Università degli Studi della Calabria

Dipartimento di Matematica

**Dottorato di Ricerca in Matematica ed Informatica**

XX Ciclo

---

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

# Answer Set Programming with Functions

Susanna Cozza

| **Supervisore** | **Coordinatore** |
|---|---|
| Prof. Nicola Leone | Prof. Nicola Leone |

---

Anno Accademico 2007 - 2008

# Answer Set Programming with Functions

**Susanna Cozza**

*Dipartimento di Matematica,*
*Università della Calabria*
87036 *Rende, Italy*
*email : cozza@mat.unical.it*

## Sommario

L'Answer Set Programming (ASP) [Gelfond and Lifschitz, 1988; 1991] è un formalismo che si è affermato, in questi ultimi anni, come potente strumento di programmazione dichiarativa per la rappresentazione della conoscenza e per il ragionamento non monotono. La semantica su cui è basato (denominata *semantica answer set*) estende la semantica dei modelli stabili (usata per i programmi logici 'normali') all'ambito della Programmazione Logica Disgiuntiva (PLD), in cui oltre alla negazione non monotona è consentito anche l'uso della disgiunzione nella testa delle regole.

Nell'ambito dell'ASP, un determinato problema computazionale viene rappresentato tramite un programma PLD che può avere zero o più modelli alternativi chiamati *answer sets*, ognuno dei quali corrisponde ad una possibile visione del dominio modellato.

I linguaggi ASP consentono di rappresentare, in maniera semplice e naturale [Eiter *et al.*, 2000], varie forme di ragionamento non monotono, planning, problemi diagnostici e, più in generale, problemi di elevata complessità computazionale [Baral and Gelfond, 1994; Lobo *et al.*, 1992; Wolfinger, 1994; Minker, 1994; Lifschitz, 1996; Eiter *et al.*, 1999; Baral, 2003].

Dopo i primi sistemi sperimentali sul calcolo dei modelli stabili [Bell *et al.*, 1994; Subrahmanian *et al.*, 1995], attualmente sono disponibili un buon numero di sistemi che supportano in maniera efficiente l'ASP [Gebser *et al.*, 2007a; Leone *et al.*, 2006; Janhunen *et al.*, 2006; Lierler, 2005; Lin and Zhao, 2004; Simons *et al.*, 2002; Anger *et al.*, 2001; East and Truszczyński, 2001; Egly *et al.*, 2000]. Tra questi, quelli di maggiore successo sono: DLV [Leone *et al.*, 2006], `GnT` [Janhunen *et al.*, 2006] (estensione PLD del più conosciuto Smodels [Simons *et al.*, 2002]), e più di recente $clasp$ [Gebser *et al.*, 2007a]. Tutto ciò, se da un lato ha consentito l'impiego dell'ASP in contesti reali, dall'altro ha messo in evidenza le limitazioni dei sistemi attualmente disponibili.

Uno dei principali limiti è dato dall'inadeguato supporto a termini complessi quali *funzioni*, *liste*, *set* ed in generale costrutti che consentano, in maniera diretta, il ragionamento su strutture dati ricorsive e domini infiniti. Infatti, anche se la semantica answer set è stata definita per linguaggi del primo ordine 'generali' quindi con anche termini funzionali, i sistemi ASP esistenti, sono essenzialmente basati su linguaggi senza funzioni. L'esigenza di estendere la PLD con un adeguato supporto ai termini funzionali è percepita fortemente dalla comunità scientifica ASP, come testimoniato dai numerosi contributi pubblicati recentemente su questo tema [Lin and Wang, 2008; Simkus and Eiter, 2007; Baselice *et al.*, 2007; Bonatti,

2004; Syrjänen, 2001]. Tuttavia, non è stato ancora proposto un linguaggio sufficientemente soddisfacente dal punto di vista linguistico (che abbia espressività elevata) ed anche adatto ad essere integrato nei sistemi ASP esistenti. Infatti, attualmente nessun sistema ASP consente un uso effettivo di termini funzionali. I simboli di funzione o sono del tutto banditi dal linguaggio oppure sono soggetti a forti vincoli sintattici che non ne consentono l'uso in ambito ricorsivo.

Obiettivo di questo lavoro di tesi è il superamento di tali limitazioni. Il contributo del lavoro è sia teorico che pratico ed ha portato all'implementazione di un sistema ASP che supporta adeguatamente i termini complessi: funzioni ma anche liste ed insiemi.

Il principale contributo teorico riguarda la definizione formale di una nuova classe di programmi PLD: i programmi *finitamente ground* ($\mathcal{FG}$). Tale classe supporta in maniera completa i simboli di funzione (essi sono ammessi anche in caso di regole ricorsive) e gode di alcune rilevanti proprietà computazionali. Viene infatti dimostrato che, per tutti i programmi appartenenti alla classe, la valutazione secondo un approccio bottom-up, consente il calcolo effettivo degli answer set e di conseguenza, sia le query ground che le query non ground sono decidibili, qualsiasi sia la forma di ragionamento scelto (coraggioso o scettico). Un ulteriore risultato riguarda l'espressività di questa classe: viene dimostrato che, qualsiasi funzione calcolabile può essere espressa tramite un programma $\mathcal{FG}$. Chiaramente, questo implica che il problema di riconoscere se un programma appartiene o meno alla classe, risulta essere indecidibile (in particolare, semidecidibile). Poiché per alcuni utenti/applicazioni è necessario avere una garanzia ''a priori'' della terminazione del programma, si è ritenuto utile identificare una sottoclasse dei programmi $\mathcal{FG}$, per la quale anche il problema del riconoscimento sia decidibile. Alcune condizioni sintattiche sono quindi state individuate come caratterizzanti per la nuova classe: i programmi *a dominio finito* ($\mathcal{FD}$).

La classe dei programmi $\mathcal{FG}$ è, per alcuni aspetti, complementare alla classe dei programmi *finitari* [Bonatti, 2004]. La prima segue un approccio bottom-up, mentre la seconda top-down. Le due classi risultano essere incomparabili, nel senso che esistono dei programmi che appartengono alla prima ma non alla seconda e viceversa. Al fine di rendere valutabili secondo l'approccio bottom-up i programmi finitari, ampliando così la classe dei programmi $\mathcal{FG}$, si è fatto ricorso alla tecnica dei *Magic Sets*. Tale tecnica, nata nell'ambito delle Basi di Dati Deduttive a scopo di ottimizzazione, consiste in un metodo di riscrittura che simula la valutazione top-down di una query, modificando il programma originale e aggiungendo nuove regole che limitano la computazione ai dati rilevanti ai fini della query. Un opportuno adeguamento dell'algoritmo di riscrittura Magic Sets è stato definito per le query finitamente ricorsive (le query il cui programma 'rilevante' è finitario ed è positivo). Dato un programma $\mathcal{P}$ ed una query ground finitamente ricorsiva $\mathcal{Q}$, i seguenti risultati sono stati dimostrati riguardo il programma riscritto $RW(\mathcal{Q}, \mathcal{P})$: la sua dimensione è lineare rispetto alla dimensione dell'input; risulta essere del tutto equivalente a quello originario ai fini della risposta alla query e soprattutto, tale programma è $\mathcal{FG}$, quindi può essere valutato bottom-up (contrariamente al programma originale).

Oltre ai contributi teorici sinora descritti, questa tesi presenta anche i risultati di un'attività più pratica, di tipo implementativo e di sperimentazione. Tale attività è stata svolta utilizzando DLV come sistema ASP di riferimento ed ha avuto come obiettivo l'estensione del linguaggio con simboli di funzione e più in generale termini complessi quali liste e set.

Un primo passo in questa direzione è stata la realizzazione del supporto a funzioni esterne (plug-in). L'utente del sistema può definire dei propri particolari predicati esterni (identificati tramite il prefisso '$\#$'), implementati come funzioni C++ ed utilizzabili all'interno di un programma DLV. Ad ogni predicato esterno $\#p$ di arità $n$ deve essere associata almeno una funzione C++ $p'$, detta 'oracolo'. Tale funzione, della cui definizione è responsabile l'utente, deve restituire un valore booleano e deve avere esattamente $n$ argomenti. Un atomo ground $\#p(a1, \ldots, an)$ sarà vero se e solo se il valore restituito dalla funzione $p'(a1, \ldots, an)$ è vero. Gli argomenti di una funzione oracolo possono essere sia di input che di output: ad argomenti di input corrispondono termini costanti o a cui è già stato assegnato un valore (termini 'bound') nel relativo predicato esterno, ad argomenti di output corrispondono termini variabili non legati ad alcun valore (termini 'unbound').

L'introduzione delle funzioni esterne nel sistema DLV ha come conseguenza la possibilità di introdurre nuove costanti simboliche nel programma logico (Value Invention) e quindi rendere eventualmente infinito l'universo di Herbrand. Di conseguenza, per i programmi con Value Invention, non è più garantita la terminazione. Sono state quindi individuate un insieme di condizioni sintattiche tali da garantire la proprietà di 'istanziazione finita' e quindi la terminazione, per un programma logico con funzioni esterne. Un riconoscitore di programmi logici che rispettano tali condizioni è stato inoltre implementato ed integrato in DLV.

Sfruttando le possibilità offerte dai predicati esterni, è stato realizzato il supporto a termini complessi di tipo funzionale. In pratica, in presenza di termini funzionali, una regola DLV viene 'riscritta' in maniera tale da tradurre la presenza del simbolo di funzione, nell'invocazione di un opportuno predicato esterno; questo si occupa poi della costruzione del termine complesso o, viceversa, dell'estrazione degli argomenti da esso. Il risultato ottenuto è che i programmi $\mathcal{FG}$, descritti in precedenza, sono pienamente supportati dal sistema. Un riconoscitore sintattico per i programmi a dominio finito consente di individuare ''a priori'' quei programmi la cui terminazione non è garantita. A discrezione dell'utente, è possibile disabilitare tale riconoscitore e sfruttare in pieno l'espressività dei programmi $\mathcal{FG}$, rinunciando però alla garanzia di terminazione. Il linguaggio effettivamente supportato dal sistema, oltre ai termini funzionali, prevede anche altre due tipologie di termini complessi: liste e set. Realizzati sfruttando lo stesso framework usato per le funzioni, liste e set sono stati corredati da una ricca libreria di funzioni di manipolazione che rendono ll loro uso molto agevole. Grazie a tali estensioni, il linguaggio di DLV è diventato sempre più adatto alla codifica immediata e compatta di problemi di rappresentazione della conoscenza.

In sintesi, i contributi principali di questo lavoro di tesi sono:

– la definizione formale della classe dei programmi finitamente ground; cioè una nuova classe di programmi logici disgiuntivi che consente l'uso dei simboli di funzione (eventualmente con ricorsione) e che gode di rilevanti proprietà computazionali quali: l'elevata espressività, la computabilità bottom-up degli answer set e quindi la decidibilità del ragionamento, anche in caso di query non ground;

– l'utilizzo della tecnica dei Magic Sets per la riscrittura di programmi finitari positivi che non risultano appartenere alla classe dei programmi finitamente ground, ma sui quali è comunque possibile valutare bottom-up delle query ground, grazie ad un'opportuna trasformazione del programma stesso;

– la realizzazione di un sistema il cui linguaggio, oltre a supportare pienamente la classe dei programmi finitamente ground, consente l'integrazione nel programma logico di sorgenti computazionali esterne e l'utilizzo agevole di termini complessi quali liste e set;

– la comparazione del nostro lavoro, in particolar modo con i programmi finitari, ma anche con altri approcci proposti in letteratura per l'estensione dei sistemi ASP con simboli di funzione.

# Contents

**Introduction** **9**

Context and Motivations . . . . . 9

Main Contributions . . . . . 11

Structure of the Thesis . . . . . 14

**1 Answer Set Programming with Functions** **16**

1.1 Syntax . . . . . 17

1.2 Semantics . . . . . 18

1.3 Knowledge Representation and Reasoning . . . . . 20

1.3.1 The Guess and Check Programming Methodology . . . . 20

1.3.2 Applications of the Guess and Check Technique . . . . . 21

1.3.3 Enhanced KRR Capabilities by Function Symbols . . . . 24

1.4 Decidability and Complexity Issues . . . . . 25

1.4.1 Hierarchies of Complexity Classes . . . . . 26

1.4.2 Complexity Results for Function-Free ASP . . . . . 29

1.4.3 Decidability and Complexity for Logic Programming with Functions . . . . . 32

**2 A Top-Down Computable Class: Finitary Programs** **34**

2.1 Top-Down Programs Evaluation . . . . . 35

2.2 Notation . . . . . 36

2.3 Finitely-Recursive Programs . . . . . 37

2.4 Finitary Programs . . . . . 39

2.5 Properties of Finitary Programs . . . . . 41

**3 A Bottom-Up Computable Class: Finitely Ground Programs** **42**

3.1 Bottom-Up Programs Evaluation . . . . . 43

3.2 Notation . . . . . 44

3.3 Finitely-Ground Programs . . . . . . . . . . . . . . . . . . . . . . 47
3.4 Properties of Finitely-Ground Programs . . . . . . . . . . . . . . 51
3.5 A Recognizable Subclass: Finite-Domain Programs . . . . . . . . 53

**4 Bottom-Up Evaluation of Finitely-Recursive Queries via Magic Sets 56**
4.1 Magic Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.2 Magic Sets and Finitely-Recursive Programs . . . . . . . . . . . . 59
4.2.1 Finitely-Recursive Queries . . . . . . . . . . . . . . . . . 60
4.2.2 Rewriting Algorithm . . . . . . . . . . . . . . . . . . . . 62
4.2.3 Examples of Finitely-Recursive Queries Rewriting . . . . 63
4.3 Properties of Rewritten Programs . . . . . . . . . . . . . . . . . . 65

**5 Extending ASP with External Functions 69**
5.1 Value Invention in ASP Programs . . . . . . . . . . . . . . . . . 71
5.1.1 External Predicates: Syntax and Semantics . . . . . . . . 72
5.2 Properties of VI Programs . . . . . . . . . . . . . . . . . . . . . 73
5.3 Dealing with Value Invention . . . . . . . . . . . . . . . . . . . . 76
5.4 A Computable Class with External Functions: VI-restricted Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.5 Recognizing VI-restricted Programs . . . . . . . . . . . . . . . . 84

**6 An ASP System with Functions Lists and Sets 86**
6.1 System Language . . . . . . . . . . . . . . . . . . . . . . . . . . 87
6.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
6.3 Experiments and Applications . . . . . . . . . . . . . . . . . . . 90

**7 Related Works 92**
7.1 Finitary Programs . . . . . . . . . . . . . . . . . . . . . . . . . . 93
7.2 $\omega$-restricted Programs . . . . . . . . . . . . . . . . . . . . . . . 94
7.3 $\mathbb{FDNC}$ Programs . . . . . . . . . . . . . . . . . . . . . . . . 95
7.4 Other Works . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

**Conclusions 97**

**A Further Examples of Finitely Ground Programs 109**
A.1 A Finitely Ground Program Simulating a Turing Machine . . . . . 109
A.2 Towers of Hanoi Example . . . . . . . . . . . . . . . . . . . . . 111

**B `VI`-restrictedness Recognizing Algorithm 113**
B.1 The Recognizer Algorithm . . . . . . . . . . . . . . . . . . . . . . 113
B.2 The IsBlocked Algorithm . . . . . . . . . . . . . . . . . . . . . . . 114

**C Lists and Sets Manipulation Library 116**
C.1 List Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
C.2 Set Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
C.3 List Predicates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118
C.4 Set Predicates . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

# Introduction

## Context and Motivations

Answer Set Programming (ASP) [Gelfond and Lifschitz, 1988; 1991], evolved significantly during the last decade, and has been recognized as a convenient and powerful paradigm for declarative knowledge representation and reasoning. This formalism is based on methods to compute models of a given logic program. The underlying semantics (the so called *answer set semantics*) is inherently non-monotonic, i.e., the set of logical consequences does, in general, not necessarily grow monotonically with increasing information, due to the use of disjunction and negation-as-failure.

The answer set semantics extends the stable model semantics in that the former is defined on a syntactically richer class of programs than the latter. More specifically, the answer set semantics is defined for Disjunctive Logic Programming (DLP), in which not only negation-as-failure may occur in the program rules, but also strong negation (often referred to as classical negation) and disjunction. On the other hand, the stable model semantics is associated with normal logic programs, in which only negation-as-failure occurs as negation operator.

In ASP, a given computational problem is represented by a DLP program that may have several alternative models (but possibly none), called *answer sets*, each corresponding to a possible view of the world.

ASP languages support the representation of problems of high computational complexity, and, importantly, the ASP encoding of a large variety of problems is often very concise, simple, and elegant [Eiter *et al.*, 2000]. Such encodings are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [Wolfinger, 1994; Minker, 1994; Lifschitz, 1996; Eiter *et al.*, 1999; Baral, 2003]. For instance, one of the attractions of ASP is its capability of allowing the natural modeling of incomplete knowledge [Baral and Gelfond, 1994; Lobo *et al.*, 1992].

Several efforts have been made in the direction of implementing efficient ASP systems. After some pioneering work on stable models computation [Bell *et al.*, 1994; Subrahmanian *et al.*, 1995], a number of modern ASP systems are now available [Gebser *et al.*, 2007a; Leone *et al.*, 2006; Janhunen *et al.*, 2006; Lierler, 2005; Lin and Zhao, 2004; Simons *et al.*, 2002; Anger *et al.*, 2001; East and Truszczyński, 2001; Egly *et al.*, 2000]. Among them, the most successful have been DLV [Leone *et al.*, 2006], Smodels [Simons *et al.*, 2002], and recently also $clasp$ [Gebser *et al.*, 2007a]. Indeed, Smodels allows for the computation of answer sets for normal logic programs, however, there is an extended prototype version for the evaluation of disjunctive logic programs as well, called `GnT` [Janhunen *et al.*, 2006].

The availability of such efficient ASP solvers have encouraged a number of applications in many real-world contexts ranging, e.g., from information integration, to frauds detection, to software configuration, and many others. On the one hand, the above mentioned applications have confirmed the viability of the exploitation of ASP for advanced knowledge-based tasks. On the other hand, they have evidenced some limitations of ASP languages and systems, that should be overcome to make ASP better suited for real-world applications even in industry.

One of the most noticeable limitations is the fact that complex terms like functions, sets and lists, are not adequately supported by current ASP languages/systems. Indeed, while answer set semantics was defined in the setting of a general first order language, current ASP frameworks and implementations, are based in essence on function-free languages. Therefore, even by using state-of-the-art systems, one cannot directly reason about recursive data structures and infinite domains, such as XML/HTML documents, lists, time, etc. This is a strong limitation, both for standard knowledge-based tasks and for emerging applications, such as those manipulating XML documents.

The strong need of extending DLP by functions is clearly perceived in the ASP community, and many relevant contributions have been recently done in this direction [Lin and Wang, 2008; Simkus and Eiter, 2007; Baselice *et al.*, 2007; Bonatti, 2004; Syrjänen, 2001]. However, we still miss a proposal which is fully satisfactory from a linguistic viewpoint (high expressiveness) and suited to be incorporated in the existing ASP systems. Indeed, at present no ASP system allows for a reasonably unrestricted usage of function terms. Functions are either required to be non recursive or subject to severe syntactic limitations, if allowed at all in ASP systems.

## Main Contributions

This thesis aims at overcoming the above mentioned limitations, towards a powerful enhancement of ASP systems by functions. The contribution is both theoretical and practical, and leads to the implementation of a powerful ASP system supporting (recursive) functions, sets, and lists, along with libraries for their manipulations.

The theoretical contributions of the work, listed next, mainly concern the proposed class of *finitely-ground* ($\mathcal{FG}$) programs and fragments thereof. This new class follows the bottom-up evaluation approach and allows for (possibly recursive) function symbols, disjunction and negation.

Since the set of ground instances of a rule might be infinite (because of the presence of function symbols in the Herbrand universe), it is crucial to try to identify those that really matter in order to compute answer sets. Supposing that a given set $S$ contains all atoms that are potentially true, we first define an operator that helps selecting, among all ground instances, those somehow 'supported' by $S$. The presence of negation allows for identifying some further rules which do not matter for the computation of answer sets, and for simplifying the bodies of some others. We then provide the definition of an operator $\Phi$ that acts on a module of a program $\mathcal{P}$ in order to:

(i) select only those ground rules whose positive body is contained in a set of ground atoms consisting of the heads of a given set of rules;

(ii) perform a further simplification among these rules exploiting the presence of negation in the body.

By properly composing consecutive applications of $\Phi^{\infty}$ to components of a program, we can obtain an 'intelligent' instantiation which drops many useless rules w.r.t. answer sets computation.

We demonstrate that $\mathcal{FG}$ programs enjoy many relevant computational properties:

- both brave and cautious reasoning are computable, even for non-ground queries;

- answer sets are computable;

- each computable function can be expressed by a $\mathcal{FG}$ program.

Since $\mathcal{FG}$ programs express any computable function, membership in this class is obviously not decidable (we prove that actually it is semi-decidable). For users/applications where termination needs to be “a priori” guaranteed, we define the class of *finite-domain ($\mathcal{FD}$)* programs:

- both reasoning and answer set generation are computable for $\mathcal{FD}$ programs (they are a subclass of $\mathcal{FG}$ programs), and, in addition,
- recognizing whether a program is an $\mathcal{FD}$ program is decidable.

The class of $\mathcal{FG}$ programs can be seen as a “dual” notion of the class of *finitary* programs. The former allows for a bottom-up computation, while the latter is suitable for a top-down evaluation. However, when comparing the computational properties of the two classes, many points in favor of $\mathcal{FG}$ programs arise. With respect to programs inclusion, the two classes are incomparable, in the sense that there exist $\mathcal{FG}$ programs that are not finitary and viceversa. In order to make all positive finitary programs bottom-up computable, and then enlarge the class of $\mathcal{FG}$ programs, the *Magic Sets* technique is exploited. This method originated in the context of Deductive Databases for optimizing query answering, and consists in a strategy for simulating the top-down evaluation of a query by properly modifying the original program. The modified version of the program narrows the computation to what is relevant for answering the query at hand. A proper adaptation of the Magic Sets rewriting algorithm is defined for finitely-recursive queries (i.e. queries whose ‘relevant’ program is finitary and positive). Given a program $\mathcal{P}$ and a ground query $\mathcal{Q}$ (for which the subset of $\mathcal{P}$ relevant for the query $\mathcal{Q}$ is finitary and positive), the following results have been proved about the output program $RW(\mathcal{Q}, \mathcal{P})$ of our algorithm:

- its size is linear w.r.t. the original program size,
- it is equivalent to the original program w.r.t. the query answer,
- it is finitely-ground.

The last result is of particular relevance, since, although a program $\mathcal{P}$ might not be finitely-ground, there can be more than one query $\mathcal{Q}$ for which $RW(\mathcal{Q}, \mathcal{P})$ is finitely-ground, thus enabling finite computation by means of the bottom-up approach.

The practical contribution of this thesis has been realized extending the DLV system in order to support the class of $\mathcal{FG}$ programs.

A first step in this direction is the definition of a formal framework, named `VI`-programs (Value Invention programs), for accommodating *external functions* in the context of ASP. Thanks to this extension, users of the system can define their own external predicates (identified by the '#' prefix) implemented as C++ functions and use it in their DLV programs. To each external predicate $p$ having arity $n$, at least one C++ function $p'$(named 'oracle') should be associated. Such function must have exactly $n$ arguments and return a boolean value. A ground atom $\#p(a_1, \ldots, a_n)$ is true if $p'(a_1, \ldots, a_n)$ is true. Oracle functions can have both input and output arguments: input arguments in case of constants or bound terms, output arguments in case of unbound terms in the corresponding external predicate. External functions gives the explicit possibility of invention of new values from external sources of computation. Since this setting could lead to non-termination of any conceivable evaluation algorithm, we identify some syntactical conditions ensuring decidability preservation. A recognizer for checking these conditions has also been defined and integrated in the DLV system.

By exploiting the `VI`-programs framework, the DLV language has been extended with support for functional terms. Actually, functions are managed through a couple of (built-in) external functions that properly 'pack' and 'unpack' a functional term. Furthermore, in a similar manner, we extend the language with list and set terms, along with a rich library of built-in functions for lists and sets manipulations. All these extensions yield a very powerful system where the user can exploit the full expressiveness of $\mathcal{FG}$ programs (able to encode any computable function), or require the finite-domain check, getting the guarantee of termination. The system is available for downloading [Calimeri *et al.*, since 2008]; it is already in use in many universities and research centers throughout the world.

Briefly, the main contributions of this work are:

- the formal definition of finitely-ground programs, i.e. a new class of disjunctive logic programs admitting (possibly recursive) functions and enjoying some relevant computational properties like: high expressiveness, bottom-up computability of answer sets and decidability of reasoning even for non ground queries;

- exploiting of the Magic Sets technique, in order to allow bottom-up computability for queries, whose relevant program is both positive and finitary but does not belong to the class of finitely-ground programs;

- implementation of proposed extensions in DLV, obtaining a very powerful system where the user can exploit the full expressiveness of finitely-ground programs (able to encode any computable function), or require the finite-domain check, getting the guarantee of termination;

- in depth comparison of our work with the finitary programs and other approach to extend ASP with functions.

## Structure of the Thesis

The thesis is organized as follows.

- Chapter 1 introduces the Answer Set Programming framework. Syntax and semantics of the underlying disjunctive logic language with functions, are reported. Then, a methodology and many examples illustrating how ASP can be used for knowledge representation and reasoning tasks are given. Finally, some decidability and complexity issues are discussed.
- Chapter 2 is devoted to *finitary* programs, a class of normal logic programs supporting function symbols and such that reasoning is decidable in case of ground queries. We first recall some main aspects of the top-down evaluation approach and the needed preliminary definitions. Then, we report the main definitions about finitely-recursive and finitary programs. Finally, computational properties enjoyed by this class of programs are illustrated.
- In Chapter 3 we introduce a bottom-up computable class of disjunctive logic programs supporting function symbols:*finitely-ground* programs. We first recall some main aspects of the bottom-up evaluation approach and introduce some needed preliminary definitions. Then, the formal definition of the class is given and some computational properties are proved. Finally, we single out a recognizable subclass: finite-domain programs.
- In Chapter 4, after describing the standard Magic Sets technique, we illustrate the motivations inducing to the use of this sort of program transformation in the context of ASP with functions and then we present the adapted rewriting algorithm and many examples of applications. Some relevant computational results are then proved.
- Chapter 5 introduces a formal framework for accommodating external source of computation in the context of ASP. `VI` programs, i.e. logic programs enriched with *external predicates* are defined. We prove that, the consistency

check of `VI` programs is, in general, undecidable and address this problem identifying a *safety* condition for granting decidability. Finally, a recognizing algorithm checking such conditions is presented.

– In Chapter 6 we illustrate the implementation of an ASP system supporting finitely-ground programs. Such system actually features an even richer language, that, besides functions, explicitly supports also complex terms such as lists and sets, and provides a large library of built-in predicates for facilitating their manipulation.

– In Chapter 7 we survey the main proposals for introducing functional terms in ASP, and we briefly discuss the related work done in other research communities.

# Chapter 1

# Answer Set Programming with Functions

In this chapter we present the *Answer Set Programming* (ASP) framework, that has been recognized as a convenient and powerful method for declarative knowledge representation and reasoning. In particular, we first define the syntax of the underlying disjunctive logic language in which function symbols are supported. The associated answer set semantics is also described. Then, we illustrate the usage of ASP for knowledge representation and reasoning and finally analyze some decidability and complexity issues.

The chapter is organized as follows:

- Sections 1.1 and 1.2 provide a formal definition of the syntax and the semantics of a disjunctive logic programming language with function symbols.
- In Section 1.3 we first illustrate the usage of DLP languages for knowledge representation and reasoning, then we show the added modelling capability offered by function symbols.
- Finally, in Section 1.4, we report the main results about the computational complexity of disjunctive logic programs and discuss the decisional issues arising in the context of ASP with functions.

## 1.1 Syntax

A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If $t_1 \ldots t_n$ are terms and $f$ is a function symbol (*functor*) of arity $n$, then: $f(t_1, \ldots, t_n)$ is a *functional term*.

Each $t_i$, $1 \leq i \leq n$, is a subterm of $f(t_1, \ldots, t_n)$. The subterm relation is reflexive and transitive, that is:

- each term is also a subterm of itself;

- if $t_1$ is a subterm of $t_2$ and $t_2$ is subterm of $t_3$ then $t_1$ is also a subterm of $t_3$.

