

Università della Calabria

Dipartimento di Matematica

**Dottorato di Ricerca in Matematica e Informatica**

XXII Ciclo

---

S.S.D. INF/01 - Informatica

Tesi di Dottorato

**Extending the ASP System DLV<sup>DB</sup> to Support  
Complex Terms and Procedural Sub-tasks**

Erika De Francesco

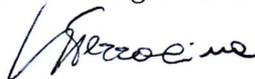


**Supervisor**

Prof. Nicola Leone

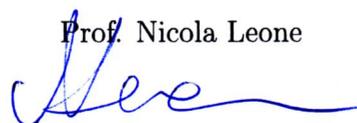


Prof. Giorgio Terracina



**Coordinatore**

Prof. Nicola Leone



---

Anno Accademico 2009 - 2010

# Extending the ASP System $DLV^{DB}$ to Support Complex Terms and Procedural Sub-tasks

Ph.D. Thesis

Erika De Francesco

## **Acknowledgements**

I would like to thank my advisors, Prof. Nicola Leone and Prof. Giorgio Terracina, who guided my research activity during these years, playing a fundamental role in the development of this thesis.

I would like also to acknowledge Dr. Claudio Panetta, with whom I worked on the DLV<sup>DB</sup> project, for his valuable help and friendship, and Exeura S.r.l., for giving me the opportunity to study for my Ph.D. program while working on many interesting business projects.

## Sommario

---

In molti scenari scientifici e di business vengono generate e memorizzate a velocità crescenti grandi quantità di dati su database centralizzati o distribuiti. Ogni giorno, in tutto il mondo, vengono eseguiti innumerevoli esperimenti scientifici che generano petabyte di dati, ed un numero crescente di applicazioni di e-commerce e di e-business memorizzano e gestiscono enormi database relativi a prodotti, clienti e transazioni. Questa crescita esplosiva dei dati ha evidenziato l'urgenza di sviluppare nuove tecniche e strumenti per estrarre nuova conoscenza e informazioni utili da questa enorme mole di dati in modo intelligente e automatico.

In questo contesto, la Programmazione Logica Disgiuntiva (DLP) con semantica dei modelli stabili, indicata anche con il termine Answer Set Programming (ASP), ha ricevuto grande interesse negli ultimi anni perchè rappresenta un potente metodo dichiarativo per i processi di Knowledge Representation and Reasoning (KRR) ed è fondamentale per eseguire un'analisi efficiente dei dati. Il successo di questa tecnica è dimostrato dall'ampio numero di applicazioni reali che la utilizzano, le quali includono, tra le altre, integrazione di informazioni, fraud detection, diagnostica e planning, motivando l'implementazione di numerosi sistemi che supportano DLP.

Il linguaggio DLP è molto espressivo e permette di modellare problemi complessi. Tuttavia, nonostante l'elevata espressività di questo linguaggio, gli attuali sistemi DLP non supportano in modo adeguato alcuni scenari reali. In particolare, le limitazioni degli attuali sistemi DLP possono essere riassunte in quattro problemi principali [62]: (i) i sistemi non sono in grado di gestire applicazioni data intensive, in quanto lavorano esclusivamente in memoria centrale; (ii) forniscono una limitata interoperabilità con i DBMS esterni; (iii) non sono idonei a modellare problemi intrinsecamente procedurali; (iv) non possono effettuare reasoning su strutture dati ricorsive come i documenti XML/HTML.

Il sistema  $DLV^{DB}$  [41] è stato ideato e sviluppato per superare le limitazioni sopra elencate, incrementando la cooperazione tra i sistemi ASP e i database.  $DLV^{DB}$  garantisce miglioramenti sostanziali sia nella valutazione dei programmi logici, sia nella gestione di dati di input ed output distribuiti su diversi database. Inoltre,  $DLV^{DB}$  presenta funzionalità avanzate che garantiscono la sua efficienza e usabilità in applicazioni reali. Tali funzionalità includono:

- Pieno supporto per il datalog disgiuntivo con negazione non stratificata e funzioni aggregate.
- Una strategia di valutazione che esegue la maggior parte delle attività di reasoning in memoria secondaria, permettendo in tal modo di eseguire reasoning in applicazioni data intensive senza degradare le prestazioni del sistema.
- Primitive per integrare i dati provenienti da database distribuiti, permettendo di specificare facilmente quali dati devono essere considerati di input e quali di output per il programma.

---

Nella sua implementazione iniziale,  $DLV^{DB}$  non supportava predicati esterni, liste, e simboli di funzione, che sono di fondamentale importanza per potenziare le capacità di KRR dei linguaggi DLP. In particolare, i simboli di funzione e le liste permettono l'aggregazione di dati atomici, la manipolazione di strutture dati complesse e la generazione di nuovi simboli, mentre i predicati esterni permettono di isolare porzioni procedurali di codice per invocarle all'interno di programmi logici dichiarativi. Il principale obiettivo di questa tesi è estendere il sistema  $DLV^{DB}$  in modo che supporti i costrutti del linguaggio sopra menzionati, al fine di incrementare il suo potere espressivo e la sua efficienza. Fornendo supporto a predicati esterni, liste e simboli di funzione,  $DLV^{DB}$  permette di effettuare reasoning direttamente su strutture dati ricorsive, come i documenti semi-strutturati. Questa è una caratteristica molto importante, sia per applicazioni standard, sia per applicazioni emergenti come quelle per la manipolazione di documenti XML.

In sintesi, i contributi di questo lavoro di tesi possono essere riassunti come segue:

- Il sistema  $DLV^{DB}$  è stato esteso per fornire pieno supporto ai predicati esterni. Poiché  $DLV^{DB}$  trasforma i programmi logici in istruzioni SQL per abilitare la loro elaborazione su database, abbiamo implementato i predicati esterni attraverso chiamate a stored function di database. Questa caratteristica migliora significativamente le capacità di elaborazione di  $DLV^{DB}$  quando il programma include sotto-task intrinsecamente procedurali che sarebbe inefficiente risolvere in modo dichiarativo.
- Un'altra estensione di  $DLV^{DB}$  è stata realizzata al fine di supportare i termini lista. Questi sono stati gestiti attraverso una riscrittura delle regole logiche usando opportuni predicati esterni. In particolare, i programmi contenenti termini lista sono automaticamente riscritti in modo che contengano soltanto termini atomici e predicati esterni. La possibilità di usare liste permette ai programmi di effettuare un reasoning efficiente su strutture dati ricorsive, caratteristica che è richiesta da molte applicazioni reali.
- Una terza estensione al sistema  $DLV^{DB}$  è stata realizzata per supportare programmi con simboli di funzione. In modo simile ai termini lista, le regole sono riscritte sostituendo ogni definizione di termine di funzione con chiamate a predicati esterni. Il supporto ai termini di funzione rappresenta un importante valore aggiunto al sistema, in quanto questi costituiscono un modo conveniente per generare domini e oggetti, permettendo una più naturale rappresentazione di problemi in questi domini.
- È stata realizzata un'ampia libreria di stored function di database per la manipolazione di liste, in modo da facilitare l'uso dei termini lista in  $DLV^{DB}$  attraverso l'uso di funzioni esterne. Ogni funzione è stata implementata per ciascuno dei DBMS attualmente usati come possibili database di lavoro per  $DLV^{DB}$ , ovvero SQLServer, PostgreSQL, MySQL e Oracle.
- Infine, è stato eseguito un ampio insieme di esperimenti per valutare le estensioni sviluppate nel sistema  $DLV^{DB}$ . Tutte e tre le estensioni migliorano in modo significativo l'espressività del linguaggio supportato. I risultati sperimentali mostrano inoltre che tali estensioni riducono in modo significativo i tempi di esecuzione rispetto a programmi  $DLV^{DB}$  equivalenti che non le sfruttano.

La presente tesi è organizzata come segue. Il Capitolo 2 presenta l'ASP basata su un linguaggio DLP esteso con aggregati, predicati esterni, termini di funzione e termini lista. Il capitolo

---

definisce prima la sintassi di tale linguaggio e la semantica ad esso associata; successivamente, esso illustra l'uso di ASP come formalismo per il KRR.

Il Capitolo 3 descrive il sistema  $DLV^{DB}$ . Il capitolo introduce dapprima gli obiettivi e l'architettura del sistema, quindi si concentra sulle direttive ausiliarie che l'utente può specificare per permettere l'interazione fra  $DLV^{DB}$  e i database esterni. Infine, descrive la strategia di valutazione delle regole DLP e la loro traduzione in istruzioni SQL.

Il Capitolo 4 analizza e confronta i più recenti sistemi ASP che supportano negazione non stratificata e altri costrutti avanzati come la disgiunzione e varie forme di constraint. I sistemi ASP sono comparati sulla base di due aspetti principali: l'espressività del linguaggio supportato e l'efficienza nel rispondere ad una query.

Il Capitolo 5 descrive le estensioni che sono state implementate per supportare i predicati esterni, i termini lista e i termini di funzione in  $DLV^{DB}$ . Il capitolo fornisce dapprima le nozioni di base delle stored function che sono usate nel nostro approccio per implementare i predicati esterni in  $DLV^{DB}$ . Successivamente, vengono descritte in dettaglio le strategie di valutazione usate per supportare i predicati esterni, le liste, e i termini di funzione.

Il Capitolo 6 presenta una valutazione sperimentale delle estensioni sviluppate. Il capitolo fornisce un insieme di test case per valutare i benefici, in termini di prestazioni ed espressività, derivanti dall'uso di tali estensioni (supporto a predicati esterni, liste e termini di funzione) in  $DLV^{DB}$ .

Infine, il Capitolo 7 conclude la tesi riassumendo i principali risultati ottenuti e delineando possibili sviluppi futuri, mentre l'Appendice A include il codice sorgente della libreria implementata per la manipolazione di liste.

