes for optimizing the rule instantiation process of **DLV**. In particular, we adapt a classical first argument indexing schema to our context and propose a more general strategy for indexing any argument, not necessarily the first one. Such a strategy allows for a kind of on demand indexing where indexes are computed during the evaluation and when needed. Moreover, when more than one argument can be indexed, we choose the most appropriate one according to a heuristic. We experiment with two heuristics: in the first one, the first "indexable" argument is chosen while the second one selects the indexable argument where it is more likely that few candidate instances will be retrieved (thus, reducing the cost of the join computation).

[2] Note that, since rules are safe [13], the join of the positive body literals allows for instantiating all rule variables.

[3] Many indexing structures have been already proposed for database systems, however they are designed to be stored in mass-memory, whereas the **DLV** instantiator works in main-memory, thus indexing has to be designed according to more strict memory limits.

We have implemented the above strategies in the instantiation procedure of **DLV** and, in order to assess their impact on the performances of the **DLV** instantiator, we have carried out an experimentation activity on a collection of benchmark programs. The results of the experiments are very positive; the new techniques allow to obtain non-trivial speed-ups w.r.t. the non indexed version, up to orders of magnitude. Moreover, as expected, on demand indexing allows for better performance on a wider range of applications; and the comparison between the two heuristics shows that execution times are strongly influenced by the selection of the indexed arguments, obtaining better results when a more refined choice is done.

2 The Language of Answer Set Programming

In this section we briefly describe the syntax and semantics of Answer Set Programming.

2.1 Syntax

A variable or a constant is a *term*. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* *not* p, where p is an atom.[4] A *(disjunctive) rule* r has the following form:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \mathit{not}\ b_{k+1}, \cdots, \mathit{not}\ b_m,$$
$$n \geq 1,\ m \geq k \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \mathit{not}\ b_{k+1}, ..., \mathit{not}\ b_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, ..., b_k, \mathit{not}\ b_{k+1}, \ldots, \mathit{not}\ b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. For a literal L, $var(L)$ denotes the set of variables occurring in L. For a conjunction (or a set) of literals C, $var(C)$ denotes the set of variables occurring in the literals in C, and, for a rule r, $var(r) = var(H(r)) \cup var(B(r))$. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r, i.e., $var(r) = var(B^+(r))$.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. A *not* -free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it.

A predicate occurring only in *facts* (rules of the form $a \leftarrow$), is referred to as an *EDB* predicate, all others as *IDB* predicates. The set of facts in which *EDB* predicates occur, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

2.2 Semantics

Let $\mathcal{P}$ be a program. The *Herbrand Universe* and the *Herbrand Base* of $\mathcal{P}$ are defined in the standard way and denoted by $U_\mathcal{P}$ and $B_\mathcal{P}$, respectively.

Given a rule r, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where $\sigma : var(r) \mapsto U_\mathcal{P}$ is a substitution mapping the variables occurring in r to constants in $U_\mathcal{P}$. We denote by $ground(\mathcal{P})$ the set of all the ground instances of the rules occurring in $\mathcal{P}$.

[4] Without loss of generality, in this paper we do not consider strong negation, which is irrelevant for the instantiation process; the symbol '*not*' denotes default negation here.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal *not* A is *true* w.r.t. I if A is false w.r.t. I; otherwise *not* A is false w.r.t. I.

Let r be a ground rule in *ground*($\mathcal{P}$). The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in$ *ground*($\mathcal{P}$) is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by MM($\mathcal{P}$).

Given a ground program $\mathcal{P}$ and an interpretation I, the *reduct* of $\mathcal{P}$ w.r.t. I is the subset $\mathcal{P}^I$ of $\mathcal{P}$, which is obtained from $\mathcal{P}$ by deleting rules in which a body literal is false w.r.t. I.

Note that the above definition of reduct, proposed in [19], simplifies the original definition of Gelfond-Lifschitz (GL) transform [20], but is fully equivalent to the GL transform for the definition of answer sets [19].

Let I be an interpretation for a program $\mathcal{P}$. I is an *answer set* for $\mathcal{P}$ if $I \in$ MM($\mathcal{P}^I$) (i.e., I is a minimal model for the program $\mathcal{P}^I$) [20, 21].

3 Instantiation of Answer Set Programs: DLV's Strategy

In this Section, we provide a short description of the overall instantiation process of the **DLV** system, and focus on the "heart" procedure of this process which produces the ground instances of a given rule.

Using advanced database techniques ([10, 11]) and a clever backjumping algorithm [12], the **DLV** instantiator efficiently generates a ground instantiation of the input that has the same answer sets as the full program instantiation, but is much smaller in general. For example, if the input program is normal and stratified, the instantiator is able to compute the single answer set of the program, namely the set of the facts and the atoms derived by the instantiation procedure.

An input program $\mathcal{P}$ is first analyzed by the parser, which also builds the extensional database from the facts in the program, and encodes the rules in the intensional database in a suitable way. Then, a rewriting procedure (see [10]), optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. At this point, another module of the instantiator computes the dependency graph [22] of $\mathcal{P}'$, its connected components, and a topological ordering of these components. Finally, $\mathcal{P}'$ is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component, according to the dependency graph.

For the instantiation of each component an improved version of the generalized semi-naive technique [13, 22] is used. Non recursive rules are evaluated by a single call to the *Instantiate* procedure described in Section 3.1, while the recursive ones are processed several times, and at each iteration only the information derived during the previous iteration is used.

3.1 Rule Instantiation

In this Section, we describe the process of rule instantiation of **DLV**. For the sake of clarity, we present a simplified version of this process, based on a classical chronolog-

ical backtracking schema; the actual **DLV** instantiation procedure exploits a more efficient backjumping technique (see, [12]).

```
Algorithm Instantiate
Input  R: Rule, I_{L_1}, ..., I_{L_n}: SetOfInstances;
Output S: SetOfTotalSubstitutions;
var L: Literal, B: ListOfAtoms, θ: Substitution;
begin
    θ = ∅;
    (* return the ordered list of the body literals (null, L_1,...,L_n, last)  *)
    B := BodyToList(R);
    L := L_1;   S := ∅;
    while L ≠ null
        if  Match(L, I_L, θ) then
            if(L ≠ last) then
                L := NextLiteral(L);
            else (* θ is a total substitution for the variables of R *)
                S := S ∪ {θ};
                L := PreviousLiteral(L); (* look for another solution *)
                θ := θ |_{PreviousVars(L)};
        else
            L := PreviousLiteral(L);
            θ := θ |_{PreviousVars(L)};
    output S;
end;

Function Match (L:Literal, I_L:SetOfInstances, var θ:Substitution):Boolean
begin
    (* take a ground instance G from I_L, if any *)
    while getInstance(I_L,G)
        (* if G is consistent with the current substitution θ, extend θ and exit *)
        if extend(G,θ) then
            return true;
    return false;
end;
```

Fig. 1. Computing the instantiations of a rule

The algorithm *Instantiate*, shown in Figure 1, generates the possible instantiations for a rule r of a program $\mathcal{P}$. When this procedure is called, for each predicate p occurring in the body of r we are given its extension, as a set I_p containing all its ground instances. We say that the mapping $\theta : var(r) \rightarrow U_{\mathcal{P}}$ is a valid substitution for r if it is valid for every positive literal occurring in its body, i.e., if for every positive literal L in $B(r)$, $\theta L \in I_L$[5] holds. In other words, we discard a priori any substitution mapping a positive body literal Q to a ground instance of Q which is not in I_q. *Instantiate* outputs such valid substitutions for r.

Note that, since the rule is safe, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal. For the sake of presentation, we assume that the body is ordered in a way that any negative literal always follows the positive atoms containing its variables. Actually, **DLV** has

[5] Meaning the extension of the predicate occurring in L.

a specialized module that computes a clever ordering of the body [11] satisfying this assumption.

Instantiate first stores the body literals $L_1, \ldots, L_n$ into an ordered list $B = (null, L_1, \cdots, L_n, last)$. Then, it starts the computation of the substitutions for r. To this end, it maintains a variable θ, initially set to $\emptyset$, representing, at each step, a partial substitution for $var(r)$.

Now, the computation proceeds as follows: For each literal L_i, we denote by $PreviousVars(L_i)$ the set of variables occurring in any literal that precedes L_i in the list B (if $i = 1$, $PreviousVars(L_i) = \emptyset$), and by $FreeVars(L_i)$ the set of variables that occurs for the first time in L_i, i.e., $FreeVars(L_i) = var(L_i) - PreviousVars(L_i)$.

At each iteration of the **while** loop, by using function *Match*, we try to find a match for a literal L_i with respect to θ, in other words, we apply θ to L_i and look for an instantiation of θL_i that matches an atom in I_{L_i}. More precisely, we look in I_{L_i} for a ground instance G which is consistent with the assignments for the variables in $PreviousVars(L_i)$, and then use G in order to extend θ to the variables in $FreeVars(L_i)$; note that, if $FreeVars(L_i) = \emptyset$, this task simply consists in checking whether θ is a valid substitution for L_i. If there is no such a substitution, then we backtrack to the previous literal in the list, or else we consider two cases: if there are further literals to be evaluated, then we continue with the next literal in the list; otherwise, θ encodes a (total) valid substitution and is thus added to the output set S. Even in this case, we backtrack for finding another solution.

4 Indexing Techniques for Rule Instantiation

A critical issue for the efficiency of the **DLV** instantiator is the task accomplished by function *Match* shown in Figure 1. As said in the previous Section, this function takes as input a literal L, its extension I_L and a partial substitution θ and tries to find a ground instance in I_L matching θ. This task, in the absence of techniques for speeding-up the retrieval of candidate instances, may be very expensive. Indeed, the size of I_L can be very large and thus, a simple approach based on linear search trough I_L leads to a drop in performance of the instantiator (also because *Match* is invoked very frequently during the instantiation).

In this Section, we describe two indexing strategies which can be exploited in order to facilitate the retrieval of instances thus allowing for a more efficient matching function.

4.1 First Argument

In the following we describe how the classical first argument indexing schema can be adapted to our context. Such indexing allows for efficiently performing the match of literals L whose first argument is *indexable*; an argument is said to be indexable if it is either a constant or a variable X such that $X \in PreviousVars(L)$ (thus, θ already contains a substitution for X).

Suitable hash structures are used for boosting the retrieval of ground instances according to values of their first term.

For each predicate p, its extension I_p is implemented by means of a list storing, according to lexicographical order, the ground instances of p. We associate to each extension I_p a sparse secondary index implemented by means of a hash map. More in detail, let $C \subseteq U_{\mathcal{P}}$ be the set of all the distinct constants appearing as first argument of some instance in I_p. An *index* is a hash map M_p that associates to each $c \in C$ (the key of the map) a pointer pt to an instance in I_p. In particular, pt

identifies the first ground instance in I_p having c as first argument and thus, due to the lexicographical order of I_p, facilitates the retrieval of all ground instances in I_p with the same characteristic.

By using these structures, the match of a literal L whose first term is instantiated with a constant c can be performed as follows: first of all, we access to the index corresponding to L using c as key. Then, we simply follow the pointer associated to c in order to directly access the instances of I_p having c as first term, and try to extend θ by using, one after the other, such instances. Note that, since the index is implemented by means of a hash map, looking up the pointer pt by its key is efficient. In particular, the average case complexity of this operation is constant time.

Such indexing schema takes advantage from the lexicographical order of the predicate extensions for creating indexes whose size is in general smaller than the corresponding extensions thus limiting the space required to store the index. However, as the following example shows, it is not general, and it allows the use of indexes only in a small range of cases.

Example 1. Consider the following program

$$r_1 : \ a(Z) \leftarrow p(X, 1, Y), q(X, Y, Z).$$
$$r_2 : \ b(T, V) \leftarrow r(U), q(T, U, V).$$

During the instantiation, the first argument indexing schema is exploited only once. Indeed, the evaluation of rule r_1 proceeds by matching first the literal $p(X, 1, Y)$ whose first argument is not indexable, since $X \notin \mathit{PreviousVars}(p(X, 1, Y))$. Then, the literal $q(X, Y, Z)$ is matched and the first argument index can be used, indeed $X \in \mathit{PreviousVars}(q(X, Y, Z))$. For the evaluation of r_2, it is easy to see that indexes can never be exploited.

4.2 On Demand Indexing

The first argument indexing schema described above allows for using sparse indexes with the advantage of limiting the memory consumption. However, it has the disadvantage of being not general and hence exploitable only in few cases.

In the following the on demand indexing strategy is described which allows for the efficient match of literals where a generic argument (not only the first one) is indexable. Such a strategy is more general than the first argument indexing but requires more space for storing indexes, since the lexicographical order of the extensions can not be exploited and, an index must contain a pointer for each instance in the extension.

Let p be a predicate, I_p be the extension of p, a an indexable argument and x_a its position in the parameter list of p. Moreover, let $C \subseteq U_{\mathcal{P}}$ be the set of all the distinct constants appearing in some instance of I_p in position x_a. An index to I_p for the argument a is a hash multi map $MM_{p,a}$ that associates with each $c \in C$ (the key of the map) a set of pointers to I_p, one for each instance having c in position x.

In our on demand indexing strategy, the argument to be indexed is not predetermined but is established during the computation. More in detail, during the evaluation of a rule, when a match of a literal L has to be performed, an argument a is chosen among all the indexable arguments of L and the index for L corresponding to a is created (if it does not exist yet).

Thus, indexes are created only if they are really exploitable and, in two different moments of the evaluation, a predicate can be associated to two different indexes,

depending on the argument that is more appropriate to be indexed. For instance, a predicate can appear in the body of two different rules, and the most convenient index to use could be different in the two cases.

Example 2. Consider the program of example 1. While the first argument indexing schema allows for using of indexes only for the match of literal $q(X, Y, Z)$ in rule r_1, the on demand indexing schema has a better behavior, since indexes can be exploited in three cases. More in detail, the matches of $p(X, 1, Y)$ in rule r_1 and of $q(T, U, V)$ of rule r_2 can be performed by using indexes on the second argument. Moreover, for the match of literal $q(X, Y, Z)$ of rule r_1, two indexes could be associated to q, one for the first argument or one for the second argument. The choice of the more appropriate index to be used can be made according to an heuristic, as described below.

In the case of a literal L having more than one indexable argument, an heuristic is used in order to choose which is the one to be indexed. In this work, we experiment with two heuristics. The first one ($\mathbf{H_1}$) is very simple and consists in the selection of the first indexable argument (in a left to right order). The second one ($\mathbf{H_2}$) allows for a more refined choice and tries to select the indexable argument where it is more likely that few candidate instances will be retrieved.

More in detail, such a choice is made by taking into account the selectivity of each indexable argument a of L, that is the number of distinct constants for a in I_L. The heuristic selects the argument with the greatest selectivity, that is, the one whose selectivity better approximates the size of I_L.

Example 3. In the previous example, we have seen that, for the match of literal $q(X, Y, Z)$ of rule r_1, the on demand indexing technique can choose among two indexes corresponding to two different arguments of q (first and second one). Suppose now that the size of the extensions of p and q are 100 and 600, respectively. Moreover, assume that the instances in I_q are the following:

$$\begin{array}{c} q(a,1,1), q(a,2,2), \ldots q(a,100,100), \\ q(b,1,101), q(b,2,102), \ldots q(b,100,200), \\ \vdots \\ q(f,1,501), q(f,2,502), \ldots q(f,100,600) \end{array}$$

It is easy to see that the selectivities of the first and the second argument of q are 6 and 100, respectively. Given these values, heuristic H_2 chooses the second one as argument to be indexed, thus suggesting a different choice w.r.t H_1 (which selects the first one). Importantly, according to such different choices, the cost of the matching operation of q varies notably. Indeed, using the index on the first argument (as H_1 suggests), we have that, for each instance of p, 100 candidate instances have to be considered for the match of q; thus, to compute the join among p and q, 10000 possible matchings have to be performed. On the contrary, the index on the second argument (as suggested by H_2) identifies, for each instance of p, 6 possible candidate instances for the match of q, thus the join among p and q is computed by considering only 600 possible matchings.

Note that, in order to limit the memory usage, besides avoiding the creation of useless indexes, a structural analysis of the input program is done to identify possible indexes previously created but not exploitable in the next steps of the computation.

More in detail, since the instantiation of the program is performed according to the dependencies among components given by the dependency graph, for the evaluation of each single component, only a subset of the predicates occurring in the program is involved. Thus, if a predicate p is involved in the instantiation of a component C and it is not necessary for the evaluation of the components following C in the topological ordering, the eventual indexes associated to p can be destroyed as soon as C has been processed. Hence, only indexes which can be possibly exploited again during the computation are maintained.

5 Experimental Results

In order to check the impact of the proposed indexing techniques on the **DLV** instantiator, we carried out an experimentation activity on a number of benchmark problems, taken from different domains. For space limitation, we do not include the code of benchmark programs; however they can be retrieved, together with the binaries used for the experiments, from our web page: `http://www.mat.unical.it/indexing/indexing.tar.gzip`. Moreover, we give below a very short description of the problems.

5.1 Benchmark Programs

We have considered several problems whose encodings are significantly hard to instantiate. Some of them are known programs which have been already used in the evaluation of ASP instantiators ([1, 23, 12]), some others are programs arising in practical applications of ASP.

InsuranceWorkflow. The goal is to emulate, by means of an ASP program, the execution of a workflow, in which each step constitutes a transformation to be applied to some data (in order to query for and/or extract implicit knowledge). Two problem instances were provided by the company EXEURA s.r.l. [24], which have been automatically generated by a software working on several American insurance data.

Scheduling. A scheduling problem for determining shift rotation of employees, ensuring appropriate days off for each employee and respecting other given constraints on the availability of some workers.

Cristal. Cristal (Cooperative Repositories & Information System for Tracking Assembly Lifecycle) is a deductive databases application that involves complex knowledge manipulations. The main purpose is to manage the gathering of production data during the ongoing construction of the Electromagnetic Calorimeter of the Compact Muon Solenoid, at the European Centre for Nuclear Research (CERN) [25].

Food. The problem here is to generate plans for repairing faulty workflows. That is, starting from a faulty workflow instance, the goal is to provide a completion of the workflow such that the output of the workflow is correct. Workflows may comprise many activities. Repair actions are compensation, (re)do and replacement of activities.

DocClass. The problem is to assign a document to one or more categories, based on its contents. In particular, the input data represent words or sequence of words appearing in the document (ngrams) and the document is classified according to the presence or the absence of given ngrams. The single problem instance was provided by EXEURA s.r.l. [24].

DataIntegration. A data integration problem. Given some tables containing discording data, find a repair where some key constraints are satisfied. The single problem instance used for these tests was originally defined within the EU project INFOMIX [26].
Hilex. The problem consists in recognizing and extracting meaningful information from unstructured web documents. This is done by combining both syntactic and semantic information, through the use of domain ontologies. A preprocessor transforms the input documents into ASP facts, extraction rules are translated into ASP, and information extraction amounts to reasoning on an ASP program, which is executed by DLV. The single problem instance was provided by the company EXEURA s.r.l. [24].
Timetabling. The problem was considered of determining a timetable for some university lectures that have to be given in a week to some groups of students. The timetable must respect a number of given constraints concerning availability of rooms, teachers, and other issues related to the overall organization of the lectures. The five instances we considered were provided by the University of Calabria; they refer to different numbers of student groups.
3-Colorability. This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors. We considered five instances representing ladder graphs with increasing number of nodes.
Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. The encoding of this problem consists of one exit rule and a recursive one. We considered five different instances which have been used at the First Answer Set Programming System Competition [23].
GrammarBasedIE. This problem has been used at the First Answer Set Programming System Competition [23]. It constitutes a part of a more complex application for recognizing and extracting meaningful information from unstructured Web documents. In particular, given a context free grammar, which specifies arithmetic equations, and a string, the problem is to determine whether the input string is an equation belonging to the language defined by the grammar and whether the equation holds. For the experiments, we used five different instances taken from the web page of the competition.