Each predicate $p$ has a fixed arity $k \geq 0$; by $p[i]$ we denote its $i$-th argument. If $t_1, \ldots, t_k$ are terms and $p$ is a *predicate* of arity $k$, then $p(t_1, \ldots, t_k)$ is an *atom*. Let $A$ be a set of atoms and $p$ be a predicate. With small abuse of notation we say that $p \in A$ if there is some atom in $A$ with predicate name $p$. An atom having $p$ as predicate name is usually referred as $a_p$. A *literal* $l$ is of the form $a$ or $\text{not } a$, where $a$ is an atom; in the former case $l$ is *positive*, and in the latter case *negative*.[1]

A *disjunctive rule* $r$ is of the form:

$$\alpha_1 \texttt{ v } \cdots \texttt{ v } \alpha_k \text{ :- } \beta_1, \cdots, \beta_n, \text{ not } \beta_{n+1}, \cdots, \text{ not } \beta_m. \tag{1.1}$$

where $m \geq 0$, $k \geq 0$; $\alpha_1, \ldots, \alpha_k$ and $\beta_1, \ldots, \beta_m$ are atoms.

The disjunction $\alpha_1 \texttt{ v } \cdots \texttt{ v } \alpha_k$ is called *head* of $r$, while the conjunction $\beta_1, \cdots, \beta_n, \text{ not } \beta_{n+1}, \cdots, \text{ not } \beta_m$ is the *body* of $r$. We denote by $H(r)$ the set $\{\alpha_1, \ldots, \alpha_k\}$ of the head atoms, and by $B(r)$ the set of body literals. In particular, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r)$ (the *positive body*) is $\{\beta_1, \cdots, \beta_n\}$ and $B^-(r)$ (*the negative body*) is $\{\text{not } \beta_{n+1}, \cdots, \text{ not } \beta_m\}$.

A rule having precisely one head literal (i.e. $k = 1$ and then $|H(r)| = 1$) is called a *normal rule*. If $r$ is a normal rule and its body is empty (i.e. $n = m = 0$ and then $B(r) = \emptyset$) then $r$ is referred to as a *fact*, and we usually omit the " :- " sign.

An *(integrity) constraint* is a rule without head literals (i.e. $k = 0$)

$$\text{:- } \beta_1, \cdots, \beta_n \text{ not } \beta_{n+1}, \cdots, \text{ not } \beta_m. \tag{1.2}$$

Under Answer Set Semantics, a constraint: $\text{:-}\, B(r)$ can be simulated through the introduction of a standard rule: $fail \text{ :- } B(r), \text{not } fail$, where $fail$ is a fresh

[1] Strong negation can be dealt with even if not explicitly supported: a constraint $\text{:- } a, \neg a.$ for each strongly negated atom $\neg a$ is added to the program, where $a$ also occurs in the program.

predicate not occurring elsewhere in the program. So, from a semantics point of view, we can assume that there are no constraints.

A rule is safe if each variable in that rule also appears in at least one positive literal in the body of that rule. For instance, the rule

$$p(X, f(Y, Z)) \text{ :- } q(Y), \text{not } s(X). \tag{1.3}$$

is not safe, because of both $X$ and $Z$. From now on we assume that all rules are safe.

A DLP program $\mathcal{P}$ is a finite set of rules. A program is safe, if each of its rules is safe. A $\text{not}$-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^{-}(r) = \emptyset$) is called *positive*. A v-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| = 1$) is called *normal logic program*. Positive normal logic programs are also called *Horn programs*. Positive normal logic programs where functional terms are not allowed are generally called *Datalog programs*.

Given a predicate $p$, a *defining rule* for $p$ is a rule $r$ such that the predicate $p$ occurs in the set of head atoms $H(r)$. If all defining rules of a predicate $p$ are facts, then $p$ is an $EDB$ *predicate*; otherwise $p$ is an $IDB$ *predicate*. [2] The set of all facts of $\mathcal{P}$ is denoted by $Facts(\mathcal{P})$; the set of instances of all $EDB$ predicates is denoted by $EDB(\mathcal{P})$ (note that $EDB(\mathcal{P}) \subseteq Facts(\mathcal{P})$). The set of all head atoms in $\mathcal{P}$ is denoted by $Heads(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} H(r)$.

A *query* $\mathcal{Q}$ is an $IDB$ atom.[3]

A program (a rule, a literal, a term, a query) is said to be *ground* if it contains no variables.

## 1.2 Semantics

The most widely accepted semantics for DLP programs is based on the notion of answer set, proposed in [Gelfond and Lifschitz, 1991] as a generalization of the concept of stable model [Gelfond and Lifschitz, 1988].

Let $\mathcal{P}$ be a disjunctive logic program, the *Herbrand universe* of $\mathcal{P}$, denoted by $U_{\mathcal{P}}$, consists of all (ground) terms that can be built combining constants and functors appearing in $\mathcal{P}$.

The *Herbrand base* of $\mathcal{P}$, denoted by $B_{\mathcal{P}}$, is the set of all ground atoms obtainable from the atoms of $\mathcal{P}$ by replacing variables with elements from $U_{\mathcal{P}}$.

[2]EDB and IDB stand for Extensional Database and Intensional Database, respectively.

[3]Note that this definition of a query is not as restrictive as it may seem, as one can include appropriate rules in the program for expressing unions of conjunctive queries (and more).

A *substitution* for a rule $r \in \mathcal{P}$ is a mapping from the set of variables of $r$ to the set $U_{\mathcal{P}}$ of ground terms. A *ground instance* of a rule $r$ is obtained applying a substitution to $r$. Given a program $\mathcal{P}$ the *instantiation (grounding)* $Ground(\mathcal{P})$ of $\mathcal{P}$ is defined as the set of all ground instances of its rules.

Given a ground program $\mathcal{P}$, an *interpretation* $I$ for $\mathcal{P}$ is a subset of $B_{\mathcal{P}}$. A positive literal $l = a$ (resp., a negative literal $l = \text{not } a$) is true w.r.t. $I$ if $a \in I$ (resp., $a \notin I$); it is false otherwise. Given a ground rule $r$, we say that $r$ is satisfied w.r.t. $I$ if some atom appearing in $H(r)$ is true w.r.t. $I$ or some literal appearing in $B(r)$is false w.r.t. $I$.

Given a ground program $\mathcal{P}$, we say that $I$ is a *model* of $\mathcal{P}$, if and only if all rules in $Ground(\mathcal{P})$ are satisfied w.r.t. $I$. A model $M$ is *minimal* if there is no model $N$ for $\mathcal{P}$ such that $N \subset M$.

**Example 1.1** The positive program $\mathcal{P}_1 = \{a \,\text{v}\, b \,\text{v}\, c.\}$ has the minimal models $\{a\}$, $\{b\}$, and $\{c\}$. Its extension $\mathcal{P}_2 = \{a \,\text{v}\, b \,\text{v}\, c. \;\; ; \;\; b \,\text{:-}\, c. \;\; ; \;\; c \,\text{:-}\, b.\}$ has two minimal models $\{a\}$ and $\{b, c\}$.

The *Gelfond-Lifschitz reduct* [Gelfond and Lifschitz, 1991] of a ground program $\mathcal{P}$, w.r.t. an interpretation $I$, is the positive ground program $\mathcal{P}^I$ obtained from $Ground(\mathcal{P})$ by:

- deleting all rules having a negative literal false w.r.t. $I$;
- deleting all negative literals from the remaining rules.

$I \subseteq B_{\mathcal{P}}$ is an *answer set* for a program $\mathcal{P}$ if and only if $I$ is a minimal model for $\mathcal{P}^I$. The set of all answer sets for $\mathcal{P}$ is denoted by $AS(\mathcal{P})$. Note that, in case of a positive program $\mathcal{P}$, $AS(\mathcal{P})$ coincides with the set of all minimal models for $\mathcal{P}$.

**Example 1.2** Given the program $\mathcal{P}_3 = \{\, a \,\text{v}\, b \,\text{:-}\, c. \;\; ; \;\; b \,\text{:-}\, \text{not } a, \text{not } c. \;\; ;$ $a \,\text{v}\, c \,\text{:-}\, \text{not } b.\}$ and $I = \{b\}$, the reduct $\mathcal{P}_3^I$ is $\{\, a \,\text{v}\, b \,\text{:-}\, c. \;\; ; \;\; b. \,\}$. It is easy to see that $I$ is a minimal model for $\mathcal{P}_3^I$, and for this reason it is also an answer set for $\mathcal{P}_3$.

Now consider $J = \{a\}$. The reduct $\mathcal{P}_3^J$ is $\{a \,\text{v}\, b \,\text{:-}\, c. \;\; ; \;\; a \,\text{v}\, c.\}$ and it can be easily verified that $J$ is a minimal model for $\mathcal{P}_3^J$, so it is also an answer set for $\mathcal{P}_3$.

If, on the other hand, we take $K = \{c\}$, the reduct $\mathcal{P}_3^K$ is equal to $\mathcal{P}_3^J$, but $K$ is not an answer set for $\mathcal{P}_3^K$. Indeed, the rule $r : a \,\text{v}\, b \,\text{:-}\, c,$ is not satisfied w.r.t. $K$ since neither some atom appearing in $H(r)$ is true w.r.t. $K$ nor some literal appearing in $B(r)$is false w.r.t. $K$. Indeed, it can be verified that $I$ and $J$ are the only answer sets of $\mathcal{P}_3$.

Given a program $\mathcal{P}$, a query $\mathcal{Q}$ and an interpretation $I$ for $\mathcal{P}$, $\vartheta(\mathcal{Q}, I)$ denotes the set containing all substitutions $\phi$ for the variables in $\mathcal{Q}$ such that $\phi(\mathcal{Q})$ is true in $I$. The answer to a query $\mathcal{Q}$ over $\mathcal{P}$, denoted by $Ans(\mathcal{Q}, \mathcal{P})$, is the set $\vartheta(\mathcal{Q}, I)$, such that $I \in AS(\mathcal{P})$. A program $\mathcal{P}$ *bravely* (resp. *cautiously*) entails a ground query $\mathcal{Q}$, denoted $\mathcal{P} \models_b \mathcal{Q}$ (resp. $\mathcal{P} \models_c \mathcal{Q}$ ) if $\mathcal{Q} \in A$ for some (resp. each) $A \in AS(\mathcal{P})$. In case $\mathcal{P}$ has a unique answer set $A$, we say that $\mathcal{P}$ entails a ground query $\mathcal{Q}$ denoted $\mathcal{P} \models \mathcal{Q}$ if $\mathcal{Q} \in A$.

## 1.3 Knowledge Representation and Reasoning

Answer Set Programming has been proved to be a very effective formalism for *Knowledge Representation and Reasoning (KRR)*. It can be used to encode problems in a highly declarative fashion, following the "Guess&Check"(**G&C**) methodology presented in [Eiter *et al.*, 2000]. In this section, we first describe the **G&C** technique and we then illustrate how to apply it on a number of examples. Finally, we show how the modelling capability of ASP is significatively enhanced by supporting function symbols.

### 1.3.1 The Guess and Check Programming Methodology

Many problems, also problems of comparatively high computational complexity ($\Sigma_2^P$-complete and $\Delta_3^P$-complete problems), can be solved in a natural manner by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so in a uniform way over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance $I$ of some problem $\mathbf{P}$, a **G&C** program $\mathcal{P}$ for $\mathbf{P}$ consists of the following two main parts:

**Guessing Part** The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for $I$.

**Checking Part** The (optional) checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program filters the solution candidates in such a way that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the admissible solutions for the problem instance $I$.

Without imposing restrictions on which rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., checking is completely integrated into the guessing part such that solution candidates are always solutions. Also, in general, the generation of the search space may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

In general, both $\mathcal{G}$ and $\mathcal{C}$ may be arbitrary collections of rules, and it depends on the complexity of the problem at hand which kinds of rules are needed to realize these parts (in particular, the checking part).

For problems with complexity in NP, often a natural **G&C** program can be designed with the two parts clearly separated into the following simple layered structure:

- The guessing part $\mathcal{G}$ consists of disjunctive rules that "guess" a solution candidate $S$.
- The checking part $\mathcal{C}$ consists of integrity constraints that check the admissibility of $S$.

Each layer may have further auxiliary predicates, for local computations.

The disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches.

It is worth remarking that the **G&C** programming methodology has also positive implications from the Software Engineering viewpoint. Indeed, the modular program structure in **G&C** allows for developing programs incrementally, which is helpful to simplify testing and debugging. One can start by writing the guessing part $\mathcal{G}$ and testing that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one adds the checking part and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ encode the admissible solutions.

## 1.3.2 Applications of the Guess and Check Technique

In this section, we illustrate the declarative programming methodology described in Section 1.3.1 by showing its application on a number of concrete examples.

Let us consider a classical NP-complete problem in graph theory, namely Hamiltonian Path.

**Definition 1.3 (HAMPATH)** 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?

Suppose that the graph $G$ is specified by using facts over predicates $node$ (unary) and $arc$ (binary), and the starting node $a$ is specified by the predicate $start$ (unary). Then, the following **G&C** program $\mathcal{P}_{hp}$ solves the problem HAMPATH:

$$\left.\begin{array}{l} inPath(X,Y)\,\texttt{v}\;outPath(X,Y) \;\text{:-}\; start(X), arc(X,Y). \\ inPath(X,Y)\,\texttt{v}\;outPath(X,Y) \;\text{:-}\; reached(X), arc(X,Y). \\ reached(X) \;\text{:-}\; inPath(Y,X). \qquad\qquad\qquad\qquad \textbf{(aux.)} \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \text{:-}\; inPath(X,Y), inPath(X,Y1), Y <> Y1. \\ \text{:-}\; inPath(X,Y), inPath(X1,Y), X <> X1. \\ \text{:-}\; node(X), \text{not}\; reached(X), \text{not}\; start(X). \end{array}\right\} \textbf{Check}$$

The two disjunctive rules guess a subset $S$ of the arcs to be in the path, while the rest of the program checks whether $S$ constitutes a Hamiltonian Path. Here, an auxiliary predicate $reached$ is used, which is associated with the guessed predicate $inPath$ using the last rule. Note that $reached$ is completely determined by the guess for $inPath$, and no further guessing is needed.

In turn, through the second rule, the predicate $reached$ influences the guess of $inPath$, which is made somehow inductively. Initially, a guess on an arc leaving the starting node is made by the first rule, followed by repeated guesses of arcs leaving from reached nodes by the second rule, until all reached nodes have been handled.

In the checking part, the first two constraints ensure that the set of arcs $S$ selected by $inPath$ meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by $S$.

A less sophisticated encoding can be obtained by replacing the guessing part with the single rule

$$inPath(X,Y)\,\texttt{v}\;outPath(X,Y) \;\text{:-}\; arc(X,Y).$$

that guesses for each arc whether it is in the path and by defining the predicate $reached$ in the checking part by rules

$$\begin{array}{l} reached(X) \;\text{:-}\; start(X). \\ reached(X) \;\text{:-}\; reached(Y), inPath(Y,X). \end{array}$$

However, this encoding is less preferable from a computational point of view, because it leads to a larger search space.

It is easy to see that any set of arcs $S$ which satisfies all three constraints must contain the arcs of a path $v_0, v_1, \ldots, v_k$ in $G$ that starts at node $v_0 = a$, and passes through distinct nodes until no further node is left, or it arrives at the starting node $a$ again. In the latter case, this means that the path is in fact a Hamiltonian Cycle (from which a Hamiltonian path can be immediately computed, by dropping the last arc).

Thus, given a set of facts $\mathcal{F}$ for $node$, $arc$, and $start$, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the corresponding graph has a Hamiltonian Path. The above program correctly encodes the decision problem of deciding whether a given graph admits a Hamiltonian Path or not.

This encoding is very flexible, and can be easily adapted to solve the *search problems* Hamiltonian Path and Hamiltonian Cycle (where the result has to be a tour, i.e., a closed path). If we want to be sure that the computed result is an *open* path (i.e., it is not a cycle), we can easily impose openness by adding a further constraint :- $start(Y), inPath(\_, Y).$ to the program (like in Prolog, the symbol '_' stands for an anonymous variable whose value is of no interest). Then, the set $S$ of selected arcs in any answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at $a$. If, on the other hand, we want to compute the Hamiltonian cycles, then we just have to strip off the literal $\text{not}\ start(X)$ from the last constraint of the program.

In the previous examples, we have seen how a search problem can be encoded in a DLP program whose answer sets correspond to the problem solutions. We next see another use of the **G&C** programming technique. We build a DLP program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the DLP program has no answer set. Such a programming scheme is useful to prove the validity of co-NP or $\Pi_2^P$ properties. We next apply the above programming scheme to a well-known problem of number and graph theory.

**Definition 1.4 (RAMSEY)** The Ramsey number $R(k, m)$ is the least integer $n$ such that, no matter how we color the arcs of the complete undirected graph (clique) with $n$ nodes using two colors, say red and blue, there is a red clique with $k$ nodes (a red $k$-clique) or a blue clique with $m$ nodes (a blue $m$-clique).

Ramsey numbers exist for all pairs of positive integers $k$ and $m$ [Radziszowski, 1994]. We next show a program $\mathcal{P}_{ramsey}$ that allows us to decide whether a given integer $n$ is <u>not</u> the Ramsey Number $R(3, 4)$. By varying the input number $n$, we can determine $R(3, 4)$, as described below. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with $n$ nodes. $\mathcal{P}_{ramsey}$ is the following **G&C** program:

$$\left.\begin{array}{l} blue(X,Y) \lor red(X,Y) \text{ :- } arc(X,Y). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \text{:- } red(X,Y),\ red(X,Z),\ red(Y,Z). \\ \text{:- } blue(X,Y),\ blue(X,Z),\ blue(Y,Z), \\ \quad blue(X,W),\ blue(Y,W),\ blue(Z,W). \end{array}\right\} \textbf{Check}$$

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red clique (i.e., a complete graph) with 3 nodes, and the second constraint eliminates the colorings containing a blue clique with 4 nodes. The program $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has an answer set if and only if there is a coloring of the edges of the complete graph on $n$ nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular $n$, then $n$ is not $R(3,4)$, that is, $n < R(3,4)$. On the other hand, if $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has no answer set, then $n \geq R(3,4)$. Thus, the smallest $n$ such that no answer set is found is the Ramsey number $R(3,4)$.

## 1.3.3 Enhanced KRR Capabilities by Function Symbols

As shown by the previous examples, ASP is particularly well-suited for modelling and solving problems that involve common-sense reasoning as well as for advanced knowledge-based tasks. As a matter of fact, this paradigm has been successfully applied to a range of applications including information integration, software configuration, reasoning about actions and change, etc.

These applications have evidenced some limitations of ASP languages and systems, that should be overcome to make ASP better suited for real-world applications even in industry. While answer set semantics, which underlies ASP, was defined in the setting of a general first order language, current ASP frameworks and implementations, like DLV [Leone *et al.*, 2006], Smodels [Simons *et al.*, 2002], clasp [Gebser *et al.*, 2007a] and other efficient solvers, are based in essence on function-free languages and resort to Datalog with negation and its extensions. Therefore, even by using state-of-the-art systems, one cannot directly reason about recursive data structures and infinite domains, such as XML/HTML documents, lists, time, etc. This is a strong limitation, both for standard knowledge-based tasks and for emerging applications, such as those manipulating XML documents.

Since one is forced to work with finite domains, potentially infinite processes cannot be represented naturally in ASP. Additional tools to simulate unbounded domains must be used. A notable example is the DLVK [Eiter *et al.*, 2003] front-end of the DLV system which implements the action language K [Eiter *et al.*, 2004]. Constants are used to instantiate a sufficiently large domain (estimated by the user) for solving the problem; this may incur high space requirements, and does not scale to large instances. Another example is given by recursive data structures like lists that can be simulated only through unnatural encodings.

Function symbols, in turn, are a very convenient means for generating infinite domains and objects, and allow for a more natural representation of problems in such domains. Recursive data structures can be immediately represented, without resorting to indirect encodings. Besides, there is no need to use constants to bound variables whose maximum value is a priori unknown like, for instance, variables representing a time or a plan length.

However, the reason why function symbols have been banned in ASP is that, allowing them, leads to undecidability of the reasoning, also for rather simple programs (see next section). This has raised the challenge to identify classes of programs retaining the decidability of the standard reasoning tasks, even in presence of functions. Different results have been achieved according to the followed evaluation strategy (top-down/bottom-up). Many examples of logic programs exploiting the expressive power offered by function symbols may be found throughout the next chapters. In particular, we will first show program examples belonging to a top-down computable class: ‘finitary programs’ (see Chapter 2) and then examples belonging to a bottom-up computable class: ‘finitely ground programs’ (see Chapter 3).

## 1.4 Decidability and Complexity Issues

In this chapter, we discuss the most relevant issues about decidability and complexity of Answer Set Programming. We first provide some preliminaries both on complexity theory and on some syntactic properties which allow us to single out computationally simpler subclasses of disjunctive logic programming languages. Then, we define the main computational problems under consideration and report the complexity results on different forms of function-free disjunctive logic programming. Finally, we analyze the decidability questions arising when function symbols are allowed.

### 1.4.1 Hierarchies of Complexity Classes

#### Polynomial Hierarchy

We assume here that the reader is familiar with the concepts of NP-completeness and complexity theory and provide only a very short reminder of the complexity classes of the polynomial hierarchy which are relevant to this chapter. For further details, the reader is referred to [Papadimitriou, 1994].

The classes $\Sigma_k^P$, $\Pi_k^P$, and $\Delta_k^P$ of the *polynomial hierarchy* ([Johnson, 1990]) are defined as follows:

$$\Delta_0^P = \Sigma_0^P = \Pi_0^P = \mathrm{P}$$

$$\text{and for all } k \geq 1,\ \Delta_k^P = \mathrm{P}^{\Sigma_{k-1}^P}, \Sigma_k^P = \mathrm{NP}^{\Sigma_{k-1}^P}, \Pi_k^P = \text{co-}\Sigma_k^P,$$

where NP$^C$ denotes the class of decision problems that are solvable in polynomial time on a nondeterministic Turing machine with an oracle for any decision problem $\pi$ in the class $C$. In particular, NP $= \Sigma_1^P$, co-NP $= \Pi_1^P$, and $\Delta_2^P = \mathrm{P}^{\mathrm{NP}}$.

The oracle replies to a query in unit time, and thus, roughly speaking, models a call to a subroutine for $\pi$ that is evaluated in unit time.

Observe that for all $k \geq 1$,

$$\Sigma_k^P \subseteq \Delta_{k+1}^P \subseteq \Sigma_{k+1}^P \subseteq \text{PSPACE}$$

where each inclusion is widely conjectured to be strict. By the rightmost inclusion above, all these classes contain only problems that are solvable in polynomial space. They allow, however, a finer grained distinction among NP-hard problems that are in PSPACE.

#### Arithmetical and Analytical Hierarchies

To characterize the complexity of logic programming with functions we also need a classification going beyond the polynomial hierarchy: the arithmetical and analytical hierarchies. We recall here just the basic definitions, referring to [Japaridze, 1994] and [Hartley Rogers, 1987] for further details.

The *arithmetical hierarchy* is a hierarchy of either (depending on the context) relations or formulas in the language of first-order arithmetic. The relations of a particular level of the hierarchy are exactly the relations defined by the formulas of that level, so the two uses are essentially the same.

The first level consists of formulas with only bounded quantifiers and is assigned the classifications $\Sigma_0^0$ and $\Pi_0^0$.

The classifications $\Sigma^0_n$ and $\Pi^0_n$ are defined inductively for every natural number $n$ using the following rules:

- If a formula $\phi$ is logically equivalent to a formula of the form $\exists n_1 \cdots \exists n_k \psi$, where $\psi$ is $\Pi^0_n$, then $\phi$ is assigned the classification $\Sigma^0_{n+1}$.
- If a formula $\phi$ is logically equivalent to a formula of the form $\forall n_1 \cdots \forall n_k \psi$, where $\psi$ is $\Sigma^0_n$, then $\phi$ is assigned the classification $\Pi^0_{n+1}$.

The subscript $n$ in the symbols $\Sigma^0_n$ and $\Pi^0_n$ indicates the number of alternations of blocks of universal and existential number quantifiers that are used in a formula. Moreover, the outermost block is existential in $\Sigma^0_n$ formulas and universal in $\Pi^0_n$ formulas. The superscript $0$ in the symbols $\Sigma^0_n$ and $\Pi^0_n$ indicates the type of the objects being quantified over. Type $0$ objects are natural numbers, and objects of type $i+1$ are functions that map the set of objects of type $i$ to the natural numbers. For example, the $\Sigma^0_1$ sets of numbers are those definable by a formula of the form $\exists n_1 \cdots \exists n_k \psi(n_1, \cdots, n_k, m)$ where $\psi$ has only bounded quantifiers. These are exactly the recursively enumerable (r.e.) sets.

Quantification over higher type objects is described by a superscript greater than $0$, as in the *analytical hierarchy*. The superscript $1$ would indicate quantification over functions from natural numbers to natural numbers. The notation $\Sigma^1_0 = \Pi^1_0$ indicates the class of formulas in the language of second-order arithmetic with no set quantifiers. As for the arithmetical hierarchy, an inductive definition can be provided for every subscript natural number n. A formula in the language of second-order arithmetic is defined to be $\Sigma^1_{n+1}$ if it is logically equivalent to a formula of the form $\exists n_1 \cdots \exists n_k \psi$ where $\psi$ is $\Pi^1_n$. A formula is defined to be $\Pi^1_{n+1}$ if it is logically equivalent to a formula of the form $\forall n_1 \cdots \forall n_k \psi$, where $\psi$ is $\Sigma^1_n$.

## Relevant Classes of Programs

Let us now report some definitions needed to identify syntactic classes of disjunctive logic programs with interesting properties.

**Definition 1.5** Functions $||\ || : B_{\mathcal{P}} \rightarrow \{0, 1, \ldots\}$ from the Herbrand base $B_{\mathcal{P}}$ to finite ordinals are called *level mappings* of $\mathcal{P}$.

Level mappings give us a useful technique for describing various classes of programs.

**Definition 1.6** A disjunctive logic program $\mathcal{P}$ is called *(locally) stratified* [Apt *et al.*, 1988; Przymusinski, 1988], if there is a level mapping $|| \ ||_s$ of $\mathcal{P}$ such that, for every rule $r$ of $Ground(\mathcal{P})$,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_s \leq ||l'||_s$;
2. For any $l \in B^-(r)$, and for any $l' \in H(r)$, $||l||_s < ||l'||_s$.
3. For any $l, l' \in H(r)$, $||l||_s = ||l'||_s$.

**Example 1.7** Consider the following two programs.

$$\begin{array}{llll} \mathcal{P}_1: & p(a) \lor p(c) \text{ :- } \text{not } q(a). & \mathcal{P}_2: & p(a) \lor p(c) \text{ :- } \text{not } q(b). \\ & p(b) \text{ :- } \text{not } q(b). & & q(b) \text{ :- } \text{not } p(a). \end{array}$$

It is easy to see that program $\mathcal{P}_1$ is stratified, while program $\mathcal{P}_2$ is not. A suitable level mapping for $\mathcal{P}_1$ is the following:

$$\begin{array}{lll} ||p(a)||_s = 2 & ||p(b)||_s = 2 & ||p(c)||_s = 2 \\ ||q(a)||_s = 1 & ||q(b)||_s = 1 & ||q(c)||_s = 1 \end{array}$$

As for $\mathcal{P}_2$, an admissible level mapping would need to satisfy $||p(a)||_s < ||q(b)||_s$ and $||q(b)||_s < ||p(a)||_s$, which is impossible.

Another interesting class of problems consists of head-cycle free programs.

**Definition 1.8** A program $\mathcal{P}$ is called *head-cycle free (HCF)* [Ben-Eliyahu and Dechter, 1994], if there is a level mapping $|| \ ||_h$ of $\mathcal{P}$ such that, for every rule $r$ of $Ground(\mathcal{P})$,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_h \leq ||l'||_h$;
2. For any pair $l, l' \in H(r)$ $||l||_h \neq ||l'||_h$.

**Example 1.9** Consider the following program $\mathcal{P}_3$.

$$\begin{array}{ll} \mathcal{P}_3: & a \lor b. \\ & a \text{ :- } b. \end{array}$$

It is easy to see that $\mathcal{P}_3$ is head-cycle free; an admissible level mapping for $\mathcal{P}_3$ is given by $||a||_h = 2$ and $||b||_h = 1$. Consider now the program

$$\mathcal{P}_4 = \mathcal{P}_3 \cup \{b \text{ :- } a.\}$$

$\mathcal{P}_4$ is not head-cycle free, since $a$ and $b$ should belong to the same level by Condition (1) of Definition 1.8, while they cannot by Condition (2) of that definition. Note, however, that $\mathcal{P}_4$ is stratified.

### 1.4.2 Complexity Results for Function-Free ASP

Three important decision problems, corresponding to three different reasoning tasks, arise in the context of Disjunctive Logic Programming:

> **Brave Reasoning**[4]. Given a program $\mathcal{P}$, and a ground atom $A$, decide whether $A$ is true in some answer set of $\mathcal{P}$ (denoted $\mathcal{P} \models_b A$).
>
> **Cautious Reasoning**[5]. Given a program $\mathcal{P}$, and a ground atom $A$, decide whether $A$ is true in all answer sets of $\mathcal{P}$ (denoted $\mathcal{P} \models_c A$).
>
> **Answer Set Checking.** Given a program $\mathcal{P}$, and a set $M$ of ground literals as input, decide whether $M$ is an answer set of $\mathcal{P}$.