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Disjunctive Datalog</b>	<b>4</b>
2.1	Syntax . . . . .	4
2.2	Semantics . . . . .	7
2.3	Knowledge representation and reasoning . . . . .	10
2.3.1	The guess and check programming methodology . . . . .	10
2.3.2	Applications of the guess and check technique . . . . .	11
2.3.3	KRR capabilities enhanced by external atoms, lists and function symbols . . . . .	13
<b>3</b>	<b>The DLV<sup>DB</sup> System</b>	<b>15</b>
3.1	System objectives . . . . .	15
3.2	System architecture . . . . .	16
3.3	Auxiliary directives . . . . .	17
3.4	Evaluation strategy . . . . .	20
3.4.1	Evaluation of non disjunctive stratified programs . . . . .	21
3.4.2	Evaluation of disjunctive programs with unstratified negation . . . . .	21
3.5	Translation from DLP to SQL . . . . .	25
3.5.1	Translating non-recursive rules . . . . .	25
3.5.2	Translating recursive rules . . . . .	31
<b>4</b>	<b>Related Work</b>	<b>33</b>
4.1	Relevant features of ASP systems . . . . .	33
4.1.1	Language expressiveness . . . . .	34
4.1.2	Optimizations . . . . .	34
4.2	ASP systems: analysis and comparison . . . . .	35
4.2.1	DLV . . . . .	35
4.2.2	S MODELS . . . . .	37
4.2.3	Cmodels . . . . .	39
4.2.4	ASSAT . . . . .	40
4.2.5	noMoRe . . . . .	41
4.2.6	SLG . . . . .	43
4.2.7	DeReS . . . . .	44
4.2.8	XSB . . . . .	45
4.2.9	claspD . . . . .	46
4.2.10	Other systems . . . . .	48

<b>5 DLV<sup>DB</sup> Extensions</b>	<b>49</b>
5.1 Database stored functions . . . . .	49
5.2 Evaluation of external predicates . . . . .	51
5.2.1 Output variable bound to other variables in the rule's body . . . . .	53
5.2.2 Output variable bound to a constant . . . . .	54
5.2.3 Output variable bound to a variable in the rule's head . . . . .	54
5.2.4 Output variable bound to an input parameter of another external predicate . . . . .	55
5.3 Evaluation of list terms . . . . .	55
5.3.1 Packing operation . . . . .	56
5.3.2 Unpacking operation . . . . .	58
5.4 Evaluation of functional terms . . . . .	60
5.4.1 Packing operation . . . . .	60
5.4.2 Unpacking operation . . . . .	61
<b>6 Evaluation</b>	<b>62</b>
6.1 External predicates . . . . .	62
6.1.1 Number conversion . . . . .	62
6.1.2 String similarity computation . . . . .	65
6.2 List terms . . . . .	68
6.2.1 Towers of Hanoi . . . . .	68
6.2.2 Graph reachability . . . . .	72
6.3 Functional terms . . . . .	75
6.3.1 Database repair . . . . .	76
<b>7 Conclusions</b>	<b>78</b>
<b>A List and Functional Terms Manipulation Library</b>	<b>79</b>
<b>Bibliography</b>	<b>89</b>

# Chapter 1

## Introduction

---

In many scientific and business scenarios, large amounts of data are generated and stored at increasing speed in local or distributed databases. Scientific experiments generating petabytes of data are daily performed in many laboratories all around the world. A growing number of e-commerce and e-business applications store and manage huge databases about products, clients and transactions. This explosive growth of data has raised an urgent need for new techniques and tools to intelligently and automatically infer useful information and knowledge from available data.

In this context, Disjunctive Logic Programming (DLP) under the answer set semantics, often referred to as Answer Set Programming (ASP), has gained lot of interest during the last few years, since it represents a powerful method for declarative Knowledge Representation and Reasoning (KRR) tasks, which are critical to perform effective data analysis. The success of this method is demonstrated by the wide number of real-world applications that include, among others, information integration, frauds detection, diagnostics and planning, also motivating the implementation of several systems supporting DLP.

The DLP language is very expressive and allows for modeling complex problems. However, despite the high expressiveness of this language, current DLP systems do not cope well with real world scenarios. In particular, the main limitations of current DLP systems can be summarized in four main issues [62]: (i) they are not capable of handling data intensive applications, since they work in main memory only; (ii) they provide a limited interoperability with external DBMSs; (iii) they are not well suited for modelling inherently procedural problems; (iv) they cannot reason about recursive data structures such as XML/HTML documents.

The  $DLV^{DB}$  system [41] has been conceived to overcome the above-mentioned limitations, by increasing the cooperation between ASP systems and databases.  $DLV^{DB}$  allows substantial improvements in both the evaluation of logic programs and the management of input and output data distributed on several databases. Moreover,  $DLV^{DB}$  presents enhanced features to improve its efficiency and usability for an effective exploitation of DLP in real-world scenarios. These features include:

- Full support to disjunctive datalog with unstratified negation, and aggregate functions.
- An evaluation strategy devoted to carry out as much as possible of the reasoning tasks in mass memory, thus enabling complex reasonings in data intensive applications without degrading performances.

- Primitives to integrate data from different databases, in order to easily specify which data are to be considered as input or as output for the program.

In its early implementation,  $DLV^{DB}$  did not support external predicates, list terms, and function symbols, which are of fundamental importance to enhance knowledge representation and reasoning capabilities of a DLP language. In particular, function symbols and lists terms allow the aggregation of atomic data, the manipulation of complex data structures and the generation of new symbols, while external predicates allow the isolation of procedural code units for calling them within declarative logic programs. The main goal of this thesis is to extend the  $DLV^{DB}$  system to let it support the above mentioned language constructs, in order to improve its knowledge modelling power. By providing support to external predicates, list terms, and functional symbols,  $DLV^{DB}$  allows to directly reason about recursive data structures, such as lists, semi-structured documents, and so on. This is a very important feature, both for standard knowledge-based tasks and for emerging applications, such as those manipulating XML documents.

The main results of this thesis can be summarized as follows:

- The  $DLV^{DB}$  system has been extended to provide full support to external predicates. Since  $DLV^{DB}$  transforms logic programs into SQL statements to enable database-oriented processing, we implemented external predicates by calls to database stored functions. This feature significantly improves the processing capabilities of  $DLV^{DB}$  when the program includes inherently procedural subtasks that would be inefficiently solved in a declarative way.
- Another extension to  $DLV^{DB}$  has been designed and implemented to support list terms. We handle list terms through a rewriting of the rules using suitable external predicates. In particular, programs containing list terms are automatically rewritten to contain only terms and external predicates. The possibility to use lists allows programs to efficiently reason about recursive data structures, a feature that is required by many real-world applications.
- A third extension to the  $DLV^{DB}$  system has been realized to support programs with functional symbols. Similarly to list terms, rules are rewritten by replacing each functional term definition with calls to external predicates. Support to functional terms represents an important added value to  $DLV^{DB}$ , since they are a very convenient means for generating domains and objects, allowing a more natural representation of problems in such domains.
- A rich library of database stored functions for lists manipulation has been realized to facilitate the use of list terms in  $DLV^{DB}$  through the use of external functions. Each function has been implemented for the DBMSs used as working databases for  $DLV^{DB}$ , namely SQLServer, PostgreSQL, MySQL and Oracle.
- Finally, a wide set of experiments has been performed to evaluate our extensions to the  $DLV^{DB}$  system. All three extensions significantly improve the expressiveness of the supported language. The experimental results show that such extensions also significantly reduce the execution times as compared to  $DLV^{DB}$  programs that do not support them.

The remainder of this chapter is organized as follows. Chapter 2 discusses the *Answer Set Programming* (ASP) framework based on a DLP language extended with aggregates, external predicates, functional terms and list terms. The chapter first defines the syntax of this language

and its associated semantics, i.e., the answer set semantics. Then, it illustrates the usage of ASP as a formalism for knowledge representation and reasoning.

Chapter 3 describes the  $DLV^{DB}$  system. The chapter starts by introducing objectives and the architecture of  $DLV^{DB}$ . Then, it focuses on the auxiliary directives the user can specify to let  $DLV^{DB}$  interact with external databases. Finally, the chapter describes the evaluation strategy of DLP rules and their translation into SQL statements.

Chapter 4 analyzes and compares the most recent ASP systems that support unstratified negation and other advanced constructs like disjunction, and various forms of constraints. The ASP systems are compared on the basis of two main aspects: the expressiveness of the supported language (e.g., its ability to express views, recursive rules, etc.), and the efficiency to answer a query (i.e., the quantity of data to be analyzed for answering a query, and the intrinsic complexity of the query itself).

Chapter 5 describes the extensions that have been implemented to support external predicates, list terms, and functional terms in  $DLV^{DB}$ . The chapter starts by providing the basic notions of database stored functions, which are used in our approach to implement the external predicates in  $DLV^{DB}$ . Then, the evaluation strategies used for supporting external predicates, lists, and functional terms are described in detail.

Chapter 6 presents an experimental evaluation of our extensions to the  $DLV^{DB}$  system. The chapter provides a set of test cases to evaluate the benefits, in terms of performance and expressiveness, deriving from the use of our extensions (support to external predicates, lists, and functional terms) in  $DLV^{DB}$ .

Finally, Chapter 7 concludes the thesis by summarizing the main results achieved and by outlining future work, while Appendix A includes the source code of the implemented library for lists manipulations.

# Chapter 2

## Disjunctive Datalog

---

Disjunctive Logic Programming (DLP) under the answer set semantics has gained lot of interest during the last years, since it represents a powerful and effective method for declarative knowledge representation and reasoning. The success of this method is demonstrated by the wide number of real-world applications that include, among others, information integration, frauds detection, and software configuration, also motivating the implementation of several systems supporting DLP.

This chapter presents the *Answer Set Programming* (ASP) framework based on a DLP language extended with aggregates, external predicates, functional terms and list terms. We assume that the language introduced here will be used to write finite-domain programs, since in this way termination is a priori guaranteed [8].

In the remainder of this chapter, we first define the syntax of this language and its associated semantics, i.e., the answer set semantics. Then, we illustrate the usage of ASP as a formalism for knowledge representation and reasoning.

### 2.1 Syntax

In this section we define the syntax of the extended DLP language following the Prolog's conventions. Based on such conventions, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants.

A *term* is either a *simple term* or a *complex term*. A *simple term* is either a constant or a variable. A *complex term* is either a *functional term* or a *list term*. If  $t_1, \dots, t_n$  are terms, then complex terms are defined as follows.

- A *functional term* is defined as  $f(t_1, \dots, t_n)$ , where  $f$  is a function symbol (also called *functor*) of arity  $n$ .
- A *list term* can be defined using one of following forms:
  - $[t_1, \dots, t_n]$  where  $t_1, \dots, t_n$  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.