5.2 Compared Methods

We implemented the indexing strategies described in the previous Section in the **DLV** instantiator and we compared the resulting prototypes by using the above benchmark problems. In particular, the following instantiators were compared:

- **noIndexes**: the **DLV** instantiator without any indexing technique;
- **1stArg**: the **DLV** instantiator enhanced with first argument indexing; [6]
- **onDemand-H$_1$**: the **DLV** instantiator with on demand indexing where the first indexable argument is chosen;
- **onDemand-H$_2$**: the **DLV** instantiator with on demand indexing where the indexable argument with greatest selectivity is chosen.

All binaries were produced by using the GNU compiler GCC 4.1.2, and the experiments were performed on a dual processor Intel Xeon HT (single core) 3.60GHz machine, equipped with 3GB of RAM and running Debian Gnu Linux 2.6.

[6] This version of the **DLV** instantiator coincides with the one of the official release October 11th 2007.

Program	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
InsuranceWorkflow1	21.58	12.96	8.15	0.51	97%
InsuranceWorkflow2	102.47	22.97	10.34	2.61	97%
Scheduling	114.54	114.52	25.65	12.52	89%
Cristal	3.65	0.67	0.45	0.26	93%
Food	135.04	115.43	57.47	49.37	63%
DocClass	–	4.48	2.40	2.42	–
DataIntegration	293.68	293.75	3.71	3.72	99%
Hilex	16.54	8.23	3.66	3.52	79%

Table 1. Results for real problems (times in seconds)

Timetabling	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
17 groups	187.28	187.75	35.94	14.39	92%
19 groups	287.18	288.46	37.19	15.11	95%
21 groups	319.66	318.98	51.51	16.54	95%
23 groups	334.06	334.02	73.85	18.45	94%
25 groups	431.99	409.61	97.14	20.15	95%

Table 2. Results for Timetabling (times in seconds)

5.3 Results and Discussion

Tables 1– 5 shows the results of our experiments. In each table, for each benchmark program P described in column 1, columns 2 –5 report the times employed to instantiate P by using the above binaries; column 6 reports the percentage gain obtained by onDemand-H_2 w.r.t noIndexes. All running times are expressed in seconds. The symbol '−' means that the instantiator did not terminate within 10 minutes.

The results confirm the intuition that the indexing techniques can be very useful for improving the efficiency of the **DLV** instantiator. Indeed, it is clear from the tables that, even a simple strategy, like the one exploited by 1^{st}Arg, allows for outperforming the instantiator noIndexes in many cases. In particular, the first argument indexing gives very relevant improvements in some benchmarks, as, for instance, DocClass and three of the instances of 3-Colorability, where noIndexes does not terminate within ten minutes, while 1^{st}Arg takes few seconds. However, there are also some cases in which the speed-up introduced by 1^{st}Arg is not so considerable, or it is not present at all. Consider for instance, Scheduling and Data Integration and the instances of Timetabling where noIndexes and 1^{st}Arg perform very similarly, or the two solved instances of Reachability where the gain is about 7%, and Food where the gain is about 15%. The reason of such different behavior of 1^{st}Arg on these problems is that the speed-up reflects how intensively first argument indexing can be used for a given benchmark. More precisely, the performance gain is low when the encodings are such that first argument indexing is exploitable only for the match of few literals.

The situation changes when looking at the results of the on demand indexing technique. Indeed, for all the tested benchmarks, the speed-up introduced by this technique is really impressive. Moreover, it is clear from the tables that the instantiator exploiting H_2 as heuristic behaves quite better than the one exploiting H_1. Indeed, OnDemand-H_1 allows for notable improvements in many cases with difference of even 2 orders of magnitude w.r.t. noIndexes (as, for instance, DataIntegration, Reachability). Moreover, it is able to solve all the problem instances within

3-Colorability	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
15000 nodes	79.13	0.98	0.97	0.97	98%
20120 nodes	222.84	0.98	0.98	0.99	99%
32300 nodes	–	3.29	3.24	3.22	–
39900 nodes	–	2.13	2.11	2.12	–
40120 nodes	–	2.11	2.08	2.08	–

Table 3. Results for 3-Colorability (times in seconds)

Reachability	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
Reachability_13	71.83	66.39	0.95	0.98	99%
Reachability_14	367.60	339.63	2.33	2.31	99%
Reachability_15	–	–	5.47	5.49	–
Reachability_16	–	–	12.61	12.58	–
Reachability_18	–	–	68.05	68.68	–

Table 4. Results for Reachability (times in seconds)

the allowed time limit. However, these considerations hold also for OnDemand-H_2; indeed, either it exhibits the same behavior of OnDemand-H_1 or perform better, as, for instance, in Timetabling, InsuranceWorkflow and other real problems. This shows that performance improvements strongly depend on the "quality" of the used index. Intuitively, when an entry in the index corresponds to a high number of candidate instances (close to the size of the extension), then indexing may not bring great benefits.

Summarizing, the tested indexing techniques allow improving the performance of the instantiator but the on demand strategy is applicable in a wider range of cases w.r.t the first argument one and gives relevant speed-ups especially when combined with an accurate choice of the argument to be indexed.

6 Conclusions and Ongoing Work

In this work we investigated the use of indexes for optimizing the rule instantiation process of **DLV**. In particular, we experimented with a classical first argument indexing schema adapted to our context and proposed an on demand indexing strategy where indexes are computed during the evaluation and the argument to be indexed is chosen according to a heuristic.

We implemented such strategies in the instantiator of **DLV** and we carried out a deep experimental analysis. The results of the experiments are very positive and confirm that the use of indexes causes the instantiation stage to achieve noticeable improvements. Moreover, the on demand indexing schema gives better results w.r.t the classical first argument schema in a wider range of cases and performance improve notably when a good choice of the argument to be indexed is made.

Currently, we are investigating the relations between the body ordering criterion exploited in **DLV** and the use of indexes. Indeed, it is easy to see that, the body ordering may have a strong impact on the use of indexes; each of the techniques described above chooses for a literal L the argument to be indexed among a set of indexable arguments, and such set depends on the position that L has in the body and, thus, on the ordering algorithm used. Hence, a different ordering criterion may

GrammarBased_IE	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
GrammarBased_IE_1	4.49	2.98	0.98	0.98	78%
GrammarBased_IE_2	7.57	3.41	0.95	0.96	87%
GrammarBased_IE_3	7.89	4.09	1.16	1.15	85%
GrammarBased_IE_4	8.84	6.22	1.74	1.74	80%
GrammarBased_IE_5	9.50	7.26	2.15	2.14	77%

Table 5. Results for GrammarBased_IE (times in seconds)

lead to make different choices and, so, to considerably influence the execution times; then, the ordering criterion could be modified in order to take into account indexes availability. However, it is well known that the instantiation time of a rule strongly depends on the order of evaluation of literals [11, 27], thus a naive ordering could have a negative impact on the instantiator performance, also overshadowing the gains brought by the indexes usage. Therefore, a clever ordering has to be conceived which allows a better use of indexes but without ignoring the principles which the current method is based on and whose effectiveness has already been assessed [11]. The design of the new ordering criterion is the subject of a future work.

References

1. 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
2. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM Transactions on Computational Logic **7**(1) (2006) 1–37
3. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings. Volume 3662 of Lecture Notes in Computer Science., Springer Verlag (2005) 447–451
4. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Morgan Kaufmann Publishers (2007) 386–392
6. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1–2) (2004) 115–137
7. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). Volume 2923 of LNAI., Springer (2004) 346–350
8. Anger, C., Konczak, K., Linke, T.: `NoMoRe`: A System for Non-Monotonic Reasoning. In Eiter, T., Faber, W., Truszczyński, M., eds.: Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings. Volume 2173 of Lecture Notes in AI (LNAI)., Springer Verlag (2001) 406–410
9. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. In Sutcliffe, G., Voronkov, A., eds.: Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005. Volume 3835 of Lecture Notes in Computer Science., Springer Verlag (2005) 95–109

10. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, ed.: Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99), Prolog Association of Japan (1999) 135–139
11. Leone, N., Perri, S., Scarcello, F.: Improving ASP Instantiators by Join-Ordering Methods. In Eiter, T., Faber, W., Truszczyński, M., eds.: Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria. Volume 2173 of Lecture Notes in AI (LNAI)., Springer Verlag (2001) 280–294
12. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Annals of Mathematics and Artificial Intelligence **51**(2–4) (2007) 195–228
13. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)
14. Demoen, B., Mariën, A., Callebaut, A.: Indexing in Prolog. In: Proceedings of the North American Conference on Logic Programming, MIT Press (1989) 1001–1012
15. Carlsson, M.: Freeze, Indexing, and other implementation issues in the WAM. In: Proceedings of the Fourth International Conference on Logic Programming, MIT Press (1987) 40–58
16. Rao, P., Sagonas, K.F., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97). Volume 1265 of Lecture Notes in AI (LNAI)., Dagstuhl, Germany, Springer Verlag (1997) 2–17
17. Wielemaker, J.: SWI-Prolog 5.1: Reference Manual, University of Amsterdam (1997-2003)
18. Costa, V.S., Sagonas, K.F., Lopes, R.: Demand-driven indexing of prolog clauses. In: Proceedings of the 23rd International Conference on Logic Programming (ICLP-2007). Volume 4670 of LNCS. (2007) 395–409
19. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In Alferes, J.J., Leite, J., eds.: Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004). Volume 3229 of Lecture Notes in AI (LNAI)., Springer Verlag (2004) 200–212
20. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
21. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. New Generation Computing **9** (1991) 401–424
22. Faber, W., Leone, N., Perri, S., Pfeifer, G.: Efficient Instantiation of Disjunctive Databases. Technical Report DBAI-TR-2001-44, Institut für Informationssysteme, Technische Universität Wien, Austria (2001) Online at `http://www.dbai.tuwien.ac.at/local/reports/dbai-tr-2001-44.pdf`.
23. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In Baral, C., Brewka, G., Schlipf, J., eds.: Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07. Volume 4483 of Lecture Notes in Computer Science., Tempe, Arizona, Springer Verlag (2007) 3–17
24. (Exeura s.r.l., homepage) `http://www.exeura.it/`.
25. (`CRISTAL` project homepage) `http://proj-cristal.web.cern.ch/`.
26. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
27. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database System Implementation. Prentice Hall (2000)

Integrating Grounding in the Search Process for Answer Set Computing

Claire Lefèvre and Pascal Nicolas

LERIA – University of Angers – France
claire.lefevre@univ-angers.fr -- pascal.nicolas@univ-angers.fr
2, bd Lavoisier – F-49045 Angers Cedex 01

Abstract. Answer Set Programming (ASP) is a very convenient paradigm to represent knowledge in Artificial Intelligence and to encode Constraint Satisfaction Problems. For that, the natural way to use ASP is to elaborate a first order logic program with default negation encoding the problem to solve. In a preliminary step this program is translated in an equivalent propositional one by a first tool: the grounder. Then, the propositional program is given to a second tool: the solver. This last one computes (if they exist) one or many answer sets (models) of the program, each answer set encoding one solution of the initial problem. Today, we can say that almost all ASP solvers follow this approach of two steps computation.
In this work, we begin by putting in evidence that sometimes the preliminary grounding phase is the only bottleneck for the answer set computation. We show that a lot of useless and counterintuitive work is done in some situations. But, our major contribution is to introduce a new approach of answer set computing that escapes the preliminary phase of rule instantiation by integrating it in the search process. Furthermore, we describe the main lines of the first implementation of our new ASP solver `ASPeRiX` developed following the introduced methodology.

1 Introduction

Answer Set Programming (ASP) is a very convenient paradigm to represent knowledge in Artificial Intelligence (AI) and to encode Constraint Satisfaction Problems (CSP). It has its roots in non monotonic reasoning and logic programming and has led to a lot of works since the seminal paper [10]. But, beyond its ability to formalize various problems from AI or CSP, ASP is also became a very interesting way to practically solve them since some efficient solvers are available. In few words, if someone wants to use ASP to solve an issue, and whatever is the variant of ASP that he uses, he has to write a logic program in a purely declarative manner in such a way that the stable models (also called answer sets) of the program represent the solutions of his original problem (see [3, 18] for more details). Usually, the program contains different kind of rules. The simplest ones are facts as $bird(tweety) \leftarrow .$, $edge(4, 10) \leftarrow .$, representing data of the particular problem. Some ones are about background knowledge as $path(X, Y) \leftarrow edge(X, Z), path(Z, Y).$ expressing a well-know property

about path in a graph for instance. Some others can be non monotonic, as $fly(X) \leftarrow bird(X), not\ penguin(X).$, for reasoning with incomplete knowledge. In other cases, especially for CSP, default negation is also used to encode alternative potential solutions of a problem as $red(X) \leftarrow v(X), not\ blue(X).$ and $blue(X) \leftarrow v(X), not\ red(X).$, expressing the two exclusive possibilities to color a vertex in a graph. Last but not least, special headless rules are used to represent constraints of the problem to solve as $\leftarrow edge(X,Y),\ red(X), red(Y).$ in order to not color with red two linked vertices.

Depending which solver is used to compute the answer sets of the program, one can also use some particular atoms for (in)equalities and simple arithmetic calculus. Other constructions using aggregates functions, cardinality constraints, weights, ... are also possible but they are out of the scope of this work in which we restrict our attention to original stable model semantics [10]. In fact, with these previous few examples we want to point out that knowledge representation in ASP is done by means of first order rules using variables. From a theoretical point of view, the models of such a first order program are those of its ground instantiation with respect to its Herbrand universe. Let us note that there exist some more recent works as [6, 14] dealing directly with first order normal logic programs without instantiating them. In these works, first order semantics are defined by means of second order logic or circumscription. But, from a practical point of view, every available ASP solver begins its work by an instantiation phase in order to obtain a propositional program. After this first phase, called *grounding*, the solver starts the real phase of answer set computation by dealing with a finite, but sometimes huge, propositional program.

The aim of our present work is to propose a new approach of answer set computation that escapes this preliminary instantiation phase by integrating it in the search process. In section 2 we examine the grounding process realized by the up-to-date ASP systems and illustrate some drawbacks of this phase. In section 3 we present our approach of answer set computation that escapes the preliminary grounding of rules and that is fully rule-oriented. In section 4 we present the first version of our new system `ASPeRiX` that implements the principles introduced in this paper. We conclude in section 5 by citing some improvements that we plan to incorporate in our system in the future.

2 Grounding in ASP

A *normal logic program* is a finite set of rules like

$$c \leftarrow a_1,\ \ldots,\ a_n,\ not\ b_1,\ \ldots,\ not\ b_m. \quad n \geq 0, m \geq 0 \tag{1}$$

where $c, a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. For a rule r (or by extension for a rule set), we note $head(r) = c$ its head, $body^+(r) = \{a_1, \ldots, a_n\}$ its positive body, $body^-(r) = \{b_1, \ldots, b_m\}$ its negative body and $body(r) = body^+(r) \cup body^-(r)$. When the negative body of a rule is not empty we say that this rule is *non-monotonic*. The Gelfond-Lifschitz reduct of a program P by an atom set X is the program $P^X = \{head(r) \leftarrow body^+(r). \mid body^-(r) \cap X = \emptyset\}$. Since it has no

default negation, such a program is definite and then it has a unique minimal Herbrand model denoted with $Cn(P)$. By definition, an *answer set* (originally called a *stable model* [10]) of P is an atom set S such that $S = Cn(P^S)$. For instance the program $\{a \leftarrow not\ b.,\ b \leftarrow not\ a.\}$ has two answer sets $\{a\}$ and $\{b\}$. Special headless rules, called *constraints*, are admitted and considered equivalent as rules like $bug \leftarrow \ldots,\ not\ bug.$ where bug is a new symbol appearing nowhere else. For instance, the program $\{a \leftarrow not\ b.,\ b \leftarrow not\ a., \leftarrow a.\}$ has one, and only one, answer set $\{b\}$.

As presented in our introduction, in many cases a problem is encoded in ASP with a logic program P containing rules with variables that we call *first order rules*. More formally, these rules are of type (1) where a_i's and b_j's are atoms, like $p(X, 3, Y)$, built from an n-ary predicate, constants and variables (no function symbol occurs in the programs considered in this work). Since answer set definition is given for propositional programs, P has to be seen as an intensional version of the propositional program $ground(P)$ defined as follows. Given a rule r, $ground(r)$ is the set of all fully instantiated rules that can be obtained by substituting every variable in r by every constant of the Herbrand universe of P and then, $ground(P) = \bigcup_{r \in P} ground(r)$.

Example 1. The program P_1 is a shorthand for the program $ground(P_1)$.

$$P_1 = \left\{ \begin{array}{l} n(1) \leftarrow .,\ n(2) \leftarrow ., \\ a(X) \leftarrow n(X),\ not\ b(X)., \\ b(X) \leftarrow n(X),\ not\ a(X). \end{array} \right\} \quad ground(P_1) = \left\{ \begin{array}{l} n(1) \leftarrow .,\ n(2) \leftarrow ., \\ a(1) \leftarrow n(1),\ not\ b(1)., \\ b(1) \leftarrow n(1),\ not\ a(1)., \\ a(2) \leftarrow n(2),\ not\ b(2)., \\ b(2) \leftarrow n(2),\ not\ a(2). \end{array} \right\}$$

$ground(P_1)$ has four answer sets $\{a(1), a(2), n(1), n(2)\}$, $\{a(1), b(2), n(1), n(2)\}$, $\{a(2), b(1), n(1), n(2)\}$ and $\{b(1), b(2), n(1), n(2)\}$ that are considered as the answer sets of P_1.

From a practical point of view, all systems available today to compute answer sets of a program follow the architecture described in figure 1. For the grounder box we can cite *Lparse* [21] and *Gringo* [9], and for the solver box *Clasp* [7] and *Smodels* [20]. A particular family of solvers are *Assat* [13], *Cmodels* [11] and *Pbmodels* [16], since they transform the answer set computation problem into a (pseudo) boolean model computation problem and use a (pseudo) SAT solver as an internal black box. Last, but not least, the system *DLV* [12], symbolized in figure 1 by the dash-line rectangle, incorporates the grounder as an internal function. Furthermore, let us mention that [5] describes an improvement of the grounding process of *DLV* by means of parallelism[1].

The main goal of each grounding system is to generate all propositional rules that can be relevant for a solver and only these ones, while preserving answer sets of the original program. But, whatever the methodology is, the grounding phase is firstly and fully processed before computing the answer sets. In the sequel, we

[1] In our study, we have chosen to use ASP systems having obtained the best results during the 2007 ASP competition [8] and not using an underlying SAT solver.