We report here the complexity results of the first three decision problems recalled in the previous subsection. These results are given for ground (i.e., propositional) DLP programs without function symbols (some remarks on the complexity of non-ground programs are provided at the end of this subsection).

An interesting issue is the impact of further syntactic restrictions on the logic program $\mathcal{P}$. Starting from normal positive programs (without negation and disjunction), we consider the effect of allowing the (combined) use of the following constructs:

- stratified negation ($\mathrm{not}_s$),
- arbitrary negation ($\mathrm{not}$),
- head-cycle free disjunction ( $\mathtt{v_h}$ ),
- arbitrary disjunction ( $\mathtt{v}$ ).

Given a set $X$ of the above syntactic elements, we denote by DLP[$X$] the fragment of DLP where the elements in $X$ are allowed. For instance, DLP[$\mathtt{v_h}, \mathrm{not}_s$] denotes the fragment allowing head-cycle free disjunction and stratified negation.

We report here, with the help of some tables, results proved in [Gottlob, 1994; Eiter and Gottlob, 1995; Eiter *et al.*, 1998; Buccafurri *et al.*, 2000; Dantsin *et al.*, 2001; Leone *et al.*, 2006].

The complexity of Brave Reasoning and Cautious Reasoning from ground DLP programs are summarized in Table 1.1 and Table 1.2, respectively.

[4]The synonym ‘credulous’ is sometimes used instead of brave
[5]The synonym ‘skeptical’ is sometimes used instead of cautious

| | $\{\}$ | $\{\mathrm{not_s}\}$ | $\{\mathrm{not}\}$ |
|---|---|---|---|
| $\{\}$ | P | P | NP |
| $\{\mathtt{v_h}\}$ | NP | NP | NP |
| $\{\mathtt{v}\}$ | $\Sigma_2^P$ | $\Sigma_2^P$ | $\Sigma_2^P$ |

Table 1.1: The Complexity of Brave Reasoning in fragments of DLP

| | $\{\}$ | $\{\mathrm{not_s}\}$ | $\{\mathrm{not}\}$ |
|---|---|---|---|
| $\{\}$ | P | P | co-NP |
| $\{\mathtt{v_h}\}$ | co-NP | co-NP | co-NP |
| $\{\mathtt{v}\}$ | co-NP | $\Pi_2^P$ | $\Pi_2^P$ |

Table 1.2: The Complexity of Cautious Reasoning in fragments of DLP

In Table 1.3, we report the results on the complexity of Answer Set Checking. The rows of the tables specify the form of disjunction allowed; in particular, $\{\}$ = no disjunction, $\{\mathtt{v_h}\}$ = head-cycle free disjunction, and $\{\mathtt{v}\}$ = unrestricted (possibly not head-cycle free) disjunction. The columns specify the support for negation. For instance, $\{\mathrm{not}_s\}$ denotes that only stratified negation is supported. Each entry of the table provides the complexity of the corresponding fragment of the language, in terms of a completeness result. For instance, $(\{\mathtt{v_h}\}, \{\mathrm{not}_s\})$ is the fragment allowing head-cycle free disjunction and stratified negation. The corresponding entry in Table 1.1, namely NP, expresses that brave reasoning for this fragment is NP-complete. The results reported in the tables represent completeness under polynomial time (and in fact LOGSPACE) reductions.

Looking at Table 1.1, we see that limiting the form of disjunction and negation reduces the respective complexity. For disjunction-free programs, brave reasoning is polynomial on stratified negation, while it becomes NP-complete if we allow unrestricted (nonmonotonic) negation. Brave reasoning is NP-complete on head-cycle free programs even if no form of negation is allowed. The complexity jumps one level higher in the Polynomial Hierarchy, up to $\Sigma_2^P$-complexity, if full disjunction is allowed. Thus, disjunction seems to be harder than negation, since the full complexity is reached already on positive programs, even without any kind of negation.

| | $\{\}$ | $\{\mathrm{not}_\mathtt{s}\}$ | $\{\mathrm{not}\}$ |
|---|---|---|---|
| $\{\}$ | P | P | P |
| $\{\mathtt{v}_\mathtt{h}\}$ | P | P | P |
| $\{\mathtt{v}\}$ | co-NP | co-NP | co-NP |

Table 1.3: The Complexity of Answer Set Checking in fragments of DLP

Table 1.2 contains results for cautious reasoning. One would expect its complexity to be symmetric to the complexity of brave reasoning, that is, whenever the complexity of a fragment is $C$ under brave reasoning, one expects its complexity to be co-$C$ under cautious reasoning (recall that co-P $=$ P, co-$\Delta_2^P = \Delta_2^P$, co-$\Sigma_2^P = \Pi_2^P$, and co-$\Delta_3^P = \Delta_3^P$).

Surprisingly, there is one exception: while full disjunction raises the complexity of brave reasoning from NP to $\Sigma_2^P$, full disjunction alone is not sufficient to raise the complexity of cautious reasoning from co-NP to $\Pi_2^P$. Cautious reasoning remains in co-NP if default negation is disallowed. Intuitively, to disprove that an atom $A$ is a cautious consequence of a program $\mathcal{P}$, it is sufficient to find *any model* $M$ of $\mathcal{P}$ (which need not be an answer set or a minimal model) which does not contain $A$. For not-free programs, the existence of such a model guarantees the existence of a subset of $M$ which is an answer set of $\mathcal{P}$ (and does not contain $A$).

The complexity results for Answer Set Checking, reported in Table 1.3, help us to understand the complexity of reasoning. Whenever Answer Set Checking is co-NP-complete for a fragment $F$, the complexity of brave reasoning jumps up to the second level of the Polynomial Hierarchy ($\Sigma_2^P$).

We close this section with briefly addressing the complexity and expressiveness of non-ground programs. A non-ground program $\mathcal{P}$ can be reduced, by naive instantiation, to a ground instance of the problem. The complexity of this ground instantiation is as described above. In the general case, where $\mathcal{P}$ is given in the input, the size of the grounding $Ground(\mathcal{P})$ is single exponential in the size of $\mathcal{P}$. Informally, the complexity of Brave Reasoning and Cautious Reasoning increases accordingly by one exponential, from P to EXPTIME, NP to NEXPTIME, $\Delta_2^P$ to EXPTIME$^{\mathrm{NP}}$, $\Sigma_2^P$ to NEXPTIME$^{\mathrm{NP}}$, etc. For disjunctive programs and certain fragments of DLP, complexity results in the non-ground case have been derived e.g. in [Eiter *et al.*, 1997; 1998]. For the other fragments, the results can be derived using complexity upgrading techniques [Eiter *et al.*, 1997; Gottlob *et al.*, 1999].

## 1.4.3 Decidability and Complexity for Logic Programming with Functions

Besides the three decision problems reported in the previous section, here we will consider a further interesting decision problem related to the consistence of a program:

> **Consistency Checking.** Given a program $\mathcal{P}$, decide whether $\mathcal{P}$ admits at least one answer set.

If function symbols are allowed in a logic program, then all these decision problems are, in general, no longer decidable.

The first result in this direction regards recursive Horn programs. A proof of the non decidability of entailment of an atom can be found e.g. in [Dantsin *et al.*, 2001] where the reduction of query answering to the Hilbert's Tenth Problem is used at this aim. Undecidability is then deduced from the undecidability of diophantine equations [Matiyasevich, 1970]. In particular, theorems in [Tärnlund, 1977] state that logic programming using binary Horn clauses with one function symbol is r.e.-complete. This statement may be proved by simulating a universal Turing machine by logic programming (where terms $f^n(c), n \geq 0$ are used for representing cell positions and time instants).

Let us now consider logic programming with negation. In particular we are interested to negation-as-failure under the stable model semantics. For an important subclass of normal logic programs, i.e. stratified programs, the complexity was determined in [Apt and Blair, 1991], where it is showed that, in this case, stratified negation yields the arithmetical hierarchy, in particular: logic programming with $n$ levels of stratified negation is $\Sigma^0_{n+1}$-complete. For a general program $\mathcal{P}$, the problem of determining whether or not there exists a stable model of $\mathcal{P}$ is $\Sigma^1_1$-complete. This result was showed in [Marek *et al.*, 1992] where this problem has been proved to be equivalent to finding a path through an infinite branching recursive tree. In [Schlipf, 1995] is also proved that reasoning in logic programming with negation under stable model semantics is $\Pi^1_1$-complete. Finally, as proved in [Eiter and Gottlob, 1995] adding disjunction in the rule heads, in case of infinite Herbrand universe, does not change the complexity of the reasoning that remains $\Pi^1_1$-complete.

If we consider some syntactic restrictions, decidability of inference can be preserved. A natural decidable fragment of logic programming with functions are normal positive nonrecursive programs (in which, intuitively, no predicate depends syntactically on itself). Their complexity is characterized in [Dantsin and Voronkov, 1997] where was proved to be NEXPTIME-complete.

Computational complexity for the problem of deciding the existence of a stable model (consistency checking) for a non ground program, has been studied for the class of $\omega$-restricted programs (see Section 7.2). In the general case, it results to be 2-NEXP-complete.

For the $\mathbb{FDNC}$ class of programs (see Section 7.3) the following computational complexity results has been established: both consistency checking for non ground programs and brave reasoning (for both ground and non ground queries) are EXPTIME-complete, while cautious reasoning is EXPSPACE-complete.

# Chapter 2

# A Top-Down Computable Class: Finitary Programs

This chapter is devoted to a class of programs adopting a top-down evaluation approach and being computable despite the presence of function symbols. In particular, we describe *finitary* programs that is a class of normal logic programs supporting function symbols and such that reasoning is decidable in case of ground queries. We first recall some main aspects of the top-down evaluation approach and the needed preliminary definitions. Then, we report the main definitions about finitely-recursive and finitary programs. Finally, computational properties enjoyed by this class of programs are illustrated.

The chapter is organized as follows:

- Section 2.1 describes the top-down evaluation approach, recalling how it works for some classes of logic programs.
- In Section 2.2 some preliminary definitions peculiar to the class of programs described in this chapter are reported.
- Section 2.3 reports the definition of finitely-recursive programs, that is a superclass of finitary programs characterized by a 'restricted' form of recursion.
- In Section 2.4 finitary programs are introduced.
- Finally, in Section 2.5, the most relevant properties enjoyed by finitary programs are reported

## 2.1 Top-Down Programs Evaluation

In Section 1.2 we presented the stable model semantics as the most widely accepted semantics to give meaning to disjunctive logic programming. This semantics is based on the notion of answer sets and is purely declarative in the sense that it does not describe at all, how an answer set is to be constructed.

One possible approach that can be followed to evaluate a logic program is based on proof-theoretic notions and is generally denoted as *top-down* evaluation method.

In mathematical logic and automated theorem proving, *resolution* is a rule of inference leading to a refutation theorem-proving technique for sentences in propositional logic and first-order logic. *SLD-resolution* (Selected literal Linear resolution strategy over Definite clauses) is a special case of a refinement of the resolution method introduced in [Kowalski and Kuehner, 1971] and known as SL-resolution. It applies to special kinds of Horn clauses called definite clauses (Horn clauses with exactly one positive literal) and is the basic inference rule used in logic programming. In essence, SLD-resolution works as follows: it starts with a list of query predicates, called the *goal* list and finds a rule whose head unifies with a goal in the goal list, say $g$. The unifying substitution is then applied to the rule and to the goals in the goal list; the goal $g$ is removed from the goal list and the body of the unifying rule is added to the goal list (this is why the literals in the body of a rule are sometimes referred to as subgoals). This process is repeated until either the goal list becomes empty (in which case we have a successful deduction) or no new unifying rules can be found (in which case we have a failure). Note that, this kind of evaluation procedure has to make two choices: one in choosing the next goal from the goal list and another in choosing the rule whose head unifies with the selected goal. The different choices that can be made by the top-down proof process implicitly define a tree, called *SLD-tree*.

SLD-resolution is the computation procedure used in Prolog [Kowalski, 1974]. Interpreters of such a language usually choose goals in a left-to-right order and rules in a sequential order that corresponds to a depth-first search of the SLD-tree, with backtracking when failure occurs. Pure Prolog was originally restricted to Horn clauses of the form: $H : -B1, , Bn.$ but it was soon extended to include negation-as-failure, in which negative conditions of the form $not(Bi)$ are shown by trying and failing to solve the corresponding positive conditions $Bi$.

A skeptical resolution calculus for the stable model semantics has been presented in [Bonatti, 2001b]. Differently from the approach generally used for this semantics, it is not based on the construction of entire stable models. This feature makes it possible to obtain concise derivations, which often involve only a strict subset of the program rules. In many cases (including all the programs whose dependency graph contains no cycles with an odd number of negative edges) the derivations may proceed in a thoroughly goal-directed way. This resolution calculus was proved sound for all programs and complete w.r.t. function-free ones. The completeness result has been extended to all finitary programs in [Bonatti, 2004]. More recently, a credulous resolution calculus for ASP has been proposed in [Bonatti *et al.*, 2008]. The approach followed in this work allows a top-down and goal directed resolution, in the same spirit as traditional SLD-resolution. The proposed credulous resolution can be used in query-answering with non ground queries and with non ground, and possibly infinite, programs. Soundness and completeness results for the resolution procedure are proved for large classes of logic programs.

Top-down evaluation implies a notion of safety different from that introduced in Section 1.1. Indeed a rule could be safely evaluated even if there are variables appearing only in the head of the rule, provided that the query to be answered supplies appropriate binding to those variables. On the other hand, a rule having variables appearing only in the body of the rule, can not be safely evaluated in case of an infinite Herbrand universe.

Another noticeable aspect of top-down evaluations is that of *relevant rules*. Given a set of rules $R$ then the rule $r \in R$ is relevant with respect to a query $\mathcal{Q}$ if $r$ is used in the top-down proof of $\mathcal{Q}$. This means that a query can be correctly answered by reasoning with a portion of the given program, i.e. the subprogram made by all the rules relevant w.r.t. the query.

## 2.2 Notation

In this section we report some preliminary definitions extracted from [Bonatti, 2004] and peculiar to the class described in this chapter. In particular, the definition of a dependency graph is exploited to introduce the notion of dependency among atoms and the concept of odd-cycle. Finally, notions of kernel atoms and relevant universe and subprogram are introduced.

Given a normal logic (disjunction-free) program $\mathcal{P}$, a labelled *dependency graph* $LDG(\mathcal{P})$ is associated to $Ground(\mathcal{P})$. The set of vertices consists of the (infinite) set of atoms in $B_{\mathcal{P}}$. The set of edges is defined as follows:

- there exists an edge labelled '+' (called positive edge) from $A_1$ to $A_2$ if and only if for some rule $r \in Ground(\mathcal{P})$, $A_1 \in H(r)$ and $A_2 \in B^+(r)$;
- there exists an edge labelled '-' (called negative edge) from $A_1$ to $A_2$ if and only if for some rule $r \in Ground(\mathcal{P})$, $A_1 \in H(r)$ and $\mathrm{not} A_2 \in B^-(r)$;

An atom $A_1$ *depends* positively (respectively negatively) on $A_2$ if there is a directed path from $A_1$ to $A_2$ in the dependency graph with an even (respectively odd) number of negative edges. Moreover, each atom depends positively on itself. If $A_1$ depends positively (respectively negatively) on $A_2$ we write $A_1 \geq_+ A_2$ (respectively $A_1 \geq_- A_2$). We write $A_1 \geq A_2$ if either $A_1 \geq_+ A_2$ or $A_1 \geq_- A_2$. If both $A_1 \geq_+ A_2$ and $A_1 \geq_- A_2$ hold then we write $A_1 \geq_\pm A_2$.

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

A *kernel* atom for a normal program $\mathcal{P}$ and a ground query $\mathcal{Q}$ is either an odd-cyclic atom or an atom occurring in $\mathcal{Q}$. The set of kernel atoms for $\mathcal{P}$ and $\mathcal{Q}$ is denoted by $K_{(\mathcal{P},\mathcal{Q})}$.

The *relevant universe* for $\mathcal{P}$ and $\mathcal{Q}$, denoted by $U_{(\mathcal{P},\mathcal{Q})}$ is the set of all ground atoms $B$ such that some kernel atom for $\mathcal{P}$ and $\mathcal{Q}$ depends on $B$. In symbols: $U_{(\mathcal{P},\mathcal{Q})} = \{B | \text{ for some } A \in K_{(\mathcal{P},\mathcal{Q})}, A \geq B\}$.

The *relevant subprogram* of $\mathcal{P}$ for a ground query $\mathcal{Q}$, denoted by $R_{(\mathcal{P},\mathcal{Q})}$, is the set of all rules in $Ground(\mathcal{P})$ whose heads belongs to $U_{(\mathcal{P},\mathcal{Q})}$. In symbols: $R_{(\mathcal{P},\mathcal{Q})} = \{r | \ r \in Ground(\mathcal{P}) \text{ and } H(r) \in U_{(\mathcal{P},\mathcal{Q})}\}$.

## 2.3 Finitely-Recursive Programs

In this section we report the definition of a class of normal logic programs allowing function symbols and characterized by a 'restricted' form of recursion. This class was firstly introduced in [Bonatti, 2001a] and [Bonatti, 2004] as a superclass of the finitary class of programs shown in the next section. Then it was studied in depth in [Baselice *et al.*, 2007].

**Definition 2.1** [Bonatti, 2004] A normal program $\mathcal{P}$ is *finitely-recursive* if and only if each atom $A \in Ground(\mathcal{P})$ depends on finitely many ground atoms. In other words, the cardinality of the set of ground atoms $\{B | A \geq B\}$ must be finite for all atoms $A$ included in $Ground(\mathcal{P})$.

Positive finitely-recursive programs enjoy all properties shown in Section 2.5. Indeed, being positive they trivially fulfill the further condition required to be

$member(X, [X|L]).$
$member(X, [Y|L]) \text{ :- } member(X, L).$

$append([\,], L, L).$
$append([X|T_1], L, [X|T_2]) \text{ :- } append(T_1, L, T_2).$

$reverse(L, R) \text{ :- } sup\_reverse(L, [\,], R).$
$sup\_reverse([\,], R, R).$
$sup\_reverse([X|T_1], L, R) \text{ :- } sup\_reverse(T_1, [X|L], R).$

Figure 2.1: Examples of positive finitely-recursive programs.

finitary (see next section). For example, most classical programs on recursive data structures such as lists and trees (e.g., predicates member, append, reverse in Fig. 2.1 are finitely-recursive positive programs). Typically, these programs are finitely-recursive because the terms occurring in the body of a rule occur also in the head, often as strict subterms of the heads arguments.

As proved in [Baselice *et al.*, 2007], finitely-recursive programs in any case share with finitary programs these nice properties:

- compactness;
- r.e.- completeness of consistency checking and cautious inference;
- completeness of the resolution calculus for skeptical stable model semantics (see [Bonatti, 2001b]).

For compactness here is intended a property similar to that enjoyed by the classical first-order logic, that is: an infinite set of formulas is inconsistent if and only if it has an inconsistent finite subset. The analogues of inconsistent finite subset of a theory are the so called *unstable kernels*. An unstable kernel for a program $\mathcal{P}$ is a set $K \subseteq Ground(\mathcal{P})$ with the following properties:

- $K$ is downward closed, that is, for each atom $A$ occurring in rules of $K$, the set $K$ contains all the rules $r \in Ground(\mathcal{P})$ such that $H(r) = A$;
- $K$ has no stable models;

A finitely-recursive program $\mathcal{P}$ has no stable model if and only if it has a finite unstable kernel. It is worth noting that this property in general is not enjoyed by nonmonotonic logics.

As already remarked in Section 2.1, a rule having variables appearing only in the body of the rule, can not be top-down safely evaluated in case of an infinite Herbrand universe. So, a program with function symbols, having a so called *local variable*, is not finitely-recursive. Indeed, in case of an infinite Herbrand domain, the possible substitutions for local variables are infinite.

**Example 2.2** The following simple program $\mathcal{P}$ is not finitely-recursive:

$$g(0,0).$$
$$p(f(X))\ \text{:-}\ g(X,Y).$$

in fact its ground version contains infinite rules deriving from the infinite possible substitution $(0, f(0), f(f(0)), f(f(f(0)))...$ and so on) for the local variable $Y$. This means that the associated dependency graph $LDG(\mathcal{P})$ has at least one node (e.g., $p(f(0))$ depending on an infinite set of other nodes. This definitely violates the condition required for a program to be finitely-recursive.

## 2.4 Finitary Programs

In this section we report the definition of *finitary programs*, a class of normal logic programs admitting unbounded (possibly infinite) domains and cyclic definitions, and such that inference is r.e.-complete.

In Section 1.4.3 we remarked as admitting function symbols, makes the inferences of normal logic programs highly undecidable. In [Bonatti, 2004] two determining factors have been identified and then proper restrictions have been enforced.

The first factor is unrestricted recursion, that makes positive programs Turing equivalent. Consequently, the answer to a negative goal $\mathrm{not}A$ is not semidecidable, in general.

The second factor is the role that odd-cycles play in query answering. For example, consider program $\mathcal{P}_1 = \{p.\ ;\ q\ \text{:-}\ \mathrm{not}q\ \}$, that has no stable models because of rule $q\ \text{:-}\ \mathrm{not}q$ (note that it makes $q$ odd-cyclic). Then, in order to decide whether this program has a stable model containing $p$, we have to consider also $q\ \text{:-}\ \mathrm{not}q$, even if this rule has no explicit syntactic relationship with $p$ (i.e., there is no path between $p$ and $q$ in the dependency graph $LDG(\mathcal{P}_1)$). In other words, a rule involved in an odd-cycle can never be ignored because it may cause the program to be inconsistent.

To keep complexity under control the definition of finitary programs imposes constraints on both these factors. Indeed, recursion is restricted and the number of possible sources of inconsistency is required to be finite.

**Definition 2.3** [Bonatti, 2004] A normal program $\mathcal{P}$ is finitary if the following conditions hold:

- $\mathcal{P}$ is finitely-recursive.
- There are finitely many odd-cyclic atoms in the dependency graph $LDG(\mathcal{P})$.

Clearly, all finitely-recursive programs that are either positive or locally stratified are finitary, because their dependency graph has no negative cycles. A simple example is the following program $\mathcal{P}_2$, defining even and odd numbers:

$$\begin{array}{l} even(0). \\ even(X) \text{ :- } \text{not } odd(X). \\ odd(s(X)) \text{ :- } \text{not } odd(X). \end{array}$$

As another example of finitary program we report the following encoding $\mathcal{P}_3$ of the SAT problem:

$$\begin{array}{ll} s(and(X,Y)) \text{ :- } s(X), s(Y). & s(A) \text{ :- } member(A,[p,q,r]), \text{not } ns(A). \\ s(or(X,Y)) \text{ :- } s(X). & ns(A) \text{ :- } member(A,[p,q,r]), \text{not } s(A). \\ s(or(X,Y)) \text{ :- } s(Y). & member(A,[A|L]). \\ s(not(X)) \text{ :- } \text{not } s(X). & member(A,[B|L]) \text{ :- } member(A,L). \end{array}$$

A ground goal $s(t)$ is bravely entailed by the program $\mathcal{P}_3$ if and only if $t$ encodes a satisfiable formula. Actually, this program handles only the propositional formulas freely generated by atoms $p, q, r$, but it can be easily generalized to unbounded formulas.

Many programs for reasoning about action and planning are finitary. In this kind of programs time can be represented explicitly via a number, rather than the history of previous actions. A function symbol is used to represent the successive time instant. Note that in other cases where function symbols are not allowed, time is represented with constants, so the number of time points must be specified a priori.

## 2.5 Properties of Finitary Programs

In this section the most relevant properties enjoyed by finitary programs are reported.

The compactness property has been proved for a larger set of programs, i.e. all those that are finitely-recursive, and has been already illustrated in Section 2.3.

All others nice properties are based on the two following preliminary results:

- If $\mathcal{P}$ is a finitary program then, for all ground query $\mathcal{Q}$, the relevant universe $U_{(\mathcal{P},\mathcal{Q})}$ and the relevant subprogram $R_{(\mathcal{P},\mathcal{Q})}$ (see Section 2.2) of $\mathcal{P}$ w.r.t. $\mathcal{Q}$ are finite.

- For all finitely-recursive programs $\mathcal{P}$ and all ground queries $\mathcal{Q}$, $\mathcal{P}$ bravely (resp. cautiously) entails $\mathcal{Q}$ if and only if $R_{(\mathcal{P},\mathcal{Q})}$ bravely (resp. cautiously) entails $\mathcal{Q}$.

First of all, decidability of the inference for all ground queries derives. More precisely, the following main theorem has been proved:

**Theorem 2.4** [Bonatti, 2004] For all finitary programs $\mathcal{P}$ and ground queries $\mathcal{Q}$, both the problem of deciding whether $\mathcal{Q}$ is a brave consequence of $\mathcal{P}$ and the problem of deciding whether $\mathcal{Q}$ is a cautious consequence of $\mathcal{P}$ are decidable.

Actually, as remarked by the author in [Bonatti, 2008], a further condition is needed to obtain decidability: "a priori" knowledge of what is the set of atoms involved in odd-cycles.

A related theorem ([Bonatti, 2004], Theorem 17) has been proved about non ground queries, but in this case the inference results to be semidecidable.

Regarding the expressive power, the class of finitary programs has been proved to be computationally complete by showing how any Turing machine $M$ can be simulated by a suitable finitary program that returns the output of $M$([Bonatti, 2004], Theorem 22).

The problem of recognizing if a program is finitary is undecidable. In particular, recognizing the validity of the first condition of Definition 2.3 has been proven to be not decidable ([Bonatti, 2004], Theorem 26), while recognizing the second condition (finite odd cycles) has been proven to be not semi-decidable ([Bonatti, 2004], Theorem 27).

# Chapter 3

# A Bottom-Up Computable Class: Finitely Ground Programs

In this chapter a class of programs adopting a bottom-up evaluation approach and being computable despite the presence of function symbols is defined. In particular, we introduce the class of *finitely-ground* programs, that is a class of disjunctive logic programs supporting function symbols and enjoying many relevant computational properties. We first recall some main aspects of the bottom-up evaluation approach and introduce some needed preliminary definitions. Then, the class of finitely-ground programs is defined and some computational properties are proved. Finally, we single out a recognizable subclass of finitely-ground programs.

The chapter is organized as follows:

- Section 3.1 describes the bottom-up evaluation approach, recalling how it works for some classes of logic programs.
- In Section 3.2 some preliminary definitions peculiar to the class of programs described in this chapter are reported.
- Section 3.3 defines the class of finitely-ground programs exploiting the notion of intelligent instantiation.
- In Section 3.4 this class of programs is characterized by identifying some key properties.
- Finally, Section 3.5, identifies a subclass, called finite-domain programs, which, besides sharing the nice decidability properties of finitely-ground programs, ensures the decidability of recognizing membership in the class.

## 3.1 Bottom-Up Programs Evaluation

Bottom-up semantics follows a complementary approach w.r.t. top-down programs evaluation. In this case there is not a goal-directed inference, but a process of model construction starting from the set of facts of the program.

The first effort to give a formal definition of such a kind of semantics was made in [Van Emden and Kowalski, 1976]. It is based on *fixpoints* of operators and regards a restricted class of logic programs i.e. Horn programs. The unique minimal model of a program is obtained as the least fixpoint of the so called *immediate consequence* operator $\mathcal{T}_{\mathcal{P}}$. This operator maps interpretations to interpretations and is defined as follows:
$\mathcal{T}_{\mathcal{P}}(I) = \{a \mid \exists r \in \mathcal{P}\ s.t.\ a \in H(r) \wedge B(r) \subseteq I\}$.
$\mathcal{T}_{\mathcal{P}}(I)$ derives an atom $a$ from a rule $r$, if the body of $r$ is true w.r.t. $I$. Intuitively, given an interpretation $I$, $\mathcal{T}_{\mathcal{P}}$ derives a set of atoms that are strictly needed to extend $I$ to a model. In [Apt and van Emden, 1982] has been proved that $\mathcal{T}_{\mathcal{P}}(\emptyset)^{\infty}$ always gives a set that is the unique minimal Herbrand model of the program.

The quest for finding a suitable semantics in the spirit of minimal models for programs containing negation turned out to be far from straightforward. One milestone has been the definition of what later became known uniformly as *perfect model semantics* for programs that can be stratified on negation [Apt *et al.*, 1988], [Van Gelder, 1988].