If  $t_1, \dots, t_n$  are terms, then  $p(t_1, \dots, t_n)$  is an *atom*. An *atom* can be defined as: an *ordinary atom*, an *aggregate atom* or an *external atom*. An *ordinary atom* is an expression  $p(t_1, \dots, t_n)$ ,

where  $p$  is a *predicate* of arity  $n$  and  $t_1, \dots, t_n$  are terms; *external* and *aggregate atom* names are conventionally preceded by “#”. Names of external atoms are often referred to as *external predicates*. *External predicates* introduce function calls in the program; we assume, by convention, that only the last variable  $O$  of an external predicate  $\#f(X_1, \dots, X_n, O)$  is considered as an output parameter, while all the other variables must be intended as input for  $f$ . An example of external predicate could be  $\#\text{concat}(X, Y, Z)$ , which takes two strings  $X$  and  $Y$  as input and returns a string  $Z$  corresponding to the concatenation of  $X$  and  $Y$ . The following special external predicates are reserved to list manipulation:

- $\#\text{head}(L, H)$ , which receives a list  $L$  and returns its head  $H$ ;
- $\#\text{tail}(L, T)$ , which receives a list  $L$  and returns its tail  $T$ .
- $\#\text{last}(L, E)$ , which receives a list  $L$  and returns its last element  $E$ ;
- $\#\text{memberNth}(L, N, E)$ , which receives a list  $L$  and an index  $N$  and returns the element  $E$  at the specified position  $N$ .

The language also supports a set of built-in predicates such as the comparative predicates equality, less-than, and greater-than ( $=, <, >$ ), as well as arithmetic predicates (like addition or multiplication). For details, refer to [23].

A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair  $\{Vars : Conj\}$ , where  $Vars$  is a list of variables and  $Conj$  is a conjunction of standard atoms.<sup>1</sup> A *ground set* is a set of pairs of the form  $\{\bar{t} : Conj\}$ , where  $\bar{t}$  is a list of constants and  $Conj$  is a ground (variable free) conjunction of standard atoms.

An *aggregate function* is of the form  $f(S)$ , where  $S$  is a set term, and  $f$  is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

An *aggregate atom* is  $f(S) \prec T$ , where  $f(S)$  is an aggregate function,  $\prec \in \{=, <, \leq, >, \geq\}$  is a predefined comparison operator, and  $T$  is a term (variable or constant) referred to as guard.

**Example 2.1.1** The following are aggregate atoms, where the latter contains a ground set and could be a ground instance of the former:

$$\begin{aligned} \#\max\{Z : r(Z), a(Z, V)\} &> Y \\ \#\max\{\langle 2 : r(2), a(2, k) \rangle, \langle 2 : r(2), a(2, c) \rangle\} &> 1 \end{aligned}$$

A *literal*  $l$  is of the form  $p$  or  $\neg p$  where  $p$  is an atom; in the first case  $l$  is a *positive literal* and in the second case  $l$  is a *negative literal*.

A *negation as failure (NAF) literal*  $\ell$  is of the form  $l$  or not  $l$ , where  $l$  is a literal; in the former case  $\ell$  is *positive*, and in the latter case *negative*. Note that an *external atom* can be negated with negation as failure, and, in this case, its variables must be safe in ordinary way.

Given a literal  $l$ , its *complementary literal*  $\neg l$  is defined as  $\neg p$  if  $l = p$  and  $p$  if  $l = \neg p$ . A set  $L$  of literals is said to be *consistent* if, for every literal  $l \in L$ , its complementary literal is not contained in  $L$ .

Rules accepted by the language are *disjunctive rules*. A *disjunctive rule*  $r$  is a formula

<sup>1</sup>Intuitively, a symbolic set  $\{X : a(X, Y), p(Y)\}$  stands for the set of  $X$ -values making  $a(X, Y), p(Y)$  true, i.e.,  $\{X \mid \exists Y \text{ s.t. } a(X, Y), p(Y) \text{ is true}\}$ .

$$a_1 \vee \cdots \vee a_n :- b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m. \quad (2.1)$$

where  $a_1, \dots, a_n$  are standard atoms,  $b_1, \dots, b_m$  are (ordinary, aggregate or external) atoms,  $n \geq 0, m \geq k \geq 0$ . The disjunction  $a_1 \vee \cdots \vee a_n$  is called the *head* of  $r$ , while the conjunction  $b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m$  is called the *body* of  $r$ . A rule without head literals (i.e.  $n = 0$ ) is usually referred to as an *integrity constraint*. A rule having exactly one head literal (i.e.  $n = 1$ ) is called a *normal rule*. If the body is empty (i.e.  $k = m = 0$ ), it is called a *fact*, and the “:-” sign is usually omitted.

If  $r$  is a rule of form (2.1), then  $H(r) = \{a_1, \dots, a_n\}$  is the set of the literals in the head and  $B(r) = B^+(r) \cup B^-(r)$  is the set of the body literals, where  $B^+(r)$  (the *positive body*) is  $\{b_1, \dots, b_k\}$  and  $B^-(r)$  (the *negative body*) is  $\{b_{k+1}, \dots, b_m\}$ .

A *disjunctive datalog program* (alternatively, *disjunctive logic program*, *disjunctive deductive database*)  $\mathcal{P}$  is a finite set of rules. A not-free program  $\mathcal{P}$  (i.e., such that  $\forall r \in \mathcal{P} : B^-(r) = \emptyset$ ) is called *positive*,<sup>2</sup> and a v-free program  $\mathcal{P}$  (i.e., such that  $\forall r \in \mathcal{P} : |H(r)| \leq 1$ ) is called *datalog program* (or *normal logic program*, *deductive database*).

A term (an atom, a rule, a program, etc.) is called *ground*, if it does not contain variables. A ground program is also called a *propositional program*.

A *global* variable of a rule  $r$  is a variable appearing in a standard or external atom of  $r$ ; all other variables are *local* variables.

A rule  $r$  is *safe* if the following conditions hold: (i) each global variable of  $r$  appears in a positive standard literal in the body of  $r$  or as output variable of an positive external atom; (ii) each local variable of  $r$  appearing in a symbolic set  $\{Vars : Conj\}$  appears in an atom of  $Conj$ ; (iii) each guard of an aggregate atom of  $r$  is a constant or a global variable. For instance, the following rules are safe:

$$\begin{aligned} p(X, f(Y, Z)) &:- q(Y), t(X), z(Z), \text{ not } s(X). \\ H(S) &:- \text{ number}(N), \#sqr(N, S). \\ H(S1) &:- \text{ number}(N), \#fatt(N, S), \#sqr(S, S1). \end{aligned}$$

The following are, vice versa, unsafe:

$$\begin{aligned} p(X, f(Y, Z)) &:- q(Y), \text{ not } s(X). \\ H(S) &:- \text{ number}(S), \#sqr(N, S). \\ H(S1) &:- \text{ number}(N), \text{ not } \#fatt(N, S), \#sqr(S, S1). \end{aligned}$$

A program  $\mathcal{P}$  is safe if all  $r \in \mathcal{P}$  are safe. In the following we assume that DLP programs are safe.

Let the *level mapping* of a program  $\mathcal{P}$  be a function  $\|\cdot\|$  from the predicates in  $\mathcal{P}$  to finite ordinals. A DLP program  $\mathcal{P}$  is called *stratified<sup>not</sup>* [3, 56], if there is a level mapping  $\|\cdot\|_s$  of  $\mathcal{P}$  such that, for every rule  $r$ :

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  $l, l' \in H(r)$ ,  $\|l\|_s = \|l'\|_s$ .

<sup>2</sup>In positive programs negation as failure (not) does not occur, while strong negation ( $\neg$ ) may be present.

A DLP program  $\mathcal{P}$  is called *stratified<sup>agg</sup>* [15], if there is a level mapping  $\|\cdot\|_a$  of  $\mathcal{P}$  such that, for every rule  $r$ :

1. If  $l$  appears in the head of  $r$ , and  $l'$  appears in an aggregate atom in the body of  $r$ , then  $\|l'\|_a < \|l\|_a$ ; and
2. If  $l$  appears in the head of  $r$ , and  $l'$  occurs in a standard atom in the body of  $r$ , then  $\|l'\|_a \leq \|l\|_a$ .
3. If both  $l$  and  $l'$  appear in the head of  $r$ , then  $\|l'\|_a = \|l\|_a$ .

**Example 2.1.2** Consider the program consisting of a set of facts for predicates  $a$  and  $b$ , plus the following two rules:

$$\begin{aligned} q(X) &:- p(X), \#count\{Y : a(Y, X), b(X)\} \leq 2. \\ p(X) &:- q(X), b(X). \end{aligned}$$

The program is stratified<sup>agg</sup>, as the level mapping  $\|a\| = \|b\| = 1, \|p\| = \|q\| = 2$  satisfies the required conditions. If we add the rule  $b(X) :- p(X)$ , then no level-mapping exists and the program becomes not stratified<sup>agg</sup>.

Intuitively, the property stratified<sup>agg</sup> forbids recursion through aggregates.

## 2.2 Semantics

The most widely accepted semantics for DLP programs is based on the notions of answer-set, proposed in [30] as a generalization of the stable model concept [29]. Given a DLP program  $\mathcal{P}$ , let:

1.  $U_{\mathcal{P}}$  denote the set of constants appearing in  $\mathcal{P}$  (*Herbrand Universe*);
2.  $B_{\mathcal{P}}$  denote the set of standard atoms constructible from the (standard) predicates of  $\mathcal{P}$  with constants in  $U_{\mathcal{P}}$  (*Herbrand Literal Base*).

Given a set  $X$ , let  $\bar{2}^X$  denote the set of all multisets over elements from  $X$ . Without loss of generality, we assume that aggregate functions map to  $I$  (the set of integers).

A *substitution* is a mapping from a set of variables  $V$  to  $U_{\mathcal{P}}$ :  $\sigma : V \rightarrow U_{\mathcal{P}}$ . Let distinguish two types of substitution:

1. *global substitution* for a rule  $r$  is a substitution from the set of global variables of  $r$  to  $U_{\mathcal{P}}$ ;
2. *local substitution* for a symbolic set  $S$  is a substitution from the set of local variables of  $S$  to  $U_{\mathcal{P}}$ .

**Example 2.2.1** Let  $\sigma = [X/a, Y/b, Z/c]$  be a substitution that replaces every instance of variables  $X, Y, Z$  with constants  $a, b, c$ , respectively. By applying  $\sigma$  to the following rule:

$$r : p(X, Y) : .q(Y, Z), t(X).$$

the following global substitution is obtained:

$$\sigma(r) : p(a, b) : .q(b, c), t(a).$$

Given a symbolic set without global variables  $S = \{Vars : Conj\}$ , the *instantiation of  $S$*  is the following ground set of pairs  $inst(S) : \{\langle \gamma(Vars) : \gamma(Conj) \rangle \mid \gamma \text{ is a local substitution for } S\}$ .