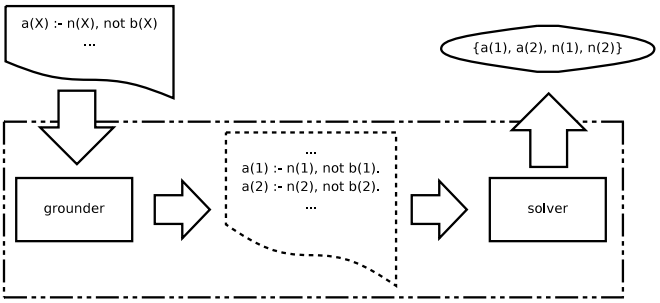


Fig. 1. Architecture of answer set computation

stress the difficulties due to this grounding process done independently of the answer set computation.

For us, the main drawback of the preliminary grounding phase is that it leads to a lot of useless and counterintuitive work in some situations that we illustrate in the following examples[2].

Example 2. From the program

$$P_2 = \left\{ \begin{array}{ll} p(1) \leftarrow ., p(2) \leftarrow ., \ldots, & p(N) \leftarrow ., \\ aa \leftarrow not\ bb., & a(X,Y) \leftarrow pa(X),\ pa(Y),\ not\ b(X,Y)., \\ bb \leftarrow not\ aa., & b(X,Y) \leftarrow pb(X),\ pb(Y),\ not\ c(X,Y)., \\ pa(X) \leftarrow aa,\ p(X)., & c(X,Y) \leftarrow a(X,Y),\ X < Y., \\ pb(X) \leftarrow bb,\ p(X)., & \leftarrow aa. \end{array} \right\}$$

DLV, *Gringo* and *Lparse* generate roughly $2.5 \times N^2$ rules.

Because of the constraint $\leftarrow aa.$ (that eliminates from the possible solutions every atom set containing aa), it is easy to see that all N rules with a positive body containing aa, like $pa(1) \leftarrow aa, p(1).$, ... are useless since they can never contribute to generate an answer set of P_2. And then, the N^2 rules with $pa(X)$ in their positive body are useless too. In defense of the actual grounders, their inability to eliminate these particular rules is not surprising since the reason justifying this elimination is the consequence of a reasoning taking into account the stable model semantics. If we refer to the figure 1 it is clear that this task is relevant to the solver box and not to the grounder box. Thus, if we want to limit as much as possible the number of rules and atoms to deal with, we have not to separate grounding and answer set computing.

Example 3. Let P_3 be the program encoding a 3-coloring problem (as given in [18]) on a N vertices graph organized as a bicycle wheel (see below). v stands for *vertex*, e for *edge*, c for *color*, *col* for *colored by*, *ncol* for *not colored by*.

[2] Sometimes, it is possible that a program given in example has to be slightly modified to respect the particular syntax of the targeted ASP solver.

$$P_3 = \left\{ \begin{array}{l} v(1) \leftarrow ., \ldots, v(N) \leftarrow ., \quad c(1) \leftarrow ., \ c(2) \leftarrow ., \ c(3) \leftarrow ., \\ e(1,2) \leftarrow ., \ldots, e(1,N) \leftarrow ., e(2,3) \leftarrow ., \ldots, e(N,2) \leftarrow ., \\ col(V,C) \leftarrow v(V), \ c(C), \ not \ ncol(V,C)., \\ ncol(V,C) \leftarrow col(V,D), \ c(C), \ C \neq D., \\ \leftarrow e(V,U), \ col(V,C), \ col(U,C). \end{array} \right\}$$ 

.

From P_3 *DLV*, *Gringo* and *Lparse* generate about $18N$ rules. If N is even then P_3 has no answer set and if N is odd then it has 6 answer sets.

Suppose that P_3 has an answer set in which there is $col(1,1)$. Obviously, all the $N-1$ constraints like $\leftarrow e(1,U), \ col(1,1), \ col(U,1)$. $\forall U \in \{2,\ldots,N\}$ are necessary because they have to be checked. But, all the other constraints like

$$\leftarrow e(1,U), \ col(1,2), \ col(U,2)., \text{ and } \leftarrow e(1,U), \ col(1,3), \ col(U,3).$$

$\forall U \in \{2,\ldots,N\}$ can be considered as useless since vertex 1 is not colored by 2 or 3. However, all these $2N-2$ constraints have been generated. So, the time consumed by this task is clearly a lost time and the memory space used by these data could have been saved. Thus, if we are seeking only one answer set, a lot of work has been done for nothing since the grounded program contains in extension informations needed for computing all solutions when only one is searched.

Beyond these particular examples, what we want to stress is that grounders generate in extension all the search space (for all potential solutions) that they give then to the solver. But, this approach is clearly not this of usual search algorithms. A classical coloring algorithm does not firstly enumerate, in extension, all possible colorations for every vertex in the graph. A CSP solver makes choices by instantiating some variables, propagates the consequences of these choices, checks the constraints and by backtracking explores its search space. But, it does not build, a priori and explicitly, all the possible tuples of variables and constraints representing the problem to solve. That is why we think that if we want to use ASP to solve very large problems we have to realize the grounding process during the search process and not before it.

Of course, a lot of work has been done about propagation and search space pruning in case of propositional programs. These tasks are well studied and efficiently implemented in the propositional case, but techniques used can not always be easily adapted for the non ground case. One reason is that the Herbrand base is not known in advance. On the other hand, this is precisely this enumeration of atoms and rules that can be useless, or even impossible, in some cases. That is the reason why we think it is relevant to explore a new way by integrating grounding in the search process, even if a lot of work remains to find tools as powerful as those of current systems.

3 A first order rule-based approach

We first present the characterization of answer sets for normal logic programs based on an abstract notion of *computation* that is proposed in [17]. In this work as in ours, a computation is a sequence of atom sets starting with the empty set.

At each step, the heads of some applicable rules[3] w.r.t. actual state are added. When no more atom can be added, one must check that the rules that have been fired are still applicable.

Definition 1. *(from [17]) Let P be a normal logic program. A* computation *for P is a sequence $\langle X_i \rangle_{i=0}^{\infty}$ of atom sets that satisfies the following conditions :*

- $X_0 = \emptyset$
- *(Revision)* $\forall i \geq 1, X_i \subseteq T_P(X_{i-1})$
- *(Persistence)* $\forall i \geq 1, X_{i-1} \subseteq X_i$
- *(Convergence)* $X_\infty = \bigcup_{i=0}^{\infty} X_i = T_P(X_\infty)$
- *(Persistence of reasons)* $\forall i \geq 1, \forall a \in X_i \backslash X_{i-1}, \exists r_a \in P$ *s.t.* $head(r_a) = a$, *and* $\forall j \geq i-1, body^+(r_a) \subseteq X_j, body^-(r_a) \cap X_j = \emptyset$

where $T_P(X) = \{a \in X \mid \exists r \in P, head(r) = a, body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}$

Theorem 1. *(from [17]) Let P be a normal logic program and X be an atom set. Then, X is an answer set of P iff there is a computation $\langle X_i \rangle_{i=0}^{\infty}$ for P such that $X_\infty = X$.*

Our approach of answer set computation follows a forward chaining that instantiates and applies one rule at each step. All along this search we deal with IN, a set of propositional atoms occurring in the seeking solution, and OUT, a set of propositional atoms not occurring in this solution. A *partial interpretation* for a program P is a pair $\langle IN, OUT \rangle$ of disjoint atom sets included in the Herbrand base of P. It defines different status for rules.

Definition 2. *Let r be a propositional rule and $I = \langle IN, OUT \rangle$ a partial interpretation. We say*

- *r is* supported *w.r.t. I when $body^+(r) \subseteq IN$,*
- *r is* unsupported *w.r.t. I when $body^+(r) \cap OUT \neq \emptyset$*
- *r is* blocked *w.r.t. I when $body^-(r) \cap IN \neq \emptyset$,*
- *r is* unblocked *w.r.t. I when $body^-(r) \subseteq OUT$,*

Let the reader note that unblocked is different from not blocked*, the negation of blocked.*

For this formal presentation every constraint (ie : headless rule) is considered given with the particular head $\bot$. The figure 2 illustrates our search procedure that starts with $IN = \emptyset$ and $OUT = \{\bot\}$ and alternates two major steps:

- a propagation step that applies the largest possible number of rules instances, and adds atoms in IN,
- a choice point that applies a non monotonic rule instance and adds atoms in IN and OUT, or does not apply it and adds a new constraint in P recording that this rule has to be blocked.

[3] An applicable rule is a supported and not blocked rule, see definition 2 below.

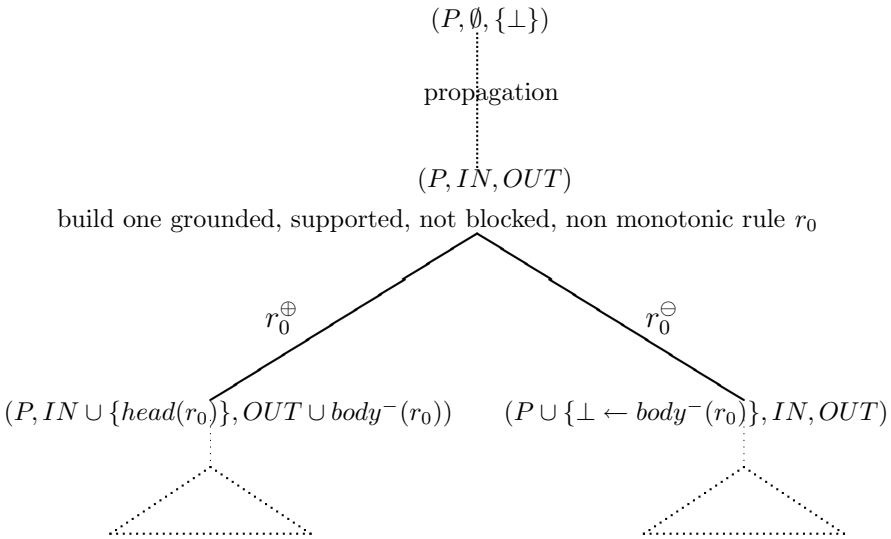


Fig. 2. Overview of the search procedure.

Note that each of these steps works with first order rules and builds ground instances on the fly by means of the two functions defined below.

Definition 3. *Let P be a set of first order rules, $\langle IN, OUT\rangle$ be a partial interpretation and R be a set of grounded (propositional) rules.*

- *γ_{pro} is a non deterministic function selecting one unique supported and unblocked rule $\gamma_{pro}(P, IN, OUT, R)$ in $ground(P) \setminus R$, or returns $false$ if no such a rule exists.*
- *γ_{sel} is a non deterministic function selecting one unique supported and not blocked rule $\gamma_{sel}(P, IN, OUT, R)$ in $ground(P) \setminus R$, or returns $false$ if no such a rule exists.*

To avoid any confusion, we insist on the fact that the set $ground(P)$, mentioned in the above definition, is not explicitly given. It is in accordance with the principal aim of our work that is to avoid its extensive construction. The two functions γ_{pro} and γ_{sel} unify first order rules of P with propositional atoms occurring in IN and OUT in order to return a new (not already occurring in R) fully grounded, supported and unblocked (or not blocked) rule.

The general principle of our new approach of answer set computing for a program P is given in algorithm 1 that must be called by $solve(P_R,\ P_K,\ \emptyset,\ \{\bot\},\ \emptyset)$, knowing that $P_K = \{r \in P \mid head(r) = \bot\}$ (the constraint set) and $P_R = P \setminus P_K$. Finally, let us note CR (for chosen rules) a set of grounded rules labelled with $\oplus$ or $\ominus$. Intuitively, a label $\oplus$ means that the rule instance has been applied and a label $\ominus$ means that it must be blocked (see figure 2). Our algorithm describes the computation of one answer set (or no one if the program is inconsistent) and

Algorithm 1: Algorithm for a first order rule-based answer set computing.

```
Function Solve(P_R, P_K, IN, OUT, CR);
repeat //Propagation phase
    r_0 ← γ_pro(P_R ∪ P_K, IN, OUT, CR);
    if r_0 then
        IN ← IN ∪ {head(r_0)};
        CR ← CR ∪ {r_0^⊕};
until ¬r_0;
if IN ∩ OUT ≠ ∅ then //Contradiction detected
    return false;
else
    r_0 ← γ_sel(P_R, IN, OUT, CR);
    if ¬r_0 then
        if γ_sel(P_K, IN, OUT, ∅) then //Constraint non satisfied
            return false;
        else//an answer set is found
            return IN;
    else//choice point
        stop ← solve(P_R, P_K, IN ∪ {head(r_0)}, OUT ∪ body⁻(r_0), CR ∪ {r_0^⊕});
        if ¬stop then
            stop ← solve(P_R, P_K ∪ {⊥ ← body⁻(r_0)}, IN, OUT, CR ∪ {r_0^⊖});
        return stop ;
```

can be easily extended to the computation of an arbitrary number (or all) of answer sets of P.

Clearly, our approach can be viewed as a particular class of computations (definition 1), obtained by restricting the principle of revision that originally enables to fire any subset of the supported and not blocked rules at each step. In our algorithm, revision alternates two steps: the propagation step that applies successively all supported and unblocked rules instances, and the choice point that selects only one supported and not blocked rule to be fired. Persistence of reasons is ensured first by adding to OUT ground atoms from the negative body of the rule choosen to be applied and, second, by stopping computation when $IN \cap OUT \neq \emptyset$. Thus, following the general framework introduced in [17] we are able to characterize our algorithm as an ***ASPeRiX** computation*[4] that we define as a sequence of partial interpretations instead of simple atom sets. If $I_1 = \langle A_1, B_1 \rangle$ and $I_2 = \langle A_2, B_2 \rangle$ are partial interpretations, $I_1 \subseteq I_2$ iff $A_1 \subseteq A_2 \wedge B_1 \subseteq B_2$.

Definition 4. *Let P be a first order normal logic program. An* ASPeRiX *computation for P is a sequence $\langle X_i \rangle_{i=0}^{\infty}$ of partial interpretations $X_i = \langle IN_i, OUT_i \rangle$ that satisfies the following conditions :*

- $X_0 = \langle \emptyset, \{\bot\} \rangle$,

[4] ASPeRiX is the name of the solver that we have developed following our algorithm 1 (see section 4).

- *(Revision)* $\forall i \geq 1, X_i = \langle IN_{i-1} \cup \{head(r_i)\}, OUT_{i-1} \rangle$
  for some rule $r_i = \gamma_{pro}(P, IN_{i-1}, OUT_{i-1}, \bigcup_{k=1}^{i-1}\{r_k\})$ *if it exists*
  else, $X_i = \langle IN_{i-1} \cup \{head(r_i)\}, OUT_{i-1} \cup body^-(r_i) \rangle$
  for some rule $r_i = \gamma_{sel}(P,\ IN_{i-1},\ OUT_{i-1},\ \bigcup_{k=1}^{i-1}\{r_k\})$ *if it exists*
  else, $X_i = X_{i-1}$,
- *(Persistence)* $\forall i \geq 1, X_{i-1} \subseteq X_i$,
- *(Convergence)* $IN_\infty = \bigcup_{i=0}^{\infty} IN_i = T_P(IN_\infty)$,
- *(Persistence of reasons)* $\forall i \geq 1, \forall a \in IN_i \backslash IN_{i-1}, \exists r_a \in ground(P)$ *s.t.* $head(r_a) = a$, *and* $\forall j \geq i-1, body^+(r_a) \subseteq IN_j, body^-(r_a) \cap IN_j = \emptyset$.

Theorem 2. *Let P be a normal logic program and A be an atom set. Then, A is an answer set of P iff there is an* `ASPeRiX` *computation $\langle X_i \rangle_{i=0}^{\infty}$ for P such that $IN_\infty = A$.*

Proof. (sketch) By restricting our attention to the sequence $\langle IN_i \rangle_{i=0}^{\infty}$, it is easy to see that an `ASPeRiX` computation is a computation and thus, by theorem 1, converges to an answer set. For the other direction, every answer set can be mapped into a computation by theorem 1. On its turn, this computation can be mapped into an `ASPeRiX` computation because persistence of reasons allows us to build a set $\{r_a \mid a \in X_\infty\}$ that can be ordered in such a way that it corresponds to the successive application of rules in an `ASPeRiX` computation.

4 A new ASP solver : `ASPeRiX`

Following our algorithm 1, we have implemented in C++ (under GPL) a new solver called `ASPeRiX` that represents a new approach of answer set computation since, up to our knowledge, all solvers are atom oriented (the choice points are about incorporating or not an atom in a potential answer set) or use an underlying SAT based solver. Apart these two categories we can cite *NoMore* [2, 15] (written in Prolog) that follows a rule based approach, as we do, and that is the closest system to ours. But, in fact *NoMore* solves a coloration problem on the rule graph of a propositional program. Thus, it needs a preliminary grounding step, when `ASPeRiX` does not. *NoMore++* [1], the *NoMore*'s successor, is developed in C++ and deals with atoms and rules conjointly but, again, a preliminary grounding is required. *GASP* [19] (a very recent work elaborated in parallel to ours) is an implementation in Prolog and Constraint Logic Programming of the notion of computation (see definition 1) that realizes the grounding during the search. *GASP*'s propagation is formally presented as a computation of well founded consequences. But, since this seems to be time consuming the implementation of *GASP* uses a variant of the propagation operator T_P that seems to be close to ours. A difference with `ASPeRiX` is about the search tree. In *GASP*, this is an n-ary tree of applicable rules, when in `ASPeRiX` we use a binary tree to apply, or not apply, one rule at a time. *GASP*'s strategy implies to add an ordering process of applicable rules to avoid producing the same answer set many times. This additional task is not necessary for `ASPeRiX` since the unique

computation of every answer set is inherent to our strategy that records the non application of a rule by introducing a constraint (see figure 2).

A deep description of our system is out of the scope of this paper. Nevertheless, we mention some of its salient features. We use dependencies between predicate symbols (predicates for short) in order to help functions γ_{pro} and γ_{sel} to build suitable rule instances. If a is an atom, we note $pred(a)$ the predicate of a, the notation is extended to atom set as usual. The *dependency graph* of a program P is a graph whose nodes are predicates occurring in P and whose arcs are $\{(p,q) \mid \exists r \in P,\ p = pred(head(r)),\ q \in pred(body(r))\}$. We say that a predicate p *depends on* q if there is a path from p to q in the dependency graph. Predicates are grouped according to maximal strongly connected components (scc for short) of the graph. Components are themselves ordered as $\langle C_1, ..., C_n \rangle$ such that if $i < j$ then no predicate in C_i depends on some predicate in C_j. In the following we speak about *rules from a scc* C for the rules whose head predicate is in C. Furthermore, we say that a predicate p is *solved* if each ground instance of rules whose head predicate is p is in CR (the set of rules selected by γ_{pro} and γ_{sel}), or is blocked, or is unsupported. Intuitively, this means that all rules concluding p are exhausted or, in other words, that the extension of the predicate p (i.e., the set of all atoms whose predicate is p belonging to IN) is entirely known. In this case, for every propositional atom a such that $pred(a) = p$, $a \notin IN \Rightarrow a \in OUT$. But, it is useless to build explicitly all such atoms and add them in OUT. Rather, we modify the definition of an unblocked rule r by :

$$\forall a \in body^-(r),\ pred(a) \text{ is solved } \wedge\, a \notin IN, \text{ otherwise } a \in OUT.$$

In `ASPeRiX`, the selection function γ_{sel} generates a rule instance issued from a first order rule belonging to the *current* scc, knowing that the first current scc is C_1. When no more such a rule exists in the current scc, all predicates from it are solved and the next scc in the order defined above becomes the current one. Note that if all predicates appearing in the negative body of a rule r are solved, to be unblocked becomes the same than to be not blocked for instances of r, and then such a rule is completely processed by γ_{pro} and can not lead to choice point. By this way, only truly non monotonic first order rules are examined by γ_{sel} and, thus, can generate choice points. In particular, if P is stratified, the first propagation phase is enough to find the only answer set if it exists.