One of the approach followed for non-stratified programs, was giving up the classical setting of models that assign two truth values and introducing a third value, intuitively representing unknown. This approach required a somewhat different definition, because in the two-valued approach, one would give a definition only for positive values, implicitly stating that all other constructs are considered to be negative. For instance, for minimal models, one minimizes the true elements, implicitly stating that all elements not contained in the minimal model will be false. With three truth values, this strategy is no longer applicable, as elements that are not true can be either false or undefined. For resolving this, Allen Van Gelder, Kenneth Ross, and John Schlipf introduced the notion of unfounded sets in [Van Gelder *et al.*, 1988], in order to define which elements of the program should be definitely false. Combining existing techniques for defining the minimal model with unfounded sets, they introduced the notion of a *well-founded* model, defined as the least fixpoint of the well-founded operator $\mathcal{W}_{\mathcal{P}}$. In this way, any program would still be guaranteed to have a single model, just like there is a unique minimal model for positive programs and a unique perfect model for

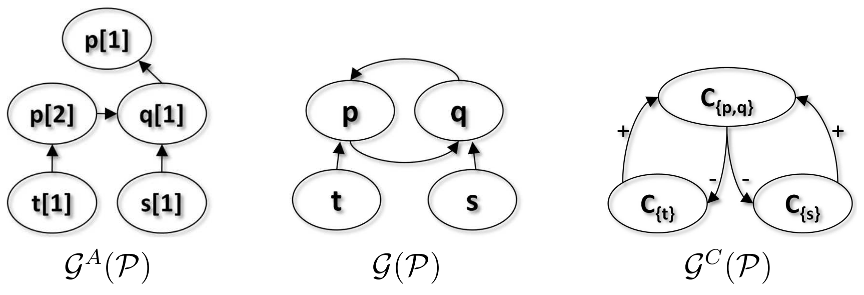
p[1]
p[2]
q[1]
t[1]
s[1]
$\mathcal{G}^A(\mathcal{P})$
p
q
t
s
$\mathcal{G}(\mathcal{P})$
C{p,q}
C{t}
C{s}
+
+
-
-
$\mathcal{G}^C(\mathcal{P})$

Figure 3.1: *Argument, Dependency* and *Component* Graphs of the program in Example 3.4.

stratified programs. Well-founded and stable models are closely related; for instance, the well-founded model of a program is contained in each stable model [Van Gelder *et al.*, 1991]. Moreover, both approaches coincide with perfect models on stratified programs.

The notion of unfounded set and the definition of the well-founded operator $\mathcal{W}_{\mathcal{P}}$ have been extended to the disjunctive case in [Leone *et al.*, 1997]. Two declarative characterizations of stable models in terms of unfounded sets are also provided. One shows that the set of stable models coincides with the family of unfounded-free models (i.e., a model is stable if and only if it contains no unfounded atoms). The other proves that stable models can be defined equivalently by a property of their false literals, as a model is stable if and only if the set of its false literals coincides with its greatest unfounded set. Moreover, a fixpoint semantics for disjunctive stable models and an algorithm for computing the stable models of programs have been given.

## 3.2 Notation

In this section we introduce some preliminary definitions needed for what presented in this chapter. In particular, we next define three different graphs that point out dependencies among arguments, predicates, and components of a program.

**Definition 3.1** The *Argument Graph* $\mathcal{G}^A(\mathcal{P})$ of a program $\mathcal{P}$ is a directed graph containing a node for each argument $p[i]$ of an IDB predicate $p$ of $\mathcal{P}$; there is an edge $(q[j], p[i])$ if and only if there is a rule $r \in \mathcal{P}$ such that:

(i) an atom $p(\bar{t})$ appears in the head of $r$;

(ii) an atom $q(\overline{v})$ appears in $B^+(r)$;

(iii) $p(\overline{t})$ and $q(\overline{v})$ share the same variable within the i-th and j-th term, respectively.

Given a program $\mathcal{P}$, an argument $p[i]$ is said to be recursive with $q[j]$ if there exists a cycle in $\mathcal{G}^A(\mathcal{P})$ involving both $p[i]$ and $q[j]$. Roughly speaking, this graph keeps track of (body-head) dependencies between the arguments of predicates sharing some variable. It is actually a more detailed version of the commonly used (predicate) dependency graph, defined below.

**Definition 3.2** The *Dependency Graph* $\mathcal{G}(\mathcal{P})$ of $\mathcal{P}$ is a directed graph whose nodes are the IDB predicates appearing in $\mathcal{P}$. There is an edge $(p_2, p_1)$ in $\mathcal{G}(\mathcal{P})$ if and only if there is some rule $r$ with $p_2$ appearing in $B^+(r)$and $p_1$ in $H(r)$, respectively.

The graph $\mathcal{G}(\mathcal{P})$ suggests to split the set of all predicates of $\mathcal{P}$ into a number of sets (called components), one for each strongly connected component (SCC)[1] of the graph itself. Given a predicate $p$, we denote its component by $comp(p)$; with a small abuse of notation, we define also $comp(l)$ and $comp(a)$, where $l$ is a literal and $a$ is an atom, accordingly.

In order to single out dependencies among components, a proper graph is defined next.

**Definition 3.3** Given a program $\mathcal{P}$ and its Dependency Graph $\mathcal{G}(\mathcal{P})$, the *Component Graph* of $\mathcal{P}$, denoted $\mathcal{G}^C(\mathcal{P})$, is a directed labelled graph having a node for each strongly connected component of $\mathcal{G}(\mathcal{P})$ and:

(i) an edge $(B, A)$, labelled "+", if there is a rule $r$ in $\mathcal{P}$ such that there is a predicate $q \in A$ occurring in the head of $r$ and a predicate $p \in B$ occurring in the positive body of $r$;

(ii) an edge $(B, A)$, labelled "-", if there is a rule $r$ in $\mathcal{P}$ such that there is a predicate $q \in A$ occurring in the head of $r$ and a predicate $p \in B$ occurring in the negative body of $r$, and there is no edge $(B, A)$, with label "+";

Self-cycles are not considered.

[1]We recall here that a strongly connected component of a directed graph is a maximal subset $S$ of the vertices, such that each vertex in $S$ is reachable from all other vertices in $S$.

**Example 3.4** Consider the following program $\mathcal{P}$, where $a$ is an EDB predicate:

$$
\begin{array}{ll}
q(g(3)). & s(X) \vee t(f(X)) \text{ :- } a(X),\ \text{not}\ q(X). \\
p(X,Y) \text{ :- } q(g(X)),\ t(f(Y)). & q(X) \text{ :- } s(X),\ p(Y,X).
\end{array}
$$

Graphs $\mathcal{G}^A(\mathcal{P})$, $\mathcal{G}(\mathcal{P})$ and $\mathcal{G}^C(\mathcal{P})$ are respectively depicted in Figure 3.1. There are three $SCC$ in $\mathcal{G}(\mathcal{P})$: $C_{\{s\}} = \{s\}$, $C_{\{t\}} = \{t\}$ and $C_{\{p,q\}} = \{p, q\}$ which are the three nodes of $\mathcal{G}^C(\mathcal{P})$.

An ordering among the rules, respecting dependencies pointed out by $\mathcal{G}^C(\mathcal{P})$, is defined next.

**Definition 3.5** A path in $\mathcal{G}^C(\mathcal{P})$ is *strong* if all its edges are labelled with "+". If, on the contrary, there is at least an edge in the path labelled with "-", the path is *weak*. A *component ordering* for a given program $\mathcal{P}$ is a total ordering $\langle C_1, \ldots, C_n\rangle$ of all components of $\mathcal{P}$ s.t., for any $C_i$, $C_j$ with $i < j$, both the following conditions hold:

(i) there are no strong paths from $C_j$ to $C_i$;

(ii) if there is a weak path from $C_j$ to $C_i$, then there must be a weak path also from $C_i$ to $C_j$.[2]

**Example 3.6** Consider the graph $\mathcal{G}^C(\mathcal{P})$ of previous example. Both $C_{\{s\}}$ and $C_{\{t\}}$ are connected to $C_{\{p,q\}}$ through a strong path, while a weak path connects: $C_{\{s\}}$ to $C_{\{t\}}$, $C_{\{t\}}$ to $C_{\{s\}}$, $C_{\{p,q\}}$ to $C_{\{s\}}$ and $C_{\{p,q\}}$ to $C_{\{t\}}$. Both $\gamma_1 = \langle C_{\{s\}}, C_{\{t\}}, C_{\{p,q\}}\rangle$ and $\gamma_2 = \langle C_{\{t\}}, C_{\{s\}}, C_{\{p,q\}}\rangle$ constitute component orderings for the program $\mathcal{P}$.

By means of the graphs defined above, it is possible to identify a set of subprograms (also called *modules*) of $\mathcal{P}$, allowing for a modular bottom-up evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate $\mathcal{P}$ if $p$ appears in $H(r)$. Once a component ordering $\gamma = \langle C_1, \ldots, C_n\rangle$ is given, for each component $C_i$ we define the *module* of $C_i$, denoted by $\mathcal{P}(C_i)$, as the set of all rules $r$ defining some predicate $p \in C_i$ excepting those that define also some other predicate belonging to a lower component (i.e., certain $C_j$ with $j < i$ in $\gamma$). A rule $r$ occurring in a module $\mathcal{P}(C_i)$ defining some predicate $q \in C_i$ is said to be *recursive* if there is some predicate $p \in C_i$ occurring in the positive body of $r$; otherwise, $r$ is said to be an *exit rule*.

[2]Note that, given the component ordering $\gamma$, $C_i$ stands for the i-th component in $\gamma$, and $C_i < C_j$ means that $C_i$ precedes $C_j$ in $\gamma$ (i.e., $i < j$).

**Example 3.7** Consider the program $\mathcal{P}$ of Example 3.4. If we consider the component ordering $\gamma_1$, the corresponding modules are:

$$\begin{array}{lcl} \mathcal{P}\ (C_{\{s\}}) & = & \{\ s(X) \vee t(f(X)) \ \text{:-}\ a(X),\ \text{not}\, q(X).\ \}; \\ \mathcal{P}\ (C_{\{t\}}) & = & \emptyset; \\ \mathcal{P}\ (C_{\{p,q\}}) & = & \{\ p(X,Y) \ \text{:-}\ q(g(X)),\ t(f(Y)).\ ;\ q(X) \ \text{:-}\ s(X),\ p(Y,X).\ ; \\ & & \ q(g(3)).\ \}. \end{array}$$

The modules of $\mathcal{P}$ are defined, according to a component ordering $\gamma$, with the aim of properly instantiating all rules. It is worth remembering that we deal only with safe rules, i.e., all variables appear in the positive body; it is therefore enough to instantiate the positive body. Furthermore, any component ordering $\gamma$ guarantees that, when $r \in \mathcal{P}\ (C_i)$ is instantiated, each nonrecursive predicate $p$ appearing in $B^+(r)$, excepting other predicates (if some) belonging to $C_i$, is defined in a lower component (i.e., in some $C_j$ with $j < i$ in $\gamma$). It is also worth remembering that, according to how the modules of $\mathcal{P}$ are defined, if $r$ is a disjunctive rule, then it is associated only to a unique module $\mathcal{P}\ (C_i)$, chosen in such a way that, among all components $C_j$ such that $comp(a) = C_j$ for some $a \in H(r)$, it always holds $i \leq j$ in $\gamma$ (that is, the disjunctive rule is associated only to the (unique) module corresponding to the lowest component among those "covering" all predicates featuring some instance in the head of $r$). This implies that the set of the modules of $\mathcal{P}$ constitute an exact partition for it.

## 3.3 Finitely-Ground Programs

In this section we introduce a subclass of DLP programs, namely finitely-ground ($\mathcal{FG}$) programs, having some nice computational properties.

Since the set of ground instances of a rule might be infinite (because of the presence of function symbols), it is crucial to try to identify those that really matter in order to compute answer sets. Supposing that $S$ contains all atoms that are potentially true, next definition singles out the relevant instances of a rule.

**Definition 3.8** Given a rule $r$ and a set $S$ of ground atoms, an *S-restricted* instance of $r$ is a ground instance $r'$ of $r$ such that $B^+(r') \subseteq S$. The set of all S-restricted instances of a program $\mathcal{P}$ is denoted as $Inst_{\mathcal{P}}(S)$.

Note that, for any $S \subseteq B_{\mathcal{P}}$, $Inst_{\mathcal{P}}(S) \subseteq Ground(\mathcal{P})$. Intuitively, this helps selecting, among all ground instances, those somehow *supported* by a given set $S$.

**Example 3.9** Consider the following program $\mathcal{P}$:

$$\begin{array}{lll} t(f(1)). & t(f(f(1))). & p(1). \\ p(f(X)) \text{ :- } p(X),\ t(f(X))). & & \end{array}$$

The set $Inst_{\mathcal{P}}(S)$ of all S-restricted instances of $\mathcal{P}$, w.r.t. $S = Facts(\mathcal{P})$ is:

$$\begin{array}{lll} t(f(1)). & t(f(f(1))). & p(1). \\ p(f(1)) \text{ :- } p(1),\ t(f(1)). & & \end{array}$$

The presence of negation allows for identifying some further rules which do not matter for the computation of answer sets, and for simplifying the bodies of some others. This can be properly done by exploiting a modular evaluation of the program that relies on a component ordering.

**Definition 3.10** Given a program $\mathcal{P}$, a component ordering $\langle C_1, \ldots, C_n \rangle$, a set $S_i$ of ground rules for $C_i$, and a set of ground rules $R$ for the components preceding $C_i$, the *simplification* $Simpl(S_i, R)$ of $S_i$ w.r.t. $R$ is obtained from $S_i$ by:

1. *deleting* each rule whose body contains some negative body literal $\text{not } a$ s.t. $a \in Facts(R)$, or whose head contains some atom $a \in Facts(R)$;
2. *eliminating* from the remaining rules each literal $l$ s.t.:
   - $l = a$ is a positive body literal and $a \in Facts(R)$, or
   - $l = \text{not } a$ is a negative body literal, $comp(a) = C_j$ with $j < i$, and $a \notin Heads(R)$.

Assuming that $R$ contains all instances of the modules preceding $C_i$, the simplification $Simpl(S_i, R)$ deletes from $S_i$ all rules whose body is certainly false or whose head is certainly already true w.r.t. $R$, and simplifies the remaining rules by removing from the bodies all literals that are true w.r.t. $R$.

**Example 3.11** Consider the following program $\mathcal{P}$:

$$\begin{array}{lll} t(1). & s(1). & s(2). \\ q(X) \text{ :- } t(X). & & p(X) \text{ :- } s(X),\ \text{not } q(X). \end{array}$$

It is easy to see that $\langle C_1 = \{q\}, C_2 = \{p\}\rangle$ is the only component ordering for $\mathcal{P}$. If we consider $R = EDB(\mathcal{P}) = \{\ t(1).\ ;\ s(1).\ ;\ s(2).\ \}$ and $S_1 = \{q(1) \text{:-}\ t(1).\}$, then $Simpl(S_1, R) = \{q(1).\}$ (i.e., $t(1)$ is eliminated from body).
Considering then $R = \{t(1).\ ;\ s(1).\ ;\ s(2).\ ;\ q(1).\}$ and $S_2 = \{\ p(1)\ \text{:-}\ s(1), \text{not}\ q(1).\ ;\ p(2)\ \text{:-}\ s(2), \text{not}\ q(2).\ \}$, after applying the simplification we have $Simpl(S_2, R) = \{p(2).\}$. Indeed, $s(2)$ is eliminated as it belongs to $Facts(R)$ and not $q(2)$ is eliminated because $comp(q(2)) = C_1$ precedes $C_2$ in the component ordering and the atom $q(2) \notin Heads(R)$; in addition, the rule $p(1)\ \text{:-}\ s(1), \text{not}\ q(1).$ is deleted, since $q(1) \in Facts(R)$.

We are now ready to define an operator $\Phi$ that acts on a module of a program $\mathcal{P}$ in order to:

(i) select only those ground rules whose positive body is contained in a set of ground atoms consisting of the heads of a given set of rules;

(ii) perform a further simplification among these rules by means of the $Simpl$ operator.

**Definition 3.12** Given a program $\mathcal{P}$, a component ordering $\langle C_1, \ldots, C_n\rangle$, a component $C_i$, the module $M = \mathcal{P}\ (C_i)$, a set $X$ of ground rules of $M$, and a set $R$ of ground rules belonging only to EDB($\mathcal{P}$) or to modules of components $C_j$ with $j < i$, let $\Phi_{M,R}(X)$ be the transformation defined as follows: $\Phi_{M,R}(X) = Simpl(Inst_M(Heads(R \cup X)), R)$.

**Example 3.13** Let $\mathcal{P}$ be the program of Example 3.4 where the extension of EDB predicate $a$ is $\{a(1)\}$. Considering the component $C_1 = \{s\}$, the module $M = \mathcal{P}\ (C_1)$, and the sets $X = \emptyset$ and $R = \{a(1)\}$, we have:

$$\begin{aligned}
\Phi_{M,R}(X) &= Simpl(Inst_M(Heads(R \cup X)), R) = \\
&= Simpl(Inst_M(\{a(1)\}), \{a(1).\}) = \\
&= Simpl(\{s(1) \vee t(f(1))\ \text{:-}\ a(1), \text{not}\ q(1).\}, \{a(1).\}) = \\
&= \{s(1) \vee t(f(1))\ \text{:-}\ \text{not}\ q(1).\}.
\end{aligned}$$

The operator defined above has the following important property.

**Proposition 3.14** $\Phi_{M,R}$ always admits a least fixpoint $\Phi^{\infty}_{M,R}(\emptyset)$.

**Proof.** The statement follows from Tarski's theorem [Tarski, 1955]), noting that $\Phi_{M,R}$ is a monotonic operator and that a set of rules forms a meet semilattice under set containment. □

By properly composing consecutive applications of $\Phi^{\infty}$ to a component ordering, we can obtain an instantiation which drops many useless rules w.r.t. answer sets computation.

**Definition 3.15** Given a program $\mathcal{P}$ and a component ordering $\gamma = \langle C_1, \ldots, C_n \rangle$ for $\mathcal{P}$, the ***intelligent instantiation*** $\mathcal{P}^{\gamma}$ of $\mathcal{P}$ for $\gamma$ is the last element $S_n$ of the sequence s.t. $S_0 = EDB(\mathcal{P})$, $S_i = S_{i-1} \cup \Phi^{\infty}_{M_i,S_{i-1}}(\emptyset)$, where $M_i$ is the program module $\mathcal{P}\ (C_i)$.

**Example 3.16** Let $\mathcal{P}$ be the program of Example 3.4 where the extension of EDB predicate $a$ is $\{a(1)\}$; considering the component ordering $\gamma = \langle C_1 = \{s\}, C_2 = \{t\}, C_3 = \{p, q\}\rangle$ we have:

- $S_0 = \{a(1).\}$
- $S_1 = S_0 \cup \Phi^{\infty}_{M_1,S_0}(\emptyset) = \{a(1).\ ;\ s(1) \,\text{v}\, t(f(1)) \text{ :- not } q(1).\}$
- $S_2 = S_1 \cup \Phi^{\infty}_{M_2,S_1}(\emptyset) = \{a(1).\ ;\ s(1) \,\text{v}\, t(f(1)) \text{ :- not } q(1).\}$
- $S_3 = S_2 \cup \Phi^{\infty}_{M_3,S_2}(\emptyset) = \{a(1).\ ;\ s(1) \,\text{v}\, t(f(1)) \text{ :- not } q(1).\ ;\ q(g(3)).\ ;\ p(3,1) \text{ :- } q(g(3)), t(f(1)).\ ;\ q(1) \text{ :- } s(1), p(3,1).\}.$

Thus, the resulting intelligent instantiation $\mathcal{P}^{\gamma}$ of $\mathcal{P}$ for $\gamma$ is:

$$\begin{array}{lll} a(1). & q(g(3)). & s(1) \,\text{v}\, t(f(1)) \text{ :- not } q(1). \\ p(3,1) \text{ :- } q(g(3)), t(f(1)). & & q(1) \text{ :- } s(1), p(3,1). \end{array}$$

We are now ready to define the class of $\mathcal{FG}$ programs.

**Definition 3.17** A program $\mathcal{P}$ is *finitely-ground* ($\mathcal{FG}$) if $\mathcal{P}^{\gamma}$ is finite, for every component ordering $\gamma$ for $\mathcal{P}$.

**Example 3.18** The program of Example 3.4 is $\mathcal{FG}$: $\mathcal{P}^{\gamma}$ is finite both when $\gamma = \langle C_{\{s\}}, C_{\{t\}}, C_{\{p,q\}}\rangle$ and when $\gamma = \langle C_{\{t\}}, C_{\{s\}}, C_{\{p,q\}}\rangle$ (i.e., for the both of two component orderings for $\mathcal{P}$).

## 3.4 Properties of Finitely-Ground Programs

In this section the class of $\mathcal{FG}$ programs is characterized by identifying some key properties.

The next theorem shows that we can compute the answer sets of an $\mathcal{FG}$ program by considering intelligent instantiations, instead of the theoretical (possibly infinite) ground program.

**Theorem 3.19** Let $\mathcal{P}$ be an $\mathcal{FG}$ program and $\mathcal{P}^\gamma$ be the intelligent instantiation of $\mathcal{P}$ w.r.t. a component ordering $\gamma$ for $\mathcal{P}$. Then, $AS(\mathcal{P}) = AS(\mathcal{P}^\gamma)$ (i.e., $\mathcal{P}$ and $\mathcal{P}^\gamma$ have the same answer sets).

**Proof.** Given $\gamma = \langle C_1, \ldots, C_n \rangle$, let denote, as usual, by $M_i$ the program module $\mathcal{P}(C_i)$, and consider the sets $S_0, \ldots, S_n$ as defined in Definition 3.15. Since $\mathcal{P} = \bigcup_{i=0}^{n} M_i$ the theorem can be proven by showing that:

$$AS(S_k) = AS(\ \bigcup_{i=0}^{k} M_i\ ) \text{ for } 1 \leq k \leq n$$

where $M_0$ denotes $EDB(\mathcal{P})$. The equation clearly holds for $k = 0$. Assuming that it holds for all $k \leq j$, we can show that it holds for $k = j + 1$. The equation above can be rewritten as:

$$AS(S_{k-1} \cup \Phi^{\infty}_{M_k, S_{k-1}}(\emptyset)) = AS(\ \bigcup_{i=0}^{k-1} M_i\ \cup M_k)\ ) \text{ for } 1 \leq k \leq n$$

The induction hypothesis allows us to assume that the equivalence $AS(S_{k-1}) = AS(\ \bigcup_{i=0}^{k-1} M_i)$ holds. A careful analysis is needed of the impact that the addition of $M_k$ to $\bigcup_{i=0}^{k-1} M_i$ has on answer sets of $S_k$; in order to prove the theorem, it is enough to show that the set $\Phi^{\infty}_{M_k, S_{k-1}}(\emptyset)$ does not drop any "meaningful" rule w.r.t. $M_k$.

If we disregard the application of the $Simpl$ operator, i.e. we consider the operator $\Phi$ performing only $Inst_{M_k}\ (Heads(S_{k-1} \cup \emptyset))$, then $\Phi^{\infty}_{M_k, S_{k-1}}(\emptyset)$ clearly generates all rules having a chance to have a true body in any answer set; omitted rules have a false body in every answer set, and are therefore irrelevant.

The application of $Simpl$ does not change the scenario: it relies only on previously derived facts, and on the absence of atoms from heads of previously derived ground rules.[3] If a fact $q$ has been derived in a previous component, then any rule

[3]Note that, due to the elimination of true literals performed by the simplification operator $Simpl$, the intelligent instantiation of a rule with a non empty body may generate some facts.

featuring $q$ in the head or $\text{not } q$ in the body is deleted, as it is already satisfied and cannot contribute to any answer set. The simplification operator also drops, from the bodies, positive atoms of lower components appearing as facts, as well as negative atoms belonging to lower components which do not appear in the head of any already generated ground rule. The presence of facts in the bodies is obviously irrelevant, and the deleted negative atoms are irrelevant as well. Indeed, by construction of the component dependency graph, while instantiating a module, all rules defining atoms of lower components have been already instantiated. Thus, atoms of lower components not appearing in the head of any generated rule, have no chances to be true in any answer set. $\square$

**Corollary 3.20** An $\mathcal{FG}$ program has finitely many answer sets, and each of them is finite.

**Theorem 3.21** Given an $\mathcal{FG}$ program $\mathcal{P}$, $AS(\mathcal{P})$ is computable.

**Proof.** Note that by Theorem 3.19, answer sets of $\mathcal{P}$ can be obtained by computing the answer sets of $\mathcal{P}^{\gamma}$ for a component ordering $\gamma$ of choice, which can be easily computed. Then, $\mathcal{P}^{\gamma}$ can be obtained by computing the sequence of fixpoints of $\Phi$ specified in Definition 3.15. Each fixpoint is guaranteed to be finitely computable, since the program is finitely-ground. $\square$

From this property, the main result below immediately follows.

**Theorem 3.22** Cautious and brave reasoning over $\mathcal{FG}$ programs are computable. Computability holds even for non-ground queries.

As the next theorem shows, the class of $\mathcal{FG}$ programs allows for the encoding of any computable function.

**Theorem 3.23** Given a recursive function $f$, there exists a DLP program $\mathcal{P}_f$ such that, for any input $x$ for $f$, $\mathcal{P}_f \cup \theta(x)$ is finitely-ground and $AS(\mathcal{P}_f \cup \theta(x))$ encodes $f(x)$, for $\theta$ a simple function encoding $x$ by a set of facts.

**Proof.** We can build a positive program $\mathcal{P}_f$, which encodes the Turing machine $M_f$ corresponding to $f$ (see Appendix A.1). For any input $x$ to $M_f$, $(\mathcal{P}_f \cup \theta(x))^{\gamma}$ is finite for any component ordering $\gamma$, and $AS(\mathcal{P}_f \cup \theta(x))$ contains an appropriate encoding of $f(x)$. $\square$

Note that recognizing $\mathcal{FG}$ programs is semi-decidable, yet not decidable:

**Theorem 3.24** Recognizing whether $\mathcal{P}$ is an $\mathcal{FG}$ program is r.e.-complete.

**Proof.** Semi-decidability is shown by implementing an algorithm evaluating the sequence given in Definition 3.15, and answering 'yes' if the sequence converges in finite time.

On the other hand, given a Turing machine $M$ and an input tape $x$, it is possible to write a corresponding program $\mathcal{P}_M$ and a set $\theta(x)$ of facts encoding $x$, such that $M$ halts on input $x$ if and only if $\mathcal{P}_M \cup \theta(x)$ is finitely-ground. The program $\mathcal{P}_M$ is the same as in the proof of Theorem 3.23 and reported in Appendix A.1. $\square$

## 3.5 A Recognizable Subclass: Finite-Domain Programs

In this section we single out a subclass of $\mathcal{FG}$ programs, called finite-domain ($\mathcal{FD}$) programs, which, besides sharing the nice decidability properties of $\mathcal{FG}$ programs, ensures the decidability of recognizing membership in the class.

We first provide the notion of finite-domain ($\mathcal{FD}$) arguments.

**Definition 3.25** Given a program $\mathcal{P}$, the set of *finite-domain arguments ($\mathcal{FD}$ arguments)* of $\mathcal{P}$ is the maximal (w.r.t. inclusion) set $FD(\mathcal{P})$ of arguments of $\mathcal{P}$ such that, for each argument $q[k] \in FD(\mathcal{P})$, every rule $r$ with head predicate $q$ satisfies the following condition. Let $t$ be the term corresponding to argument $q[k]$ in the head of $r$. Then,

- either $t$ is variable-free, or
- $t$ is a subterm[4] of (the term of) some $\mathcal{FD}$ argument of a positive body predicate, or
- every variable appearing in $t$ also appears in (the term of) a $\mathcal{FD}$ argument of a positive body predicate which is not recursive with $q[k]$.

If all arguments of the predicates of $\mathcal{P}$ are $\mathcal{FD}$, then $\mathcal{P}$ is said to be an $\mathcal{FD}$ *program*.

[4]The condition can be made less strict considering other notions, as, e.g., the *norm* of a term [Bossi *et al.*, 1994; Bruynooghe *et al.*, 2007; Schreye and Decorte, 1994].

Intuitively, $\mathcal{FD}$ arguments can range only on a finite set of different ground values. Observe that $FD(\mathcal{P})$ is well-defined; indeed, it is easy to see that there always exists, and it is unique, a maximal set satisfying Definition 3.25 (trivially, given two sets $A_1$ and $A_2$ of $\mathcal{FD}$ arguments for a program $\mathcal{P}$, the set $A_1 \cup A_2$ is also a set of $\mathcal{FD}$ arguments for $\mathcal{P}$).

**Example 3.26** The following is an example of $\mathcal{FD}$ program:

$$q(f(0)). \qquad q(X) \text{ :- } q(f(X)).$$

Indeed q[1] is the only argument in the program and it is an $\mathcal{FD}$ argument, since the two occurrences of $q[1]$ in a rule head satisfy first and second condition of Definition 3.25, respectively.

**Example 3.27** The following is not an $\mathcal{FD}$ program:

$$\begin{array}{ll} q(f(0)). & q(X) \text{ :- } q(f(X)). \\ s(f(X)) \text{ :- } s(X). & v(X) \text{ :- } q(X),\ s(X). \end{array}$$

We have that all arguments belong to $FD(\mathcal{P})$, except for $s[1]$. Indeed, $s[1]$ appears as head argument in the third rule with term $f(X)$, and:$(i)$ $f(X)$ is not variable-free; $(ii)$ $f(X)$ is not a subterm of some term appearing in a positive body $\mathcal{FD}$ argument; $(iii)$ there is no positive body predicate which is not recursive with $s$ and contains $X$.