A *ground instance* of a rule  $r$  is obtained in two steps: (i) a global substitution  $\sigma$  for  $r$  is first applied over  $r$ ; (ii) every symbolic set  $S$  in  $\sigma(r)$  is replaced by its instantiation  $inst(S)$ . The instantiation  $Ground(\mathcal{P})$  of a program  $\mathcal{P}$  is the set of all possible instances of the rules of  $\mathcal{P}$ .

**Example 2.2.2** Consider the following program  $\mathcal{P}_1$ :

$$\begin{aligned} & q(1) \vee p(2, 2). \\ & q(2) \vee p(2, 1). \\ & t(X) :- q(X), \#sum\{Y : p(X, Y)\} > 1. \\ & l([Z, Z]) :- q(Z). \\ & f(g(W)) :- q(W). \end{aligned}$$

The instantiation  $Ground(\mathcal{P}_1)$  is the following:

$$\begin{aligned} & q(1) \vee p(2, 2). \\ & t(1) :- q(1), \#sum\{\langle 1:p(1, 1) \rangle, \langle 2:p(1, 2) \rangle\} > 1. \\ & l([1, 1]) :- q(1). \\ & f(g(1)) :- q(1). \\ & q(2) \vee p(2, 1). \\ & t(2) :- q(2), \#sum\{\langle 1:p(2, 1) \rangle, \langle 2:p(2, 2) \rangle\} > 1. \\ & l([2, 2]) :- q(2). \\ & f(g(2)) :- q(2). \end{aligned}$$

An *interpretation  $I$*  for a DLP program  $\mathcal{P}$  is a consistent set of standard ground atoms, that is  $I \subseteq B_{\mathcal{P}}$ . A positive literal  $A$  is true w.r.t.  $I$  if  $A \in I$ , it is false otherwise. A negative literal  $\text{not } A$  is true w.r.t.  $I$ , if  $A \notin I$ , it is false otherwise. An interpretation also provides a meaning for aggregate literals.

Given an interpretation  $I$ , a standard ground conjunction is true (resp. false) w.r.t.  $I$  if all its literals are true. The meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Given an aggregate function  $f(S)$ , the valuation  $I(S)$  of  $S$  w.r.t.  $I$  is the multiset of the first constant of the elements in  $S$  whose conjunction is true w.r.t.  $I$ .

More precisely, let  $I(S)$  denote the multiset  $[t_1 \mid \langle t_1, \dots, t_n : Conj \rangle \in S \wedge Conj \text{ is true w.r.t. } I]$ . The valuation  $I(f(S))$  of an aggregate function  $f(S)$  w.r.t.  $I$  is the result of the application of  $f$  on  $I(S)$ . If the multiset  $I(S)$  is not in the domain of  $f$ ,  $I(f(S)) = \perp$  (where  $\perp$  is a fixed symbol not occurring in  $\mathcal{P}$ ).

An instantiated aggregate atom  $A = f(S) \prec k$  is *true w.r.t.  $I$*  if: (i)  $I(f(S)) \neq \perp$ , and, (ii)  $I(f(S)) \prec k$  holds; otherwise,  $A$  is false. An instantiated aggregate literal  $\text{not } A = \text{not } f(S) \prec k$  is *true w.r.t.  $I$*  if: (i)  $I(f(S)) \neq \perp$ , and, (ii)  $I(f(S)) \prec k$  does not hold; otherwise,  $A$  is false.

Given an interpretation  $I$ , a rule  $r$  is *satisfied w.r.t.  $I$*  if some head atom is true w.r.t.  $I$  whenever all body literals are true w.r.t.  $I$ . An interpretation  $M$  is a *model* of a DLP program  $\mathcal{P}$  if all  $r \in Ground(\mathcal{P})$  are satisfied w.r.t.  $M$ . A model  $M$  for  $\mathcal{P}$  is (subset) *minimal* if no model  $N$  for  $\mathcal{P}$  exists such that  $N \subset M$ .

In the following, a generalization of the Gelfond-Lifschitz transformation to programs with aggregates is provided, as it is defined in [22].

**Definition 2.2.3** [22] Given a ground DLP program  $\mathcal{P}$  and a total interpretation  $I$ , let  $\mathcal{P}^I$  denote the transformed program obtained from  $\mathcal{P}$  by deleting all rules in which a body literal is false w.r.t.  $I$ .  $I$  is an answer set of a program  $\mathcal{P}$  if it is a minimal model of  $\text{Ground}(\mathcal{P})^I$ .

The set of all answer sets for  $\mathcal{P}$  is denoted by  $\mathcal{AS}(\mathcal{P})$

**Example 2.2.4** Consider the following two programs:

$$\begin{aligned} P_1 &: \{p(a) :- \#count\{X : p(X)\} > 0.\} \\ P_2 &: \{p(a) :- \#count\{X : p(X)\} < 1.\} \end{aligned}$$

$\text{Ground}(P_1) = \{p(a) :- \#count\{\langle a : p(a) \rangle\} > 0.\}$  and  $\text{Ground}(P_2) = \{p(a) :- \#count\{\langle a : p(a) \rangle\} < 1.\}$ ; consider also interpretations  $I_1 = \{p(a)\}$  and  $I_2 = \emptyset$ . Then,  $\text{Ground}(P_1)^{I_1} = \text{Ground}(P_1)$ ,  $\text{Ground}(P_1)^{I_2} = \emptyset$ , and  $\text{Ground}(P_2)^{I_1} = \emptyset$ ,  $\text{Ground}(P_2)^{I_2} = \text{Ground}(P_2)$  hold.

$I_2$  is the only answer set of  $P_1$  (because  $I_1$  is not a minimal model of  $\text{Ground}(P_1)^{I_1}$ ), whereas  $P_2$  admits no answer set ( $I_1$  is not a minimal model of  $\text{Ground}(P_2)^{I_1}$ , and  $I_2$  is not a model of  $\text{Ground}(P_2) = \text{Ground}(P_2)^{I_2}$ ).

Note that any answer set  $A$  of  $\mathcal{P}$  is also a model of  $\mathcal{P}$  because  $\text{Ground}(\mathcal{P})^A \subseteq \text{Ground}(\mathcal{P})$ , and rules in  $\text{Ground}(\mathcal{P}) - \text{Ground}(\mathcal{P})^A$  are satisfied w.r.t.  $A$ .

List terms are managed by using a set of function calls. Such functions are used to rewrite programs containing list terms, in order to let them contain only terms and function calls. For example, the rule  $q(H):- \text{dom}(H), \text{list}(T), \text{list}([H|T])$ . is translated into  $q(H):- \text{dom}(H), \text{list}(T), \text{list}(L), \#head(L,H), \#tail(L,T)$ .

As mentioned in Section 2.1, given an external atom  $\#f(X_1, \dots, X_n, O)$  used in a rule  $r$ , only the last variable  $O$  is used as an output parameter, while all the other variables are intended as input for  $f$ ; this corresponds to the function call  $f(X_1, \dots, X_n) = O$ .

The handling of external atoms is carried out during the generation of the  $\text{Ground}(\mathcal{P})$  through the function calls introduced above. To illustrate this process, let us consider the following program:

$$\begin{aligned} & \text{person}(id1, "John", "Smith"). \\ & \text{mergedNames}(ID, N) :- \text{person}(ID, FN, LN), \#concat(FN, LN, N). \end{aligned}$$

The corresponding ground program is:

$$\begin{aligned} & \text{person}(id1, "John", "Smith"). \\ & \text{mergedNames}(id1, "JohnSmith") :- \text{person}(id1, "John", "Smith"). \end{aligned}$$

where the second term of predicate  $\text{mergedNames}$  has been produced by calling function  $\#concat$ .

In the example above, the function call introduces new constant values (i.e., "John Smith") in the program, which can generate an infinite domain. To avoid the generation of infinite-sized answer sets, the programs must be value-invention restricted (cfr. [7]), in other words, new values possibly introduced by external atoms must not propagate through recursion.

## 2.3 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/Optimize (“Guess and Check”) methodology presented in [17]. In this section, we first describe the Guess/Check/Optimize technique and we then illustrate how to apply it on a number of examples. Finally, we show how the modelling capability of ASP is significantly enhanced by supporting function symbols.

### 2.3.1 The guess and check programming methodology

Several problems, including problems of comparatively high computational complexity ( $\Sigma_2^P$ -complete and  $\Delta_3^P$ -complete problems), can be solved in a natural way 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 Guess/Check/Optimize 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 Guess/Check/Optimize 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 Guess/Check/Optimize programming methodology has also positive implications from the Software Engineering viewpoint. Indeed, the modular program

structure in Guess/Check/Optimize 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.

### 2.3.2 Applications of the guess and check technique

In this section, we illustrate the declarative programming methodology described in Section 2.3.1 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 2.3.1** [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 Guess/Check/Optimize program  $\mathcal{P}_{hp}$  solves the problem HAMPATH:

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

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

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

$$inPath(X, Y) \vee outPath(X, Y) :- arc(X, Y).$$

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

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

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  $v_0, v_1, \dots, v_k$  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  $:- \text{start}(Y), \text{inPath}(-, Y)$ . 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 Guess/Check/Optimize 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 2.3.2** [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  $m$  [57]. We next show a program  $\mathcal{P}_{ramsey}$  that allows us to decide whether a given integer  $n$  is not 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 Guess/Check/Optimize program:

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

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  $R(3, 4)$ .

### 2.3.3 KRR capabilities enhanced by external atoms, lists and function symbols

This section describes, through a set of examples, how knowledge representation and reasoning (KRR) capabilities of a DLP language are enhanced by introducing lists, external atoms, and function symbols. In particular we show how function symbols and lists terms allow to aggregate atomic data, manipulate complex data structures and generate new symbols, and external atoms allow to isolate units of a procedural program and call them from declarative logic programs.

For example the term  $notify(delete(F))$  in the atom  $request(S, I, notify(delete(F)), T1)$  is a functional term. Other functional terms are  $father(X)$  and  $mother(X)$  in the atom  $family(X, father(X), mother(X))$  or  $f(g(X))$  in  $p(f(g(X)))$ . Lists can be profitably exploited in order to explicitly model collections of objects where position matters, and repetitions are allowed. For example the term  $[b, o, b]$  in the atom  $noun([b, o, b])$  is a list term. Other examples are  $[red, green, blue]$  or  $[Sun \mid [Mon, Tue, Wed, Fri, Sat, Sun]]$ .