The rest of the section gives some measures of performances of the first version of our system: `ASPeRiX 0.1`[5]. In no case it is a deep evaluation, but the results reported below illustrate that our approach reaches our introductory goals and that the methodology is promising. Since we want to compare our system to others by taking into account the two steps of ASP computation (grounding and solving), we have formed the following couples of systems : *Gringo 1.0.0+Clasp 1.0.5*, *Lparse 1.1.1+Smodels 2.32* and *Dlv Oct 11 2007*. We did not include the system *GASP* [19] in our comparisons because for program $P_{locstrat}$ (see example 4) it returns two times each stable model and for P_2 (see example 6) the consumed time is prohibited even when the size of the problem is very

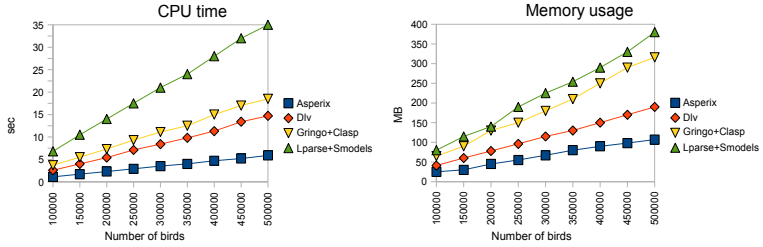
[5] Available at `http://www.info.univ-angers.fr/pub/claire/asperix` with all examples used for tests.

small. All the systems have been run with their default options on an Intel Core Duo T2400, 1.83 Ghz, with 2GB of RAM under Linux Ubuntu 7.10. For every example, when we report the used memory, it is the sum of maximum amount of memory used by the grounder and the parser, since these two processes run simultaneously. For CPU time, it is the same, we cumulate durations spent by grounder and solver[6].

Example 4. Program P_{birds} encodes a taxonomy about flying (f) and non flying (nf) birds (b) such penguins (p), super penguins (sp) and ostriches (o).

$$P_{birds} = \left\{ \begin{array}{lll} p(X) \leftarrow sp(X)., & b(X) \leftarrow p(X)., & b(X) \leftarrow o(X)., \\ f(X) \leftarrow b(X),\ not\ p(X),\ not\ o(X)., & f(X) \leftarrow sp(X)., & \\ nf(X) \leftarrow p(X),\ not\ sp(X)., & nf(X) \leftarrow o(X). & \end{array} \right\}$$

We add to this program the atoms encoding N birds with 10% of ostriches, 20% of penguins whose half of them are super penguins. The CPU time and the memory usage needed to compute the unique answer set of P_{birds} are summarized in the pictures below.



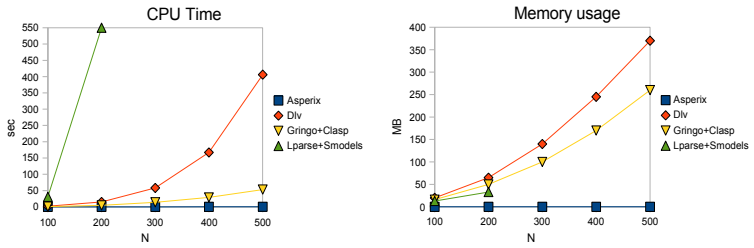
To complete these results, we mention that for P_{birds} 70 % of time is spent by the grounder and 30 % is spent by the solver for the 3 couples of systems (*Gringo 1.0.0+Clasp 1.0.5*, *Lparse 1.1.1+Smodels 2.32* and *Dlv Oct 11 2007*). We can remark that in a certain way this 30 % of time is useless since each grounder provides the solver with a set of rules with empty bodies (ie. facts) because of the nature of this example. For this kind of example, the well founded semantics coincides with the stable model semantics. So we have also used the system XSB[7] that is able to compute the well-founded semantics of such a program. But, for 100000 birds XSB needs more than 500 seconds to compute the result.

Example 5. For the locally stratified program

$$P_{locstrat} = \left\{ \begin{array}{ll} p(1) \leftarrow ., \ldots, p(N) \leftarrow ., & \\ a(X) \leftarrow p(X),\ not\ b(X)., & aa(X,Y) \leftarrow a(X),\ p(Y),\ not\ a(Y)., \\ b(X) \leftarrow p(X),\ not\ a(X)., & bb(X,Y) \leftarrow b(X),\ p(Y),\ not\ b(Y). \end{array} \right\}$$

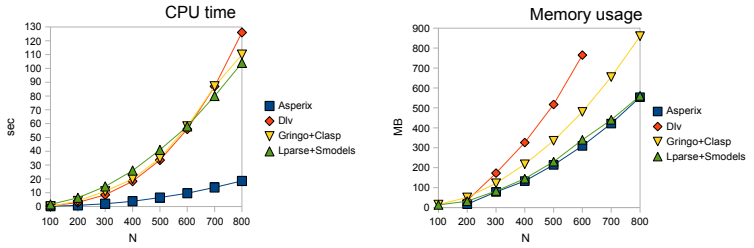
the performances of the systems to compute one answer set are reported below. In this case, the time spent by the grounders is very small (few percents) with respect to the whole consumed time.

[6] Memory usage that exceeds 1GB and CPU time that exceeds 500s are not reported.
[7] XSB (`http://xsb.sourceforge.net`) is a logic programming and deductive database system.



The preceding examples illustrate the ability of `ASPeRiX` to manage very efficiently (locally) stratified programs as P_{birds} and $P_{locstrat}$. On their side, for a definite or stratified program, all intelligent grounders do not generate all grounded instances of rules but compute in fact the answer set of the program and the solver has nothing to do. On the contrary, for $P_{locstrat}$ that is only locally stratified, a solver is necessary to deal with predicates a and b. But, once the choices are made, a and b are solved and the rest of the program becomes stratified and can be easily evaluated by `ASPeRiX`. But, seeing instantiation as a pretreatment forces the grounders to generate all ground instances of rules for aa and bb and the used memory quickly becomes prohibitive.

Example 6. (Example 2 continued) The computation of one answer set of P_2 leads to the following results.



For such an example, memory usages of all solvers (including `ASPeRiX`) are comparable. One reason is that all atoms generated by the grounders must also be generated by `ASPeRiX` since all N^2 atoms $b(\_,\_)$ and N^2 atoms $c(\_,\_)$ have to be included in sets IN and OUT respectively. Nevertheless, `ASPeRiX` is much faster than other systems.

Example 7. (Example 3 continued) For the program P_3 about 3-coloration of a bicycle wheel with $N = 1001$ the computations of one and all the 6 solutions take the following times (in sec).

nb of answer sets	`ASPeRiX`	*Dlv*	*Gringo+Clasp*	*Lparse+Smodels*
1	17	0.38	0.46	1
all (6)	> 500	6	0.52	5.5

As we can see, `ASPeRiX` is not efficient for this example. But, if we use another encoding of the same problem with the program

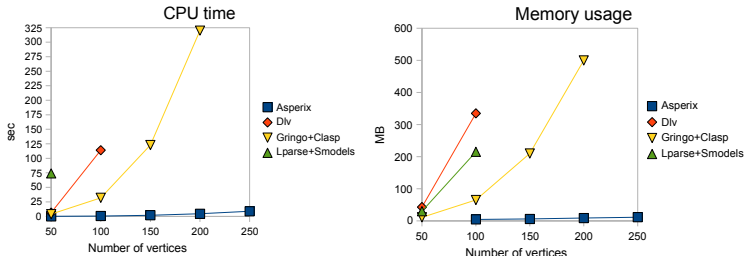
$$P_7 = \left\{ \begin{array}{l} v(1) \leftarrow ., \ldots, v(N) \leftarrow ., \quad c(1) \leftarrow ., \ c(2) \leftarrow ., \ c(3) \leftarrow ., \\ e(1,2) \leftarrow ., \ldots, e(1,N) \leftarrow ., e(2,3) \leftarrow ., \ldots, e(N,2) \leftarrow ., \\ col(V,C) \leftarrow v(V), \ c(C), \ not \ ncol(V,C)., \\ ncol(V,C) \leftarrow col(V,D), \ c(C), \ C \neq D., \\ ncol(V,C) \leftarrow col(U,C), \ e(U,V)., \\ ncol(V,C) \leftarrow col(U,C), \ e(V,U)., \\ colored(V) \leftarrow col(V,C)., \\ \leftarrow v(V), \ not \ colored(V)., \\ col(U,C3) \leftarrow ncol(U,C1), \ ncol(U,C2), \\ \qquad\qquad C1 \neq C2, \ c(C3), \ C3 \neq C1, \ C3 \neq C2. \end{array} \right\}$$

then the performances become the following.

nb of answer sets	`ASPeRiX`	*Dlv*	*Gringo+Clasp*	*Lparse+Smodels*
1	1.1	58	0.8	1.9
all (6)	6	134	0.9	16.2

Here, `ASPeRiX` benefits from the added rules, while the performances of others degrade perhaps because of the amount of new rules to deal with. So, adding some knowledge about the problem to solve, as rules propagating the choices made during the search of a solution (like the last rule in P_7) seems to be profitable for `ASPeRiX` and not always for the other ASP systems.

Example 8. In the two first diagrams below, we show the results of the computation of one answer set of a program encoding the Hamiltonian cycle problem in a complete graph.



In the following table, we show the results for the computation of all answer sets of the same program. Let us note that in this last case there are $(N-1)!$ solutions to compute if the graph has N vertices. Then, we can see that every system spends a linear time with respect to the number of solutions to compute.

N	$(N-1)!$	`ASPeRiX`	*Dlv*	*Gringo+Clasp*	*Lparse+Smodels*
6	120	0.1	0.05	0.02	0.02
7	720	0.7	0.25	0.07	0.23
8	5040	5.1	1.9	0.4	1.5
9	40320	45	17.4	3.5	13.28
10	362880	445	182	36	138

Example 8 illustrates a strange phenomenon. Sometimes, solving a trivial problem, as finding one Hamiltonian cycle in a complete graph, is impossible

for ASP systems. This is very counterintuitive since, in whole generality, in CSP the more the problem has solutions, the easier it is to find one of them. Again, the bottleneck for traditional ASP systems seems to come from the huge number of rules and atoms that are generated in first, delaying and making the resolution more difficult than it should be. On its side, `ASPeRiX` is particularly efficient when we seek one solution among very numerous ones. When we seek all solutions, `ASPeRiX` looses a part of its efficiency, certainly by doing, and redoing, same unifications many times. Despite this point, performances of `ASPeRiX` and other solvers are of the same order of magnitude with respect to the number of solutions.

5 Conclusion

In a first part of this work we have shown some difficulties encountered by ASP systems to deal with programs whose grounding may lead to a huge number of instantiated rules. We have argued that some of these rules are useless with respect to ASP semantics and that it is a reason to not separate rule grounding and answer set computing. Then, we have elaborated a new approach that escapes the preliminary phase of grounding. Our methodology deals with first order rules following a forward chaining with unification process realized on the fly. Furthermore, we have implemented a new ASP solver `ASPeRiX`. We have reported some evaluations of its performances illustrating that our approach is a promising alternative for computing answer sets. In particular, we think that it is well adapted to programs containing rules (with head, not constraints) encoding choice propagation in case of combinatorial problems as it has been illustrated in example 7.

Obviously, this is only a first step towards a first order ASP solver efficient for any kind of programs and we plan to improve our approach in two directions. First, by taking into account ASP semantics, we may integrate better propagation strategies, earlier detection of unsatisfiable constraints, intelligent backtracking, ... in order to prune more efficiently the search space. Second, on the side of the software development, we may improve our unification techniques, first order rule management and tuples handling.

References

1. C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The nomore++ system. In *Proceedings of the 5th Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, pages 422–426, 2005.
2. C. Anger, K. Konczak, and T. Linke. nomore: Non-monotonic reasoning with logic programs. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 521–524, 2002.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

4. C. Baral, G. Brewka, and J. S. Schlipf, editors. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *LNCS*. Springer, 2007.
5. F. Calimeri, S. Perri, and F. Ricca. Experimenting with parallelism for the instantiation of asp programs. *to appear in Journal of Algorithms*, 2008.
6. P. Ferraris, J. Lee, and V. Lifschitz. A new perspective on stable models. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 372–379, 2007.
7. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392, 2007.
8. M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, and M. Truszczynski. The first answer set programming system competition. In Baral et al. [4], pages 3–17.
9. M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In Baral et al. [4], pages 266–271.
10. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.
11. E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.
12. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.
13. F. Lin and Y. Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.
14. F. Lin and Y. Zhou. From answer set logic programming to circumscription via logic of gk. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 441–446, 2007.
15. T. Linke. Graph theoretical characterization and computation of answer sets. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 641–648. Morgan Kaufmann, 2001.
16. L. Liu and M. Truszczynski. Pbmodels - software to compute stable models by pseudoboolean solvers. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *LPNMR*, volume 3662 of *LNCS*, pages 410–415. Springer, 2005.
17. Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczynski. Logic programs with abstract constraint atoms: The role of computations. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *LNCS*, pages 286–301. Springer, 2007.
18. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
19. Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp: Answer set programming with lazy grounding. Convegno Italiano di Logica Computazionale, 2008.
20. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
21. T. Syrjnen. Implementation of local grounding for logic programs for stable model semantics. Technical report, Helsinki University of Technology, 1998.

FO(ID) as an extension of DL with rules

Joost Vennekens* and Marc Denecker

{joost.vennekens, marc.denecker}@cs.kuleuven.be
Dept. of Computer Science, K.U. Leuven, Belgium

Abstract. One of the important topics under active investigation in the area of description logic (DL) is that of adding rules to the language. In this paper, we observe that the logic FO(ID), which was developed as an integration of ASP and classical logic, provides an interesting source of inspiration for such extensions, remaining closer to the DL philosophy than e.g. SWRL. In particular, FO(ID) seems well-suited as an upperbound to a hierarchy of increaslingly expressive extensions of DL with rules. We demonstrate this by defining two interesting sublogics of FO(ID), called $\mathcal{ALCI}$(ID) and guarded $\mathcal{ALCI}$(ID).

1 Introduction

Over the past decades, description logics have emerged as an important knowledge representation technology. More recently, they have also had a significant impact on industry, most notably with the adoption of OWL as a W3C standard. In current research, we find a trend to investigate extensions of OWL with *rules*, as done in e.g. [10]. In fact, the hierarchical Semantic Web architecture already prescribes a *Rule layer* on top of the *Ontology layer* formed by OWL.

Traditionally, research on description logics has always recognized the importance of the trade-off between expressivity and computational complexity, as well as the fact that different applications require this trade-off to be made differently. This has lead to the development of an entire hierarchy of logics, ranging from for instance the tractable DL-Lite to OWL DL, or even further to the undecidable OWL Full. In keeping with this tradition, a reasonable goal for the research into extensions of OWL with rules might be to come up with a hierarchy of logics, combining increasingly expressive description logics with increasingly expressive kinds of rules.

In this paper, we want to call attention to the language FO(ID) [3][1] and argue that it provides an ideal upperbound for such a hierarchy. FO(ID) is a knowledge representation language that adds inductive definitions to first-order logic. It represents these definitions in the same way as they typically appear in mathematical texts, i.e., as an enumeration of a set of cases in which the defined relation(s) holds; in FO(ID), each of these cases is represented by a rule. For instance, in a textbook on propositional logic, we might find the inductive

* Joost Vennekens is a postdoctoral researcher of the FWO.
[1] This paper refers to FO(ID) as FO(NMID).

definition shown in Figure 1, which can be represented in FO(ID) by the following set of *definitional rules*:

$$\left\{ \begin{array}{c} \forall i, p\ Sat(i,p) \leftarrow p \in i. \\ \forall i, f_1, f_2\ Sat(i, or(f_1,f_2)) \leftarrow Sat(i,f_1) \vee Sat(i,f_2). \\ \forall i, f\ Sat(i, not(f)) \leftarrow \neg Sat(i,f). \end{array} \right\}$$

This definition defines the predicate $Sat/2$ in terms of the predicate $\in/2$ and the term-building functors $or/2$ and $not/1$. We refer to this last kind of symbols—$\in/2$, $or/2$ and $not/1$—as the *open* symbols of the definition, and to the other ones as the *defined* predicates.

The definitional rules of FO(ID) cannot simply be reduced to implications or equivalences, since this would fail to treat even the most basic inductive definitions (e.g., that of a transitive closure) correctly. Instead, they are interpreted according to the *well-founded model semantics* for logic progams (parametrized on the interpretation of the open predicate). As argued in [4], this ensure that—also in the presence of negation—their meaning is indeed what one expects from an inductive definition[2].

Definition 1. *Let I be an interpretation for a set of propositions Σ. For a propositional formula φ in alphabet Σ, we define the relation I* satisfies *φ, denoted $I \models \varphi$, by the following induction over the subformula order:*

- *For an atom $p \in \Sigma$, $I \models p$ iff $p \in I$;*
- *$I \models \varphi_1 \vee \varphi_2$ iff $I \models \varphi_1$ or $I \models \varphi_2$;*
- *$I \models \neg\varphi$ iff $I \not\models \varphi$.*

Fig. 1. The inductive definition of satisfaction in propositional logic.

A theory in FO(ID) now consists of, on the one hand, a set of such definitions and, on the other hand, a set of regular first-order logic formulas. We remark that, therefore, rules themselves are *not* FO(ID) formulas, but definitions, i.e., *sets* of rules. The semantics of the first-order formulas is standard. The semantics of the definitions is presented formally in the next section, but can be intuitively summarized as follows: each definition expresses a certain relation between its defined predicates and its open symbols; therefore, such a definition Δ is satisfied in a first-order structure S, written $S \models \Delta$, iff S interprets the defined predicates in the way that its interpretation of the open predicates dictates.

The origins of FO(ID) lie in the area of logic programming. In particular, FO(ID) is very closely related to the Answer Set Programming (ASP) paradigm. The precise relation between FO(ID) and ASP is discussed in detail in [13]. One of the main contributions that FO(ID) provides to this area is to show how a

[2] The correspondence between the precise, well-understood but *informal* concept of an inductive definion and the *formal* object that is the well-founded semantics can of course, by nature, never be proven formally.

tight integration of logic programming rules into classical logic can be achieved in a conceptually clean way. Therefore, FO(ID) offers a way of combining ASP and DL which is, unlike the hybrid approaches of, e.g., [6] and [5], a true semantic integration. This is useful because, while hybrid approaches have the advantage of technical flexibility, they are somewhat unsatisfactory from a philosophical, knowledge theoretical perspective, since they do not shed much light on the relation between the meaning of statements expressed in the two languages. Moreover, hybrid approaches also provide little modeling methodology to guide users when constructing a joint LP/DL representation of some domain, since they have a hard time satisfactorily answering the question of which knowledge to represent in DL and which in LP.