By the following theorems we now point out two key properties of $\mathcal{FD}$ programs.

**Theorem 3.28** Recognizing whether $\mathcal{P}$ is an $\mathcal{FD}$ program is decidable.

**Proof.** An algorithm deciding whether $\mathcal{P}$ is $\mathcal{FD}$ or not can be defined as follows. Arguments of predicates in $\mathcal{P}$ are all supposed to be $\mathcal{FD}$ at first. If at least one rule is found, such that for an argument of an head predicate none of the three conditions of Definition 3.25 holds, then $\mathcal{P}$ is recognized as not being an $\mathcal{FD}$ program. If no such rule is found, the answer is positive. $\square$

**Theorem 3.29** Every $\mathcal{FD}$ program is an $\mathcal{FG}$ program.

**Proof.** Given an $\mathcal{FD}$ program $\mathcal{P}$, it is possible to find *a priori* an upper bound for the maximum nesting level[5] of the terms appearing in $\mathcal{P}^{\gamma}$, for any component ordering $\gamma$ for $\mathcal{P}$. This is given by $max\_nl = (n+1) * m$, where $m$ is the maximum nesting level of the terms in $\mathcal{P}$, and $n$ is the number of components in $\gamma$. Indeed, given that $\mathcal{P}$ is an $\mathcal{FD}$ program, it is easy to see that the maximum nesting level cannot increase because of recursive rules, since, in this case, the second condition of Definition 3.25 forces a sub-term relationships between head and body predicates. Hence, the maximum nesting level can increase only because of body-head dependencies among predicates of different components. We can now compute the set of all possible ground terms $t$ obtained by combining all constants and function symbols appearing in $\mathcal{P}$, such that the nesting level of $t$ is less or equal to $max\_nl$. This is a finite set, and clearly a superset of the ground terms appearing in $\mathcal{P}^{\gamma}$. Thus, $\mathcal{P}^{\gamma}$ is necessarily finite. $\Box$

The results above allow us to state the following properties for $\mathcal{FD}$ programs.

**Corollary 3.30** Let $\mathcal{P}$ be an $\mathcal{FD}$ program, then:

1. $\mathcal{P}$ has finitely many answer sets, and each of them is finite.
2. $AS(\mathcal{P})$ is computable;
3. skeptical and credulous reasoning over $\mathcal{P}$ are computable. Computability holds even if the query at hand is not ground.

[5]The nesting level of a ground term is defined inductively as follows: (i) a constant term has nesting level zero; (ii) a functional term $f(t_1, \ldots, t_n)$ has nesting level equal to the maximum nesting level among $t_1, \ldots, t_n$ plus one.

# Chapter 4

# Bottom-Up Evaluation of Finitely-Recursive Queries via Magic Sets

In this chapter the Magic Sets technique is described and, a suitable adaptation of the rewriting algorithm underlying this technique is presented. The Magic Sets transformation originate as an optimization technique in the context of deductive database. Here, we use it in order to allow a safe bottom-up evaluation of finitely ground queries. After describing the standard technique, we illustrate the motivations inducing to the use of this sort of program transformation in the context of ASP with functions and then we present the adapted rewriting algorithm and many examples of applications. Some relevant computational results are then proved.

The chapter is organized as follows:

- Section 4.1 describes the standard Magic Sets technique.
- In Section 4.2 we first motivate the use of this technique in the context of ASP with functions and then present an adaptation of the standard rewriting algorithm, supplying also several examples of rewritten programs.
- Finally, in Section 4.3 some key properties about the program outputted by the presented rewriting algorithm are proved.

## 4.1 Magic Sets

The *Magic Sets* method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of additional rules, which narrow the computation to what is relevant for answering the query. Intuitively, the goal of the Magic Sets method is to use the constants appearing in the query to reduce the size of the instantiation by eliminating 'a priori' a number of ground instances of the rules which cannot contribute to the derivation of the query goal.

This method has originally been defined in [Bancilhon *et al.*, 1986] for non-disjunctive Datalog (i.e. with no function symbols) queries only. Afterwards, many generalizations have been proposed. In the context of ASP, the generalization to the disjunctive case [Cumbo *et al.*, 2004] and to Datalog with (possibly unstratified) negation [Faber *et al.*, 2007] are worth remembering.

We next provide a brief and informal description of the Magic Sets rewriting technique. The reader is referred to [Ullman, 1989] for a detailed presentation. The method is structured in four main phases which are informally illustrated below by example, considering the query $path(1, 5)$ on the following program:

$$\begin{array}{l} path(X, Y) \text{ :- } edge(X, Y). \\ path(X, Y) \text{ :- } edge(X, Z), path(Z, Y). \end{array}$$

**1. Adornment Step:** The key idea is to materialize, by suitable adornments, binding information for $IDB$ predicates which would be propagated during a top-down computation. Adornments are strings of the letters b and f, denoting 'bound' and 'free' respectively, for each argument of an $IDB$ predicate. First, adornments are created for query predicates so that an argument occurring in the query is adorned with the letter b if it is a constant, or with the letter f if it is a variable. The query adornments are then used to propagate their information into the body of the rules defining it, simulating a top-down evaluation. It is worth noting that adorning a rule may generate new adorned predicates. Thus, the adornment step is repeated until all adorned predicates have been processed, yielding the *adorned program*.

For simplicity of the presentation, we next adopt the 'basic' Magic Sets method as defined in [Bancilhon *et al.*, 1986], in which binding information within a rule comes only from the adornment of the head predicate, from $EDB$ predicates in the (positive) rule body, and from constants. In other words, an adornment of type b is induced by a constant, or by a variable occurring either as an argument in a position of type b in the head predicate or in an $EDB$ predicate. On the contrary, in the so-called ''generalized'' Magic Sets method [Beeri and Ramakrishnan, 1987], bindings may also be generated by $IDB$ predicates in rule bodies. In particular, an appropriate strategy for Sideways Information Passing (SIP) has to be specified for each rule, fixing the body ordering and the way in which bindings are generated. In this respect, the 'basic' method uses a particular, predetermined SIP for all rules.

**Example 4.1** Adorning the query $path(1,5)$ generates the adorned predicate $path^{bb}$ since both arguments are bound, and the adorned program is:

$$\begin{array}{l} path^{bb}(X,Y) \;\text{:-}\; edge(X,Y). \\ path^{bb}(X,Y) \;\text{:-}\; edge(X,Z), path^{bb}(Z,Y). \end{array}$$

**2. Generation Step:** The adorned program is used to generate *magic rules*, which simulate the top-down evaluation scheme and single out the atoms which are relevant for deriving the input query. Let the *magic version* $\text{magic}(p^{\alpha}(\bar{t}))$ for an adorned atom $p^{\alpha}(\bar{t})$ be defined as the atom $magic\_p^{\alpha}(\bar{t}')$, where $\bar{t}'$ is obtained from $\bar{t}$ by eliminating all arguments corresponding to an $f$ label in $\alpha$, and where $magic\_p^{\alpha}$ is a new predicate symbol obtained by attaching the prefix '$magic\_$' to the predicate symbol $p^{\alpha}$. Then, for each adorned atom $A$ in the body of an adorned rule $r_a$, a magic rule $r_m$ is generated such that (i) the head of $r_m$ consists of $\text{magic}(A)$, and (ii) the body of $r_m$ consists of the magic version of the head atom of $r_a$, followed by all the ($EDB$) atoms of $r_a$ which can propagate the binding on $A$.

**3. Modification Step:** The adorned rules are subsequently modified by including magic atoms generated in Step 2 in the rule bodies, which limit the range of the head variables avoiding the inference of facts which cannot contribute to deriving the query. The resulting rules are called *modified rules*. Each adorned rule $r_a$ is modified as follows. Let $H$ be the head atom of $r_a$. Then, atom $\text{magic}(H)$ is inserted in the body of the rule, and the adornments of all other predicates are stripped off. We would like to point out that stripping off the adornments serves mainly for facilitating the equivalence proofs; one may also leave the adornments (also in the query) intact, as it was done in the original definition of Magic Sets.

**4. Processing of the Query:** Let the query goal be the adorned $IDB$ atom $g^{\alpha}$, the *magic seed* or query rule $\text{magic}(g^{\alpha})$. (a fact) is produced. For instance, in our example we generate $magic\_path^{bb}(1,5)$.

The complete rewritten program consists of the magic, modified, and query rules.

**Example 4.2** The complete rewriting of our example program is:

$magic\_path^{bb}(1, 5).$
$magic\_path^{bb}(Z, Y)$ :- $magic\_path^{bb}(X, Y), edge(X, Z).$
$path(X, Y)$ :- $magic\_path^{bb}(X, Y), edge(X, Y).$
$path(X, Y)$ :- $magic\_path^{bb}(X, Y),\ edge(X, Z),\ path(Z, Y).$

Note that the adorned rule $path^{bb}(X, Y)$ :- $edge(X, Y)$. does not produce any magic rule, since it does not contain any adorned predicate in its body. Hence, we only generate $magic\_path^{bb}(Z, Y)$ :- $magic\_path^{bb}(X, Y), edge(X, Z)$. Moreover, the last two rules above are those resulting from the modification of the adorned program in Example 4.1.

In this rewriting, $magic\_path^{bb}(X, Y)$ represents the start- and end-nodes of all potential sub-paths of paths from $1$ to $5$. Therefore, when answering the query, only these sub-paths will be actually considered in bottom-up computations.

## 4.2 Magic Sets and Finitely-Recursive Programs

As remarked in Section 2.3, positive finitely-recursive programs are the simplest finitary programs. Being finitary, they enjoy all nice properties of this class of programs. In particular, consistency checking is decidable as well as reasoning if ground queries are considered (reasoning is semi-decidable in case of non ground queries). Unfortunately, even if a program $\mathcal{P}$ is finitely-recursive, it is not suited for the bottom-up evaluation for two main reasons:

1. A bottom-up evaluation of a finitely-recursive program would generate some new terms at each iteration, thus iterating for ever.

   **Example 4.3** Consider the following program defining the natural numbers:

   $nat(0).$
   $nat(s(X))$ :- $nat(X).$

   The program is positive and finitely-recursive, so every ground query (such as for instance $nat(s(s(s(0))))?$) can be answered in a top-down fashion; but its bottom-up evaluation would iterate for ever, as, for any positive integer $n$, the n-th iteration would derive the new atom $nat(s^n(0))$.

2. Finitely-recursive programs do not enforce the range of an head variable to be restricted by a body occurrence (i.e., bottom-up safety is not required). A bottom-up evaluation of these unsafe rules would cause the derivation of non-ground facts which is not admissible.

**Example 4.4** Consider the following program, defining the reachability among vertices of a graph:

$$\begin{array}{l} reachable(X, X). \\ reachable(X, Y) \text{ :- } reachable(X, Z), arc(Z, Y). \end{array}$$

The program is positive and finitely-recursive, so any ground query can be computed top-down; but its bottom-up evaluation is unfeasible, since the first iteration would generate $\{reachable(X, X)\}$ representing an infinite set of atoms. In this case, $node(X)$ could be added to the body of the first rule, rendering safe the variable $X$ and then making possible the program bottom-up evaluation. But, this is not always the case, as shown in the next example.

**Example 4.5** Interpreting the function symbol $s$ as the successor of a natural number, the following program defines the comparison operator 'less than' between two natural numbers:

$$\begin{array}{l} lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } lessThan(X, Y). \end{array}$$

The program is positive and finitely-recursive, thus any ground query can be easily answered top-down. For instance, the query $lessThan(s(0), s(s(0)))?$ results true, whereas the query $lessThan(s(s(0)), s(s(0)))?$ is false. Bottom-up evaluation of this program is unfeasible, since the first iteration would generate the set consisting of an infinite number of atoms having the form $\{lessThan(X, s(X))\}$.

### 4.2.1 Finitely-Recursive Queries

A finitely-recursive program is such that *every* ground query on it, depends only on a finite set of other ground atoms. There are programs that can not be classified as being finitely-recursive but, there exists a subset of all possible ground queries, for which the above mentioned property holds.

**Example 4.6** Consider the following program:

$$\begin{array}{l} lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } lessThan(X, Y). \\ q(f(f(0))). \\ q(X) \text{ :- } q(f(X)). \\ r(X) \text{ :- } lessThan(X, Y), q(X). \end{array}$$

If the whole program is considered, we note that it is surely not finitely-recursive. In fact, considering a whatsoever constant value $c$, we have that both atoms likes $q(c)$ and atoms likes $r(c)$, depend on an infinite number of other atoms. In the former case, because of a never terminating recursion, in the latter, because of: both the local variable $Y$ and the dependence on $q$ atoms. Nevertheless, if we consider query atoms having as predicate $lessThan$, these continue to depend only on a finite set of other atoms.

Here and throughout the next section, we restrict our attention to the case of a ground query on a logic program, such that the subprogram 'relevant' in order to answer the query is positive and finitely-recursive. Exploiting the notion of dependency among atoms and dependency graph of a program ($LDG(\mathcal{P})$) defined in Section 2.2, this kind of queries can be defined as follows:

**Definition 4.7** Given a program $\mathcal{P}$ and a *ground query* $\mathcal{Q}$ over $\mathcal{P}$ we say that $\mathcal{Q}$ is *finitely-recursive* if and only if:

- $\mathcal{Q}$ depends only on a finite set $A$ of other ground atoms in $Ground(\mathcal{P})$, i.e starting from the node in $LDG(\mathcal{P})$ corresponding to the atom of $\mathcal{Q}$, only a finite set $A$ of nodes is reachable;
- called $R_{\mathcal{Q}}(\mathcal{P})$ be the set of rule $r \in Ground(\mathcal{P})$ such that $H(r) \in A$, $R_q(\mathcal{P})$ is a positive program.

In other words, we can say that a ground query $\mathcal{Q}$ on a program $\mathcal{P}$ is finitely-recursive if and only if the relevant subprogram of $\mathcal{P}$ for $\mathcal{Q}$ ($R_{(\mathcal{P},\mathcal{Q})}$) is positive and finitely-recursive.

For instance, if we consider the program of Example 4.6, we have that all atoms likes $lessThan(c_1, c_2)$, where $c_1$ and $c_2$ are constant values, are examples of finitely-recursive queries on that program.

In the next subsection we design a suitable adaptation of the Magic Sets rewriting technique for logic programs with functions, that allows for a bottom-up evaluation of finitely-recursive queries.

### 4.2.2 Rewriting Algorithm

If we restrict our attention to finitely-recursive queries, some steps of the Magic Sets technique reported in Section 4.1 can be significantly simplified. In particular, the adornment phase is no longer needed since all the $IDB$ predicates involved in the query evaluation would have a completely bound adornment. In fact:

- the query is ground, so it would have a completely bound adornment;
- all rules involved in a top-down evaluation of the query can not have local variables (i.e. variables appearing only in the body of the rule) since the relevant subprogram is supposed to be finitely-recursive (see Section 2.3). So, starting from a ground query, a complete bound adornment from the head to all the $IDB$ predicates of the body would be propagated.

In the generation step, it is not necessary to include any other atom, different from the magic version of the head atom, in the body of the generated magic rule. Again, this is due to the absence of local variables, so that all the needed bindings are provided through the magic version of the head atom.

The algorithm $MS_{FR}$ in Figure 4.1 implements the Magic Sets method for finitely-recursive queries. Its input is a positive finitely-recursive program $\mathcal{P}$ and a ground query $\mathcal{Q}$. The algorithm $MS_{FR}$ outputs a program $RW(\mathcal{Q}, \mathcal{P})$ consisting of a set of *modified* and *magic* rules (denoted by $\mathit{modifiedRules}$ and $\mathit{magicRules}$, respectively). The algorithm generates modified and magic rules on a rule-by-rule basis. To this end, it exploits a stack $S$ for storing all predicates that are still to be used for propagating the query binding. At first, the set of magic rules is initialized with the magic version of the query and the query atom is pushed on $S$. At each step, an element $u$ is removed from $S$. If the predicate $u$ has not been already considered (the auxiliary variable $\mathit{Done}$ is used to check about this), all the rules defining $u$ are processed one-at-a-time. For each of such rules, if its body is not empty or there is at least a variable in the rule, then a modified rule is created and a set of magic rules are generated (one for each $IDB$ atom in the body). Moreover, every $IDB$ predicate appearing in the body is pushed on the stack $S$. In case the rule is a fact, i.e. the body is empty and the are no variables, it is added to the $\mathit{modifiedRules}$ set as it is. Finally, after all the predicates involved in the query evaluation have been processed, the algorithm outputs the program $RW(\mathcal{Q}, \mathcal{P})$ obtained as the union of all modified rules and generated magic rules.

```
Input: a program $\mathcal{P}$ and a finitely-recursive query $\mathcal{Q} = \mathtt{g}(\bar{\mathtt{c}})$? on $\mathcal{P}$
Output: the rewritten program $RW(\mathcal{Q},\mathcal{P})$.
Main Vars: $S$: stack of predicates to rewrite;
        $modifiedRules(\mathcal{Q},\mathcal{P})$,$magicRules(\mathcal{Q},\mathcal{P})$: set of rules;
        $Done$: set of predicates;
begin
 1. $modifiedRules(\mathcal{Q},\mathcal{P})$ := $\emptyset$; $Done$ := $\emptyset$
 2. $magicRules(\mathcal{Q},\mathcal{P})$ := $\{\mathtt{magic\_g}(\bar{\mathtt{c}}).\}$;
 3. $S$.push(g);
 4. while $S \neq \emptyset$ do
 5.     $u$ := $S$.pop();
 6.     if $u \notin Done$ then
 7.         $Done$ := $Done \cup \{u\}$;
 8.         for each $r \in \mathcal{P}$ : $r$ is a defining rule for $u$ do
 9.             if $B(r) \neq \emptyset$ or $Vars(r) \neq \emptyset$ then
                    // let $r$ be $\mathtt{u}(\bar{\mathtt{t}})$ :- $\mathtt{v_1}(\overline{\mathtt{t_1}}), ..., \mathtt{v_n}(\overline{\mathtt{t_n}})$.
10.                 $modifiedRules(\mathcal{Q},\mathcal{P})$ := $modifiedRules(\mathcal{Q},\mathcal{P}) \cup$
                        $\{\mathtt{u}(\bar{\mathtt{t}})$ :- $\mathtt{magic\_u}(\bar{\mathtt{t}}), \mathtt{v_1}(\overline{\mathtt{t_1}}), ..., \mathtt{v_n}(\overline{\mathtt{t_n}}).\}$;
11.                 for each $v_i$ : $v_i \in B(r)$ and $v_i \in IDB(\mathcal{P})$ do
12.                     $magicRules(\mathcal{Q},\mathcal{P})$ := $magicRules(\mathcal{Q},\mathcal{P}) \cup$
                            $\{\mathtt{magic\_v_i}(\overline{\mathtt{t_i}})$ :- $\mathtt{magic\_u}(\bar{\mathtt{t}}).\}$;
13.                     $S$.push($v_i$);
14.                 end for
15.             else
16.                 $modifiedRules(\mathcal{Q},\mathcal{P})$ := $modifiedRules(\mathcal{Q},\mathcal{P}) \cup r$
17.             end if
18.         end for
19.     end if
20. end while
21. $RW(\mathcal{Q},\mathcal{P})$:=$magicRules(\mathcal{Q},\mathcal{P}) \cup modifiedRules(\mathcal{Q},\mathcal{P})$;
22. return $RW(\mathcal{Q},\mathcal{P})$;
end.
```

Figure 4.1: Magic Sets rewriting algorithm for finitely-recursive queries

### 4.2.3 Examples of Finitely-Recursive Queries Rewriting

At first, we will consider the finitely-recursive query $\mathcal{Q} = nat(s(s(0)))$? on the program $\mathcal{P}$ of Example 4.3. For this example, we will depict, step by step, the execution performed by the $MS_{FR}$ algorithm.

**Example 4.8** After initialization of variables, the algorithm (lines $1-2$) generates the first magic rule deriving from the query:

$$magic\_nat(s(s(0))).$$

After, the predicate $nat$ is pushed onto the stack $S$ (line 3) and the first iteration of the cycle (line 4) starts. The predicate $nat$ is extracted from the stack and is marked as already done (lines $5-7$). In this case, this is the only predicate to be considered. All defining rules for $nat$ are then processed (lines $8-18$). The first rule defining $nat$ is a fact ($nat(0).$): both conditions of line 9 are false (here $Vars(r)$ is used to denote the set of variables occurring in the rule $r$), so the rule is added to the $ModifiedRules$ set (line 16) unchanged. The second rule defining $nat$ is a recursive rule. First of all, the modified rule:

$$nat(s(X)) \text{ :- } magic\_nat(s(X)), nat(X).$$

is added to the $ModifiedRules$ set (line 10). Then, the following magic rule for the $nat$ atom occurring in the body is generated, and the $nat$ predicate is pushed onto the stack $S$ (lines $11-14$):

$$magic\_nat(X) \text{ :- } magic\_nat(s(X)).$$

Then, the second iteration starts but it immediately ends, as the predicate extracted from the stack $S$ is the already considered predicate $nat$. Finally, being $S$ empty, there are no further iterations and the algorithm outputs the following complete rewritten program $RW(\mathcal{Q},\mathcal{P})$:

$$\begin{array}{l} magic\_nat(s(s(0))). \\ magic\_nat(X) \text{ :- } magic\_nat(s(X)). \\ nat(0). \\ nat(s(X)) \text{ :- } magic\_nat(s(X)), nat(X). \end{array}$$

**Example 4.9** Applying the algorithm to the program $\mathcal{P}$ of Example 4.5 w.r.t. the query $\mathcal{Q} = lessThan(s(s(0)), s(0))$? we obtain the following rewritten program $RW(\mathcal{Q},\mathcal{P})$:

$$\begin{array}{l} magic\_lessThan(s(s(0)), s(0)). \\ magic\_lessThan(X,Y) \text{ :- } magic\_lessThan(X, s(Y)). \\ lessThan(X, s(X)) \text{ :- } magic\_lessThan(X, s(X)). \\ lessThan(X, s(Y)) \text{ :- } magic\_lessThan(X, s(Y)), lessThan(X,Y). \end{array}$$

All queries on the programs for manipulating lists, reported in Section 2.3, are finitely-recursive and then can be rewritten using the $MS_{FR}$ algorithm.

**Example 4.10** Let us consider the query: $\mathcal{Q} = reverse([a,b,c,d],[d,c,b,a])$? on the following program $\mathcal{P}$:

$reverse(L,R)$ :- $sup\_reverse(L,[\,],R)$.
$sup\_reverse([\,],R,R)$.
$sup\_reverse([X|T_1],L,R)$ :- $sup\_reverse(T_1,[X|L],R)$.

The output rewritten program $RW(\mathcal{Q},\mathcal{P})$ is:

$magic\_reverse([a,b,c,d],[d,c,b,a])$.
$magic\_sup\_reverse(L,[\,],R)$ :- $magic\_reverse(L,R)$.
$magic\_sup\_reverse(T_1,[X|L],R)$ :- $magic\_sup\_reverse([X|T_1],L,R)$.
$reverse(L,R)$ :- $magic\_reverse(L,R), sup\_reverse(L,[\,],R)$.
$sup\_reverse([\,],R,R)$ :- $magic\_sup\_reverse([\,],R,R)$.
$sup\_reverse([X|T_1],L,R)$ :- $magic\_sup\_reverse([X|T_1],L,R)$,
$sup\_reverse(T_1,[X|L],R)$

## 4.3 Properties of Rewritten Programs

Let $RW(\mathcal{Q},\mathcal{P})$ denote the output of the $MS_{FR}$ algorithm, having as input a program $\mathcal{P}$ and a finitely-recursive query $\mathcal{Q}$. Next, we are going to prove some relevant results about the $RW(\mathcal{Q},\mathcal{P})$ program. First of all we give a query equivalence property.

**Theorem 4.11** Given a ground query $\mathcal{Q}$ on a program $\mathcal{P}$, if $\mathcal{Q}$ is finitely-recursive on $\mathcal{P}$, then $\mathcal{P} \models \mathcal{Q}$ if and only if $RW(\mathcal{Q},\mathcal{P}) \models \mathcal{Q}$.

**Proof.** Query equivalence has already been proved for the 'standard' Magic Sets technique (see e.g. [Ullman, 1989]). The algorithm presented in section 4.2 differs from the standard one for some aspects but all of them have no consequences on the correctness of the transformation. We next recall the differences against the standard Magic Sets technique, and illustrate why the introduced changes do not affect the query equivalence result:

1. Adornment is not performed because the structure of finitely-recursive queries implies that only a completely bound adornment would be derived (see Section 4.2.2). Anyway, all the $IDB$ predicates involved in the top-down query evaluation are correctly identified and processed, likewise the standard algorithm does. Both rules modification and magic rules generation are performed considering all processed $IDB$ predicate as having an implicit completely bound adornment.

2. Only the magic version of the head atom is included in the body of each generated magic rule. According to the standard technique, all $EDB$ atoms which can propagate the binding on variables occurring in the currently processed atom, should be added to the body. In case of a finitely-recursive query, all variables occurring in the body of a rule, necessarily occurs also in its head (no local variables are admitted). So, we know 'a priori' that no further atom is needed.

3. The $MS_{FR}$ algorithm acts on a rule-by-rule basis, instead of performing the different phases on the entire program. This approach, adopted also in [Cumbo *et al.*, 2004] and [Faber *et al.*, 2007], allows to improve efficiency, as each rule of the original program is processed just once. The resulting rewritten program is not affected by this change.

□

Next, we are going to prove a result about the efficiency of the rewriting algorithm. At this aim, we need to introduce the definition of what we mean for size of a program.

**Definition 4.12** Let $\mathcal{P}$ be a (non ground) logic program. The *size* of $\mathcal{P}$, denoted by $\|\mathcal{P}\|$, is the number of atoms occurring in $\mathcal{P}$. It is worth noting that, if the same atom occurs in two different rules of $\mathcal{P}$ it is counted twice.

For instance, the program $\mathcal{P}$ in Example 4.10 has size $\|\mathcal{P}\| = 5$.

**Theorem 4.13** Given a finitely-recursive query $\mathcal{Q}$ on a program $\mathcal{P}$, the size of $RW(\mathcal{Q}, \mathcal{P})$ is linear in the size of the input $\mathcal{P}$ and $\mathcal{Q}$, in symbols: $\|RW(\mathcal{Q}, \mathcal{P})\| = O(\|\mathcal{P}\|)$.

**Proof.** $RW(\mathcal{Q}, \mathcal{P})$ is obtained as the union of two set of rules: $\mathit{modifiedRules}(\mathcal{Q}, \mathcal{P})$ and $\mathit{magicRules}(\mathcal{Q}, \mathcal{P})$.

In the worst case, the number of atoms in the first set is given by the number of atoms in $\mathcal{P}$ plus as many atoms as the number of rules in $\mathcal{P}$ (at most one magic atom is added for each rule in $\mathcal{P}$). So, $\|modifiedRules(\mathcal{Q}, \mathcal{P})\|$ is definitely equal to $O(\|\mathcal{P}\|)$.

Let us consider now the $\mathit{magicRules}(\mathcal{Q}, \mathcal{P})$ program. For each $IDB$ atom occurring in the body of a rule in $\mathcal{P}$, at most one magic rule with exactly two atoms is generated. Then, in the worst case, the number of atoms in $\mathit{magicRules}(\mathcal{Q}, \mathcal{P})$ is not greater than $2 \cdot \|\mathcal{P}\|$.

From considerations done for these two sets, immediately follows the statement $\|RW(\mathcal{Q},\mathcal{P})\| = O(\|\mathcal{P}\|)$. $\Box$

The next theorem points out a relationship between finitely-recursive queries and finitely-ground programs.

**Theorem 4.14** Given a ground query $\mathcal{Q}$ on a program $\mathcal{P}$, if $\mathcal{Q}$ is finitely-recursive on $\mathcal{P}$, then $RW(\mathcal{Q},\mathcal{P})$ is finitely-ground.

**Proof.** Let $P_{magic}$ be the set of predicates defined by rules in $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$ and let $P_{mod}$ be the set of predicates defined by $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$. We observe that $P_{magic} \cap P_{mod} = \emptyset$ and all component ordering $\gamma$ for $RW(\mathcal{Q},\mathcal{P})$ (see Definition 3.5) will be such that: if $p \in P_{magic}$ and $C_i = \mathit{comp}(p)$ then $C_i$ will precede every component $C_j = \mathit{comp}(q)$ with $q \in P_{mod}$. This means that, in the modular bottom-up evaluation performed by the intelligent instantiation (see Definition 3.15), all rules in $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$ will precede rules in $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$.