**Example 2.3.3** A list can be used to model a string of characters, as shown by the following facts:

$$\begin{aligned} &noun([b, o, b]). \\ &noun([t, h, o, m, a, s]). \\ &noun([n, a, t, a, n]). \end{aligned}$$

Given the facts above, the following rule:

$$palindromicNoun(X) :- noun(X), \#reverse(X, X).$$

allows to easily deduce nouns that are palindromic by using the external atom  $\#reverse$  that receives in input a list and returns another list that contains the same elements in reverse order. The program consisting of the rule above generates the following answer set:

$$\{palindromicNoun([b, o, b]), palindromicNoun([n, a, t, a, n])\}$$

**Example 2.3.4** Suppose that a system administrator wants to model the following security policy about file deletion: 'a certain subject  $S$  is permitted to delete a file  $F$  if at time  $T0$  he sends a request to target institution  $I$ , notifying the intention of deleting  $F$ , and there is no explicit request from  $I$  to  $S$  to retain  $F$  in the next ten time units'. The following rule can be used to naturally express this policy thanks to proper functional terms.

$$\begin{aligned} &permitted(S, delete(F), T1) :- request(S, I, notify(delete(F)), T0), \\ &notrequestInBetween(I, S, retain(F), T0, T1), T1 = T0 + 10. \end{aligned}$$

**Example 2.3.5** 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:

$$\begin{aligned} \text{path}([X, Y]) &:- \text{edge}(X, Y). \\ \text{path}([X|[Y|W]]) &:- \text{edge}(X, Y), L == [Y|W], \text{path}(L), \#member(X, L, false). \end{aligned}$$

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* 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.

**Example 2.3.6** Let us consider the famous "Towers of Hanoi" puzzle. Assume that a possible move is represented by a predicate *possibleMove*, featuring seven attributes: the first one represents the move number; the second, third, and fourth attributes represent the states of the three pegs before applying the current move; the fifth, sixth, and seventh attributes represent the last state of the three pegs after the move has been applied. We can encode a possible move from the first peg to the second peg by means of the following rule:

$$\begin{aligned} \text{possibleMove}(I1, [X|S1], S2, S3, S1, [X|S2], S3) &:- \text{possibleState}(I, [X|S1], S2, S3), \\ \text{legalMoveNumber}(I), \#succ(I, I1), \text{legalStack}([X|S2]) \end{aligned}$$

Roughly, the top element of the first peg can be moved on top of the second peg if: (i) the current peg is admissible, i.e. this state can be reached after applying a sequence of "I" moves (*possibleState*(*I*, [*X|S1*], *S2*, *S3*)); (ii) the number "I" is in the range of allowed move number (*legalMoveNumber*(*I*)); (iii) the new resulting configuration for the second stack is legal, i.e. there is no larger disc on top of the smaller one (*legalStack*([*X|S2*])).

# Chapter 3

## The DLV<sup>DB</sup> System

---

The DLP language is very expressive and allows for modeling complex combinatorial problems. However, despite the high expressiveness of this language, the success of DLP systems is still dimmed when the applications of interest become data intensive, since they work only in main memory.

The DLV<sup>DB</sup> system, described in this chapter, is an extension of the DLV system [42] allowing both the instantiation of logic programs directly on databases, and the handling of input and output data distributed on several databases in order to combine the expressive power of DLP with the efficient data management features of DBMSs.

This chapter is organized as follows. First, we introduce objectives and the architecture of the DLV<sup>DB</sup> system. Then, we focus on the auxiliary directives the user can specify to let DLV<sup>DB</sup> interact with external databases. Finally, we describe the evaluation strategy of DLP rules and their translation into SQL statements.

### 3.1 System objectives

The main limitations of current DLP systems in real world scenarios can be summarized in four main issues [62]: (i) they are not capable of handling data intensive applications, since they work in main memory only; (ii) they provide a limited interoperability with external DBMSs; (iii) they are not well suited for modelling inherently procedural problems; (iv) they cannot reason about recursive data structures and infinite domains, such as XML/HTML documents.

DLV<sup>DB</sup> has been conceived to increase the cooperation between ASP systems and databases. It allows substantial improvements in both the evaluation of logic programs and the management of input and output data distributed on several databases. Moreover, DLV<sup>DB</sup> presents enhanced features to improve its efficiency and usability in the contexts outlined above, for an effective exploitation of DLP in real world scenarios. These features include:

- Full support to disjunctive datalog with unstratified negation, and aggregate functions.
- An evaluation strategy devoted to carry out as much as possible of the reasoning tasks in mass memory, thus enabling complex reasonings in data intensive applications without degrading performances.
- Primitives to integrate data from different databases, in order to easily specify which data is to be considered as input or as output for the program.

In order to make the above features possible, various challenges had to be faced:

- Data intensive applications usually must access, and modify, data stored in autonomous enterprise databases and these should be accessed also by other applications.
- Evaluating the stable models of an ASP program directly in mass-memory data-structures, could be highly inefficient.
- Using the main memory to accommodate both the input data (hereafter, EDB) and the inferred data is usually impossible for data intensive applications due to the limited amount of available main memory.

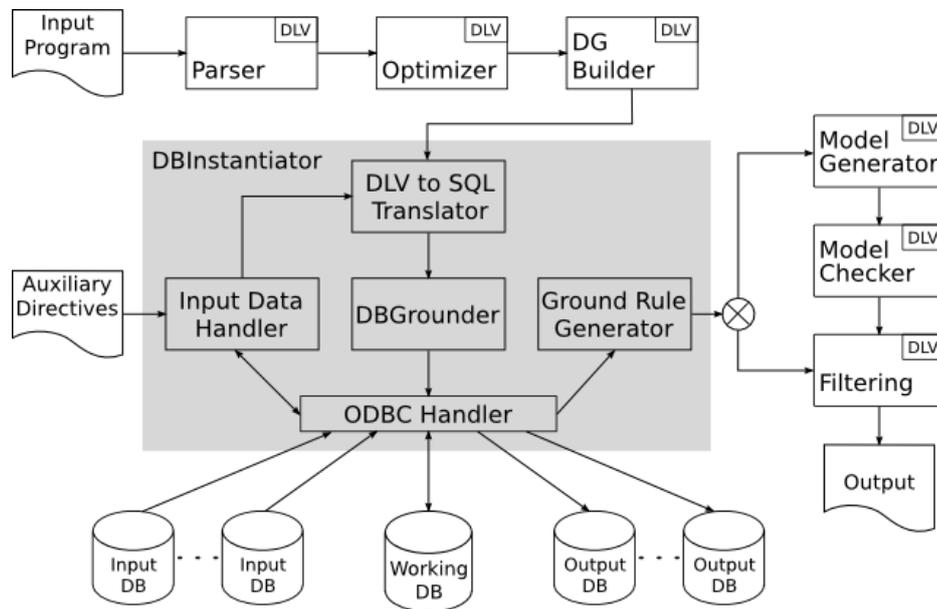
In order to face the first challenge, DLV<sup>DB</sup> is interfaced with external databases via ODBC. ODBC allows a very straightforward way to access and manipulate data over, possibly distributed, databases. Moreover, in order to properly carry out the evaluation of a program, it is necessary to specify the mappings between input and output data and program predicates, as well as to specify whether the temporary relations possibly needed for the mass-memory evaluation should be maintained or deleted at the end of the execution. These can be specified by some *Auxiliary Directives*, more details on these directives are discussed in Section 3.3. Note that the first challenge makes the adoption of deductive systems integrating proprietary DBMSs not effective.

To face with the second challenge, a mixed strategy is adopted, as outlined in Section 3.4. In brief, the evaluation is divided in two distinct phases: grounding and model generation. The grounding is completely performed in the database, while the model generation is carried out in main memory; this allows also to address the third challenge. In fact, in several cases, only a small portion of the ground program is actually needed for the model generation phase, since most of the inferred data is “stable”, i.e. it belongs to every stable model, and is already derived during the grounding phase.

## 3.2 System architecture

In this section we present the general architecture of the DLV<sup>DB</sup> system [41]. It has been designed as an extension of the DLV system [42], and combines the expressive power of DLV (and the optimization strategies implemented in it) with the efficient data management features of DBMSs [25]. Figure 3.1 illustrates the architecture of the DLV<sup>DB</sup> system. In the figure, the boxes marked with the DLV label represent the components already present in DLV.

An input program  $\mathcal{P}$  is first analyzed by the *Parser* which encodes the rules in the intensional database (IDB) in a suitable way and builds an extensional database (EDB) in main-memory data structures from the facts specified directly in the program (if any). As for facts already stored in database relations, no EDB is produced in main-memory. After this, the *Optimizer* applies a rewriting procedure in order to get a program  $\mathcal{P}'$ , equivalent to  $\mathcal{P}$ , that can be instantiated more efficiently and that can lead to a smaller ground program. The *Dependency Graph Builder* computes the dependency graph of  $\mathcal{P}'$ , its connected components and a topological ordering of these components. Finally, the *DB Instantiator* module, the core of the DLVDB system, is activated. The *DB Instantiator* module receives: (i) the IDB and the EDB (if not empty) generated by the parser; (ii) the Dependency Graph (DG) generated by the dependency graph

Figure 3.1: Architecture of DLV<sup>DB</sup>.

builder; *(iii)* the auxiliary directives specifying the needed interactions between DLV<sup>DB</sup> and the databases.

Within the *DB Instantiator*, each DLV rule is translated into SQL statement by the *DLV to SQL Translator*; then the SQL statements are executed on the *Working Database* by the *DB-Grounder*. The *Input Data Handler* receives the auxiliary directives defined by the user and performs all mappings and settings needed for a correct translation and execution of the SQL statements produced by the *DLV to SQL Translator*. Finally, the *Ground Rule Generator* extracts data from the database, creates ground rules (if any) and loads them to the *Model Generator*. The *Model Generator* produces some “candidate” answer sets (models), the stability of which are verified by the *Model Checker*; then each stable model is printed on output. Note that, if no ground rule is generated after grounding, it means that all the program has been solved by the grounder, then the *Ground Rule Generator* prints directly on output the (unique) stable model found.

Note that all the instantiation steps are performed directly on the working database through the execution of SQL statements and no data is loaded in main-memory from the databases in any phase of the grounding process. Communication with databases is performed via ODBC. This allows DLV<sup>DB</sup> both to be independent from a particular DBMS and to handle databases distributed over the Internet.

### 3.3 Auxiliary directives

Auxiliary directives are the way DLV<sup>DB</sup> allows the interaction between the system and a set of possibly distributed databases. Such directives must be expressed following the grammar represented in Figure 3.2.