From the DL point-of-view, the main appeal of FO(ID) in this context is it is not really an *extension* of description logics with some distinct notion of rules, but rather a general embodiement of the foundational ideas of description logic themselves, which happens to include, in a very natural way, a particular form of rules. Indeed, the origin of description logics lies with Brachman and Levesque's observation [2] that two distinct forms of knowledge are of great importance for knowledge-based systems: *terminological* knowledge, which defines the meaning of relations and concepts; and *assertional* knowledge, which states properties of the world. We find the same distinction in FO(ID), which offers both definitions (to express terminological knowledge) and first-order logic formulas (to express assertional knowledge). Moreover, in both cases, the representation is arguably as general as one could want, allowing, on the one hand, most kinds of inductive definitions that are typically found in mathematical texts [4] and, on the other hand, full first-order logic. Therefore, FO(ID) indeed seems a suitable upperbound, as we have claimed above.

In particular, we would argue that FO(ID) stays closer to the DL philosophy than, for instance, the language of SWRL [10], which is currently one of the most popular extensions of OWL with rules. SWRL extends OWL with Horn clauses, i.e., implications in which the antecedent is a conjunction of atoms and the consequent is either an atom or is absent. The meaning of such a rule is that of a standard FO implication. The example of a SWRL-rule most commonly found in the literature is the following:

$$\forall x, y, z \; Brother(x, z) \wedge Parent(z, y) \supset Uncle(x, y)$$

One striking thing about this rule is that it does not define the relation $Uncle$ in terms of the relations $Parent$ and $Brother$: instead, it only states the *sufficient* condition that all brothers of parents are uncles, but does not express that this condition is also *necessary*—and as a matter of fact, this is something which cannot be expressed in SWRL. Here, SWRL clearly departs from the original DL philosophy. Indeed, to quote the DL handbook [1] about the '$\equiv$'-connective: "This form of definition is much stronger than the ones used in other kinds of representations of knowledge, which typically impose only necessary conditions; the strength of this kind of declaration is usually considered a characteristic feature of DL knowledge bases."

Because FO(ID) extends first-order logic, it also allows us to write down such a material implication, which means that it is possible to state only the sufficient condition if this is our intention. However, it also provides explicit support for defining concepts. For example, both the necessary and sufficient condition are correctly expressed by the FO(ID) definition consisting of the single rule:

$$\{\forall x, y\ Uncle(x, y) \leftarrow \exists z\ Brother(x, z) \wedge Parent(z, y).\}$$

Indeed, this FO(ID) definition can be shown to be equivalent to the first-order logic formula:

$$\forall x, y\ Uncle(x, y) \Leftrightarrow \exists z\ Brother(x, z) \wedge Parent(z, y).$$

If we also want our concept of an uncle to include uncles-by-marriage, we might define it instead by the following set of two rules:

$$\left\{\begin{array}{l} \forall x, y\ Uncle(x, y) \leftarrow \exists z\ Brother(x, z) \wedge Parent(z, y). \\ \forall x, y\ Uncle(x, y) \leftarrow \exists z\ Husband(x, z) \wedge Aunt(z, y). \end{array}\right\}$$

This is then equivalent to the first-order logic formula:

$$\begin{aligned} \forall x, y\ Uncle(x, y) \Leftrightarrow & (\exists z\ Brother(x, z) \wedge Parent(z, y)) \\ & \vee (\exists z\ Husband(x, z) \wedge Aunt(z, y)). \end{aligned}$$

This illustrates one particular advantage that the rule-based representation has over the standard equivalence: it is more *elaboration tolerant*, because if we discover a new case in which the defined relation holds, we simply need to add an additional rule to its definition, without touching the old ones.

FO(ID) can also express *inductive* definitions, which cannot be expressed in first-order logic (and therefore, *a fortiori*, also not in SWRL). An example is the following definition of the *Ancestor* relation:

$$\left\{\begin{array}{l} \forall x, y\ Anc(x, y) \leftarrow \exists z\ Anc(x, z) \wedge Anc(z, y). \\ \forall x, y\ Anc(x, y) \leftarrow Parent(x, y). \end{array}\right\}$$

One of the other characteristic features of FO(ID) is that, unlike SWRL, it also allows the use of negation in rule bodies. For instance, we might define:

$$\{\forall x\ OnlyChild(x) \leftarrow \neg\exists y\ Sibling(x, y).\}$$

Moreover, negation can also be combined with recursion, as shown by the third rule in the above definition of *Sat*. Unlike SWRL, the rules of FO(ID) can therefore be used to express *non-monotone inductive* definitions.

To summarize, this paper looks at FO(ID), a general integration of FO and logic programming, which has the same philosophical underpinnings as description logic itself. We propose this language as a foundation for the development of extensions of description logics, such as OWL, with rules. To be more precise, we argue that FO(ID) would serve well as an upperbound to a hierarchy of

logics that add increasingly expressive forms of rules to increasingly expressive description logics. This also suggest that we can develop less expressive extensions of DL with rules, which might be more useful in practice, by restricting the general language FO(ID) as needed to regain certain desirable properties. We believe that, in general, this is better than the opposite approach of gradually extending a small tractable language into a more expressive one, since it allows the trade-off between expressivity and complexity to be made more conciously and informedly, which reduces the risk of creating an *ad hoc* language, whose boundaries are decided more by coincidence than design.

This paper will try to back up our claims, by considering the basic description logic $\mathcal{ALCI}$ and presenting two different extensions of $\mathcal{ALCI}$ with rules, both of which are sublogics of FO(ID). Our goal here will be to regain two desirable properties of description logics. The first such property is the intuitive *syntactic sugar* of description logics, which hides away many of the complexities of classical logic and enforces adherence to a useful, concept-centric style of knowledge representation. We will define a sublogic of FO(ID), called $\mathcal{ALCI}$(ID), for which a similar syntactic sugar exists. The second property is the *decidability* of deductive inference. This too is an important property for many applications—even though it is not always needed, as is witnessed by the fact that OWL Full and SWRL (unless restricted to strongly safe rules) are both undecidable. We will show that $\mathcal{ALCI}$(ID) has a decidable *guarded fragment*. Finally, we will also briefly discuss the relation between our FO(ID) based approach and several other approaches to combining ASP or LP with DL.

2 Preliminaries: FO(ID)

This section summarizes the definition of FO(ID). Syntactically, a definition in FO(ID) is a set of *definitional rules*, which are of the form:

$$\forall \boldsymbol{x}\ P(\boldsymbol{x}) \leftarrow \varphi(\boldsymbol{x}).$$

Here, $\varphi(\boldsymbol{x})$ is an FO formula whose free variables are $\boldsymbol{x}$, and '$\leftarrow$' is a new symbol, the *definitional implication*, which is to be distinguished from the material implication of FO, which we denote by '$\subset$'. We refer to $P(\boldsymbol{x})$ as the *head* of the rule r, denoted $head(r)$, and to the formula φ as its *body*, denoted $body(r)$.

An FO(ID) *formula* is any expression that can be formed by combining atoms and definitions, using the standard FO connectives and quantifiers. The meaning of the FO formulas and boolean connectives is standard; we therefore only need to define the semantics of a definition in order to define that of FO(ID). Originally, this was formulated in logic programming terminology, defining the semantics of a definition by (roughly speaking) its well-founded model. More recently, however, [4] introduced a new, equivalent characterization of this semantics, which is more natural and easier understand. Therefore, we will present this new characterization.

Let us first introduce some semantical concepts. An *interpretation* $\mathcal{S}$ for a vocabulary Σ consists of a non-empty domain D, a mapping from each function

symbol f/n to an n-ary functions on D, and a mapping from each predicate symbols P/n to a relation $R \subseteq D^n$. A *three-valued* interpretation ν is the same as a two-valued one, except that it maps each predicate symbol P/n to a function P^ν from D^n to the set of truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. Such a ν assigns a truth value to each logical atom $P(\boldsymbol{c})$, namely $P^\nu(c_1^\nu, \ldots, c_n^\nu)$. This assignment can be extended to an assignment $\nu(\varphi)$ of a truth value to each formula φ, using the standard Kleene truth tables for the logical connectives:

φ, ψ	$\mathbf{t}$, $\mathbf{t}$	$\mathbf{t}$, $\mathbf{f}$	$\mathbf{t}$, $\mathbf{u}$	$\mathbf{u}$, $\mathbf{f}$	$\mathbf{u}$, $\mathbf{u}$	$\mathbf{f}$, $\mathbf{f}$
$\varphi \vee \psi$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{u}$	$\mathbf{u}$	$\mathbf{f}$

φ	$\mathbf{t}$	$\mathbf{u}$	$\mathbf{f}$
$\neg\varphi$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{t}$

and so on.

The three truth values can be partially ordered according to *precision*: $\mathbf{u} \leq_p \mathbf{t}$ and $\mathbf{u} \leq_p \mathbf{f}$. This order induces also a precision order $\leq_p$ on interpretations: $\nu \leq_p \nu'$ if for each predicate P/n and tuple $\boldsymbol{d} \in D^n$, $P^\nu(\boldsymbol{d}) \leq_p P^{\nu'}(\boldsymbol{d})$.

For a predicate P/n and a tuple $\boldsymbol{d} \in D^n$ of domain elements, we denote by $\nu[P(\boldsymbol{d})/\mathbf{v}]$ the three-valued interpretation ν' that coincides with ν on all symbols apart from P/n, and for which $P^{\nu'}$ maps $\boldsymbol{d}$ to $\mathbf{v}$ and all other tuples $\boldsymbol{d}'$ to $P^\nu(\boldsymbol{d}')$. We also extend this notation to sets $\{P_1(\boldsymbol{d}_1), \ldots, P_n(\boldsymbol{d}_n)\}$ of such pairs.

Our goal is to define when a (two-valued) interpretation $\mathcal{S}$ is a model of a definition Δ. We call the predicates that appears in the head of a rule of Δ its *defined predicates* and we denote the set of all these by $Def(\Delta)$; all other symbols are called *open* and the set of open symbols is written $Op(\Delta)$. The purpose of Δ is now to define the predicates $Def(\Delta)$ in terms of the symbols $Op(\Delta)$, i.e., we should assume the interpretation of $Op(\Delta)$ as given and try to construct a corresponding interpretation for $Def(\Delta)$. Let O be the restriction $\mathcal{S}|_{Op(\Delta)}$ of $\mathcal{S}$ to the open symbols. We are now going to construct a sequence of three-valued interpretations $(\nu_\alpha^O)_{0 \leq \alpha \leq \beta}$, for some ordinal β, each of which extends O; we will use the limit of such a sequence to interpret $Def(\Delta)$.

- ν_0^O assigns $O(P(\boldsymbol{d})) \in \{\mathbf{t}, \mathbf{f}\}$ to $P(\boldsymbol{d})$ if $P \in Op(\Delta)$ and $\mathbf{u}$ if $P \in Def(\Delta)$;
- ν_{i+1}^O is related to ν_i^O in one of two ways:
  - Either $\nu_{i+1}^O = \nu_i^O[P(\boldsymbol{d})/\mathbf{t}]$, such that Δ contains a rule $\forall \boldsymbol{x}\ P(\boldsymbol{x}) \leftarrow \varphi(\boldsymbol{x})$ with $\nu_i^O(\varphi[\boldsymbol{d}]) = \mathbf{t}$
  - Or $\nu_{i+1}^O = \nu_i^O[U/\mathbf{f}]$, where U is any *unfounded set*, meaning that it consists of pairs of predicates P/n and tuple $\boldsymbol{d} \in D^n$ for which $P^{\nu_i^O}(\boldsymbol{d}) = \mathbf{u}$, and for each rule $\forall \boldsymbol{x}\ P(\boldsymbol{x}) \leftarrow \varphi(\boldsymbol{x})$, we have that $\nu_{i+1}^O(\varphi[\boldsymbol{d}]) = \mathbf{f}$.
- For each limit ordinal λ, ν_λ^O is the least upper bound w.r.t. $\leq_p$ of all ν_δ^O for which $\delta < \lambda$.

We call such a sequence a *well-founded induction* of Δ in O. Each such sequence eventually reaches a limit ν_β^O. It was shown in [4] that all sequences reach the same limit. It is now this ν_β^O that tell us how to interpret the defined predicates. To be more precise, we define that:

$$\mathcal{S} \models \Delta \text{ iff } \mathcal{S}|_{Def(\Delta)} = \nu_\beta^O|_{Def(\Delta)}, \text{ with } O = \mathcal{S}|_{Op(\Delta)}.$$

Note that if there is some predicate P/n for which some tuple $\boldsymbol{d} \in D^n$ of domain elements is still assigned $\mathbf{u}$ by ν_β, the definition has no models extending O. Intuitively, this means that, for this particular interpretation of its open symbols, Δ does not manage to unambiguously define the predicates $Def(\Delta)$, due to some non well-founded use of negation.

In the rest of this paper, we will only consider relational vocabularies, that is, there will be no function symbols of arity > 0.

3 $\mathcal{ALCI}$(ID)

In this section, we present a fragment of FO(ID) that extends the description logic $\mathcal{ALCI}$ with rules, while retaining a DL-like syntactic sugar.

We start from the usual connectives $\sqcup$, $\sqcap$, $\cdot^{-}$, $\neg$, $\exists$ and $\forall$. We will also allow the use of $\sqcup, \sqcap$ and $\neg$ to form, respectively, disjunctions, conjunctions and negations of *roles*, instead of applying them only to concepts.

In addtion, we introduce the following new connectives to construct roles:

- The operator '.' takes the join of two binary relation, that is, $R.S$ is the relation consisting of all pairs (x, y) for which there exits a z such that $R(x, z)$ and $S(z, y)$.
- The operator '$\times$' constructs the binary relation that is the Cartesian product of two unary relations, that is, $C \times D$ contains all pairs (x, y) for which $C(x)$ and $D(y)$.

These connectives can mapped to classical logic in a straightforward way; details are shown in Figure 2.

φ	$\langle\varphi\rangle$
$R.S$	$\exists x\ \langle R\rangle(x, z) \wedge \langle S\rangle(z, y)$
$C \times D$	$\langle C\rangle(x) \wedge \langle D\rangle(y)$

Fig. 2. The mapping $\langle\cdot\rangle$ of $\mathcal{ALCI}$(ID) to FO(ID).

In addition to these concept/role-building operators, we also have the usual '$\sqsubseteq$'-connective. We introduce the new connective '$\leftarrow$' to represent a definitional rule. For instance, we can define the concept $Uncle$ by the following set of two definitional rules:

$$\left\{ \begin{array}{l} Uncle \leftarrow Brother.Parent \\ Uncle \leftarrow Husband.Aunt \end{array} \right\}$$

In general, if R is a role symbol and φ some description of a binary relation (that is, $\langle\varphi\rangle$ is a formula in two free variables), then the $\mathcal{ALCI}$(ID) statement $R \leftarrow \varphi$ represents the FO(ID) rule $\forall x, y\ R(x, y) \leftarrow \langle\varphi\rangle(x, y)$. Similary, if C

is a concept symbol and φ a description of a unary relation ($\langle\varphi\rangle$ has one free variable), then $C \leftarrow \varphi$ stands for $\forall x\ C(x) \leftarrow \langle\varphi\rangle(x)$.

$\mathcal{ALCI}$ is typically also defined to have the equivalence symbol '$\equiv$', which abbreviates two inclusions. Analogously, we also introduce the new symbol '$\doteq$' to represent definitions containing only a single rule, i.e., a statement $C \doteq \varphi$ stands for the definition $\{C \leftarrow \varphi\}$. We can, for instance, rephrase the above definition of $Uncle$ as

$$Uncle \doteq Brother.Parent \sqcup Husband.Aunt$$

If such a definition using '$\doteq$' is not inductive (i.e., the role/concept on the left-hand side does not appear in the right-hand side), it is simply equivalent to a normal equivalence '$\equiv$'. However, unlike'$\equiv$', this '$\doteq$' also works for definitions which *are* inductive, such as:

$$Ancestor \doteq Parent \sqcup Ancestor.Ancestor$$

Therefore, '$\doteq$' eliminates the need for a transitive closure construct such as $\cdot^+$ or the reflexive-transitive closure $\cdot^*$ of e.g. [11]; it can also replace non-nested uses of the explicit least fixpoint constructor μ.

A '$\doteq$'-statement abbreviates a definition containing only a single rule. In general, if a definition contains multiple rules with the same predicate in the head, these can always be replaced by a single rule whose body is the disjunction of the bodies of the original rules, as illustrated by the above definition of $Uncle$. Therefore, each definition which defines only a single predicate can be stated either in the rule-based format or using '$\doteq$'. The advantage of '$\doteq$' is that it offers a more compact representation. On the other hand, the rule-based format is more elaboration tolerant, since rules can more easily be added or removed. Therefore, it is more suited for definitions which are likely to change. For instance, a bank might define the class of persons eligable for a loan as consisting of people with a large income, people who own a house and people with a good credit history. Each time the bank now tightens or relaxes its policy, certain rules would have to be removed or added to this definition. Therefore, the rule-based representation seems more appropriate in this case.

The rule-based representation is also more general than '$\doteq$', since it also allows definitions by *simultaneous* induction. For instance, given a two-player game whose move tree is described by the role $Parent$, we can define the nodes in which I move and the nodes in which my opponent moves by the following simultaneous induction (assuming I start):

$$\left\{\begin{array}{l} MyMove \leftarrow (\exists Parent.HisMove) \sqcup (\forall Parent.\bot) \\ HisMove \leftarrow \exists Parent.MyMove \end{array}\right\}$$

Syntax. In summary, the syntax of $\mathcal{ALCI}$(ID) is formally defined as follows. A *role* is either a role name or one of the following expressions: R^-, $\neg R$, $R \sqcup S$, $R \sqcap S$, $R.S$, $R \times S$, where R and S are roles. A *concept* is either $\top$, $\bot$, a concept

name, or one of the following: $C \sqcup D$, $C \sqcap D$, $\neg C$, $\exists R.C$, $\forall R.C$, where C, D are concepts and R is a role. A *formula* of $\mathcal{ALCI}$(ID) is then either an inclusion $X \sqsubseteq Y$, an equivalence $X \equiv Y$, a single-rule definition $X \doteq Y$, where either both X and Y are concepts or both are roles, or a multiple-rule definition $\{X \leftarrow Y_1; \ldots; X \leftarrow Y_n\}$, where either all of $X, Y_1, \ldots, Y_n$ are concepts or all are roles.

The semantics of the language is formally defined by the mapping to FO(ID) described above.

4 An example

Fig. 3. The result of selecting ball 2.

In this section, we will illustrate the language $\mathcal{ALCI}$(ID) by presenting a formalization of a simple game, sometimes found on mobile phones and such. The player is presented a grid of coloured balls. He makes a move by selecting one of these balls. The effect of this is that the entire colour-group to which the ball belongs disappears; the remaining balls then fall down, yielding the next position of the game, as depicted in Figure 3. The goal of the game is to remove all balls from the grid in such a way as to score as many points as possible. However, we will not discuss how points are scored, but instead focus only on how to specify the effect of a move on the state of the board. To be more concrete, we will define the new state in terms of both the old state and the move made by the player. It is clear that, in principle, this is the bulk of the domain dependent information that is needed to be able to plan out a winning strategy—the rest can be handled by some generic game-playing planner.

To make things more concrete, let us first fix a representation for a state of the game. We will represent a grid by two binary relations: a relation Up and a relation $Left$, both with the obvious meaning. The starting grid in Figure 3, for instance, would correspond to the following interpretations:

$$\begin{aligned} Up = &\{(5,1),(9,5),(6,2),(10,6),(7,3),\\ &(11,7),(8,4),(12,8)\};\\ Left = &\{(1,2),(2,3),(3,4),(5,6),(6,7),\\ &(7,8),(9,10),(10,11),(11,12)\}. \end{aligned}$$

We represent the player's move by a unary predicate $Chosen$; the move made in Figure 3 would correspond to $Chosen = \{2\}$. Our goal is now to define roles Up' and $Left'$, representing the next state of the game, in terms of the original position described by Up, $Left$ and $Chosen$.