We know that $\mathcal{Q}$ is finitely-recursive, i.e. it depends only on a finite set of other ground atoms in $Ground(\mathcal{P})$. But rules in $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$ are appositely built in order to exclusively derive atoms on which the query $\mathcal{Q}$ depends. So, starting from the ground query atom $\mathcal{Q}$, each element of the sequence $S_i$ in Definition 3.15 is necessarily finite for modules of $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$. But, magic atoms give binding to all variables occurring in the head of rules in $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$, restricting their domain of possible values, so elements $S_i$ will be finite also for all modules of $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$. This is enough to prove that $RW(\mathcal{Q},\mathcal{P})$ is finitely-ground. $\Box$

This last result is relevant because it implies that all nice properties of finitely-ground programs hold for rewritten finitely-recursive queries too. In particular, bottom-up computability of the answer set and then decidability of reasoning.

**Example 4.15** If we consider the program $RW(\mathcal{Q},\mathcal{P})$ of Example 4.8 we can observe that it is safe (according to the bottom-up safety notion) and its resulting intelligent instantiation is finite:

$magic\_nat(s(s(0))).$
$magic\_nat(s(0))$ :- $magic\_nat(s(s(0))).$
$magic\_nat(0)$ :- $magic\_nat(s(0)).$
$nat(0).$
$nat(s(s(0)))$ :- $magic\_nat(s(s(0))), nat(s(0)).$
$nat(s(0))$ :- $magic\_nat(s(0)), nat(0).$

The above ground program has the following unique finite answer set: $\{magic\_nat(s(s(0))), magic\_nat(s(0)), magic\_nat(0), nat(0), nat(s(0)), nat(s(s(0)))\}$. The answer to the query $\mathcal{Q}$ is then 'yes'.

**Example 4.16** Let us now consider the program $RW(\mathcal{Q}, \mathcal{P})$ of Example 4.9. Also this program, differently from the originating $\mathcal{P}$ program, can be safely bottom-up evaluated. Its intelligent instantiation is finite and results to be:

$magic\_lessThan(s(s(0)), s(0)).$
$magic\_lessThan(s(s(0)), 0)$ :- $magic\_lessThan(s(s(0)), s(0)).$

The above ground program has the following unique finite answer set: $\{magic\_lessThan(s(s(0)), s(0)), magic\_lessThan(s(s(0)), 0)\}$. So, the answer to the query $\mathcal{Q}$ is 'no'.

# Chapter 5

# Extending ASP with External Functions

This chapter introduces a formal framework for accommodating external source of computation in the context of Answer Set Programming. In this framework, the notion of `VI` programs (where `VI` stands for Value Invention) is introduced. In practice, `VI` programs are logic programs enriched with the notion of *external predicates*. External predicates model the mechanism of value invention by taking input from a given set of values and returning (possibly newly invented) values. These are computed by means of an associated evaluation function (called *oracle*). We prove that, although assuming as decidable the external functions defining oracles, the consistency check of `VI` programs is, in general, undecidable. Therefore, it is important to investigate on (nontrivial) sub-classes of decidable programs. We address this problem identifying a *safety* condition for granting decidability of `VI` programs.

The chapter is organized as follows:

- Section 5.1 surveys the usage of value invention in logic programming and introduce the syntax and semantics of elements we added to the standard disjunctive language in order to support value invention.
- In Section 5.2 we investigate the consequences of allowing value invention, in terms of undecidability of consistency checking for `VI` programs.
- In Section 5.3 we show how to cope with value invention, providing a characterization of those programs that can be computed even if external sources of computation are exploited.

- Section 5.4 introduces a safety condition defining the class of ltop-restricted programs. This class enjoys the ‘finite grounding property’ characterizing those programs that can be computed with a finite ground program. Decidability of consistency checking is thus ensured.
- Finally, in Section 5.5 we show that `VI`-restrictedness can be checked in polynomial time in the size of the non-ground program.

## 5.1 Value Invention in ASP Programs

The notion of 'value invention' has been formerly adopted in the database field (see e.g. [Abiteboul and Vianu, 1991; Cabibbo, 1996]) for denoting mechanisms aimed at allowing the introduction of new domain elements in a logic based query language. Indeed, applications of logic programming often need to deal with a universe of symbols which is not known a priori. We can divide these demands in two main categories:

(*i*) 'Constructivist' demands: the majority of logic programming languages has the inherent capability to build new symbols from pre-existing ones, e.g. by means of traditional constructs like functional terms. Furthermore, manipulating and creating complex data structures other than simple constant symbols, such as sets, or lists, is a source of value invention. Also, controlled value invention constructs have been proposed in order to deal with the creation of new object identifiers in object oriented deductive databases [Hull and Yoshikawa, 1990].

(*ii*) 'Externalist' demands: in this setting, non-predictable external sources of knowledge have to be dealt with. For instance, in the Semantic Web area, rule based languages must explicitly embrace the case where ontologies and the universe of individuals is external and not known a priori [Eiter *et al.*, 2005], or is explicitly assumed to be open [Heymans *et al.*, 2005].

Whatever popular semantics is chosen for a rule based logic program (*well-founded, answer sets, first order*, etc.), both of the above settings are sources of undecidability that are difficult to cope with.

*Top-down solvers* (such as SLD solvers) do not usually address this issue: the programmer is requested to carry the burden of ensuring termination. In order to achieve this, she has to have a good knowledge of the evaluation strategy implemented in her specific adopted system, since termination is often algorithm-dependent.

On the other hand, *bottom-up solvers* (such as DLV or Smodels for the Answer Set Semantics [Leone *et al.*, 2006; Simons *et al.*, 2002]), and in general, languages derived from Datalog, are instead conceived for ensuring algorithm independent decidability and full declarativity. To this aim, the implementation of such languages relies on the explicit choice of computing a ground version of a given program. Unfortunately, in a context where value invention is explicitly allowed, grounding a program against an infinite set of symbols leads to an infinite ground program, which obviously cannot be built in practice.

Although we take *Answer Set* as the reference semantics, our framework relies on the traditional notion of ground program. Thus, results about VI-restricted programs can be be adapted to semantics other than Answer Set Programming, such as the Well-Founded Semantics [Ross, 1989], or else.

Next, we show how syntax and semantics described in Section 1.1 and 1.2 are affected by the addition of elements needed to support value invention.

## 5.1.1 External Predicates: Syntax and Semantics

*External* predicate names follow the same syntax of *ordinary* predicate names, but they are prefixed with the character '#'. Likewise, an atom is called *external* if its predicate name is external otherwise we call it ordinary. For instance, $node(X)$, and $\#succ(X, Y)$ are atoms; the first is ordinary, whereas the second is external.

Atoms appearing in the head of a rule must be ordinary, while atoms appearing in the body can be both ordinary and external.

We call a rule $r$ *ordinary*, if it contains only ordinary atoms.

A *VI program* is a finite set $\mathcal{P}$ of rules that can, in case, include also non ordinary rules; it is *ordinary* if all rules are ordinary.

**Example 5.1** The following is an example of a short VI program:

$$\begin{array}{rcl} mustChangePasswd(Usr) & :- & passwd(Usr, Pass), \\ & & \#strlen(Pass, Len), \# < (Len, 8). \end{array}$$

The *Herbrand base* $B_{\mathcal{P}}$ of a VI program $\mathcal{P}$, will also include all possible ground versions of external atoms occurring in $\mathcal{P}$. An *interpretation* $I$ for $\mathcal{P}$ is a pair $\langle S, F \rangle$ where:

- $S \subseteq B_{\mathcal{P}}$ is a *consistent* set of ordinary atoms; we say that $I$ (or by small abuse of notation, $S$) is a *model* of an ordinary atom $a \in B_{\mathcal{P}}$, denoted $I \models a$ ($S \models a$), if $a \in S$.

- $F$ is a mapping associating with every external predicate name $\#e$, a decidable $n$-ary function (which we call *oracle*) $F(\#e)$ assigning each tuple $(x_1, \ldots, x_n)$ either 0 or 1, where $n$ is the fixed arity of $\#e$, and $x_i \in U_{\mathcal{P}}$. $I$ (or, by small abuse of notation, $F$) is a *model* of a ground external atom $a = \#e(x_1, \ldots, x_n)$, denoted $I \models a$ ($F \models a$), if $F(\#e)(x_1, \ldots, x_n) = 1$.

**Example 5.2** We give an interpretation $I = \langle S, F \rangle$ such that the external predicate $\#strlen$ is associated to the oracle $F(\#strlen)$, and $F(\#<)$ to $\#<$. Intuitively these oracles are defined according to the expected semantics for a function computing the length of a string and for the 'less than' comparison operator, respectively. So that, for example, $\#strlen(pat4dat, 7)$ and $\#<(7, 8)$ are satisfied by $I$, whereas $\#strlen(pat4dat, 8)$ and $\#<(10, 8)$ are not.

The following is a ground version of the program in Example 5.1:

$$\begin{array}{rcl} mustChangePasswd(frank) & :- & passwd(frank, pat4dat), \\ & & \#strlen(pat4dat, 7), \#<(7, 8). \end{array}$$

The semantics of a program $\mathcal{P}$ is then defined as in Section 1.2 but, for a fixed $F$, a model $M = \langle S, F \rangle$ is minimal if there is no model $N = \langle T, F \rangle$ such that $S \subset T$. We call $\mathcal{P}$ *F-satisfiable* if it has some answer set for a fixed function mapping $F$, i.e. if there is some interpretation $\langle S, F \rangle$ which is an answer set. We will assume in the following to deal with a fixed set $F$ of functions mappings for external predicates.

## 5.2 Properties of VI Programs

If the domain of constants $U_{\mathcal{P}}$ for a program $\mathcal{P}$ is infinite, the Herbrand base $B_{\mathcal{P}}$ of $\mathcal{P}$ is then infinite too. This is the case of programs with function symbols and, in general it is the case of programs allowing value invention.

It is of interest to tailor cases where a finite portion of $U_{\mathcal{P}}$ is enough to evaluate the semantics of a given program. In the following we reformulate some results regarding splitting sets. The notion of splitting set has been introduced in [Lifschitz and Turner, 1994] in order to provide a technique for decomposing a given ground program $\mathcal{P}$, so that its answer sets can be computed from the answer sets of two separate programs. This technique is commonly adopted for enabling modular computation of the answer sets. Here, splitting sets are exploited as a tool for decomposing $\mathcal{P}$ into a finite part $\mathcal{P}'$ and an infinite part $\mathcal{P}''$. Then, we identify classes of programs for which $\mathcal{P}''$ is provable to be always consistent, since it has a single, empty, answer set. Thus, computing the answer sets of $\mathcal{P}$ can be reduced to computing the answer sets of $\mathcal{P}'$.

**Definition 5.3** Let $\mathcal{P}$ be a `VI` program. A *splitting set* is a set of atoms $A \in B_{\mathcal{P}}$ such that for each atom $a \in A$, if $a \in H(r)$ for some $r \in Ground(\mathcal{P})$, then all the atoms occurring in $B(r)$ and in $H(r)$ belongs also to $A$. The *bottom* $b_A(\mathcal{P})$

is the set of rules $\{r \mid r \in Ground(\mathcal{P})$ and $H(r) \subseteq A\}$. A literal whose atom belongs to $A$ is said $A$-literal. Given an interpretation $I$, the *residual* $r_A(\mathcal{P}, I)$ is a program obtained from $Ground(\mathcal{P})$ by deleting all the rules whose body contains an $A$-literal not satisfied by $I$, and removing from the remaining rules all the $A$-literals.

**Example 5.4** Consider the following program $\mathcal{P}$:

$$\begin{array}{l} r \text{ :- } p. \\ r \text{ :- } q. \\ p \text{ :- } \#e(a,b), \text{not } q. \\ q \text{ :- } \text{not } p. \end{array}$$

$A$ = $\{p,\ q,\ \#e(a,b)\}$ is a *splitting set* for $\mathcal{P}$. The *bottom* $b_A(\mathcal{P})$ is the set containing the last two rules of $\mathcal{P}$. Consider the interpretation $I = \langle\{p, r\}, F\rangle$, where the oracle $F(\#e)$, associated to the external predicate $\#e$, is defined such that $\#e(a, b)$ is satisfied by $I$. The *residual* $r_A(\mathcal{P} \setminus b_A(\mathcal{P}), I)$ is the program consisting of the single rule:

$$r.$$

We reformulate here the splitting theorem as given in [Bonatti, 2004].

**Theorem 5.5** *(Splitting theorem [Lifschitz and Turner, 1994])* Let $\mathcal{P}$ be a program and $A$ be a splitting set. Then, $M \in AS(\mathcal{P})$ if and only if $M$ can be split in two disjoint sets $I$ and $J$, such that $I \in AS(b_A(\mathcal{P}))$ and $J \in AS(r_A(\mathcal{P}) \setminus b_A(\mathcal{P})), I)$.

We consider now an interesting subclass of `VI` programs, namely *vi-safe* programs; we will exploit the above theorem in order to prove that each vi-safe program can be evaluated simply taking into account only the constants originally appearing in the program itself.

**Definition 5.6** Let $r$ be a rule. A variable $X$ is *vi-safe* in $r$ if it appears in some *ordinary* atom $a \in B^+(r)$. A rule $r$ is *vi-safe* if each variable $X$ appearing in $r$ is vi-safe. A program $\mathcal{P}$ is *vi-safe* if each rule $r \in \mathcal{P}$ is vi-safe.

Note that the notion of *vi-safety* makes distinction between ordinary and external atoms, while *safety* as defined in Section 1.1, does not. That is, for making a variable safe, it is necessary its appearance in a positive literal, while vi-safety requires a variable to appear explicitly in an ordinary (non-external) atom. Both conditions are syntactic. As an important semantic consequence, vi-safety prevents completely the appearance of new symbols in a given program, as next theorem shows.

**Theorem 5.7** Let $\mathcal{P}$ be a vi-safe `VI` program. Let $U$ be the set of constants appearing in $\mathcal{P}$ and let $AS_U(\mathcal{P})$ be the set of answer sets obtained restricting the universe of constants $U_{\mathcal{P}}$ to $U$. Then $AS_U(\mathcal{P}) = AS(\mathcal{P})$.

**Proof.** Let's denote with $Ground_U(\mathcal{P})$ the ground program obtained by replacing variables with elements of $U$ and with $A$ the set of ground atoms appearing in $Ground_U(\mathcal{P})$. Assuming $\mathcal{P}$ as vi-safe, it is easy to see that $A$ is a finite splitting set for $\mathcal{P}$. Furthermore, $Ground_U(\mathcal{P}) = b_A(\mathcal{P})$. For each $M \in AS_U(\mathcal{P})$, we have that $r_A(Ground(\mathcal{P})) \setminus b_A(\mathcal{P}), M)$ is consistent and its only answer set is the empty set (indeed, no rule can be ever satisfied unless the variables are bound to constants appearing in $U$). Thus $M \cup \emptyset \in AS(\mathcal{P})$ (Theorem 5.5). Viceversa, assuming an answer set $M \in AS(\mathcal{P})$ is given, same arguments easily lead to conclude that $M \in AS_U(\mathcal{P})$. $\Box$

In case a vi-safe program is given, the above theorem allows to consider as the set of 'relevant' constants only those values explicitly appearing in the program at hand. The semantics of a vi-safe program $\mathcal{P}$ can be evaluated by means of the following algorithm:

1. compute $Ground_U(\mathcal{P})$, where $U$ is defined as in Theorem 5.7;
2. remove from $Ground_U(\mathcal{P})$ all the rules containing at least one external literal $e$ such that $F \not\models e$, and remove from each rule all the remaining external literals, obtaining a reduced ground program that we will call $\overline{Ground_U}(\mathcal{P})$.
3. evaluate the answer sets of $\overline{Ground_U}(\mathcal{P})$ by means of a standard Answer Set solver.

It is worth pointing out that, assuming the complexity of computing oracles is polynomial in the size of their arguments, this algorithm has the same complexity as computing $Ground_U(\mathcal{P})$[1].

[1]Assuming rules can have unbounded length, grounding a disjunctive logic program is in the worst case exponential in the size of the Herbrand base (see e.g. [Leone *et al.*, 2001]).

**Example 5.8** Consider the program in Example 5.1 and the following two facts:

$$passwd(jack, short).$$
$$passwd(bill, longpasswd).$$

If we consider a set $U$ including constants appearing in the program plus a finite subset of natural numbers, the resulting ground program after step 1 contains, among the others, the rules:

$$\begin{array}{lll} mustChangePasswd(jack) & \text{:-} & passwd(jack, short), \\ & & \#strlen(short, 5), \#<(5, 8). \\ mustChangePasswd(bill) & \text{:-} & passwd(bill, longpasswd), \\ & & \#strlen(longpasswd, 10), \#<(10, 8). \end{array}$$

Step 2 is such that only one of the two above rules is kept (after modification):

$$mustChangePasswd(jack) \text{ :- } passwd(jack, short).$$

that no longer contains external atoms.

## 5.3 Dealing with Value Invention

Although 2-valued oracles are important for clarifying the given semantics, we aim at introducing the possibility to specify *functional oracles*, keeping anyway the simple reference semantics given previously.

**Example 5.9** For instance, assume that $U_{\mathcal{P}}$ contains encoded values that can be interpreted as natural numbers and that the external predicate $\#sqr$ is defined such that the atom $\#sqr(X, Y)$ is satisfied whenever $Y$ encodes a natural number representing the square of the natural number $X$; we want to extract a series of squared values from this predicate; consider the short program

$$number(2).$$
$$square(Y) \text{ :- } number(X), \#sqr(X, Y).$$

In the presence of unsafe rules as in the above example, Theorem 5.7 ceases to hold: it is indeed unclear whether there is a finite set of constants which the program can be grounded on. In the above example, we can intuitively conclude that the set of meaningful constants is $\{2, 4\}$. Nonetheless, it is in general undecidable, given a computable oracle $f$, to establish whether a given set $S$ contains all and only those tuples $\bar{t}$ such that $f(\bar{t}) = 1$.

In the new setting we are going to introduce, it is also very important that an external atom brings knowledge from external sources of computation, in terms of new symbols added to a given program. We extend our framework with the possibility of explicitly computing missing values on demand. Although restrictive, this setting is not far from a realistic scenario where external predicates are defined by means of generic partial functions.

**Definition 5.10** Let $\#p$ be an external predicate name of arity $n$, and let $F(\#p)$ be its oracle function. A *pattern* is a list of $i$'s and $o$'s, where a $i$ represents a placeholder for a constant (or a bounded variable), and an $o$ is a placeholder for a variable. Given a list of terms, the corresponding pattern is given by replacing each constant with a $i$, and each variable with a $o$. Positions where $o$ appears are called *output* positions whereas those denoted with $i$ are called *input* positions. For instance, the pattern related to the list of terms $(X, a, Y)$ is $(o, i, o)$.

Let $pat$ be a pattern of length $n$ having $k$ placeholders $i$ (*input positions*), and $n-k$ placeholders of $o$ type (*output positions*). A *functional oracle* $F(\#p)[pat]$ for the pattern $pat$, associated with the external predicate $\#p$, is a partial function taking $k$ constant arguments and returning a finite relation of arity $n-k$, and such that $d_1, \ldots, d_{n-k} \in F(\#p)[pat](c_1, \ldots, c_k)$ if and only if $F(\#p)(h_1, \ldots, h_n) = 1$, where for each $l(1 \leq l \leq n)$, $h_l = c_j$ if the $j$-th $i$ value occurs in position $l$ in $pat$, otherwise $h_l = d_j$ if the $j$-th $o$ value occurs in position $l$ in $pat$. Let $pat[j]$ be the $j$-th element of a pattern $pat$. Let $output_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = o$ for each $X_j \in \overline{X}$, and $input_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = i$ for each $X_j \in \overline{X}$.

An external predicate $\#p$ might be associated to one or more functional oracles 'consistent' with the originating 2-valued one. For instance, consider the $\#sqr$ external predicate, defined as mentioned above. We can have two functional oracles, $F(\#sqr)[i, o]$ and $F(\#sqr)[o, i]$. The two functional oracles are such that, e.g. $F(\#sqr)[i, o](3) = 9$ and $F(\#sqr)[o, i](16) = 4$, consistently with the fact that $F(\#sqr)(3, 9) = F(\#sqr)(4, 16) = 1$, whereas $F(\#sqr)[o, i](5)$ is set as undefined since $F(\#sqr)(X, 5) = 0$ for any natural number $X$.

For the sake of simplicity, in the sequel, given an external predicate $\#e$, we will assume that it comes equipped with its oracle $F(\#e)$ (called also *base oracle*) and exactly one functional oracle $F(\#e)[pat_{\#e}]$, having pattern $pat_{\#e}$. It is worth noting that this does not cause any loss of generality: indeed, having an external predicate with two (or more) different functional oracles is equivalent to having two (or more) different external predicates with one functional oracle each, and using the proper one every time a particular oracle is desired.

Once functional oracles are given, it is important to investigate which are the cases where they can be used for computing the actual set of ground instances of a given rule.

To this end, we introduce the notions of weakly safe variable and weakly safe rule. Intuitively, a variable is weakly safe if its domain, although not explicitly bound to the domain of an ordinary atom, can be computed indirectly through a functional oracle.

For instance, the second rule of Program 5.9 is not vi-safe (since $Y$ is not vi-safe), but is such that, intuitively, the domain of $Y$ can be computed once the domain of $X$ is known, provided a proper oracle $F(\#sqr)[i, o]$ is given for $\#sqr$. The following definition captures this intuition.

**Definition 5.11** Let $r$ be a rule. A variable $X$ is *weakly safe in* $r$ if either

- $X$ is vi-safe (i.e. it appears in some ordinary atom of $B^+(r)$); or
- $X$ appears in some external atom $\#e(\overline{T})$, the functional oracle of $\#e$ is $F(\#e)[pat]$, $X$ appears in output position with respect to $pat$ and each variable $Y$ appearing in input position in the same atom is weakly safe.

A weakly safe variable $X$ is *free* if it appears in $B^+(r)$ only in output position of some external atom. A rule $r$ is weakly safe if each variable $X$ appearing in some atom $a \in B(r)$ is weakly safe. A program $\mathcal{P}$ is weakly safe if each rule $r \in \mathcal{P}$ is weakly safe.

**Example 5.12** Assume that $\#sqr$ is associated to the functional oracle $F(\#sqr)$ $[i, o]$ defined above. The second rule of Program 5.9 is weakly safe ($X$ is vi-safe, while $Y$ appears in output position in the atom $\#sqr(X, Y)$). The same rule is not weakly safe if we consider the functional oracle $F(\#sqr)[o, i]$.

**Proposition 5.13** Given a `VI` program $\mathcal{P}$, it can be checked in polynomial time whether $\mathcal{P}$ is weakly safe.

**Proof.** Simply observe that for each rule $r \in \mathcal{P}$ it can be checked in time linear in the number of atoms of $r$ whether the patterns of the functional oracle associated to each external atom occurring in $\mathcal{P}$ make the rule vi-safe or not. $\square$

Weakly safe rules can be grounded with respect to functional oracles as follows.

**Definition 5.14** Let $I = \langle S, F \rangle$ be an interpretation. We call $ins(r, I)$ the set of ground instances $r_\theta$ of $r$ for which $I \models B^+(r_\theta)$, and such that $I$ is a model for the set of external atoms in $r_\theta$.

**Proposition 5.15** Let $I$ be a finite interpretation, and $r$ be a weakly safe rule. $ins(r, I)$ is finite.

**Proof.** Indeed, given the set of functional oracles associated to each external atom, any ground rule $r'$ which is member of $ins(r, I)$ can be generated by the following algorithm:

1. replace positive literals of $r$ with a consistent nondeterministic choice of matching ground atoms from $I$; let $\theta$ the resulting variable substitution;
2. until $\theta$ instantiates all the variables of $r$:
    - pick from $r\theta$ an external atom $\#e(\overline{X})\theta$ such that $\theta$ instantiates all the variables $X \in input_{pat}(\overline{X})$.
    - choose nondeterministically a tuple $\langle a_1, \ldots, a_k \rangle \in F(input_{pat}(\overline{X}\theta))$, then update $\theta$ by assigning $a_1, \ldots, a_k$ to $output_{pat}(\overline{X}\theta)$;
3. return $r' = r\theta$.

□

Weakly safe rules have the important property of producing a finite set of *relevant* ground instances provided that we know a priori the domain of positive ordinary body atoms. Although desirable, weak safety is intuitively not sufficient in order to guarantee finiteness of answer sets and decidability. For instance, it is easy to see that the program:

$$square(2).$$
$$square(Y) \text{ :- } square(X), \#sqr(X, Y).$$

has the infinite set of atoms $\{square(2), square(4), \ldots\}$ as answer set.

## 5.4 A Computable Class with External Functions: VI-restricted Programs

The introduction of new symbols in a logic program by means of external atoms is a clear source of undecidability, nonetheless it is desirable in a variety of contexts.

Our approach investigates which programs, allowing value invention, can be solved by means of a finite ground program having a finite set of models of finite size.

**Definition 5.16** A class of `VI` programs $\mathcal{V}$ has the *finite grounding* property if, for each $\mathcal{P} \in \mathcal{V}$ there exists a finite set $U \subset U_{\mathcal{P}}$ such that $AS_U(\mathcal{P}) = AS(\mathcal{P})$, where $AS_U(\mathcal{P})$ is the set of answer sets obtained restricting the universe of constants to $U$.

This class of programs (having the *finite grounding property*) is unluckily not recognizable in finite time.

**Theorem 5.17** Recognizing the class of all the `VI` programs having the finite grounding property is undecidable.

**Proof.** Given a Turing machine $\mathcal{T}$ and an input string $x$ we can build a suitable `VI` program $\mathcal{P}_{\mathcal{T},x}$ encoding $\mathcal{T}$ and $x$. $\mathcal{T}(x)$ terminates if and only if $\mathcal{P}_{\mathcal{T},x}$ has the finite grounding property. Indeed, if $\mathcal{T}(x)$ terminates, the content of a finite set of symbols $U$, such that Definition 5.16 is applicable, can be inferred from the finite number of transitions of $\mathcal{T}(x)$. Viceversa, if $U$ is given, the evolution of $\mathcal{T}(x)$ until its termination can be mimicked by looking at the answer sets of $Ground_U(P_{\mathcal{T},x})$. Hence the result follows. □

Note that the above theorem holds under the assumption that functional oracles might have an infinite co-domain, although functional oracles are supposed to associate, to each fixed combination of input values, a finite number of combination of values in output. Also, it is assumed to deal with weakly safe programs.

The intuition leading to our definition of `VI`-restrictedness, is based on the idea of controlled propagation of new values throughout a given program. Assume the following `VI` program is given ($\#b$ has a functional oracle with pattern $[i, o]$):

$a(k, c).$
$p(X, Y)$ :- $a(X, Y).$
$p(X, Y)$ :- $s(X, Y), a(Z, Y).$
$s(X, Y)$ :- $p(Z, X), \#b(X, Y).$

The last rule of the program generates new symbols by means of the $Y$ variable, which appears in the second attribute of $s(X, Y)$ and in output position of $\#b(X, Y)$. This situation is per se not a problem, but we observe that values of $s[2]$ are propagated to $p[2]$ by means of the last but one rule, and $p[2]$ feeds input values to $\#b(X, Y)$ in the last rule. This occurs by means of the binding given by the $X$ variable. The number of ground instances to be considered for the above program is thus in principle infinite, due to the presence of this kind of cycles between attributes.

We introduce the notion of *dangerous* rule for those rules that propagate new values in recursive cycles, and of *dangerous* attributes for those attributes (e.g. $s[2]$) that carry new information in a cycle.

Actually, the above program can be reconducted to an equivalent finite ground program: we can observe that $p[2]$ takes values from the second and third rule above. In both cases, values are given by bindings to $a[2]$ which has, clearly, a finite domain. So, the number of input values to $\#b(X, Y)$ is bounded as well. In some sense, the 'poisoning' effect of the last (dangerous) rule, is canceled by the fact that $p[2]$ limits the number of symbols that can be created.

In order to formalize this type of scenarios we introduce the notion of *savior* and *blocked* attributes. $p[2]$ is *savior* since all the rules where $p$ appears in the head can be proven to bring values to $p[2]$ from blocked attributes, or from constant values, or from other savior attributes. Also, $s[2]$ is dangerous but *blocked* with respect to the last rule, because of the indirect binding with $p[2]$, which is savior. Note that an attribute is considered blocked with respect to a given rule. Indeed, $s[2]$ might not be blocked in other rules where $s$ appears in the head.

We define now an *attribute dependency graph* useful to track how new symbols propagate from an attribute to another by means of bindings of equal variables. Note that, this dependency graph is different from the one introduced in Section 3.2: both graphs have attributes as vertices but, in the graph defined next, external predicates are also considered.

**Definition 5.18** The *attribute dependency graph* $AG(\mathcal{P})$ associated to a `VI` program $\mathcal{P}$ is defined as follows. For each predicate $p \in \mathcal{P}$ of arity $n$, there is a node for each predicate attribute $p[i](1 \leq i \leq n)$, and, looking at each rule $r \in \mathcal{P}$, there are the following edges:

- $(q[j], p[i])$, if $p$ appears in some atom $a_p \in H(r)$, $q$ in some ordinary atom $a_q \in B^+(r)$ and $q[j]$ and $p[i]$ share the same variable in $a_q$ and $a_p$ respectively.