As shown in the figure, the auxiliary directives can be subdivided in four sections, namely

```

Auxiliary-Directives ::=
  Init-section
  [Table-definition]+
  [Query-section]?
  [Final-section]*

Init-section ::=
  USEDB DatabaseName:UserName:Password [System-Like]?.

Table-definition ::=
  [USE TableName [( AttrName [, AttrName]* )]]?
  [AS ( SQL-Statement )]?
  [FROM DatabaseName:UserName:Password]?
  [MAPTO PredName [( SqlType [, SqlType]* )]]? ]??.
  |
  CREATE [VIEW]? TableName [( AttrName [, AttrName]* )]]?
  [MAPTO PredName [( SqlType [, SqlType]* )]]? ]?
  [KEEP_AFTER_EXECUTION]?.]

Query-section ::= QUERY TableName.

Final-section ::=
  [DBOUTPUT DatabaseName:UserName:Password.
  |
  OUTPUT [APPEND | OVERWRITE]? PredName [AS AliasName]?
  [IN DatabaseName:UserName:Password.]

System-Like ::=
  LIKE [POSTGRES | ORACLE | DB2 | SQLSERVER | MYSQL]

```

Figure 3.2: Grammar of the auxiliary directives.

the *init section*, the *table definition* section, the *query section* and the *final section*.

The *init section* defines the working database on which the instantiation process will be carried out. This database, like all the other ones, is accessed via an ODBC connection that must be previously set up and, consequently, username and password for the connection must be provided here. The `System-Like` option can be used here to specify the DBMS that implements the working database, in order to exploit the corresponding SQL dialect. If this option is omitted, a generic DBMS supporting standard SQL is assumed.

The *table definition* section specifies the mappings between logic program predicates and database tables (or SQL views). The user can specify two options in this phase, namely the `USE` and the `CREATE` option.

The `USE` option must be specified if the table `TableName` already exists and the data it contains must be taken as input (facts) for the instantiation process. Note that this data might reside on a database different from the working database (this situation can be specified with the option `[FROM DatabaseName]`). If the data is available, but it is not in a single table, `TableName` can be filled with the result of the SQL query specified in the option `[AS (SQL-Statement)]`.

The `CREATE` option must be used when the table must be created in the working database from the output computed by the program; if a table with the same name is present in the database it is first removed and then redefined. If the user specifies the option `KEEP_AFTER_EXECUTION`, the table is kept in the working database at the end of the instantiation process; otherwise it is removed.

The `VIEW` option of the `CREATE` statement can be used to specify that tables associated with intermediate predicates (i.e. corresponding to neither input nor output data) should be maintained virtual. This possibility ensure both space saving and performance improvement, especially for those programs that have a hierarchical structure. Note that the `VIEW` option cannot be used with predicates that are both recursive and unsolved (a definition of *unsolved predicate* is given in Section 3.4).

The `MAPTO` option can be used to link a table to a predicate. The number of attributes of the table must be equal to the arity of the associated predicate. In a table definition, attribute type declarations are needed only if the program contains built-in predicates or aggregate functions, in order to ensure their correct management.

If a predicate is not explicitly mapped into a table, two cases are possible: *i* there is a table in the working database with the same name of the predicate and compatible attributes; *ii* no corresponding table exists in the working database. In the first case, the system automatically creates a `USE` mapping between the predicate and its matching table, whereas in the second case, a `CREATE` mapping is automatically generated.

The *query section* can be used to specify the database table devoted to store the results of the query possibly present in the program.

The *final section* allows to copy either the entire output of the instantiation or only some of the tables on a database that is different from the working database. If the user does not specify any table name, the entire output is copied; otherwise, only the specified tables are copied. In the latter situation, the user can choose to append the data created during the instantiation to those already present in the target table (through the option `APPEND`) or to overwrite the (possibly existing) data in the target table (using the option `OVERWRITE`); if no option is specified, `OVERWRITE` is the default. The definition of the target table is the same as the source one except for the name which can be changed to `AliasName`. The output procedures can be carried out only if the system is forced to produce one model only.

**Example 3.3.1** Assume that a travel agency asks to derive all the destinations reachable by an airline company either by using its vectors or by exploiting code-share agreements. Suppose that the direct flights of each company are stored in a relation `flight_rel(Id, From, To, Company)` of the database `dbAirports`, whereas the code-share agreements between companies are stored in a relation `codeshare_rel(Company1, Company2, FlightId)` of an external database `dbCommercial`; if a code-share agreement holds between the company `c1` and the company `c2` for `flightId`, it means that the flight `flightId` is actually provided by a vector of `c1` but can be considered also carried out by `c2`. Finally, assume that, for security reasons, it is not allowed to travel agencies to directly access the databases `dbAirports` and `dbCommercial`, and, consequently, it is necessary to store the output result in a relation `composedCompanyRoutes` of a separate database `dbTravelAgency` supposed to support travel agencies. The program that can derive all the connections is:

- (1)  $destinations(From, To, Comp) :- flight(Id, From, To, Comp).$
- (2)  $destinations(From, To, Comp) :- flight(Id, From, To, C2),$   
 $codeshare(C2, Comp, Id).$
- (3)  $destinations(From, To, Comp) :- destinations(From, T2, Comp),$   
 $destinations(T2, To, Comp).$

In order to exploit data residing in the above mentioned databases, we should map the predicate *flight* with the relation `flight_rel` of `dbAirports` and the predicate *codeshare* with the relation `codeshare_rel` of `dbCommercial`. Finally, we have to map the predicate *destinations* with the relation `composedCompanyRoutes` of `dbTravelAgency`.

Now suppose that, due to a huge size of input data, we need to perform the program execution in mass-memory (on a DBMS). In order to carry out this task, the auxiliary directives shown in Figure 3.3 should be used. They allow to specify the mappings between the program predicates and the database relations introduced previously.

```

USEDDB dlvdb:myname:mypasswd.

USE flight_rel (Id, From, To, Company)
FROM dbAirports:airportUser:airportPasswd
MAPTO flight (integer, varchar(255), varchar(255), varchar(255)).

USE codeshare_rel (Company1, Company2, FlightId)
FROM dbCommercial:commUser:commPasswd
MAPTO codeshare (varchar(255), varchar(255), integer).

CREATE destinations_rel (From, To, Company)
MAPTO destinations (varchar(255), varchar(255), varchar(255))
KEEP_AFTER_EXECUTION.

OUTPUT destinations AS composedCompanyRoutes
IN dbTravelAgency:agencyName:agencyPasswd.

```

Figure 3.3: Auxiliary directives for Example 3.3.1.

### 3.4 Evaluation strategy

DLV<sup>DB</sup> instantiates DLP programs directly on a database. The instantiation process basically consists of two steps: (i) the translation of DLP rules in SQL statements, (ii) the definition of an efficient query plan based on a Semi-Naive evaluation.

The evaluation strategy is based on a sharp distinction existing between the grounding of the DLP program and the generation of its stable models. Then, two distinct approaches can be adopted whether the input program is non disjunctive and stratified (in this case everything can be evaluated on the DBMS) or not. In the following we describe how the evaluation is performed in the two cases.

### 3.4.1 Evaluation of non disjunctive stratified programs

It is well known that if a program is non disjunctive and stratified, it has a unique stable model corresponding exactly to its ground instantiation. The evaluation of these kinds of program consists in the translation of each rule into a corresponding SQL statement and in the composition of a suitable query plan on the DBMS; the evaluation of recursive rules is carried out using a semi-naïve approach.

Before starting the evaluation of a (possibly optimized) program  $\mathcal{P}$ , its connected components and their topological order (i.e., the Dependency Graph of  $\mathcal{P}$ ) are computed. Then, the program is evaluated one component at a time, starting from the lowest ones in the topological order. This process is iterated until no new ground instance can be derived from the component rules.

At each iteration, the instantiation of a rule consists of the execution of the SQL statement associated with it. One of the main objectives in the implementation of DLV<sup>DB</sup> has been that of associating one single (non recursive) SQL statement with each rule of the program (either recursive or not), without the support of main-memory data structures for the instantiation. This allows DLV<sup>DB</sup> to fully exploit optimization mechanisms implemented in the DBMSs and to minimize the “out of memory” problems caused by limited main-memory dimensions.

The translation of DLP to SQL will be addressed more in detail in Section 3.5.

### 3.4.2 Evaluation of disjunctive programs with unstratified negation

In presence of disjunctive rules or unstratified negation in a program  $\mathcal{P}$ , the ground instantiation of  $\mathcal{P}$  is not sufficient to compute its stable models. Then, grounding and model generation phases must both be carried out. Also in this case, the evaluation strategy adopted carries out the grounding completely in the database, through the execution of suitable SQL queries. This phase generates two kinds of data: ground atoms (facts) valid in every stable model (and thus not requiring further elaboration in the model generation phase) and ground rules, summarizing possible values for a predicate and the conditions under which these values can be inferred.

Facts compose the so called *solved* part of the program, whereas ground rules form the *residual program*, not completely solved by the grounding. As pointed out earlier, one of the main challenges in DLV<sup>DB</sup> is to load the smallest amount of information as possible in main memory; therefore, the residual program generated by the system should be as small as possible.

Model generation is then carried out in main memory with the technique described in [42].

**Definition 3.4.1** Let  $p$  be a predicate of a program  $\mathcal{P}$ ,  $p$  is said to be *unsolved* if at least one of the following holds: (i) it is in the head of a disjunctive rule; (ii) it is the head of at least one rule involved in unstratified negation; (iii) the body of a rule having  $p$  as head contains at least one unsolved predicate.  $p$  is said to be *solved* otherwise.

In the evaluation strategy, a ground predicate is associated with facts only in the ground program and, thus, with *certainly-true* values, i.e. values true in every stable model. On the contrary, an unsolved predicate  $p$  may be defined by both facts (certainly-true values) and ground rules; the latter identify *possibly-true* values for  $p$ , i.e. the domain of values  $p$  may assume in stable models.

Given an unsolved predicate  $p$  we indicate the set of its certainly-true values as  $p^s$  and the set of its possibly-true values as  $p^u$ .

**Example 3.4.2** Consider the following program:

$$\begin{aligned} & q(1, 2). \\ & p(3). \\ & p(X) \vee p(Y) :- q(X, Y). \end{aligned}$$

Here  $q$  is a solved predicate, whereas  $p$  is an unsolved predicate; in particular,  $q(1, 2)$  is a certainly-true value for  $q$ ,  $p(3)$  is a certainly-true value for  $p$ , whereas  $p(1)$  and  $p(2)$  are possibly-true values of  $p$ . Then,  $p^s = \{3\}$ , whereas  $p^u = \{1, 2\}$ .

As previously pointed out, rules having an unsolved predicate may generate ground rules in the instantiation. Since the goal is to generate the smallest residual program as possible, ground rules are “epurated” of certainly-true values.