We first define some useful auxilary relations. We begin by defining $Above$ as the transitive closure of Up:

$$Above \doteq Up \sqcup Above.Above$$

A ball is next to another ball if it is either to the left, to the right, underneath, or on top of it:

$$NextTo \doteq Left \sqcup Right \sqcup Up \sqcup Below$$

Of course, $Right$ and $Below$ are simply the inverses of, respectively $Left$ and Up:

$$Right \doteq Left^- \qquad Below \doteq Up^-$$

The balls that disappear after the move are the chosen ball itself and all balls belonging to the same colour-group:

$$Disappears \doteq Chosen \sqcup \exists InColourGroup.Chosen$$

Being in the same colour group means being connected through a sequence of balls of the same colour:

$$InColourGroup \doteq (SameColour \sqcap NextTo) \sqcup InColourGroup.InColourGroup$$

This of course requires us to define when two balls have the same colour. This will be the case if one ball has a colour, that is also the colour of the other ball.

$$SameColour \doteq HasColour.HasColour^-$$

Having defined which balls disappear, we can now easily define which balls remain as the complement thereof:

$$Remains \doteq \neg Disappears$$

We now define the relation $Above'$, i.e., the "above"-relation as it will hold in the next state. This will hold for any two remaining balls that were originally above each other.

$$Above' \doteq Above \sqcap (Remains \times Remains)$$

We can now define the relation Up' as the intransitive relation of which $Above'$ is the transitive closure:

$$Up' \doteq Above' \sqcap \neg(Above'.Above')$$

We define an auxiliary concept $OnGround'$ as consisting of those balls that will be on the ground in the new situations:

$$OnGround' \doteq Remains \sqcap \neg\exists Below.Remains$$

Having already defined Up', all that remains is to define also $Left'$. Let us first define when a ball is in the column to the left of some other ball.

$$InLeftColumn \doteq Left \sqcup Left.(Above \sqcup Above^{-})$$

We now define $Left'$ as consisting of all pairs of balls for which $InLeftColumn$ holds and which are both on the same height in the new situation:

$$Left' \doteq InLeftColumn \sqcap ((OnGround' \times OnGround') \sqcup Up'^{-}.Left'.Up').$$

Note that this too is an inductive definition: the relation $Left'$ is first defined for those pairs of balls that are both in the first row, then for those in the second row, and so on.

This now concludes our representation of this game. We remark that this example makes heavy use of the characteristic features of FO(ID): the ability to express inductive definitions and to use negation in such definitions.

5 Inference in a fixed and finite domain

For basic description logics such as $\mathcal{ALCI}$, deductive inference (i.e., answering the question "is a formula φ a logical consequence of a theory T?") is decidable, and even tractable. While the possibility to efficiently perform this inference task—and related ones such as subsumption or consistency checking—has certainly contributed to the appeal of DL, the need for fast deductive inference is not absolute. Indeed, sometimes DL theories are not (primarily) intended to be used by a computer, but serve instead as a means of communication between people. For instance, in the context of software development, [18] uses a DL theory to represent an agreement between business experts and software engineers about a common model of the application domain. Even though in such cases we might still be interested in, e.g., consistency of the theory, it is here less of a requirement that this can be verified fully automatically.

Moreover, even when the goal is to perform inference tasks on a theory, this does not necessarily preclude the use expressive or even undecidable languages. Indeed, recent trends in computational logic, in particular ASP, have shown that show that many real-world problems can be solved by considering other, "cheaper" inference tasks than traditional deduction. The central observation here is that the solution to many interesting problems can be naturally characterized as (part of) a finite structure/interpretation satisfying a given theory. By performing reasoning in a fixed, finite domain (for instance, a finite Herbrand universe), instead of in an *open domain*, the complexity of reasoning tasks tends to decreases dramatically. Even deduction for full first-order logic (that is, deciding whether a formula φ holds in all models of a theory T whose domain is some given, finite set D) is then only co-NP.

Extending ASP's model generation paradigm, [14] considered the inference task of *model expansion* for FO(ID): given an interpretation $\mathcal{S}$ for some part Σ_0 of the alphabet Σ of a theory T, extend $\mathcal{S}$ with an interpretation for the

remaining symbols $\Sigma \setminus \Sigma_0$, in such a way that the resulting interpretation is a model of T. Since the given interpretation $\mathcal{S}$ for Σ_0 already determines the domain, there is no need to consider the possibility of unknown objects. They showed that model expansion for FO(ID) captures the complexity class NP.

To illustrate the usefulness of this inference task, let us consider again the example of the previous section. Here, we defined the next state of a game ($Left'$ and Up') in terms of its old state ($Left$ and Up) and a given move ($Chosen$). We can therefore compute a new state of the game by performing model expansion on the structure $\mathcal{S}$ for the alphabet $\Sigma_0 = \{Left, Up, Chosen\}$ that represents the old state and the chosen move; the interpretation of $Left'$ and Up' in the resulting expanded model then gives us the new state. Moreover, because our representation of the game defines all of its predicates except the ones in Σ_0, this computation can actually be done in polynomial times.

6 Guarded $\mathcal{ALCI}$(ID)

As discussed in the previous section, there are interesting applications for DL, and its extensions with rules, which only require reasoning with a fixed domain and for which the undecidability of FO(ID) is therefore not a problem. However, in other circumstances, open domain reasoning can still be necessary. Therefore, in this section we will develop a decidable fragment of FO(ID).

Our fragment will be based on the *guarded fragment* of FO. We recall that an FO formula φ is *guarded* if every one of its quantified subformulas is either of the form $\exists \boldsymbol{x}\ G(\boldsymbol{x}, \boldsymbol{y}) \wedge \varphi(\boldsymbol{x}, \boldsymbol{y})$ or $\forall \boldsymbol{x}\ G(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \varphi(\boldsymbol{x}, \boldsymbol{y})$, where $G(\boldsymbol{x}, \boldsymbol{y})$ is an atom, called the *guard* of the formula, such that $free(\varphi(\boldsymbol{x}, \boldsymbol{y})) \subseteq free(G(\boldsymbol{x}, \boldsymbol{y}))$.

We now define a similar guarded fragment of FO(ID). This fragment will allow only theories consisting of precisely one definition and precisely one first-order formula.

Definition 2. *Let T be an FO(ID) theory, consisting of precisely one definition Δ and one FO formula φ. T is* guarded *if*

- *φ is a guarded formula;*
- *for each rule $\forall \boldsymbol{x}\ P(\boldsymbol{x}) \leftarrow \psi$, it holds that ψ is a guarded formula and $\boldsymbol{x} \subseteq free(\psi)$;*
- *none of the guards is defined by Δ.*

Exploiting a theorem from [7], we prove the following result in the appendix to this paper.

Theorem 1. *The guarded fragment of FO(ID) is decidable. More precisely, deductive reasoning for FO(ID) is 2*EXPTIME*-complete.*

This of course raises the question of how this guarded fragment relates to the $\mathcal{ALCI}$(ID)-fragment defined above. It is easy to see that neither is subsumed by the other. For instance, the $\mathcal{ALCI}$(ID) formula $P \doteq P.P$ translates to

$$\{\forall x, y\ P(x, y) \leftarrow \exists z\ P(x, z) \wedge P(z, y).\},$$

which is not in our guarded fragment. Conversely, the guarded fragment is not subsumed by $\mathcal{ALCI}$(ID) either, since, for instance, the latter only allows unary and binary predicates, while the former allows arbitrary arities.

The two fragments do, however, have a significant intersection. Let us consider a theory T in $\mathcal{ALCI}$(ID) and examine which additional conditions need to be imposed in order to ensure that its translation will be guarded.

The guarded fragment of FO(ID) only allows theories to contain a *single* definition, whereas an $\mathcal{ALCI}$(ID) theory can have multiple definitions. In [19], we examined when a set of definitions $(\Delta_i)_{1\leq i\leq n}$ can be merged into a single equivalent definition $\cup_i \Delta_i$. We showed there that the following condition is sufficient.

Condition 1 *There exists a partial order $\leq$ on the predicates of the theory, such that for all Δ_i and $P \in Def(\Delta_i)$:*

- *if $Q \in Def(\Delta_i)$ then $P \geq Q$;*
- *if Q appears in Δ_i but does not belong to $Def(\Delta_i)$, then $P > Q$ (i.e., $P \geq Q$ and $P \not\leq Q$).*

By imposing this condition on our theories, we are therefore able to merge all definitions into a single definition, thus already satisfying that particular requirement of our guarded fragment. Looking at the translation of $\mathcal{ALCI}$(ID) into FO(ID), it is quite clear that the only other cause for falling outside of the guarded fragment is the fact that defined predicates might be used as guards. We therefore also need the following restriction:

Condition 2 *For every construct $\exists R.C$, $\forall R.C$ or $R.S$ that appears in T, it must be the case that R is not defined, i.e., T does not contain any formulas of the form $R \doteq \varphi$ or any definitions in which a rule $R \leftarrow \varphi$ appears.*

We will say that an $\mathcal{ALCI}$(ID) theory is *guarded* if it satisfies both Condition 1 and Condition 2.

Theorem 2. *Let T be a guarded $\mathcal{ALCI}$(ID) theory and let T' be the result of merging all definitions in $\langle T\rangle$ into a single definition. Then T' is equivalent to $\langle T\rangle$ itself and belongs the guarded fragment of FO(ID).*

To conclude this section, we recall that for SWRL, it has been shown that decidability can be regained by a restriction to *(strongly) safe* rules [16]: a SWRL-rule is safe if every one of its variables appears in an atom whose predicate is not used anywhere in the TBox of the OWL theory (i.e., it is only allowed to appear in other SWRL rules or in the ABox). This restriction has a somewhat similar flavour to our Condition 2, in that it also requires that variables are "guarded" by atoms about which there is not "too much" information present in the theory.

7 Related Work

[6] presents an integration of DL and Answer Set Programs. In [5], a combination of DL and logic programs under the well-founded semantics is investigated. Unlike FO(ID)'s strong semantic integration of LP and DL, these two approaches

foster a strong separation between the two components: essentially, they allow a logic program to pose queries to a description logic theory, with the latter acting more-or-less as a black box towards the former. In contrast, FO(ID) can offer a full semantic integration in which both logic programs and description logic axioms are first-class citizens. As mentioned in the introduction, we feel that such a strong intregration is more satisfactory from a knowledge theoretic point-of-view.

Description logic programs [8] and $\mathcal{DL}$+log [17] are two strong integrations, which both consider only a quite restricted language: Grosof et al. focuses on the intersection, rather than the union, of DL and LP, and Rosati only consids Datalog. By contrast, FO(ID) allows full first-order logic in the body of rules.

[15, 12, 9] both provide strong semantic integrations for a more general kind of rules. Even though we do not yet know the precise formal relation between these languages and FO(ID), they seem to be about equally expressive. The main difference with our approach is of a philosophical nature: FO(ID) makes a clear distinction between its definitions, which are meant to express terminological knowledge, and its first-order formulas, which are meant to express assertional knowledge; the other two languages do not address the question of which kind of knowledge should be expressed by which constructs in their language.

8 Conclusions and future work

FO(ID) is a knowledge representation language which integrates ASP-style logic programming and classical logic. It shares the epistemological and philosophical foundations of description logic, by acknowledging the distinction between terminological and assertional knowledge, and the need to represent both in a knowledge based system. To represent terminological knowledge, it offers a definition construct which is more general than that of description logics, allowing a natural, rule-based representation of most of the common forms of (inductive) definitions found in e.g. mathematical texts. So, FO(ID) actually contains two rule-like constructs, namely material implications and definitional rules, and makes a clear distinction between the meaning of these two, both at the formal and informal level. This makes FO(ID) a suitable setting for a semantic study of extensions of description logics with rules. In particular, we have argued that it makes a good upperbound to an expressivity hierachy of such extensions, or, to put it the other way around, that we could construct such a hierarchy by restricting FO(ID) in various appropriate ways. We have tried to demonstrate the appeal of this approach, by defining the language $\mathcal{ALCI}$(ID) as syntactic sugar for a certain fragment of FO(ID). Like FO(ID) itself, this language is not decidable. However, this does not preclude it from having important computational applications, such as solving various types of fixed finite domain problems. Because other applications of course do require general deductive reasoning, we have also defined *guarded* $\mathcal{ALCI}$(ID) as a decidable fragment.

There are two obvious ways in which this work could be extended. First of all, this paper has considered just two languages in between $\mathcal{ALCI}$ and full FO(ID).

This is obviously still far from a thorough analysis of all the possibilities. Second, we have only looked at the basic description logic $\mathcal{ALCI}$. More expressive logics, such as $\mathcal{SHOIN}$(D) which underlies OWL-DL, could of course be extended with rules by the same method by which we have extended $\mathcal{ALCI}$ in this paper. How the expressivity of the underlying description logic affects the complexity of the resulting extension with rules is another interesting question for future research.

References

1. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook. Theory, Implementation and Applications*. Cambridge University Press, 2002.
2. Ronald J. Brachman and Hector J. Levesque. Competence in Knowledge Representation. In *Proc. of AAAI*, pages 189–192, 1982.
3. Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions. *Transactions On Computational Logic (TOCL)*, 2008.
4. Marc Denecker and Joost Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In *Proc. LPNMR*, 2007.
5. W. Drabent, J. Henriksson, and J. Maluszynski. HD-rules: a hybrid system interfacing Prolog with DL-reasoners. In *2nd I'nal Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services*, 2007.
6. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proc. KR*, 2004.
7. E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *Proc. LICS*, 1999.
8. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *Proc. WWW*, 2003.
9. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Open answer set programming with guarded programs. *ACM TOCL*, 4, 2008.
10. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantics web rule language combining OWL and RuleML, 2004. W3C Submission, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.
11. Maurizio Lenzerini. Tbox and abox reasoning in expressive description logics. In *In Proc. of KR*, pages 316–327. Morgan Kaufmann, 1996.
12. Thomas Lukasiewicz. A novel combination of answer set programming with description logics for the semantic web. In *Proc. of European Semantic Web Conference (ESWC)*, 2007.
13. Maarten Mariën, David Gilis, and Marc Denecker. On the relation between ID-Logic and Answer Set Programming. In *Proc. of JELIA*, 2004.
14. David Mitchell and Eugenia Ternovska. A framework for representing and solving np search problems. In *Proc. of AAAI*, 2005.
15. B. Motik and R. Rosati. A faithful integration of description logics with logic programming. In *Proc. of IJCAI*, 2007.
16. B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. In *Proc. of the 3rd International Semantic Web Conference (ISWC)*, 2004.
17. Riccardo Rosati. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. KR*, pages 68–78, 2006.
18. M. Vanden Bossche, P. Ross, I. MacLarty, B. Van Nuffelen, and N. Pelov. Ontology driven software engineering for real life applications. In *Proc. SWESE)*, 2007.
19. Joost Vennekens and Marc Denecker. An algebraic account of modularity in ID-logic. In *Proc. LPNMR*, 2005.

Classical Logic Event Calculus as Answer Set Programming

Joohyung Lee and Ravi Palla

School of Computing and Informatics
Arizona State University, Tempe, AZ, USA
{joolee, Ravi.Palla}@asu.edu

Abstract. Recently, Ferraris, Lee and Lifschitz presented a generalized definition of a stable model that applies to the syntax of arbitrary first-order sentences, under which a logic program is viewed as a special class of first-order sentences. The new definition of a stable model is similar to the definition of circumscription, and can even be characterized in terms of circumscription. In this paper, we show the opposite direction, that is, how to embed circumscription into the new stable model semantics, and based on this, how to turn some versions of the classical logic event calculus into the general language of stable models. By turning the latter to answer set programs under certain conditions, we show that answer set solvers can be used for classical logic event calculus reasoning, allowing more expressive query answering than what can be handled by the current SAT-based implementations of the event calculus. We prove the correctness of our translation method and compare our work with the related work by Mueller.

1 Introduction

Recently, Ferraris, Lee and Lifschitz [1] presented a generalized definition of a stable model that applies to the syntax of arbitrary first-order sentences. Under this framework, a logic program is viewed as a special class of first-order sentences, in which negation as failure (not) is identified with classical negation ($\neg$) under the stable model semantics. The new definition of a stable model is given by a translation into second-order logic, and does not refer to grounding to define the meaning of variables. This allowed to lift the notion of stable models to a special class of first-order models, not restricted to Herbrand models. The new definition is similar to the definition of circumscription [2; 3], and was even characterized in terms of circumscription [1], extending the work by Lin [4]. The same characterization was also independently given in [5].

The opposite direction, turning (parallel) circumscription into the stable model semantics, was shown in [6], limited to the propositional case. In this paper, we start with generalizing this result: turning first-order circumscription into the generalized language of stable models. This leads to the following natural question: how are the formalisms for reasoning about actions and change that are based on circumscription, related to the stable model semantics? Recall that as nonmonotonic formalisms, circumscription and the stable model semantics

have served to provide (different) solutions to the frame problem. A group of action formalisms, such as the classical logic event calculus [7] and temporal action logic [8], take (monotonic) first-order logic as the basis, augmented with circumscription to handle the frame problem. On the other hand, action language $\mathcal{A}$ and many of its descendants [9] refer to logic programs under the stable model semantics (a.k.a. answer set programs) as the underlying formalism. Although there have been some papers that relate classical logic based action formalisms to each other (e.g., [10; 11]), not much work was done in relating them to action languages and to answer set programs.

As an initial step, we show how to turn the classical logic event calculus into the general language of stable models. Note that the event calculus is a family of languages with some variance. Here we consider versions of the event calculus that are based on classical logic, one defined by Miller and Shanahan [12], and the other by Mueller [13], which is a simplified version of the former. The fact that circumscription can be reduced to completion [14] under certain syntactic conditions ([15, Proposition 2]) allowed efficient satisfiability solvers (SAT) to be used for event calculus reasoning [16; 13], similar to the idea of SAT-based answer set programming. Interestingly, early versions of the event calculus [17] were based on logic programs but this was the time before the invention of the stable model semantics, while more extensive later developments of the event calculus were carried out under the classical logic setting. Our work here can be viewed as turning back to the logic program tradition, in the modern form of answer set programming. This is not only interesting from a theoretical perspective, but also interesting from a computational perspective, as it allows answer set solvers to be used for event calculus reasoning. In contrast to the SAT-based approaches from [16; 13] which rely on completion and hence cannot allow certain recursive axioms in the event calculus, we show that the answer set programming approach handles all the axioms correctly, modulo grounding. Our work shows that the new language of stable models is a suitable nonmonotonic formalism as general as circumscription to be applied in commonsense reasoning, with the unique advantage of having efficient ASP solvers as computational tools.

Our work is motivated by Erik Mueller's work that is available on the webpage `http://decreasoner.sourceforge.net/csr/ecas/`, where a few example answer set programs were used to illustrate that event calculus like reasoning can be done in answer set programming. However, this was a kind of "proof of concept" [1] and no formal justification was provided.