- $(q[j], \#p[i])$, if $q$ appears in some ordinary atom $a_q \in B^+(r)$, $\#p$ in some external atom $a_{\#p}$, $q[j]$ and $\#p[i]$ share the same variable in $a_q$ and $a_{\#p}$ respectively, and $i$ is an input position for the functional oracle of $\#p$;
- $(\#q[j], \#p[i])$, if $\#q$ appears in some external atom $a_{\#q}$, $\#p$ in some external atom $a_{\#p}$, $\#q[j]$ and $\#p[i]$ share the same variable in $a_{\#q}$ and $a_{\#p}$ respectively, $j$ is an output position for the functional oracle of $\#q$, $i$ is an input position for the functional oracle of $\#p$;
- $(\#p[j], \#p[i])$, if $\#p$ appears in some external atom $a_{\#p}$, $\#p[j]$ and $\#p[i]$ both have a variable in $a_{\#q}$ and $a_{\#p}$ respectively, $j$ is an input position for the functional oracle of $\#p$, and $i$ is an output position for the functional oracle of $\#p$;
- $(\#q[j], p[i])$, if $p$ appears in some atom $a_p \in H(r)$, $\#q$ in some external atom $a_{\#q}$, $\#q[j]$ and $p[i]$ share the same variable in $a_{\#q}$ and $a_p$ respectively, and $j$ is an output position for the functional oracle of $\#q$;

**Example 5.19** The `VI` attribute dependency graph induced by the first three rules of the following `VI` program is depicted in Figure 5.1:

$$
\begin{array}{l}
\mathit{trusted}(X,U) \text{ :- } \#\mathit{rdf}(``\mathit{myurl}'', X, ``\mathit{trusts}'', U).\\
\mathit{url}(X,U) \text{ :- } \#\mathit{rdf}(``\mathit{myurl}'', X, ``\mathit{seealso}'', U), \mathit{trusted}(X,U).\\
\mathit{url}(X,U) \text{ :- } \mathit{url}(\_, U1), \#\mathit{rdf}(U1, X, ``\mathit{seealso}'', U), \mathit{trusted}(X,U).\\
\mathit{connected}(X,Y) \text{ :- } \mathit{url}(X,U), \#\mathit{rdf}(U, X, ``\mathit{knows}'', Y).\\
\mathit{connected}(X,Y) \text{ :- } \mathit{connected}(X,Z), \mathit{url}(Z,U), \#\mathit{rdf}(U, Z, ``\mathit{knows}'', Y).
\end{array}
$$

**Definition 5.20** Let $\mathcal{P}$ be a weakly safe program. Then[2]:

- A rule $r$ *poisons* an attribute $p[i]$ if some atom $a_p \in H(r)$ has a free variable $X$ in position $i$. $p[i]$ is said to be *poisoned* by $r$. For instance, $\mathit{connected}[2]$ is poisoned by the last rule.
- A rule $r$ is *dangerous* if it poisons an attribute $p[i]$ $(p \in H(r))$ appearing in a cycle in $AG(\mathcal{P})$. Also, we say that $p[i]$ is *dangerous*. For instance, the last rule is dangerous since $\mathit{connected}[2]$ is poisoned and appears in a cycle.

[2]All examples in this definition refer to program of Example 5.19. Also, we assume that $\#rdf$ has functional oracle with pattern $[i, o, o, o]$

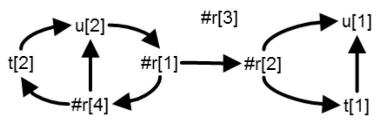

Figure 5.1: `VI` Attributes Dependency Graph *(Predicate names shortened to the first letter).*

- Let $r$ be a dangerous rule. A dangerous attribute $p[i]$ (bounded in $H(r)$ to a variable name $X$), is *blocked* in $r$ if for each external atom $a_{\#e} \in B(r)$ where $X$ appears in output position, each variable $Y$ appearing in input position in the same atom is *savior*. $Y$ is *savior* if it appears in some predicate $q \in B^+(r)$ in position $i$, and $q[i]$ is *savior*.
- An attribute $p[i]$ is *savior* if at least one of the following conditions holds for each rule $r \in \mathcal{P}$ where $p \in H(r)$.
  - $p[i]$ is bound to a ground value in $H(r)$;
  - there is some savior attribute $q[j]$, $q \in B^+(r)$ and $p[i]$ and $q[j]$ are bound to the same variable in $r$;
  - $p[i]$ is blocked in $r$.

  For instance, the dangerous attribute $connected[2]$ of the last rule is blocked since the input variables $U$ and $Z$ are savior (indeed they appear in $url[2]$ and $url[1]$).
- A rule is `VI`-restricted if all its dangerous attributes are blocked. $\mathcal{P}$ is said to be *`VI`-restricted* if all its *dangerous* rules are *`VI`-restricted.*

**Theorem 5.21** `VI`-restricted programs have the finite grounding property.

**Proof.** Let $\mathcal{P}$ be a `VI`-restricted program. We show how to compute a finite ground program $gr_{\mathcal{P}}$ such that $AS(\mathcal{P}) = AS_U(gr_{\mathcal{P}})$, where $U$ is the set of constants appearing in $gr_{\mathcal{P}}$.

Let's call $GA$ the set of *active ground atoms*, initially containing all atoms appearing in some fact of $\mathcal{P}$. The program $gr_{\mathcal{P}}$ can be constructed by an algorithm $\mathcal{A}$ that repeatedly updates $gr_{\mathcal{P}}$ (initially empty) with the output of $ins(r, I)$ (Definition 5.14) for each rule $r \in \mathcal{P}$, where $I = \langle GA, F\rangle$; all atoms belonging to the head of some rule appearing in $gr_{\mathcal{P}}$ are then added to $GA$. The iterative process stops when $GA$ is not updated anymore. That is, $gr_{\mathcal{P}}$ is the least fixed point of the operator

$$T_{\mathcal{P}}(X) = \{\bigcup_{r \in \mathcal{P}} ins(r, I) \mid I = \langle GA, F \rangle, \text{ and } GA = atoms(X)\}$$

where $X$ is a set of ground rules and $atoms(X)$ is the set of ordinary atoms appearing in $X$. $T^{\infty}_{\mathcal{P}}(\emptyset)$ is finite in case $\mathcal{P}$ is VI-restricted. Indeed, $gr_{\mathcal{P}}$ might not cease to grow only in case an infinite number of new constants is generated by the presence of external atoms. This may happen only because of some *dangerous* rule having some *poisoned* attributes. However, in a *VI-restricted* program all poisoned attributes are *blocked* in dangerous rules where they appear, i.e. they depend from savior attributes. Now, for a given savior attribute $p[i]$, the number of symbols that appear in position $i$ in an atom $a_p$ such that $a_p \in T^{\infty}_{\mathcal{P}}(\emptyset)$ is finite. This means that only a finite number of calls to functional oracles is made by $\mathcal{A}$, each one producing a finite output.

Because of the way it has been constructed, it is easy to see that $GA = atoms(gr_{\mathcal{P}})$ is a splitting set [Lifschitz and Turner, 1994], for $Ground(\mathcal{P})$. Based on this, it is possible to observe that no atom $a \notin GA$ can be in any answer set, and to conclude that $AS_U(\mathcal{P}) = AS(\mathcal{P})$, where $U$ is the set of constants appearing in $GA$. $\square$

## 5.5 Recognizing VI-restricted Programs

An algorithm recognizing VI-restricted programs is reported in Appendix B.1. The idea is to iterate through all *dangerous rules* trying to prove that all of them are *VI-restricted.* In order to prove VI-restriction for rules, we incrementally build the set of all *savior attributes*; this set is initially filled with all attributes which can be proven to be savior (i.e. they do not depend from any dangerous attribute). This set is updated with a further attribute $p[i]$ as soon as it is proved that each dangerous attribute which $p[i]$ depends on is blocked. The set $RTBC$ of rules to be checked initially consists of all dangerous rules, then the rules which are proven to be VI-restricted are gradually removed from $RTBC$. If an iteration ends and nothing new can be proven the algorithm stops. The program is VI-restricted if $RTBC$ is empty at the last iteration.

The algorithm consumes polynomial time in the size of a program $\mathcal{P}$: let $m$ be the total number of rules in $\mathcal{P}$, $n$ the number of different predicates, $k$ the maximum number of attributes over all predicates, and $l$ the maximum number of atoms in a single rule. $O(n * k)$ is an upper bound to the total number of different attributes, while $O(l * k)$ is an upper bound to the number of variables in a rule.

A naive version of the $isBlocked$ function (see Appendix B.2) has complexity $O(n * l * k^2)$. The $recognizer$ function iterates $O(n * k)$ times over an inner cycle which performs at most $O(m * k * l)$ steps (when all attributes are initially in $NSA$ and only one attributes can be stated as savior at each step); each inner step iterates over all rules in $RTBC$, which are at most $m$; and for each rule all free variables must be checked (this requires $O(k * l)$ checks, in the worst case).

# Chapter 6

# An ASP System with Functions Lists and Sets

In this chapter we illustrate the implementation of an ASP system supporting the class of DLP programs presented in Chapter 3. Such system actually features an even richer language, that, besides functions, explicitly supports also complex terms such as lists and sets, and provides a large library of built-in predicates for facilitating their manipulation. Thanks to such extensions, the resulting language becomes even more suitable for easy and compact knowledge representation tasks.

The chapter is organized as follows:

- Section 6.1 introduces the peculiar features of the fully extended language, with the help of some sample programs.
- In Section 6.2 we explain how function symbols and, in general, complex terms have been implemented exploiting the value invention framework on top of the DLV system.
- Finally, in Section 6.3 results about some preliminary tests are reported and applications already exploiting the implemented system are mentioned.

## 6.1 System Language

We next informally point out the peculiar features of the fully extended language, with the help of some sample programs.

In addition to simple and functional terms, there might be also *list* and *set* terms; a term which is not simple is said to be *complex*.

A *list term* can be of two different forms:

- $[t_1, \ldots, t_n]$, where $t_1, \ldots, t_n$ are terms;
- $[h|t]$, where $h$ (the head of the list) is a term, and $t$ (the tail of the list) is a list term.

Examples for list terms are: $[jan, feb, mar, apr, may, jun]$, $[jan \mid [feb, mar, apr, may, jun]]$, $[[jan, 31] \mid [[feb, 28], [mar, 31], [apr, 30], [may, 31], [jun, 30]]]$.

Set terms are used to model collections of data having the usual properties associated with the mathematical notion of set. They satisfy idempotence (i.e., sets have no duplicate elements) and commutativity (i.e., two collections having the same elements but with a different order represent the same set) properties.

A *set term* is of the form: $\{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ are ground terms.

Examples for set terms are: $\{red, green, blue\}$, $\{[red, 5], [blue, 3], [green, 4]\}$, $\{\{red, green\}, \{red, blue\}, \{green, blue\}\}$. Note that duplicated elements are ignored, thus the sets: $\{red, green, blue\}$ and $\{green, red, blue, green\}$ are actually considered as the same.

As already mentioned, in order to easily handle list and set terms, a rich set of built-in functions and predicates is provided. Functional terms prefixed by a $\#$ symbol are *built-in* functions. Such kind of functional terms are supposed to be substituted by the values resulting from the application of a functor to its arguments, according to some predefined semantics. For this reason, built-in functions are also referred to as *interpreted* functions. Atoms prefixed by $\#$ are, instead, instances of *built-in* predicates. Such kind of atoms are evaluated as true or false by means of operations performed on their arguments, according to some predefined semantics[1]. Some simple built-in predicates are also available, such as the comparative predicates equality, less-than, and greater-than $(=, <, >)$ and arithmetic predicates like successor, addition or multiplication, whose meaning is straightforward. A pair of simple examples about complex terms and proper manipulation

[1]The specification of the entire library for lists and sets manipulation is available at [Calimeri *et al.*, since 2008].

functions follows. Another interesting example, i.e., the Hanoi Tower problem, is reported in Appendix A.2.

**Example 6.1** Given a directed graph, a *simple path* is a sequence of nodes, each one appearing exactly once, such that from each one (but the last) there is an edge to the next in the sequence. The following program derives all simple paths for a directed graph, starting from a given $edge$ relation:

$$path([X, Y]) \text{ :- } edge(X, Y).$$
$$path([X|[Y|W]]) \text{ :- } edge(X, Y),\ path([Y|W]),\ \text{not } \#member(X, [Y|W]).$$

The first rule builds a simple path as a list of two nodes directly connected by an edge. The second rule constructs a new path adding an element to the list representing an existing path. The new element will be added only if there is an edge connecting it to the head of an already existing path. The external predicate $\#member$ (which is part of the above mentioned library for lists and sets manipulation) allows to avoid the insertion of an element that is already included in the list; without this check, the construction would never terminate in the presence of circular paths. Even if not an $\mathcal{FD}$ program, it is easy to see that this is an $\mathcal{FG}$ program; thus, the system is able to effectively compute the (in this case, unique) answer set.

**Example 6.2** Let us imagine that the administrator of a social network wants to increase the connections between users. In order to do that, (s)he decides to propose a connection to pairs of users that result, from their personal profile, to share more than two interests. If the data about users are given by means of EDB atoms of the form $user(id,\ \{interest_1, \ldots, interest_n\})$, the following rule would compute the set of common interests between all pairs of users:

$$sharedInterests(U_1, U_2, \#intersection(S_1, S_2)) \text{ :- } user(U_1, S_1), user(U_2, S_2), U_1 \neq U_2.$$

where the interpreted function $\#intersection$ takes as input two sets and returns their intersection. Then, the predicate selecting all pairs of users sharing more than two interests could be defined as follows:

$$proposeConnection(pair(U_1, U_2)) \text{ :- } sharedInterests(U_1, U_2, S), \#card(S) > 2.$$

Here, the interpreted function $\#card$ returns the cardinality of a given set, which is compared to the constant 2 by means of the built-in predicate "$>$".

## 6.2 Implementation

The presented language has been implemented on top of the state-of-the-art ASP system DLV [Leone *et al.*, 2006]. Complex terms have been implemented by using a couple of built-in predicates for packing and unpacking them (see below). These functions, along with the library for lists and sets manipulation reported in Appendix C, have been incorporated in DLV by exploiting the framework introduced in Chapter 5.

In particular, support for complex terms is actually achieved by suitably rewriting the rules they appear in. The resulting rewritten program does not contain complex terms any more, but a number of instances of proper built-in predicates. We briefly illustrate in the following how the rewriting is performed in case of functional terms; the cases of list and set terms are treated analogously. Firstly, any functional term $t = f(X_1, \ldots, X_n)$, appearing in some rule $r$ of a program $\mathcal{P}$, is replaced by a fresh variable $Ft$ and then, one of the following atom is added to $B(r)$:

- $\#function\_pack(Ft, f, X_1, \ldots, X_n)$ if $t$ appears in $H(r)$;
- $\#function\_unpack(Ft, f, X_1, \ldots, X_n)$ if $t$ appears in $B(r)$.

This transformation is applied to the rule $r$ until no functional terms appear in it. The role of an atom $\#function\_pack$ is to build a functional term starting from a functor and its arguments; while an atom $\#function\_unpack$ acts unfolding a functional term to give values to its arguments. So, the former binds the $Ft$ variable, provided that all other terms are already bound, the latter binds (checks values, in case they are already bound) the $X_1, \ldots, X_n$ variables according to the binding for the $Ft$ variable (the whole functional term). More precisely, the arguments of the $\#function\_pack/unpack$ built-ins are:

1. the fresh variable $Ft$ representing the whole functional term;
2. the function symbol $f$;
3. all of the arguments: $X_1, \ldots, X_n$ for the original functional term.

**Example 6.3** The rule: $p(f(f(X))) \,\text{:-}\, q(X, g(X, Y))$. will be rewritten as follow:

$$\begin{aligned} p(Ft_1) \;\text{:-}\;\; & \#function\_pack(Ft_1, f, Ft_2), \#function\_pack(Ft_2, f, X), \\ & q(X, Ft_3), \#function\_unpack(Ft_3, g, X, Y). \end{aligned}$$

Note that rewriting the nested functional term $f(f(X))$ requires two $\#function\_pack$ atoms in the body: (i) for the inner $f$ function having $X$ as argument and (ii) for the outer $f$ function having as argument the fresh variable $Ft_2$, representing the inner functional term.

The resulting ASP system is indeed very powerful: the user can exploit the full expressiveness of $\mathcal{FG}$ programs (plus the ease given by the availability of complex terms), at the price of giving the guarantee of termination up. In this respect, it is worth stating that the system grounder fully complies with the definition of *intelligent* instantiation introduced in this work (see Section 3.3 and Definition 3.15). This implies, among other things, that the system is guaranteed to terminate and correctly compute all answer sets for any program resulting as finitely-ground. Nevertheless, the system features a syntactic $\mathcal{FD}$ programs recognizer, based on the algorithm sketched in Theorem 3.28. This kind of finite-domain check, which is active by default, ensures a priori computability for all accepted programs, without the need for knowing the membership to $\mathcal{FG}$ programs class.

The system prototype, called DLV-complex, is available at [Calimeri *et al.*, since 2008]; the above mentioned library for list and set terms manipulation is available for free download as well, together with a reference guide and a number of examples.

## 6.3 Experiments and Applications

Some preliminary tests have been carried out in order to measure how much the new features cost in terms of performances. These experiments have been done on a Double-Intel-Xeon-HT (single core) 3.60GHZ-3GBRAM.

In particular, we evaluated costs of the following three steps of a computation:

(1) *Rewriting.* We have considered a program with this set of rules:

$p1(f(X))$ :- $a(X)$.

$\vdots$

$p10000(f(X))$:- $a(X)$.

that after rewriting becomes:

$p1(F)$ :- $a(X), \#function\_pack(F, f, X))$.

$\vdots$

$p10000(F)$ :- $a(X), \#function\_pack(F, f, X))$..

Rewriting ten thousand rules with functions, took only $0.252750$. Each rewriting took only $0.000025275$ seconds on average. Considering that the rewriting is usually done mostly on the non-ground rules, rewriting-time seems to be negligible.

(2) *Instantiation.* Each call to $\#function\_pack(F, f, X)$ takes less than $10^{-5}$ seconds, and corresponds approximately to $1.5$ of the time taken by a call to a comparison built-in (like $X < "c"$), much less than matching a standard predicate. The time taken by a call to $\#function\_unpack$ is basically the same.

(3) *Answer-Sets Computation.* We compared a program including, besides some facts for $a$, the disjunctive rule:
$p(f(X)) \vee q(f(X))$ :- $a(X)$.
with the analogous program without function symbols:
$p(X) \vee q(X)$ :- $a(X)$.

The residual instantiation has precisely the same size, and the times for the computation of the answer sets is identical.

In sum, from the preliminary experiments it seems that: (i) rewriting times are negligible; (ii) the cost of evaluating function terms (pack/unpack functions) is low (about $1.5$ times a comparison built-in as '$<$'); (iii) there is no overhead at all on answer-sets computation. Therefore, the system can profitably deal with real-world problems. For instance, in the area of self-healing Web Services DLV-complex is already exploited for the computation of minimum cardinality diagnoses [Friedrich and Ivanchenko, 2008], and functional terms are here employed to replace existential quantification. In summary, the introduction of functions brings only a little overhead, while it offers a significant gain in terms of knowledge-modeling power and program clarity. In some cases, the better problem encoding obtained through functions can bring also a significant computational gain.[2] (We expect that this happens also when, thanks to functions, we can better enforce arguments "types" cutting down the search space significantly.)

[2]For instance, the encoding for Tower of Hanoi reported in [Calimeri *et al.*, since 2008] against the classical guess-and-check encoding (a disjunctive version of the Smodels program exploited for the First ASP competition [Gebser *et al.*, 2007b]) allows one to enjoy nice speedups and to scale much better while increasing the number of disks.

# Chapter 7

# Related Works

Functional terms are widely used in logic formalisms stemming from first order logic. Introduction and treatment of functional terms (or similar constructs) have been studied indeed in several fields, such as Logic Programming and Deductive Databases. In the ASP community, the treatment of functional terms has recently received quite some attention. In this chapter we focus on the main proposals for introducing functional terms in ASP, and we briefly discuss the related work done in other research communities.

The chapter is organized as follows:

- Section 7.1 is about finitary programs, the class of programs described more in deep in Chapter 2. In this section, relationships with our work are discussed.
- In Section **??** we compare finetely-ground programs with $\omega$-restricted programs, a previously existent bottom-up computable class.
- Section 7.3 is about $\mathbb{FDNC}$ programs, a recently proposed class of logic programs allowing for function symbols, disjunction and non-monotonic negation under answer set semantics.
- Finally, in Section 7.4 we relate our work with some other classes of programs proposed both in ASP and in other research communities.

## 7.1 Finitary Programs

Finitary programs [Bonatti, 2004; Baselice *et al.*, 2007] are a major contribution to the introduction of recursive functional terms (and thus infinite domains) in logic programming under answer set semantics. For this reason, we reported definitions and properties about this class of programs, more widely in Chapter 2. Here we discuss how finitary programs are related to our work.

The class of finitary programs can be seen as a “dual” notion of the class of finitely-ground programs. The former is suitable for a top-down evaluation, while the latter allows for a bottom-up computation. Comparing the computational properties of the two classes, we observe:

- Both finitary programs and finitely-ground programs can express any computable function.

- Ground queries are decidable for both finitary and finitely-ground programs; however, for finitary programs, to obtain decidability one needs to additionally know (“a priori”) what is the set of atoms involved in odd-cycles [Bonatti, 2008].

- Answer sets on finitely-ground programs are computable, while they are not computable on finitary programs. The same holds for nonground queries.

- Recognizing if a program is finitely-ground is semi-decidable; while recognizing if a program is finitary is undecidable (see Section 2.5).

Finitary and $\mathcal{FG}$ programs are not comparable: there are finitary programs that are not finitely-ground, and finitely-ground programs that are not finitary. The syntactic restrictions imposed by the two notions somehow come from the underlying computational approaches (top-down vs bottom-up).

Finitary programs impose that all rule variables must occur in the head; while finitely-ground programs require that all rule variables occur in the positive body. Therefore, $p(X,Y)$ :- $q(X)$. is safe for finitary programs, while it is not for finitely-ground programs (as $Y$ is not range-restricted).

On the contrary, $p(X,Y)$ :- $q(X,V), r(V,Y)$ is safe for finitely-ground programs, while it is not admissible for finitary programs (because of the “local” variable $V$).

Similarly, for the nesting level of the functions: it cannot increase head-to-body for finitary programs, while it cannot increase body-to-head for finitely-ground programs. For instance, a program with the rule: $p(X)$ :- $p(f(X))$. is not finitary, while a program with the rule: $p(f(X))$ :- $p(X)$. is not finitely-ground.

Importantly, finitary programs are or-free; while finitely-ground programs allow for disjunctive rules. The class of finitary programs has been extended to the disjunctive case in [Bonatti, 2002]. To this end, a third condition on the disjunctive heads is added to the definition of finitary programs, in order to guarantee the decidability of ground querying.

Concluding, we observe that the bottom-up nature of the notion of $\mathcal{FG}$ programs allows for an immediate implementation of this class in ASP systems (as ASP instantiators are based on a bottom-up computational model). Indeed, we were able to enhance DLV to deal with finitely-ground by small changes in its instantiator, keeping the database optimization techniques which rely on the bottom-up model and significantly improve the efficiency of the instantiation. While an ASP instantiator should be replaced by a top-down grounder to deal with finitary programs.

## 7.2 $\omega$-restricted Programs

The class of $\omega$-restricted programs [Syrjänen, 2001] allows for function symbols under answer set semantics. They have been effectively implemented into Smodels [Simons *et al.*, 2002] - a very popular ASP system.

The notion of $\omega$-restricted program relies on the concept of $\omega$-stratification. An $\omega$-stratification corresponds, essentially, to a traditional stratification w.r.t. negation (see Section 1.4.1), extended by the (uppermost) $\omega$-stratum, which contains all predicates depending negatively on each other (basically, this stratum contains entirely the unstratified part of the program).

In order to avoid infiniteness/undecidability, programs must fulfill some syntactic conditions w.r.t. an $\omega$-stratification. In particular, each variable appearing in a rule must also occur in a positive body literal belonging to a *strictly* lower stratum than the head.

The above restrictions are strong enough to guarantee the computability of answer sets, yet losing recursive completeness. Thus, $\omega$-restricted programs are strictly less expressive than both finitary and $\mathcal{FG}$ programs (which can express all computable functions).

From a merely syntactic viewpoint, the class of $\omega$-restricted programs is uncomparable with that of finitary programs, while it is strictly contained in the class of $\mathcal{FD}$ programs (and thus, of $\mathcal{FG}$ programs). Indeed, if a program $\mathcal{P}$

is $\omega$-restricted, then each variable appearing in a rule head fulfills Condition 3 of Definition 3.25 (thus, $\mathcal{P}$ is $\mathcal{FD}$). On the contrary, there are $\mathcal{FD}$ programs that are not $\omega$-restricted: for instance, the $\mathcal{FD}$ program made of the single rule $p(X)$ :- $p(f(X))$ is $\mathcal{FD}$ but it is not $\omega$-restricted.

## 7.3 $\mathbb{FDNC}$ Programs

The class $\mathbb{FDNC}$ of logic programs [Simkus and Eiter, 2007] allows for function symbols, disjunction and non-monotonic negation under answer set semantics.

In order to retain the decidability of the standard reasoning tasks, the structure of any rule must be chosen among one out of seven predefined forms. These syntactic restrictions ensure that programs have a *forest-shaped model* property.

Answer sets of $\mathbb{FDNC}$ programs are in general infinite, but have a finite representation which can be exploited for knowledge compilation and fast query answering. The class of $\mathbb{FDNC}$ programs is less expressive than both finitary and finitely-ground programs.

From a syntactic viewpoint, $\mathbb{FDNC}$ programs are uncomparable with both finitary and finitely-ground programs. Notably, $\mathbb{FDNC}$ programs are finitely-recursive, but not necessarily finitary.

## 7.4 Other Works

Recently, in [Lin and Wang, 2008], functions have been proposed as a tool for obtaining a more direct and compact representation of problems, and for improving the performance of ASP computation by reducing the size of resulting ground programs. The class of programs which is considered is strictly contained in $\omega$-restricted programs: indeed, predicates as well as functions must range over finite domains, which must be explicitly (and extensively) provided.

The idea of $\mathcal{FG}$ programs is also related to termination studies of SLD-resolution for Prolog programs (see e.g. [Schreye and Decorte, 1994; Bossi *et al.*, 1994; Bruynooghe *et al.*, 2007]). In this context, several notion of *norm* for complex terms were introduced.

Intuitively, proving that norms of sub-goals are non-increasing during top-down evaluation ensures decidability of a given program. Note that such techniques can not be applied in a straightforward way to our setting, for a series of technical differences. First, propagation of norm information should be studied from rules bodies to heads while traditional termination analysis works the other way around. Also, top-down termination analysis often integrates right recursion avoidance techniques, which are not required in the context of ASP.

As for the deductive database field, we recall that one of the first comprehensive proposals has been $\mathcal{LDL}$ [Naqvi and Tsur, 1989], a declarative language featuring a non-disjunctive logic programming paradigm based on bottom-up model query evaluation. $\mathcal{LDL}$ provides a rich data model including the possibility to manage complex objects, lists and sets. The language allows for a stratified form of negation, while functional terms are managed by means of "infinite" base relations computed by external procedures; proper restrictions (called constraints) and checks based on structural properties of the program (interdependencies between arguments) ensure that a finite number of tuples are generated for each relation.

With respect to the enriched language presented in Chapter 6, it is worth remembering that the book [Baral, 2003] showed examples of how certain kinds of reasoning about sets and lists could be captured by propositional ASPs, thus giving examples of programs with function symbols that can be rewritten as propositional ASPs.

# Conclusions

This thesis regards Answer Set Programming, a very powerful and expressive formalism which is quite popular in the areas of non-monotonic reasoning and logic programming.

Although many efficient ASP solvers have been developed in the last years, encouraging a number of applications in many real-world contexts, some ASP limitations still hold. Indeed, while answer set semantics was defined in the setting of a general first order language, current ASP frameworks and implementations, are based in essence on function-free languages. Therefore, even by using state-of-the-art systems, one cannot directly reason about recursive data structures and infinite domains, such as XML/HTML documents, lists, time, etc. Several contributions published recently, witness as ASP community clearly perceives the need of supporting functions in order to make ASP better suited for real-world applications. However, we still miss a proposal which is fully satisfactory from a linguistic viewpoint (high expressiveness) and suited to be incorporated in the existing ASP systems. Indeed, at present no ASP system allows for a reasonably unrestricted usage of function terms. Functions are either required not to be recursive or subject to severe syntactic limitations, if allowed at all in ASP systems.