**Definition 3.4.3** A *simplified ground rule* (g-rule in the following) of a program  $\mathcal{P}$  is a ground rule not involving any certainly-true values of  $\mathcal{P}$ .

**Example 3.4.4** From the previous example, the (only) ground rule that can be generated is  $p(1) \vee p(2) :- q(1, 2)$ . However this is not a simplified rule since it involves  $q(1, 2)$  which is a certainly-true value. Then, the corresponding g-rule is simply  $p(1) \vee p(2)$ .

It is now possible to illustrate the evaluation strategy implemented in the DLV<sup>DB</sup> system. Consider a program  $\mathcal{P}$  composed of non ground rules of the form:

$$\alpha_1 \vee \dots \vee \alpha_k :- \beta_1, \dots, \beta_n, \text{not } \beta_{n+1}, \dots, \text{not } \beta_m, \gamma_1, \dots, \gamma_p, \text{not } \gamma_{p+1}, \dots, \text{not } \gamma_q. \quad (3.1)$$

where  $\beta_i$  (resp.  $\gamma_j$ ) are solved (resp. unsolved) predicates. The evaluation is carried out in five steps:

**Step 1.** Translate  $\mathcal{P}$  in an equivalent program  $\mathcal{P}'$ ;

**Step 2.** Translate each rule of  $\mathcal{P}'$  in a corresponding SQL statement;

**Step 3.** Compose and execute the query plan of statements generated in Step 2 on the DBMS;

**Step 4.** Generate the residual program and load it in the Model Generator of DLV;

**Step 5.** Execute the residual program in main memory and show the results.

**Step 1.** The objective of this step is to “prepare” rules of  $\mathcal{P}$  to be translated in SQL almost straightforwardly, in order to generate a residual programs as small as possible. In more detail, for each rule  $r$  in  $\mathcal{P}$  three kinds of rule are generated:

- A. If the head of  $r$  has one atom only ( $k = 1$ ), a rule (hereafter denoted as A-rule) is created for deriving only certainly-true values of  $r$ 's head; note that if  $k > 1$  no certainly-true values can be derived from  $r$ .

- B. A set of rules (hereafter, B-rules) supporting the generation of the g-rules of  $r$ . The heads of these rules contain both the variables of unsolved predicates in the body of  $r$  and the variables in the head of  $r$ . Ground values obtained for these variables with B-rules are then used to instantiate  $r$  with possibly-true values only.
- C. A set of rules (hereafter, C-rules) for generating the set of possibly-true values of unsolved predicates as projections on B-rules obtained previously.

Given a generic rule defined as (3.1), the corresponding A-rule has the form:

$$\alpha_1^s :- \beta_1, \dots, \beta_n, \text{not } \beta_{n+1}, \dots, \text{not } \beta_m, \gamma_1^s, \dots, \gamma_p^s, \text{not } \gamma_{p+1}^s, \text{not } \gamma_{p+1}^u, \dots, \text{not } \gamma_q^s, \text{not } \gamma_q^u. \quad (3.2)$$

where for positive unsolved predicates only certainly-true values ( $\gamma_1^s, \dots, \gamma_p^s$ ) are considered, whereas for negated unsolved predicates both certainly-true and possibly-true values ( $\gamma_{p+1}^s, \dots, \gamma_q^s, \gamma_{p+1}^u, \dots, \gamma_q^u$ ) must be taken into account.

**Example 3.4.5** Consider the following program, which will be exploited as a running example throughout the rest of the section:

$$\begin{aligned} r1 : & \quad q(1, 2). \\ r2 : & \quad p(Y, X) \vee t(X) :- q(X, Y). \\ r3 : & \quad q(X, Y) :- p(X, Y), \text{not } t(X). \end{aligned}$$

Here both  $p$ ,  $q$ , and  $t$  are unsolved. The A-rules derived for this program are:

$$\begin{aligned} r1.A : & \quad q^s(1, 2). \\ r3.A : & \quad q^s(X, Y) :- p^s(X, Y), \text{not } t^s(X), \text{not } t^u(X). \end{aligned}$$

where rule  $r2$  does not contribute since it is disjunctive and cannot generate certainly-true values.

B-rules play a key role, since they allow the generation of the residual program. In particular, their role is to identify the set of values for variables in unsolved predicates of the body of  $r$ , generating possibly-true values of the head of  $r$ . Then,  $r$  is seen as a template for generating its g-rules, and ground values derived by the corresponding B-rules are used to instantiate  $r$ .

Note that in order to generate a possibly true value for a normal rule, at least one possibly true value must be involved in its body, whereas disjunctive rules always generate possibly-true values. Moreover, in order to properly generate g-rules (i.e. ground rules involving possibly-true values only) the system must be able to track, for each truth value of a B-rule, which predicates of  $r$  contributed with a certainly-true value and which ones with a possibly-true value.

This issue is addressed by first labeling each unsolved predicate  $\gamma_j$  of  $r$  alternatively with a 0 or with a 1, where a 0 indicates to take its  $\gamma_j^s$ , whereas a 1 indicates to consider its  $\gamma_j^u$ . Then, each binary number between 1 and  $2^q-1$  for normal rules and between 0 and  $2^q-1$  for disjunctive rules<sup>1</sup> corresponds to a labeling stating the combination of values to be considered. For each labeling, a corresponding B-rule is generated starting from the definition of  $r$  and substituting each unsolved predicate  $\gamma_j$  with  $\gamma_j^s$  (resp.,  $\gamma_j^u$ ) if the corresponding label is 0 (resp., 1).

The only exception is caused by negated unsolved predicates. In fact, if  $\gamma_j$  is negated and labeled with a 1, it must be put in the B-rule without negation. In fact, negated certainly-true

<sup>1</sup> Recall that  $q$  is the number of unsolved predicates in the body of  $r$ .

values surely invalidate the satisfiability of the g-rule, whereas negated possibly-true values may invalidate the rule only if the model generator sets them to true.

It is worth noticing that this labeling approach makes significantly easier the generation of simplified ground rules; in fact, it is sufficient to consider only the values of predicates labeled with 1 and not derived to be certainly-true by other rules.

Finally, in order to allow a proper reconstruction of g-rules from B-rules, a mapping between the variables of the B-rules and the variables of  $r$  is maintained.

**Example 3.4.6** From rules  $r2$  and  $r3$  introduced in the previous example, the following B-rules are derived. Original rules are re-proposed in parenthesis for the sake of comprehension; labels are reported in rule names. Variable mapping is trivial and not reported.

$$\begin{aligned}
 & (r2 : p(Y, X) \vee t(X) :- q(X, Y).) \\
 r2.B(0) : & B-rule_{r2}(X, Y) :- q^s(X, Y). \\
 r2.B(1) : & B-rule_{r2}(X, Y) :- q^u(X, Y). \\
 & (r3 : q(X, Y) :- p(X, Y), not t(X).) \\
 r3.B(01) : & B-rule_{r3}(X, Y) :- p^s(X, Y), t^u(X). \\
 r3.B(10) : & B-rule_{r3}(X, Y) :- p^u(X, Y), not t^s(X). \\
 r3.B(11) : & B-rule_{r3}(X, Y) :- p^u(X, Y), t^u(X).
 \end{aligned}$$

Finally, C-rules are simple projections on the B-rule heads over the attributes of the corresponding predicate.

**Example 3.4.7** From rules  $r2$  and  $r3$  introduced previously and from the corresponding B-rules the system generates:

$$\begin{aligned}
 r2.C_p : & p^u(Y, X) :- B-rule_{r2}(X, Y). \\
 r2.C_t : & t^u(X) :- B-rule_{r2}(X, Y). \\
 r3.C_q : & q^u(X, Y) :- B-rule_{r3}(X, Y).
 \end{aligned}$$

Note that in the example above, the same B-rule predicate (B-rule<sub>r2</sub>) is used to generate possibly-true values of two predicates ( $p^u$  and  $t^u$ ); this follows directly from the fact that  $r2$  is a disjunctive rule involving  $p$  and  $t$ .

**Step 2.** Translation of the rules obtained in Step 1 into SQL is carried out with the technique already presented in Section 3.5 and used also for non disjunctive and stratified programs. As an example, rule  $r3.A$  introduced above is translated into<sup>2</sup>:

```

INSERT INTO qs (SELECT ps.att1, ps.att2, FROM ps
WHERE ps.att1 NOT IN (SELECT * FROM ts)
AND ps.att1 NOT IN (SELECT * FROM tu))

```

**Step 3.** In order to compile the query plan, the dependency graph  $D$  associated with  $\mathcal{P}$  is considered [43]. In particular,  $D$  allows the identification of a partially ordered set  $\{\text{Comp}_i\}$  of program components where lower components must be evaluated first.

<sup>2</sup>Here and in the following we use the notation  $x.att_i$  to indicate the  $i$ -th attribute of the table  $x$ . Actual attribute names are determined at runtime.

Then, given a component *Comp* and a rule *r* in *Comp*, if *r* is not recursive, then the corresponding portion of query plan is as follows<sup>3</sup>: (1) evaluate (if present) the A-rule associated with *r*; (2) evaluate each B-rule obtained from *r*; (3) for each predicate in the head of *r* evaluate the corresponding C-rule.

If *r* is recursive, the portion of query plan above must be included in a fix-point semi-naïve evaluation, as described in [65].

**Step 4 and 5.** The generation of the residual program requires the analysis of values derived by B-rules only. Then, for each rule *r* and each corresponding B-rule (say, *r*.B(L)), first predicates having label 0 in L are purged from *r*, then *r* is instantiated with values of *r*.B(L). During this phase a further check is carried out to verify if some predicate value has been derived as certainly-true by other rules. In this case the predicate is removed from the g-rule for that instance. The residual program is then loaded in main memory for the generation of stable models. Note that each answer set found on this residual program shall be enriched with certainly-true values determined during the grounding.

**Example 3.4.8** The residual program generated for our running example is:

$$\begin{aligned} p(2, 1) \vee t(1). \\ p(1, 2) \vee t(2) :- q(2, 1). \\ q(2, 1) :- p(2, 1), \text{not } t(2). \end{aligned}$$

Note that the first g-rule does not involve *q* since it derives from *r*.B(0), having *q*(1, 2) as certainly-true value.

### 3.5 Translation from DLP to SQL

As pointed out in the previous section, the evaluation strategy in DLV<sup>DB</sup> is based on the translation of each DLP rule (recursive or not) into a single non recursive SQL statement. It is worth recalling that the system maps each predicate of the input program into a database table. This section describes the functions used to perform the translations. The functions are presented in pseudocode and, for the sake of presentation clarity, they omit some details; moreover, since there is a one-to-one correspondence between the predicates in the logic program and the tables in the database, in the following, when this is not confusing, we use the terms predicate and table interchangeably.