The paper is organized as follows. In the following two sections, we review the syntax of the event calculus and the generalized language of stable models. In Section 4, we present the language RASPL^M ("Many-sorted extension of Reductive Answer Set Programming Language"), for which syntactically similar codes are accepted by LPARSE,[2] the front-end of SMODELS and several other answer set solvers. We show how to turn circumscription into the first-order language of stable models in Section 5, and how to turn a description in the event calculus into a RASPL^M program in Section 6. We compare our method with Mueller's work in Section 7.

[1] Personal communication with Erik Mueller.

[2] `http://www.tcs.hut.fi/Software/smodels`

2 Review of the Event Calculus

Since the notion of equivalence under classical logic is weaker than the notion of equivalence under the stable model semantics, classically equivalent formulas do not necessarily have the same stable models. Thus any translation from classical logic based formalisms into the stable model semantics will need to fix the syntax of the former. Here we follow the syntax of the classical logic event calculus as described in [18, Chapter 2].

We assume a many-sorted first-order language, which contains an *event* sort, a *fluent* sort, and a *timepoint* sort. A *fluent term* is a term whose sort is a fluent, an *event term* is a term whose sort is an event and a *timepoint term* is a term whose sort is a time point. A *condition* in the event calculus is defined recursively as follows:

- A comparison ($\tau_1 < \tau_2$, $\tau_1 \leq \tau_2$, $\tau_1 \geq \tau_2$, $\tau_1 > \tau_2$, $\tau_1 = \tau_2$, $\tau_1 \neq \tau_2$) for terms τ_1, τ_2 is a condition;
- If f is a fluent term and t is a timepoint term, then $\mathit{HoldsAt}(f,t)$ and $\neg\mathit{HoldsAt}(f,t)$ are conditions;
- If γ_1 and γ_2 are conditions, then $\gamma_1 \wedge \gamma_2$ and $\gamma_1 \vee \gamma_2$ are conditions;
- If v is a variable and γ is a condition, then $\exists v\gamma$ is a condition.

In all the subsequent sections, we will use e and e_i to denote event terms, f and f_i to denote fluent terms, t and t_i to denote timepoint terms, and γ and γ_i to denote conditions. We understand formula $F \leftrightarrow G$ as shorthand for $(F \rightarrow G) \wedge (G \rightarrow F)$; formula $\top$ as shorthand for $\bot \rightarrow \bot$; formula $\neg F$ as shorthand for $F \rightarrow \bot$.

An event calculus domain description is defined as

$$\mathrm{CIRC}[\Sigma\ ;\ \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \mathrm{CIRC}[\Delta_1 \wedge \Delta_2\ ;\ \mathit{Happens}]$$
$$\wedge \mathrm{CIRC}[\Theta\ ;\ \mathit{Ab}_1, \ldots, \mathit{Ab}_n] \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge E$$

where

- Σ is a conjunction of axioms of the form

  $$\gamma \rightarrow \mathit{Initiates}(e,f,t)$$
  $$\gamma \rightarrow \mathit{Terminates}(e,f,t)$$
  $$\gamma \rightarrow \mathit{Releases}(e,f,t)$$
  $$\gamma \wedge \pi_1(e,f_1,t) \rightarrow \pi_2(e,f_2,t) \qquad \text{(“effect constraint”)}$$
  $$\gamma \wedge [\neg]\mathit{Happens}(e_1,t) \wedge \cdots \wedge [\neg]\mathit{Happens}(e_n,t) \rightarrow \mathit{Initiates}(e,f,t)$$
  $$\gamma \wedge [\neg]\mathit{Happens}(e_1,t) \wedge \cdots \wedge [\neg]\mathit{Happens}(e_n,t) \rightarrow \mathit{Terminates}(e,f,t)$$

  where each π_1 and π_2 is either *Initiates* or *Terminates*;
- Δ_1 is a conjunction of axioms of the form $\mathit{Happens}(e,t)$ and *temporal ordering formulas* which are comparisons between timepoint terms;
- Δ_2 is a conjunction of axioms of the form

  $$\gamma \rightarrow \mathit{Happens}(e,t)$$
  $$\sigma(e,t) \wedge \pi_1(e_1,t) \wedge \cdots \wedge \pi_n(e_n,t) \rightarrow \mathit{Happens}(e,t)$$
  $$\mathit{Happens}(e,t) \rightarrow \mathit{Happens}(e_1,t) \vee \cdots \vee \mathit{Happens}(e_n,t) \qquad \text{(“disjunctive event axiom”)}$$

where σ is *Started* or *Stopped* and each π_j $(1 \leq j \leq n)$ is either *Initiated* or *Terminated*. Predicates *Started*, *Stopped*, *Initiated* and *Terminated* are defined as follows:

$$Started(f,t) \overset{def}{\leftrightarrow} (HoldsAt(f,t) \vee \exists e(Happens(e,t) \wedge Initiates(e,f,t))) \quad (CC_1)$$

$$Stopped(f,t) \overset{def}{\leftrightarrow} (\neg HoldsAt(f,t) \vee \exists e(Happens(e,t) \wedge Terminates(e,f,t))) \quad (CC_2)$$

$$Initiated(f,t) \overset{def}{\leftrightarrow} (Started(f,t) \vee \neg\exists e(Happens(e,t) \wedge Terminates(e,f,t))) \quad (CC_3)$$

$$Terminated(f,t) \overset{def}{\leftrightarrow} (Stopped(f,t) \vee \neg\exists e(Happens(e,t) \wedge Initiates(e,f,t))) \quad (CC_4)$$

- Θ is a conjunction of axioms of the form $\gamma \rightarrow Ab_i(\ldots,t)$;
- Ω is a conjunction of unique name axioms ;
- Ψ is a conjunction of axioms of the form [3]

$$\gamma, \quad \gamma_1 \rightarrow \gamma_2, \quad \gamma_1 \leftrightarrow \gamma_2$$
$$Happens(e,t) \rightarrow \gamma$$
$$Happens(e_1,t) \wedge \gamma \wedge [\neg]Happens(e_2,t) \rightarrow \bot \ ;$$

- Π is a conjunction of trajectory axioms and anti-trajectory axioms of the form
$$\gamma \rightarrow (Anti)Trajectory(f_1,t_1,f_2,t_2) \ ;$$
- Γ is a conjunction of observations of the form $HoldsAt(f,t)$ and $ReleasedAt(f,t)$;
- E is a conjunction of the event calculus axioms *DEC* or *EC*. [4]

As shown, a classical logic event calculus description may contain existential quantifiers; some parts of the description are circumscribed on a partial list of predicates, while some others are not circumscribed. These features look different from logic programs. Nonetheless we show that the classical logic event calculus can be embedded into logic programs.

3 Review of the New Stable Model Semantics and the New Splitting Theorem

Under the new definition of stable models presented in [19] that is applicable to arbitrary first-order sentences, a logic program is identified as a universal formula, called the *FOL-representation*. First, we identify the logical connectives—the comma, the semicolon, and *not* with their counterparts in classical logic $\wedge$,

[3] The last formula is a minor rewriting of the formula from [18] which is $Happens(e_1,t) \wedge \gamma \rightarrow [\neg]Happens(e_2,t)$. This rewriting simplifies the later presentation.

[4] Due to lack of space, we refer the reader to [18, Chapter 2] for these axioms.

$\vee$ and $\neg$. The *FOL-representation of a rule Head* $\leftarrow$ *Body* is the universal closure of the implication *Body* $\rightarrow$ *Head*. The *FOL-representation of a program* is the conjunction of the FOL-representations of its rules. For example, the FOL-representation of the program

$$\begin{array}{l} p(a) \\ q(b) \\ r(x) \leftarrow p(x), \mathit{not}\ q(x) \end{array}$$

is

$$p(a) \wedge q(b) \wedge \forall x((p(x) \wedge \neg q(x)) \rightarrow r(x)) \tag{1}$$

We review the new definition of stable models from [19]. Let $\mathbf{p}$ be a list of distinct predicate constants $p_1, \ldots, p_n$, and let $\mathbf{u}$ be a list of distinct predicate variables $u_1, \ldots, u_n$ of the same length as $\mathbf{p}$. By $\mathbf{u} = \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \leftrightarrow p_i(\mathbf{x}))$, where $\mathbf{x}$ is a list of distinct object variables of the same arity as the length of p_i, for all $i = 1, \ldots n$. By $\mathbf{u} \leq \mathbf{p}$ we denote the conjunction of the formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \rightarrow p_i(\mathbf{x}))$ for all $i = 1, \ldots n$, and $\mathbf{u} < \mathbf{p}$ stands for $(\mathbf{u} \leq \mathbf{p}) \wedge \neg(\mathbf{u} = \mathbf{p})$.

For any first-order sentence $F(\mathbf{p})$, expression $\mathrm{SM}[F; \mathbf{p}]$ 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 $p_1, \ldots, p_n$ of predicate constants that are called *intensional*, $\mathbf{u}$ is a list $u_1, \ldots, u_n$ of distinct predicate variables corresponding to $\mathbf{p}$, and $F^*(\mathbf{u})$ is defined recursively:

– $$p_i(t_1, \ldots, t_m)^* = \begin{cases} u_i(t_1, \ldots, t_m) & \text{if } p_i \text{ belongs to } \mathbf{p}, \\ p_i(t_1, \ldots, t_m) & \text{otherwise;} \end{cases}$$

– $(t_1 = t_2)^* = (t_1 = t_2)$;
– $\bot^* = \bot$;
– $(F \wedge G)^* = F^* \wedge G^*$;
– $(F \vee G)^* = F^* \vee G^*$;
– $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
– $(\forall x F)^* = \forall x F^*$;
– $(\exists x F)^* = \exists x F^*$.

As before, we understand formula $F \leftrightarrow G$ as shorthand for $(F \rightarrow G) \wedge (G \rightarrow F)$; formula $\top$ as shorthand for $\bot \rightarrow \bot$; formula $\neg F$ as shorthand for $F \rightarrow \bot$.

$\mathrm{SM}[F]$ defined in [1] is identical to $\mathrm{SM}[F; \mathbf{p}]$ where intensional predicate constants $\mathbf{p}$ range over all predicate constants that occur in F. According to [1], the models of $\mathrm{SM}[F]$ whose signature σ consists of the object, function and predicate constants occurring in F are called the *stable models* of F. Among those stable models we call the Herbrand models of signature σ, the *answer sets* of F. The definition of stable models is closely related to the definition of quantified equilibrium model [20; 1]. The answer sets of a logic program Π are defined as the answer sets of the FOL-representation of Π. Proposition 1 from [1] shows that, for normal logic programs, this definition is equivalent to the definition of answer sets from [21].

As shown in [19], the extended notion of SM by a partial list of intensional predicates is not essential in the sense that it can be rewritten so that intensional predicates become exactly those that occur in the formula. By $Choice(\mathbf{p})$ we denote the conjunction of "choice formulas" $\forall \mathbf{x}(p(\mathbf{x}) \vee \neg p(\mathbf{x}))$ for all predicate constants p in $\mathbf{p}$ where $\mathbf{x}$ is a list of distinct variables whose length is the same as the arity of p; by $False(\mathbf{p})$ we denote the conjunction of $\forall \mathbf{x} \neg p(\mathbf{x})$ for all predicate constants p in $\mathbf{p}$; by $pr(F)$ we denote the list of all predicate constants occurring in F.

Proposition 1 ([19])

$$\mathrm{SM}[F; \mathbf{p}] \leftrightarrow \mathrm{SM}[F \wedge Choice(pr(F) \setminus \mathbf{p})] \wedge False(\mathbf{p} \setminus pr(F))$$

is logically valid.

However, it is convenient to describe our main results and the following splitting theorem using the generalized notion of SM.

Recall that the occurrence of one formula in another is called *positive* if the number of implications containing that occurrence in the antecedent is even, and *negative* otherwise. We say that an occurrence of a subformula or a predicate constant in a formula F is *strictly positive* if the number of implications in F containing that occurrence in the antecedent is 0. For example, in (1), both occurrences of q are positive, but only the first is strictly positive. By the head predicates of F, denoted by $h(F)$, we mean the set of predicate constants that have at least one strictly positive occurrence in F. We call a formula *negative* if it has no strictly positive occurrences of predicate constants. We say that a predicate constant p *depends* on a predicate constant q in an implication $G \to H$ if

- p has a strictly positive occurrence in H, and
- q has a positive occurrence in G that does not belong to any occurrence of a negative formula in G.

The *predicate dependency graph* of a formula F is the directed graph such that

- its vertices are the predicate constants occurring in F, and
- it has an edge from a vertex p to a vertex q if p depends on q in an implication that has a strictly positive occurrence in F.

A nonempty finite subset $\mathbf{l}$ of V is called a *loop* of F if the subgraph of the predicate dependency graph of F induced by $\mathbf{l}$ is strongly connected.

We say that F and G *interact* on $\mathbf{p}$ if $F \wedge G$ has a loop $\mathbf{l}$ such that

- $\mathbf{l}$ is contained in $\mathbf{p}$,
- $\mathbf{l}$ contains an element of $h(F)$, and
- $\mathbf{l}$ contains an element of $h(G)$.

The following theorem shows how formula $\mathrm{SM}[F \wedge G; \mathbf{p}]$ can be split :

Theorem 1 ([22]) *If F and G don't interact on $\mathbf{p}$, then $\mathrm{SM}[F \wedge G; \mathbf{p}]$ is equivalent to*

(a) $\mathrm{SM}[F; \mathbf{p} \setminus h(G)] \wedge \mathrm{SM}[G; \mathbf{p} \setminus h(F)]$, *and to*
(b) $\mathrm{SM}[F; \mathbf{p} \setminus h(G)] \wedge \mathrm{SM}[G; \mathbf{p} \cap h(G)]$, *and to*
(c) $\mathrm{SM}[F; \mathbf{p} \cap h(F)] \wedge \mathrm{SM}[G; \mathbf{p} \cap h(G)] \wedge \mathit{False}(p \setminus h(F) \setminus h(G))$.

The theorem will be used to justify our translation method.

4 RASPL^M Programs

The definition of SM above can be easily extended to many-sorted first-order languages, similar to the extension of circumscription to many-sorted first-order languages (Section 2.4 of [15]). We define RASPL^M programs as a special class of sentences under this extension, which are essentially a many-sorted extension of RASPL-1 programs from [23]. We assume that the underlying signature contains an integer sort and contains several built-in symbols, such as integer constants, built-in arithmetic functions $+$, $-$, and comparison operators $<$, $\leq$, $>$ $\geq$. Since we do not need counting aggregates in this paper, for simplicity, we will assume that every "aggregate expression" is an atom or a negated atom. That is, a *rule* is an expression of the form

$$A_1 \,;\ldots;\, A_k \leftarrow\ A_{k+1}, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n, \\ \mathit{not}\ \mathit{not}\ A_{n+1}, \ldots, \mathit{not}\ \mathit{not}\ A_p$$

($0 \leq k \leq m \leq n \leq p$), where each A_i is an atom, possibly equality or comparisons. A *program* is a finite list of rules.

The "choice rule" of the form $\{A\} \leftarrow\ \mathit{Body}$ where A is an atom, stands for $A \leftarrow\ \mathit{Body}, \mathit{not}\ \mathit{not}\ A$.

The semantics of a RASPL^M program is understood by turning it into its corresponding many-sorted FOL-representation, as in RASPL-1. The integer constants and built-in symbols will be evaluated in the standard way, and we will consider only those "standard" interpretations. The answer sets of a RASPL^M program are the Herbrand interpretations of the signature consisting of object, function and predicate constants occurring in the program, that satisfies $\mathrm{SM}[F]$, where F is the FOL-representation of the program.

Though RASPL^M programs have no implementation, syntactically similar codes are accepted by LPARSE, whose language is essentially many-sorted.

5 Turning Circumscription to SM

Definition 1. *For any list* $\mathbf{p}$ *of predicate constants, and any formulas* G *and* H *in each of which every occurrence of predicate constants from* $\mathbf{p}$ *is strictly positive, we call implication* $G \rightarrow H$ canonical *w.r.t.* $\mathbf{p}$.

Proposition 2 *Let* F *be the universal closure of a conjunction of canonical implications w.r.t.* $\mathbf{p}$. *Then*

$$\mathrm{SM}[F; \mathbf{p}] \leftrightarrow \mathrm{CIRC}[F; \mathbf{p}]$$

is logically valid.

Note that in the syntax of the event calculus described in Section 2, all axioms in Σ are already canonical implications w.r.t. *Initiates*, *Terminates*, *Releases*; all axioms in $\Delta_1 \wedge \Delta_2$ are canonical implications w.r.t. *Happens*; all axioms in Θ are canonical implications w.r.t. Ab_i.[5]

The proof of Proposition 2 is immediate from the following lemma, which can be proved by induction.

Lemma 1. *For any formula F in which every occurrence of predicate constants from $\mathbf{p}$ is strictly positive,*

$$(\mathbf{u} \leq \mathbf{p}) \rightarrow (F^*(\mathbf{u}) \leftrightarrow F(\mathbf{u}))$$

is logically valid, where $\mathbf{u}$ is a list of distinct predicate variables of the same length as $\mathbf{p}$.

6 Turning Event Calculus Descriptions to SM

Theorem 2 *Given an event calculus description, let F be the conjunction of $\Omega, \Psi, \Pi, \Gamma$ and E, and let $\mathbf{p}$ be the set of all predicates (other than equality and comparisons) occurring in the event calculus description. The following theories are equivalent:*

(a) $\text{CIRC}[\Sigma; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \text{CIRC}[\Delta; \mathit{Happens}] \wedge \text{CIRC}[\Theta; Ab_1, \ldots, Ab_n] \wedge F$;

(b) $\text{SM}[\Sigma; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \text{SM}[\Delta; \mathit{Happens}] \wedge \text{SM}[\Theta; Ab_1, \ldots, Ab_n] \wedge F$;

(c) $\text{SM}[\Sigma \wedge \Delta \wedge \Theta \wedge F; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}, \mathit{Happens}, Ab_1, \ldots, Ab_n]$;

(d) $\text{SM}[\Sigma \wedge \Delta \wedge \Theta \wedge F \wedge \mathit{Choice}(\mathbf{p} \backslash \{\mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}, \mathit{Happens}, Ab_1, \ldots, Ab_n\})]$.

Proof. *Between (a) and (b):* Follows immediately from Proposition 2.

Between (b) and (c): Note first that F is equivalent to $\text{SM}[F; \emptyset]$. Since Σ, Δ, Θ, F do not interact on $\{\mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}, \mathit{Happens}, Ab_1, \ldots, Ab_n\}$, from Theorem 1 (b) (applying it multiple times), it follows that (b) and (c) are equivalent.

Between (c) and (d): Follows immediately from Proposition 1. ∎

6.1 Turning Event Calculus Descriptions to RASPL^M Programs

The formulas in Theorem 2 may still contain existential quantifiers, which are not allowed in RASPL^M programs. The following procedure turns an event calculus description into a RASPL^M program by eliminating existential quantifiers using new atoms.

Definition 2 (Translation EC2ASP).

[5] We understand an axiom such as $\mathit{Happens}(e, t)$ as an abbreviation for implication $\top \rightarrow \mathit{Happens}(e, t)$.

1. *Simplify all the definitional axioms of the form*

$$\forall \mathbf{x}(p(\mathbf{x}) \overset{def}{\leftrightarrow} \exists \mathbf{y} G(\mathbf{x}, \mathbf{y})) \tag{2}$$

except for $CC_1 - CC_4$, where $\mathbf{y}$ is a list of all free variables in G that are not in $\mathbf{x}$, as $\forall \mathbf{xy}(G(\mathbf{x}, \mathbf{y}) \rightarrow p(\mathbf{x}))$.