In this thesis we focus on the definition of a new class of logic programs allowing for (possibly recursive) function symbols, disjunction and negation. We demonstrate that this class of programs, called finitely-ground programs, is highly expressive and enjoys many relevant computational properties. In particular, answer sets are bottom-up computable and then both brave and cautious reasoning are decidable, even for non-ground queries. Some syntactic conditions have been identified to tailor a subclass of finitely-ground programs (finite-domain programs), in order to guarantee "a priori" termination when needed. Furthermore, the Magic Sets technique is exploited, in order to make all positive finitary programs bottom-up computable, and then enlarge the class of finitely-ground programs.

This theoretical work has had practical application in the extension of the DLV system, one of the most famous system implementing the ASP formalism. First, external functions have been integrated in the context of ASP through the possibility of defining external predicates. Then, a couple of (built-in) external functions have been implemented to manage functional terms. Furthermore, other two different types of complex terms such as lists and sets have enriched the DLV supported language. All these extensions yield a very powerful system where the user can exploit the full expressiveness of finitely-ground programs (able to encode any computable function), or require the finite-domain check, getting the guarantee of termination. The system prototype, examples and manual are available for downloading [Calimeri *et al.*, since 2008].

Future work will focus on extending the rewriting algorithm based on Magic Sets, in order to consider also partially ground queries and programs with negation.

# Bibliography

[Abiteboul and Vianu, 1991] Serge Abiteboul and Victor Vianu. Datalog Extensions for Database Queries and Updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[Anger *et al.*, 2001] Christian Anger, Kathrin Konczak, and Thomas Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 406–410. Springer Verlag, September 2001.

[Apt and Blair, 1991] Krzysztof R. Apt and Howard A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundamenta Informaticae*, 14(3):339–343, 1991.

[Apt and van Emden, 1982] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[Apt *et al.*, 1988] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Minker [1988], pages 89–148.

[Bancilhon *et al.*, 1986] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM.

[Baral and Gelfond, 1994] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[Baral, 2003] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[Baselice *et al.*, 2007] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On Finitely Recursive Programs. In *23rd International Conference on Logic Programming (ICLP-2007)*, volume 4670 of *LNCS*, pages 89–103. Springer, 2007.

[Beeri and Ramakrishnan, 1987] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 269–284, New York, NY, USA, 1987. ACM.

[Bell *et al.*, 1994] Colin Bell, Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *Journal of the ACM*, 41:1178–1215, 1994.

[Ben-Eliyahu and Dechter, 1994] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[Bonatti *et al.*, 2008] Piero A. Bonatti, Enrico Pontelli, and Tran Cao Son. Credulous resolution for answer set programming. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 418–423, Chicago, Illinois, USA, July 2008. AAAI Press.

[Bonatti, 2001a] Piero A. Bonatti. Reasoning with Infinite Stable Models. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 603–610, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[Bonatti, 2001b] Piero A. Bonatti. Resolution for Skeptical Stable Model Semantics. *Journal of Automated Reasoning*, 27(4):391–421, 2001.

[Bonatti, 2002] Piero A. Bonatti. Reasoning with infinite stable models II: Disjunctive programs. In *Proceedings of the 18th International Conference on Logic Programming (ICLP 2002)*, volume 2401 of *LNCS*, pages 333–346. Springer, 2002.

[Bonatti, 2004] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[Bonatti, 2008] Piero A. Bonatti. Erratum to: Reasoning with infinite stable models [artificial intelligence 156 (1) (2004) 75-111]. *Artificial Intelligence*, 172(15):1833–1835, 2008.

[Bossi *et al.*, 1994] Annalisa Bossi, Nicoletta Cocco, and Massimo Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124(2):297–328, 1994.

[Bruynooghe *et al.*, 2007] Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):10, 2007.

[Buccafurri *et al.*, 2000] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[Cabibbo, 1996] Luca Cabibbo. Expressiveness of Semipositive Logic Programs with Value Invention. In *Logic in Databases*, pages 457–474, 1996.

[Calimeri *et al.*, since 2008] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. `DLV-Complex` homepage, since 2008. `http://www.mat.unical.it/dlv-complex`.

[Cumbo *et al.*, 2004] Chiara Cumbo, Wolfgang Faber, and Gianluigi Greco. Enhancing the magic-set method for disjunctive datalog programs. In *Proceedings of the the 20th International Conference on Logic Programming – ICLP'04*, volume 3132 of *Lecture Notes in Computer Science*, pages 371–385, 2004.

[Dantsin and Voronkov, 1997] Evgeny Dantsin and Andrei Voronkov. Complexity of Query Answering in Logic Databases with Complex Values. In Sergei I. Adian and Anil Nerode, editors, *Proceedings of Logical Foundations of Computer Science, 4th International Symposium, LFCS'97*, volume 1234 of *Lecture Notes in Computer Science*, pages 56–66, Yaroslavl, Russia, July 1997. Springer.

[Dantsin *et al.*, 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[East and Truszczyński, 2001] Deborah East and Mirosłaq Truszczyński. Propositional Satisfiability in Answer-set Programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, volume 2174 of *Lecture Notes in AI (LNAI)*, pages 138–153. Springer Verlag, 2001.

[Egly *et al.*, 2000] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), Austin, TX, USA*, pages 417–422. AAAI Press / MIT Press, July 30 – August 3 2000.

[Eiter and Gottlob, 1995] Thomas Eiter and Georg Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[Eiter *et al.*, 1997] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, and Domenico Saccá. Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases. *Theoretical Computer Science*, 206(1–2):181–218, October 1998.

[Eiter *et al.*, 1999] Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The DLV System. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[Eiter *et al.*, 2000] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[Eiter *et al.*, 2003] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. *Artificial Intelligence*, 144(1–2):157–211, March 2003.

[Eiter *et al.*, 2004] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, April 2004.

[Eiter *et al.*, 2005] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI) 2005*, pages 90–96, Edinburgh, UK, August 2005.

[Faber *et al.*, 2007] Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences*, 73(4):584–609, 2007.

[Friedrich and Ivanchenko, 2008] G. Friedrich and V. Ivanchenko. Diagnosis from first principles for workflow executions. Technical report, Alpen Adria University, Applied Informatics, Klagenfurt, Austria, 2008. http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf.

[Gebser *et al.*, 2007a] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392. Morgan Kaufmann Publishers, January 2007.

[Gebser *et al.*, 2007b] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosław Truszczyński. The first answer set programming system competition. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 3–17, Tempe, Arizona, May 2007. Springer Verlag.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a Source of Expression Complexity. *Annals of Pure and Applied Logic*, 97(1–3):231–260, 1999.

[Gottlob, 1994] Georg Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. In Maurice Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS '94)*, pages 23–42, Ithaca NY, 1994. MIT Press.

[Hartley Rogers, 1987] Jr. Hartley Rogers. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, USA, 1987.

[Heymans *et al.*, 2005] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. In *Proceedings of the Second European Semantic Web Conference, ESWC 2005*, volume 3532 of *Lecture Notes in Computer Science*, pages 392–407, 2005.

[Hull and Yoshikawa, 1990] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, Brisbane, Queensland, Australia*, pages 455–468. Morgan Kaufmann, August 1990.

[Janhunen *et al.*, 2006] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, January 2006.

[Japaridze, 1994] Giorgi Japaridze. The logic of the arithmetical hierarchy. *Annals of Pure and Applied Logic*, 66(2):89–112, 1994.

[Johnson, 1990] David S. Johnson. A Catalog of Complexity Classes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Pub., 1990.

[Kowalski and Kuehner, 1971] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[Kowalski, 1974] Robert A. Kowalski. Predicate Logic as Programming Language. In *IFIP Congress*, pages 569–574, 1974.

[Leone *et al.*, 1997] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[Leone *et al.*, 2001] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP Instantiators by Join-Ordering Methods. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 280–294. Springer Verlag, September 2001.

[Leone *et al.*, 2006] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[Lierler, 2005] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[Lifschitz, 1996] Vladimir Lifschitz. Foundations of Logic Programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[Lin and Wang, 2008] Fangzhen Lin and Yisong Wang. Answer Set Programming with Functions. In *Proceedings of Eleventh International Conference on*

*Principles of Knowledge Representation and Reasoning (KR2008)*, 2008. To appear.

[Lin and Zhao, 2004] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[Lobo *et al.*, 1992] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[Marek *et al.*, 1992] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. How complicated is the set of stable models of a recursive logic program? *Annals of Pure and Applied Logic*, 56(1-3):119–135, 1992.

[Matiyasevich, 1970] Yuri Matiyasevich. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191:279–282, 1970. In Russian. English Translation in: Soviet Mathematical Doklady 11, 354-357.

[Minker, 1988] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[Minker, 1994] Jack Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[Naqvi and Tsur, 1989] Shamim Naqvi and Shalom Tsur. *A logical language for data and knowledge bases*. Computer Science Press, Inc., New York, NY, USA, 1989.

[Papadimitriou, 1994] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Przymusinski, 1988] Teodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[Radziszowski, 1994] Stanislaw P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.

[Ross, 1989] Kenneth A. Ross. The Well-Founded Semantics for Disjunctive Logic Programs. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object-Oriented Databases*, pages 385–402. Elsevier Science Publishers B. V., 1989.

[Schlipf, 1995] John S. Schlipf. The Expressive Powers of Logic Programming Semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995. Abstract in Proc. PODS 90, pp. 196–204.

[Schreye and Decorte, 1994] Danny De Schreye and Stefaan Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[Simkus and Eiter, 2007] Mantas Simkus and Thomas Eiter. FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR2007)*, volume 4790 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2007.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[Subrahmanian *et al.*, 1995] V.S. Subrahmanian, Dana Nau, and Carlo Vago. WFS + Branch and Bound = Stable Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, June 1995.

[Syrjänen, 2001] Tommi Syrjänen. Omega-restricted logic programs. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 2001. Springer-Verlag.

[Tärnlund, 1977] Sten-Åke Tärnlund. Horn Clause Computability. *BIT Numerical Mathematics*, 17(2):215–226, 1977.

[Tarski, 1955] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

[Ullman, 1989] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, 1989.

[Van Emden and Kowalski, 1976] Maarten H. Van Emden and Robert A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[Van Gelder *et al.*, 1988] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proceedings of the Seventh Symposium on Principles of Database Systems (PODS'88)*, pages 221–230, 1988.

[Van Gelder *et al.*, 1991] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Van Gelder, 1988] Allen Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In Minker [1988], pages 1149–1176.

[Wolfinger, 1994] Bernd Wolfinger, editor. Workshop: *Disjunctive Logic Programming and Disjunctive Databases*, Berlin, August 1994. German Society for Computer Science (GI), Springer. $13^{th}$ IFIP World Computer Congress, Hamburg, Germany.

# Appendix A

# Further Examples of Finitely Ground Programs

## A.1 A Finitely Ground Program Simulating a Turing Machine

We next show how a Turing Machine can be encoded by a suitable DLP program simulating its computation. It is worth noting that this encoding is actually executable; it is available for download at [Calimeri *et al.*, since 2008], together with the system prototype.

Let $M$ be a Turing Machine given by the 4-uple $\langle K, \Sigma, \delta, s_0 \rangle$([Papadimitriou, 1994]), where $K$ is a finite set of states, $s_0 \in K$ is the initial state, $\Sigma$ is a finite set of symbols constituting the alphabet (with $\sqcup \notin \Sigma$ standing for the blank symbol), and $\delta : K \times \Sigma \to K \times \Sigma \times \{l, r, \lambda\}$ is the transition function describing the behavior of the machine. Given the current state and the current symbol, $\delta$ specifies the next state, the symbol to be overwritten on the current one, and the direction in which the cursor will move on the tape ($l, r, \lambda$ standing for left, right, stay, respectively). Besides the initial state, there is another special state, which is called final state; the machine halts if the machine reaches this state at some point. Each configuration of $M$ can be encoded in a program $\mathcal{P}_M$ by means of the following predicates.

- $tape(P, Sym, T)$: the tape position $P$ stores the symbol $Sym$ at time step $T$. For each time step, there is an instance of such predicate for every actually used position of the tape.
- $position(P, T)$: the head of $M$ reads the position $P$ on tape at time step $T$.

$position$ has a single true ground instance for each time step.

- $state(St, T)$: at time step $T$ $M$ is in the state $St$. $state$ has a single true ground instance for each time step.

$\mathcal{P}_M$ encodes the transition function $\delta$ in the following way: For each $St_c$, $Sym_c$, $St_n$, $Sym_n$, $D$, such that $\delta(St_c, Sym_c) = (St_n, Sym_n, D)$ we add to $P_M$ a fact of the form $delta(St_c$, $Sym_c$, $St_n$, $Sym_n$, $D)$. The initial input is encoded by a proper set of facts describing all tape positions at the first time step (facts of the form $tape(P, Sym, 0)$ ), a fact of the form $state(s_0, 0)$, and a fact of the form $position(P, 0)$ where $P$ is the initial position of the head. The rules defining the evolution of the machine configurations are reported next. For the sake of readability, we exploit some comparison built-ins, that could be easily simulated by means of suitable predicates.

$$\begin{array}{lll}
(r_1) & position(P, s(T)) & \text{:-}\ position(s(P), T), state(St, T), \\
& & \quad tape(s(P), Sym, T), delta(St, Sym, \_, \_, l). \\
(r_2) & position(s(P), s(T)) & \text{:-}\ position(P, T), state(St, T), \\
& & \quad tape(P, Sym, T), delta(St, Sym, \_, \_, r). \\
(r_3) & position(P, s(T)) & \text{:-}\ position(P, T), state(St, T), \\
& & \quad tape(P, Sym, T), delta(St, Sym, \_, \_, \lambda). \\
(r_4) & state(St1, s(T)) & \text{:-}\ position(P, T), state(St, T), \\
& & \quad tape(P, Sym, T), delta(St, Sym, St1, \_, \_). \\
(r_5) & tape(P, Sym1, s(T)) & \text{:-}\ position(P, T), state(St, T), \\
& & \quad tape(P, Sym, T), delta(St, Sym, \_, Sym1, \_). \\
(r_6) & tape(P, Sym, s(T)) & \text{:-}\ position(P1, T), tape(P, Sym, T), P \neq P1. \\
(r_7) & tape(P, \sqcup, T) & \text{:-}\ position(P, T), lastUsedPos(L, T), P > L. \\
(r_8) & lastUsedPos(L, s(T)) & \text{:-}\ lastUsedPos(L, T), position(P, T), P \leq L. \\
(r_9) & lastUsedPos(P, s(T)) & \text{:-}\ lastUsedPos(L, T), position(P, T), P > L.
\end{array}$$

First three rules encode how the tape position changes according to the transition function; the fourth updates the state. Rule $r_5$ updates, for each time step, the current tape position with the new symbol to be stored, with rule $r_6$ stating that all other positions remain unchanged. Rules $r_7, r_8, r_9$ allow to manage the semi-infinite tape. Indeed, the whole tape is not explicitly encoded; rather, each tape position is initialized with a blank symbol when reached for the first time (moving right, the tape being limited at left).

Given a valid tape $x$ encoded by means of a set $X$ of facts having the form $tape(p, s, 0)$, one can show that the computation of $(P_M \cup X)^\gamma$ follows in one-to-one correspondence the computation of $M$ on the tape $x$. $\gamma$ is unique and contains a single component $C$ having a corresponding module $M$. We have that $S_0 = EDB(\mathcal{P}_M)$, and $S_1 = S_0 \cup \Phi^{\infty}_{M,S_0}(\emptyset)$. Let $\Phi(t) = \Phi^{t}_{M,S_0}(\emptyset)$. Then, the value of $\Phi(t)$ directly corresponds to the step $t$ of $M$. It is easy to note that, at step $t+1$, $\Phi(t+1)$ can be larger than $\Phi(t)$ only if, at step $t$, $\Phi(t)$ contains an atom $state(st, t)$ for $st$ not a final state. In such a case by means of rules $r_1$ through $r_5$, new atoms of form $position(p, sym, t+1), state(st, t+1), tape(p, sym, t+1)$ are added to $\Phi(t+1)$.

## A.2 Towers of Hanoi Example

We report next an $\mathcal{FG}$ program encoding the famous Towers of Hanoi puzzle. This program, as well as other examples, is available online at [Calimeri *et al.*, since 2008].

```
% ——————————————————- begin of logic program ——————————————————-

#include ⟨ListAndSet⟩

%—— initial settings ——
number_of_moves(15).
largest_disc(4).
initial_state(towers([4, 3, 2, 1], [ ], [ ])).
goal(towers([ ], [ ], [4, 3, 2, 1])).
disc(X) :- largest_disc(X).
disc(X) :- disc(#succ(X)), X! = 0.
legalStack([ ]).
legalStack([T]) :- disc(T).
legalStack([T|[T1|S]]) :- legalStack([T1|S]), disc(T), T > T1.

% —— possible states ——
possible_state(0, towers(S1, S2, S3)) :- initial_state(towers(S1, S2, S3)).
possible_state(I, towers(S1, S2, S3)) :- possible_move(I, _, towers(S1, S2, S3)),
                                         legalStack(S1), legalStack(S2), legalStack(S3).

% —— possible moves ——
% from stack one to stack two.
possible_move(#succ(I), towers( [X|S1], S2, S3), towers(S1, [X|S2], S3)) :-
                                    possible_state(I, towers([X|S1], S2, S3)),
                                    legalMoveNumber(I), legalStack([X|S2]).
% from stack one to stack three.
possible_move(#succ(I), towers([X|S1], S2, S3), towers(S1, S2, [X|S3])) :-
                                    possible_state(I, towers([X|S1], S2, S3)),
                                    legalMoveNumber(I), legalStack([X|S3]).
% from stack two to stack one.
possible_move(#succ(I), towers(S1, [X|S2], S3), towers([X|S1], S2, S3)) :-
```

$$\begin{aligned}
&possible\_state(I,\ towers(S1,[X|S2],S3)),\\
&legalMoveNumber(I),\ legalStack([X|S1]).
\end{aligned}$$

% from stack two to stack three.

$possible\_move(\#succ(I),\ towers(S1,[X|S2],S3),\ towers(S1,S2,[X|S3]))$ :-
$possible\_state(I,\ towers(S1,[X|S2],S3)),$
$legalMoveNumber(I),\ legalStack([X|S3]).$

% from stack three to stack one.

$possible\_move(\#succ(I),\ towers(S1,S2,[X|S3]),\ towers([X|S1],S2,S3))$ :-
$possible\_state(I,\ towers(S1,S2,[X|S3])),$
$legalMoveNumber(I),\ legalStack([X|S1]).$

% from stack three to stack two.

$possible\_move(\#succ(I),\ towers(S1,S2,[X|S3]),\ towers(S1,[X|S2],S3))$ :-
$possible\_state(I,\ towers(S1,S2,[X|S3])),$
$legalMoveNumber(I),\ legalStack([X|S2]).$

%—— actual moves ——

% a solution exists if and only if there is a '$possible\_move$' leading to the goal.
% in this case, starting from the goal, we proceed backward to the initial state to single out the full
% set of moves.

$move(I,towers(S1,S2,S3))$ :- $goal(towers(S1,S2,S3)),$
$possible\_state(I,towers(S1,S2,S3)).$

$move(I,towers(S1,S2,S3)) \vee nomove(I,towers(S1,S2,S3))$ :-
$move(\#succ(I),\ towers(A1,A2,A3)),$
$possible\_move(\#succ(I),\ towers(S1,S2,S3),\ towers(A1,A2,A3)).$

%—— precisely one move at each step ——

$moveStepI(I)$ :- $move(I,\_).$
:- $legalMoveNumber(I),$ not $moveStepI(I).$
:- $legalMoveNumber(I),\ move(I,T1),\ move(I,T2),\ T1! = T2.$
$legalMoveNumber(0).$
$legalMoveNumber(\#succ(I))$ :- $legalMoveNumber(I), number\_of\_moves(J), I < J.$

% ———————————— end of logic program ————————————

By invoking the system at the command line as follows:

$ $\langle DLV * executable\rangle\ \ \langle programfilename\rangle\ \ -fdnocheck\ \ -N = 15\ \ -filter = move$

the next (unique) answer set is output:

$\{\ move(15,towers([\,],[\,],[4,3,2,1])), move(14,towers([\,],[4],[3,2,1])),$
$move(13,towers([3],[4],[2,1])), move(12,towers([4,3],[\,],[2,1])),$
$move(11,towers([4,3],[2],[1])), move(10,towers([3],[2],[4,1])),$
$move(9,towers([\,],[3,2],[4,1])), move(8,towers([\,],[4,3,2],[1])),$
$move(7,towers([1],[4,3,2],[\,])), move(6,towers([4,1],[3,2],[\,])),$
$move(5,towers([4,1],[2],[3])), move(4,towers([1],[2],[4,3])),$
$move(3,towers([2,1],[\,],[4,3])), move(2,towers([2,1],[4],[3])),$
$move(1,towers([3,2,1],[4],[\,])), move(0,towers([4,3,2,1],[\,],[\,]))\ \}$

# Appendix B

# VI-restrictedness Recognizing Algorithm

## B.1 The Recognizer Algorithm

We next show an algorithm for recognizing if a program admitting external predicate is VI-restricted. Several comment lines (i.e. text preceded by the '%' sign) are inserted in the pseudo code implementing the algorithm, in order to explain both variables usage and crucial steps performed. The algorithm exploits some auxiliary functions having a quite evident meaning and whose implementation is straightforward. Conversely, a very relevant role is played by the 'isBlocked' function, explained in more details in Appendix B.2.

```
Bool Function recognizer ( var $SA$: Set{ Attr };
                              % SA is initialized with provable savior attributes
                              % (i.e. attributes that do not depend from
                              % dangerous attributes.
                           var $NSA$: Set{ pair⟨ Attr, Set{ Attr } ⟩ };
                              % NSA is initialized with attributes which cannot be
                              % proven to be savior, each of which is associated with the
                              % set of dangerous attributes that prevent them to be savior.
                           var RTBC : Set{ Rule } ) % Set of dangerous rules to check.
    Bool $NSA\_Updated = true$;
    While ( $NSA\_Updated$ ) do % Try to prove $VI$-restriction when some change occurs.
        $NSA\_Updated = false$;
        For each Rule $r \in RTBC$ do % free(r) = free variables appearing in the rule $r$.
                Set{Var} $varsTBC$ = free( $r$ );
                Bool $allBlocked = true$;
                For each Var $v \in varsTBC$ do
                    % isBlocked tells if v is blocked in r by means of attributes
                    % currently in SA.
                    If ( isBlocked( v, r, SA ) ) then
                           % headAttr returns reference to the head attribute of r
                           % containing v.
                           Attr $p[i]$ = headAttr( $v$, $r$ );
                           % update processes the $NSA$ set, deleting $p[i]$ from each set $S$.
                           % such that $p[i] \in S$ and $\langle q[j], S\rangle \in NSA$.
                           % Then each attribute $q[j]$ such that $\langle q[j], S\rangle \in NSA$
                           % and $S = \emptyset$ is moved from $NSA$ to $SA$.
                           update( $NSA$, $SA$, $p[i]$ );
                           % A change occurred, so we have to continue cycling.
                           $NSA\_Updated = true$;
                    Else % At least one free variable can't be proved as blocked.
                           $allBlocked = false$;
                    EndIf
                EndFor
                If ( $allBlocked$ ) then
                    $RTBC$.delete($r$); % $r$ is VI-restricted: can be deleted from RTBC.
                EndIf
        EndFor
    EndWhile
    If ( $RTBC == \emptyset$ ) then
        Return $true$
    Else % Display the set of rules that can't be proved as VI-restricted.
        printINSAne( $RTBC$ )
        Return $false$
    EndIf
EndFunction
```

## B.2 The IsBlocked Algorithm

Briefly, recalling Definition 5.20, it is enough to find at least an external atom in the body of the given rule such that the variable appears in output position and all the variables in input position are blocked.

```
Bool Function isBlocked ( $v$: Var; % The variable to be checked as blocked.
                          $r$: Rule; % Current rule.
                          $SA$: Set{ Attr } ) % Savior attributes.
    % The set of all external predicate atoms in the positive body,
    % including the free variable $v$.
    Set{External_Atom} $EAS$ = externalAtomsWithFreeVar( $r$, $v$ );
    Bool $isB = false$;
    External_Atom $\#b = EAS$.first();
    % At least one external predicate must have all of its input variables blocked.
    While ( $!isB$ && $\#b \neq EAS$.end() ) do
        % The set of input variables for the current external predicate.
        Set{ Var } $inputVarsTBC$ = inputPatternVars( $\#b$ );
        Bool $allVarsBlocked = true$;
        Var $currInputVar = inputVarsTBC$.first();
```

```
            % Check savior property for all variables included in $inputVarsTBC$.
            While ( $allVarsBlocked$ && $currInputVar \neq inputVarsTBC$.end() ) do
                  % All the attributes of standard positive atom in the rule,
                  % having as variable $currInputVar$.
                  Set{ Attr } $potentiallySavior$ =
                        attrsWithVar( $currInputVar$, $r$ );
                  Bool $saviorAttrFound = false$;
                  Attr $currAttr = potentiallySavior$.first();
                  % One savior attribute is sufficient.
                  While ( !$saviorAttrFound$ &&
                        $currAttr \neq potentiallySavior$.end() ) do
                     If ( $currAttr \in SA$ ) then
                        $saviorAttrFound = true$;
                     Else
                        $currAttr = potentiallySavior$.next();
                     EndIf
                  EndWhile
                  If ( $saviorAttrFound$ ) then
                     % Check the next input variable.
                     $currInputVar = inputVarsTBC$.next();
                  Else
                     % This input variable is currently not blocked.
                     $allVarsBlocked = false$;
                  EndIf
            EndWhile
            If ( $allVarsBlocked$ ) then
                  % An external predicate atom having
                  % all its input variables blocked has been found.
                  $isB = true$;
            Else
                  % Try the next external atom including the free variable $v$.
                  $\#b = EAS$.next();
            EndIf
      EndWhile
      return $isB$;
EndFunction
```

# Appendix C

# Lists and Sets Manipulation Library

## C.1 List Functions

**#append**
Meaning: return the list resulting from the concatenation of the elements of two lists
Example: #append([a,b,c],[f,m]) results to [a,b,c,f,m]

**#delete**
Meaning: return the list obtained deleting all occurrences of an element in a list
Example: #delete(b,[a,b,c,b,e]) results to [a,c,e]

**#delNth**
Meaning: return the list obtained deleting an element in a specified position in a list
Example: #delNth([a,b,c,b,e],3) results to [a,b,b,e]

**#head**
Meaning: return the first element in a list
Example: #head([a,b,c,b,e]) results to a

**#insLast**
Meaning: return the list obtained inserting an element as last in a list
Example: #insLast([a,b,c,b,e],3) results to [a,b,b,e]

**#insNth**

Meaning: return the list obtained inserting an element in a specified position in a list
Example: #insNth([a,b,c,h,e],s,2) results to [a,s,b,c,h,e]

**#last**
Meaning: return the last element of a list
Example: #last([a,b,c]) results to c

**#length**
Meaning: return the number of elements of a list
Example: #length([a,b,c]) results to 3

**#memberNth**
Meaning: return the element in the specified position in a list
Example: #memberNth([a,b,c,h,e],3) results to c

**#reverse**
Meaning: return the list obtained reversing the order of the elements in a lists
Example: #reverse([a,b,c,h,e]) results to [e,h,c,b,a]

**#select**
Meaning: return the list obtained after selection of the first occurrence of an element in a list
Example: #select(b,[a,b,c,b,e]) results to [a,c,b,e]

**#tail**
Meaning: return the list obtained after deletion of its first element
Example: #tail([a,b,c,h,e]) results to [b,c,h,e]

## C.2 Set Functions

**#card**
Meaning: return the number of elements of a set
Example: #card({a,b,c}) results to 3

**#delete**

Meaning: return the set obtained deleting an element from a set
Example: #delete({a,r,t},r) results to {a,t}

**#difference**
Meaning: return the set resulting from the difference between two sets
Example: #difference({a,r,t},{f,r,s}) results to {a,t}

**#insert**
Meaning: return the set obtained inserting an element into a set
Example: #insert({a,b,c,e,h},d) results to {a,b,c,d,e,h}

**#intersection**
Meaning: return the set resulting from the intersection of two sets
Example: #intersection({a,r,t},{f,r,s}) results to {r}

**#union**
Meaning: return the set resulting from the union of two sets
Example: #union({a,r,t},{f,r,s}) results to {a,f,r,s,t}

## C.3 List Predicates

#**member**
Meaning: is true if the first argument is a member of the list given as second argument
Example: #member(c,[a,b,c]) is valuated as true

#**sublist**
Meaning: is true if the first argument is a sublist of the second argument
Example: #sublist([c,b],[a,b,c,b,e]) is valuated as true

## C.4 Set Predicates

#**isEmpty**
Meaning: is true if the argument is an empty set

Example: #isEmpty({}) is valuated as true

**#member**
Meaning: is true if the first argument is a member of the set given as second argument
Example: #member(c,{a,b,c}) is valuated as true

**#subset**
Meaning: is true if the first argument is a subset of the second argument
Example: #subset({a,c},{a,b,c,b,e}) is valuated as true