2. *For each axiom that contains existential quantifiers, repeat the following until there are no existential quantifiers:*
   (a) *Replace maximal negative occurrences of $\exists y G(y)$ in the axiom by $G(z)$ where z is a new variable.*
   (b) *Replace maximal positive occurrences of $\exists y G(\mathbf{x}, y)$ in the axiom, where $\mathbf{x}$ is the list of all free variables of $\exists y G(\mathbf{x}, y)$, by the formula $\neg\neg p_G(\mathbf{x})$ where p_G is a new predicate constant, and add the axiom*

$$\forall \mathbf{x} y(G(\mathbf{x}, y) \rightarrow p_G(\mathbf{x})). \tag{3}$$

3. *Add choice formulas $\forall x(p(\mathbf{x}) \vee \neg p(\mathbf{x}))$ for all the predicate constants p except for $\{Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n, \mathbf{p_1}, \mathbf{p_2}\}$ where*
   - *$\mathbf{p_1}$ is a list of all predicate constants p considered in Step 1.*
   - *$\mathbf{p_2}$ is a list of all new predicate constants p_G introduced in Step 2.*
4. *Apply the conversion from [24] that turns programs with nested expressions into disjunctive logic programs.*

For example, consider *DEC5* axiom:

$$\begin{aligned} \forall f t((&HoldsAt(f,t) \wedge \neg ReleasedAt(f,t{+}1) \wedge \\ &\neg\exists e(Happens(e,t) \wedge Terminates(e,f,t))) \rightarrow HoldsAt(f,t{+}1)). \end{aligned} \tag{4}$$

In order to eliminate the positive occurrence of $\exists e(Happens(e,t) \wedge Terminates(e,f,t))$ in the formula, we apply Step 2(b), introducing the formula

$$\forall e f t(Happens(e,t) \wedge Terminates(e,f,t) \rightarrow q(f,t)),$$

and replacing (4) with

$$\forall f t((HoldsAt(f,t) \wedge \neg ReleasedAt(f,t+1) \wedge \neg\neg\neg q(f,t)) \rightarrow HoldsAt(f,t+1)),$$

from which $\neg\neg\neg q(f,t)$ is simplified as $\neg q(f,t)$ by Step 4.[6]

We will present the proof in the next section. Here we attempt to give the idea of the translation. Step 1 can be dropped without affecting the correctness, but it yields a more succinct transformation. The simplification does not apply for $CC_1 - CC_4$ since these axioms, together with other axioms in the description, may yield loops (A more detailed explanation follows in the next section). Step 2 (a) is one of the steps in prenex normal form conversion. Instead of Skolemization, which will introduce an infinite Herbrand universe, Step 2 (b) eliminates existential quantifiers using new atoms. The transformation yields a set of implications where each antecedent and consequent are formed

[6] In general we put $\neg\neg$ in front of $p_G(\mathbf{y})$ in order to prevent from introducing unnecessary loops. A more precise explanation is given in the proof of Theorem 3.

from atoms by allowing $\neg$, $\wedge$, and $\vee$ nested arbitrarily, similar to the syntax of a program with nested expressions from [24]. The transformation that turns a program with nested expressions into a disjunctive logic program from [24] can be straightforwardly extended to turn these set of implications into a RASPL^M program.

Turning the resulting RASPL^M program further into the input language of LPARSE requires minor rewriting, such as moving equality or negated atoms to the body (e.g., $\neg p(\mathbf{t}) \leftarrow \ldots$ into $\leftarrow \ldots, p(\mathbf{t})$), and adding domain predicates in the body for all variables occurring in the rule. [7]

6.2 Proof of the Correctness of the Translation

The following theorem states the correctness of the translation.

Theorem 3 *Let T be an event calculus domain description, and let Π be a RASPL^M program obtained by applying the translation* EC2ASP *to T. The stable models of Π restricted to the signature of T are precisely the models of T.*

We will use the following fact for the proof.

Lemma 2. *Let F be a first-order formula, let p be a predicate constant not occurring in F, let $G(\mathbf{x})$ be a subformula of F where $\mathbf{x}$ is the list of all free variables of $G(\mathbf{x})$, and let F' be a formula obtained from F by replacing an occurrence of $G(\mathbf{x})$ with $\neg\neg p(\mathbf{x})$. The models of $F' \wedge \forall \mathbf{x}(G(\mathbf{x}) \leftrightarrow p(\mathbf{x}))$ restricted to the signature of F are exactly the models of F.*

We will also use the proposition below that relates SM to completion, extending the results of Propositions 6,8 from [1].

Let Π be a finite set of rules that have the form

$$A \leftarrow F \tag{5}$$

where A is an atom and F is a first-order formula (that may contain quantifiers). We say that Π is in normal form w.r.t. a set $\mathbf{p}$ of predicate constants if, for each predicate constant p in $\mathbf{p}$, there is exactly one rule

$$p(\mathbf{x}) \leftarrow F \tag{6}$$

where $\mathbf{x}$ is the list of object variables whose length is the same as the arity of p and F is a formula. It is clear that every program whose rules have the form (5) can be turned into a normal form w.r.t $\mathbf{p}$. Given a program Π in normal form w.r.t. $\mathbf{p}$, the *completion* of Π w.r.t $\mathbf{p}$ is the conjunction of the universal closure of formulas obtained from Π by replacing (6) with

$$p(\mathbf{x}) \leftrightarrow \exists \mathbf{y} F$$

where $\mathbf{y}$ is the list of free variables occurring in F that are not in $\mathbf{x}$.

We say that a first-order formula F is tight on $\mathbf{p}$ if the subgraph of the dependency graph of F induced by $\mathbf{p}$ is acyclic.

[7] If we are only interested in answer sets, rather than stable models (note the distinction made in Section 3), UNA axioms can be disregarded.

Proposition 3 *Let Π be a program in normal form w.r.t.* $\mathbf{p}$ *and let F be the FOL-representation of Π. If F is tight on* $\mathbf{p}$*, then* $\mathrm{SM}[F;\mathbf{p}]$ *is equivalent to the completion of Π w.r.t.* $\mathbf{p}$*.*

Proof of Theorem 3 Assume that T is

$$\begin{array}{l}\mathrm{CIRC}[\Sigma; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \mathrm{CIRC}[\Delta; \mathit{Happens}] \\ \wedge\ \mathrm{CIRC}[\Theta; Ab_1, \ldots, Ab_n] \wedge F\ ,\end{array}$$

which is equivalent to

$$\begin{array}{l}\mathrm{SM}[\Sigma; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \mathrm{SM}[\Delta; \mathit{Happens}] \\ \wedge\ \mathrm{SM}[\Theta; Ab_1, \ldots, Ab_n] \wedge F\end{array}$$

by Theorem 2.

Let D_1 be the set of all definitions of the form (2) except for $CC_1 - CC_4$. Let Step $1'$ be the transformation that turns each formula (2) in D_1 into

$$\mathrm{SM}[\forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x}));\ p], \tag{7}$$

and let Step $2'$ be a modification of Step 2 in EC2ASP by introducing

$$\forall \mathbf{x}y(G(\mathbf{x},y) \leftrightarrow p_G(\mathbf{x})) \tag{8}$$

instead of (3) in Step 2 (b).

Let Σ', Δ', Θ', F' be the formulas obtained from Σ, Δ, Θ, F by applying Steps $1'$ and $2'$. By Proposition 3, Step $1'$ is an equivalent transformation. By Lemma 2, the models of

$$\begin{array}{l}\mathrm{SM}[\Sigma'; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \mathrm{SM}[\Delta'; \mathit{Happens}] \\ \wedge\ \mathrm{SM}[\Theta'; Ab_1, \ldots, Ab_n] \wedge F'\end{array} \tag{9}$$

restricted to the signature of T are precisely the models of T. Let D_2 be the set of all definitions (8) that are introduced in Step $2'$. Note that these formulas are introduced only for some formulas in F (other than D_1), and not for Σ, Δ and Θ; formulas Σ', Δ' and Θ' are obtained by applying Step 2 (a) only since, according to the syntax of the event calculus (Section 2), every occurrence of existential quantification is negative in each of Σ, Δ and Θ. Let F'' be the axioms in F' excluding D_2 and all formulas (7) for D_1.

By Proposition 3 again, each formula (8) in D_2 is equivalent to

$$\mathrm{SM}[\forall \mathbf{x}y(G(\mathbf{x},y) \rightarrow p_G(\mathbf{x}));\ p_G].$$

Consequently (9) is equivalent to

$$\begin{array}{l}\mathrm{SM}[\Sigma'; \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \mathrm{SM}[\Delta'; \mathit{Happens}] \\ \wedge\ \mathrm{SM}[\Theta'; Ab_1, \ldots, Ab_n] \wedge \mathrm{SM}[F''; \emptyset] \\ \wedge \bigwedge_{(2)\in D_1} \mathrm{SM}[\forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x}));\ p] \\ \wedge \bigwedge_{(8)\in D_2} \mathrm{SM}[\forall \mathbf{x}y(G(\mathbf{x},y) \rightarrow p_G(\mathbf{x}));\ p_G].\end{array} \tag{10}$$

Since Σ' and Δ' do not interact on $\{Initiates, Terminates, Releases, Happens\}$, by Theorem 1 (b), formula (10) is equivalent to

$$\begin{array}{l} \mathrm{SM}[\Sigma' \wedge \Delta'; Initiates, Terminates, Releases, Happens] \\ \wedge\ \mathrm{SM}[\Theta'; Ab_1, \ldots, Ab_n] \wedge \mathrm{SM}[F''; \emptyset] \\ \wedge \bigwedge_{(2)\in D_1} \mathrm{SM}[\forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x}));\ p] \\ \wedge \bigwedge_{(8)\in D_2} \mathrm{SM}[\forall \mathbf{x}y(G(\mathbf{x},y) \rightarrow p_G(\mathbf{x}));\ p_G]. \end{array} \tag{11}$$

Similarly, by applying Theorem 1 multiple times, it is clear that formula (11) is equivalent to

$$\begin{array}{l} \mathrm{SM}[\Sigma' \wedge \Delta' \wedge \Theta' \wedge F''; Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n] \\ \wedge\ \mathrm{SM}[\bigwedge_{(2)\in D_1} \forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x}));\ \mathbf{p_1}] \\ \wedge\ \mathrm{SM}[\bigwedge_{(8)\in D_2} \forall \mathbf{x}y(G(\mathbf{x},y) \rightarrow p_G(\mathbf{x}));\ \mathbf{p_2}] \end{array} \tag{12}$$

where $\mathbf{p_1}$ is a list of all predicate constants p defined in D_1, and $\mathbf{p_2}$ is a list of all new predicate constants p_G defined in D_2.

According to the syntax of the event calculus (Section 2), $\Sigma' \wedge \Delta' \wedge \Theta' \wedge F''$ and

$$\bigwedge_{(2)\in D_1} \forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x}))$$

do not interact on

$$\{Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n, \mathbf{p_1}\},$$

so that, by Theorem 1 (b), (12) is equivalent to

$$\begin{array}{l} \mathrm{SM}[\Sigma' \wedge \Delta' \wedge \Theta' \wedge F'' \wedge \bigwedge_{(2)\in D_1} \forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x})); \\ \qquad Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n, \mathbf{p_1}] \\ \wedge\ \mathrm{SM}[\bigwedge_{(8)\in D_2} \forall \mathbf{x}y(G(\mathbf{x},y) \rightarrow p_G(\mathbf{x})); \mathbf{p_2}]. \end{array} \tag{13}$$

From the fact that every occurrence of a predicate constant from $\mathbf{p_2}$ in F'' is preceded with $\neg\neg$, we conclude that the formula in the first SM and the formula in the second SM in (13) do not interact on

$$\{Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n, \mathbf{p_1}, \mathbf{p_2}\}$$

so that, by Theorem 1 (b), (13) is equivalent to

$$\begin{array}{l} \mathrm{SM}[\Sigma' \wedge \Delta' \wedge \Theta' \wedge F'' \wedge \bigwedge_{(2)\in D_1} \forall \mathbf{xy}(G(\mathbf{x},\mathbf{y}) \rightarrow p(\mathbf{x})) \\ \wedge \bigwedge_{(8)\in D_2} \forall \mathbf{x}y(G(\mathbf{x},y) \rightarrow p_G(\mathbf{x})); \\ Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n, \mathbf{p_1}, \mathbf{p_2}] \end{array} \tag{14}$$

Formula (14) is exactly the formula obtained from Steps 1 and 2. The rest of the proof follows immediately from Proposition 1, and a straightforward extension of Proposition 7 from [24]. ∎

If we were to treat $CC_1 - CC_4$ same as the other definitional axioms, then Theorem 1 (b) won't justify that (12) is equivalent to (13), since there may be a

loop, such as {*Started*,*Happens*,*Initiated*}, on which Δ' and $CC_1 - CC_4$ interact. Indeed, (12) and (13) are not equivalent in general if $CC_1 - CC_4$ were regarded to belong to D_1.

7 Comparison with Mueller's Work

7.1 Comparison with Mueller's ASP Approach

The answer set programming approach on the webpage

`http://decreasoner.sourceforge.net/csr/ecas/`

illustrates the idea using some examples only, and misses formal justification. Still we observe a few differences.[8]

First, these examples use classical negation, while our method does not. We do not need both negations—negation as failure (*not*) and classical negation (¬)—to embed the two-valued event calculus into answer set programming. Second, no choice rules were used, which resulted in limiting attention to temporal projection problems—to determine the states that result from performing a sequence of actions. On the other hand, our approach can handle not only temporal projection problems, but also planning and postdiction problems. To solve a planning problem *Happens* should not be minimized. Adding choice rules for *Happens* is a way to exempt it from minimization in logic programs. The following example is a logic program counterpart of the planning problem example on page 244 of [18].

```
agent(james).
fluent(awake(A)) :- agent(A).
event(wakeUp(A)) :- agent(A).
initiates(wakeUp(A), awake(A), T) :- agent(A).
:- holdsAt(awake(james), 0).
holdsAt(awake(james), 1).
:- releasedAt(F, 0).
0 {happens(E, T)} 1 :- T<1.
```

When the above program along with the *DEC* axioms is provided as input, SMODELS returns an answer set that contains `happens(wakeup(james), 0)`.

More examples can be found from `http://reasoning.eas.asu.edu/ecasp`.

7.2 Comparison with the *DEC* Reasoner

The *DEC* reasoner[9] is an implementation of the event calculus written by Erik Mueller. The system reduces event calculus reasoning into satisfiability and calls SAT solvers [25]. Since circumscription is not always reducible to completion, some event calculus axioms like effect constraints and disjunctive event axioms

[8] Our version of EC/DEC axioms is available at `http://reasoning.eas.asu.edu/ecasp`.

[9] `http://decreasoner.sourceforge.net/`.

(Section 2) cannot be handled by the *DEC* reasoner. For example, consider the following event calculus axioms that describe the indirect effects of the agent walking from one room to another on the objects that he is holding:

$$\begin{array}{l} HoldsAt(Holding(a,o),t) \wedge Initiates(e,InRoom(a,r),t) \\ \qquad \rightarrow Initiates(e,InRoom(o,r),t) \\ HoldsAt(Holding(a,o),t) \wedge Terminates(e,InRoom(a,r),t) \\ \qquad \rightarrow Terminates(e,InRoom(o,r),t) \end{array} \tag{15}$$

Since these axioms involve non-trivial loops, they cannot be reduced to completion. On the other hand, the logic program corresponding to (15) can be directly handled by answer set solvers.

8 Conclusion

Our contributions are as follows.

- We showed how to embed circumscription into the new language of stable models.
- Based on it we showed how to turn the classical logic event calculus into answer set programs, and proved the correctness of the translation. This approach can handle the *full* version of the event calculus, modulo grounding.

We plan to implement this transformation method, and compare it with the *DEC* reasoner. Another future work is to extend the embedding method to other action formalisms, such as temporal action logics and situation calculus.

Acknowledgements

We are grateful to Vladimir Lifschitz, Erik Mueller and anonymous referees for useful comments on the ideas of this paper and several pointers to earlier work. We are also grateful to Tae-Won Kim for useful discussions. The authors were partially supported by the National Science Foundation under Grant IIS-0839821.

References

1. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). (2007) 372–379
2. McCarthy, J.: Circumscription—a form of non-monotonic reasoning. Artificial Intelligence **13** (1980) 27–39,171–172
3. McCarthy, J.: Applications of circumscription to formalizing common sense knowledge. Artificial Intelligence **26** (1986) 89–116
4. Lin, F.: A Study of Nonmonotonic Reasoning. PhD thesis, Stanford University (1991)

5. Lin, F., Zhou, Y.: From answer set logic programming to circumscription via logic of GK. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). (2007)
6. Lee, J., Lin, F.: Loop formulas for circumscription. Artificial Intelligence **170** (2006) 160–185
7. Shanahan, M.: A circumscriptive calculus of events. Artif. Intell. **77** (1995) 249–284
8. Doherty, P., Gustafsson, J., Karlsson, L., Kvarnström, J.: TAL: Temporal action logics language specification and tutorial.[10] Linköping Electronic Articles in Computer and Information Science ISSN 1401-9841 **3** (1998)
9. Gelfond, M., Lifschitz, V.: Action languages.[11] Electronic Transactions on Artificial Intelligence **3** (1998) 195–210
10. Mueller, E.T.: Event calculus and temporal action logics compared. Artif. Intell. **170** (2006) 1017–1029
11. Belleghem, K.V., Denecker, M., Schreye, D.D.: On the relation between situation calculus and event calculus. J. Log. Program. **31** (1997) 3–37
12. Miller, R., Shanahan, M.: The event calculus in classical logic - alternative axiomatisations. Electron. Trans. Artif. Intell. **3** (1999) 77–105
13. Mueller, E.T.: Event calculus reasoning through satisfiability. J. Log. Comput. **14** (2004) 703–730
14. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press, New York (1978) 293–322
15. Lifschitz, V.: Circumscription. In Gabbay, D., Hogger, C., Robinson, J., eds.: Handbook of Logic in AI and Logic Programming. Volume 3. Oxford University Press (1994) 298–352
16. Shanahan, M., Witkowski, M.: Event calculus planning through satisfiability. J. Log. Comput. **14** (2004) 731–745
17. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
18. Mueller, E.: Commonsense reasoning. Elsevier (2006)
19. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence (2008) To appear.
20. Pearce, D., Valverde, A.: A first order nonmonotonic extension of constructive logic. Studia Logica **80** (2005) 323–348
21. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proceedings of International Logic Programming Conference and Symposium, MIT Press (1988) 1070–1080
22. Ferraris, P., Lee, J., Lifschitz, V., Palla, R.: Two types of splitting in the theory of stable models. Unpublished Draft (2008)
23. Lee, J., Lifschitz, V., Palla, R.: A reductive semantics for counting and choice in answer set programming. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI). (2008) 472–479
24. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence **25** (1999) 369–389
25. Mueller, E.T.: A tool for satisfiability-based commonsense reasoning in the event calculus. In Barr, V., Markov, Z., eds.: FLAIRS Conference, AAAI Press (2004)

[10] `http://www.ep.liu.se/ea/cis/1998/015/` .
[11] `http://www.ep.liu.se/ea/cis/1998/016/` .