In order to provide examples for the presented functions, we use the following reference schema:

$$\begin{aligned} \textit{employee}(\textit{Ename}, \textit{Salary}, \textit{Dep}, \textit{Boss}) \\ \textit{department}(\textit{Code}, \textit{Director}) \end{aligned}$$

storing information about the employees of the departments of a given company. Specifically, each employee has associated a *Boss* who is, in her turn, an employee.

#### 3.5.1 Translating non-recursive rules

The translation of non-recursive rules is performed by the *TranslateNonRecursiveRule* function, which is shown in Figure 3.4.

<sup>3</sup>Here, for the sake of simplicity, we refer to rules, indicating that the corresponding SQL statements must be evaluated in the database.

---

```

Function TranslateNonRecursiveRule( $r$ : DLPA rule): SQL statement
begin
   $SQL := ""$ ;
  if (hasAggregate( $r$ )) then
     $SQL := SQL + \text{TranslateAggregateRule}(r)$ ;
   $SQL := SQL + \text{"INSERT INTO " + head}(r) + \text{"("}$ ;
  if (isPositive( $r$ )) then
     $SQL := SQL + \text{TranslatePositiveRule}(r)$ ;
  else if (hasNegation( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithNegation}(r)$ ;
  else if (hasBuilt-In( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithBuilt-In}(r)$ ;
  else if (hasNegationAndBuilt-In( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithNegationAndBuilt-In}(r)$ ;
   $SQL := SQL + \text{"})"$ ;
  return  $SQL$ ;
end.

```

---

Figure 3.4: Function TranslateNonRecursiveRule

---

```

Function TranslatePositiveRule( $r$ : DLPA rule): SQL statement
begin
   $SQL := \text{"SELECT " + head.attr}(r) +$ 
     $\text{"FROM " + body}^+(r) +$ 
     $\text{"WHERE " + joinConditions}(r) +$ 
     $\text{"AND " + bodyConstantConditions}(r) +$ 
     $\text{"EXCEPT (SELECT * FROM " + head}(r) + \text{"})"$ ;
  return  $SQL$ ;
end.

```

---

Figure 3.5: Function TranslatePositiveRule

The function receives a rule  $r$  as input and returns the corresponding SQL statement, depending on the rule typology. Function *head* receives the rule  $r$  and returns the table associated with the atom in its head; this task is carried out by considering the mappings specified in the auxiliary directives. Function *isPositive*( $r$ ) (resp., *hasNegation*( $r$ ), *hasBuilt-In*( $r$ ), *hasNegationAndBuilt-In*( $r$ ), *hasAggregate*( $r$ )) receives a rule  $r$  and returns *true* if  $r$  is a positive rule (resp., contains negated atoms, contains built-in functions, contains both negated atoms and built-in functions, contains aggregate functions), *false* otherwise. In the following we describe in detail the translation functions used in function *TranslateNonRecursiveRule*<sup>4</sup>.

### Translating Positive Rules

The translation of a positive rule into an SQL statement is performed by function *TranslatePositiveRule*, as shown in Figure 3.5. The SELECT part of the statement is determined by the variable bindings between the head and the body of the rule. The FROM part is determined by the predicates composing the body of the rule. Variable bindings between body atoms and constants determine the WHERE conditions of the statement. Finally, an EXCEPT part is added in order to eliminate tuple duplications.

---

<sup>4</sup>*TranslateRuleWithNegationAndBuilt-In* will be not described since it is a straightforward fusion of *TranslateRuleWithNegation* and *TranslateRuleWithBuilt-In*.

---

```

Function TranslateRuleWithNegation(r: DLPA rule): SQL statement
begin
  SQL := "SELECT " + head_attr(r) +
    "FROM " + body+(r) +
    "WHERE " + joinConditions(r) +
    "AND " + bodyConstantConditions(r);
  for each p in body-(r)
    SQL := SQL + "AND " + negativeAttr(r,p) +
      "NOT IN (SELECT * FROM " + p + ")";
  SQL := SQL + "EXCEPT (SELECT * FROM " + head(r) + ")";
  return SQL;
end.

```

---

Figure 3.6: Function TranslateRuleWithNegation

Function *head\_attr(r)* returns the list of attributes of the atoms (associated to tables) in the body of *r* specified also in the head of *r*. The function returns this list in the proper order and also handles possible constant values specified in the rule head. Function *body<sup>+</sup>(r)* returns the list of tables corresponding to the (positive) atoms present in the body of *r*. Function *joinConditions(r)* derives the join conditions to be specified among the involved tables from the positions and the names of the variables specified in the body of *r*, whereas function *bodyConstantConditions(r)* handles possible constants specified in the body of *r*.

**Example 3.5.1** Consider the following query:

$$q_0(Ename) :- employee(Ename, 100.000, Dep, Boss), department(Dep, rossi).$$

which returns all the employees working at the department whose chief is *rossi* and having a yearly salary of 100.000 euros. The corresponding SQL statement is the following, where the notation *t.att<sub>i</sub>* is used to indicate the *i<sup>th</sup>* attribute of table *t*:

```

INSERT INTO q0 (
  SELECT employee.att1
  FROM employee, department
  WHERE employee.att3 = department.att1
    AND department.att2='rossi'
    AND employee.att2=100.000
  EXCEPT
    (SELECT * FROM q0))

```

### Translating rules with negated atoms

Rules with negated atoms are translated into an SQL statement by function *TranslateRuleWithNegation*, which is shown in Figure 3.6.

The construction of the SQL statement is carried out as follows: the positive part of the rule is handled in the same way as function *TranslatePositiveRule*; then, each negated atom is handled by a corresponding NOT IN part in the statement. Function *body<sup>-</sup>(r)* returns the tables corresponding to the negated atoms in the body of *r*, while function *negativeAttr(r,p)* singles out those attributes of positive atoms in *r* bound to attributes of the negated atom *p*.

**Example 3.5.2** The following program computes (using the goal *topEmployee*) the employees who have no other boss than the director.

$$\begin{aligned} \text{topEmployee}(Ename) & \quad :- \quad \text{employee}(Ename, Salary, Dep, Boss), \\ & \quad \text{department}(Dep, Boss), \\ & \quad \text{not otherBoss}(Ename, Boss). \\ \text{otherBoss}(Ename, Boss) & \quad :- \quad \text{employee}(Ename, Salary, Dep, Boss), \\ & \quad \text{employee}(Boss, Salary, Dep, Boss1). \end{aligned}$$

The first rule above is translated into the following SQL statement:

```
INSERT INTO topEmployee (
  SELECT employee.att1
  FROM employee, department
  WHERE (employee.att3=department.att1)
        AND (employee.att4=department.att2)
        AND (employee.att1, employee.att4)
        NOT IN (SELECT otherBoss.att1, otherBoss.att2 FROM otherBoss )
  EXCEPT
  (SELECT * FROM topEmployee))
```

### Translating rules with built-in predicates

The language defined in Section 2.1 defines some built-in predicates, such as comparative and arithmetic predicates. When running a program containing built-in predicates, the range of admissible integer values must be fixed. We fulfill this requirement in the working database by adding a restriction based on the maximum value allowed for integer variables (referred to as #maxint). Moreover, in order to allow mathematical operations among attributes, DLV<sup>DB</sup> requires the types of attributes to be properly defined in the database.

The function for translating rules containing built-in predicates is a trivial variation of the function for translating positive rules and, consequently, it will not be shown here. As a matter of facts, the presence of a built-in predicate in the rule implies just to add a corresponding condition in the WHERE part of the SQL statement.

**Example 3.5.3** The program:

$$q_1(Ename) : -\text{employee}(Ename, Salary, Dep, Boss), Salary > 100.000$$

is translated into the following SQL statement:

```
INSERT INTO q1 (
  SELECT employee.att1
  FROM employee
  WHERE employee.att2 > 100.000 )
```

If the variables specified in the built-in atoms are not bound to any other variable of the atoms in the body, a #maxint value is used to bind that variable to its admissible range of values.

### Translating rules with aggregate atoms

As shown in Section 2.1, specific safety conditions must hold for each rule containing aggregate atoms, in order to guarantee the computability of the corresponding rule. As an example, aggregate atoms can not contain predicates mutually recursive with the head of the rule they are placed in. This implies that the truth values of each aggregate function can be computed once and for all before evaluating the corresponding rule, which can be, in its turn, recursive.

The optimization process that rewrites input programs before their execution, automatically rewrites each rule containing some aggregate atom in such a way that it follows a standard format. Specifically, given a generic rule of the form:

$$head \quad :- \quad body, f(\{Vars : Conj\}) \prec Rg.$$

where  $Conj$  is a generic conjunction and  $Rg$  is a guard, the system automatically translates this rule into a pair of rules of the form

$$\begin{aligned} auxAtom & \quad :- \quad Conj, BindingAtoms. \\ head & \quad \quad :- \quad body, f(\{Vars : auxAtom\}) \prec Rg. \end{aligned}$$

where  $auxAtom$  is a standard rule containing both  $Conj$  and the atoms ( $BindingAtoms$ ) necessary for the bindings of  $Conj$  with  $body$  and/or  $head$ . Note that  $auxAtom$  contains only those attributes of  $Conj$  that are strictly necessary for the computation of  $f$  and, consequently, it may have far less (and can not have more) attributes than those present in  $Conj$ .

We rely on this standardization to translate this kind of rules into SQL. Clearly only the second rule, containing the aggregate function, is handled by the function presented in the following; in fact, the first rule is automatically translated by one of the functions presented previously.

The objective of the translation is to create an SQL view  $auxAtom\_supp$  from  $auxAtom$  which contains all the attributes necessary to bind  $auxAtom$  with the other atoms of the original rule, and a column storing the results of the computation of  $f$  over  $auxAtom$ . The original aggregate atom is then replaced by this view, and guard conditions are suitably translated by logic conditions between variables. At this point, the resulting rule is a standard rule not containing aggregate functions and can be then translated by one of the functions previously presented. The translation function,  $TranslateAggregateRule$ , is shown in Figure 3.7.

Function  $aggr\_atom(r)$  returns the aggregate atoms present in  $r$ ;  $aux\_atom(a)$  returns the auxiliary atom corresponding to  $Conj$  of  $a$  and automatically generated by the optimizer. Function  $bound\_attr(a)$  yields in output the attributes of the atom  $a$  bound with attributes of the other atoms in the rule, while  $aggr\_attr(a)$  returns the attribute which the aggregation must be carried out onto (the first variable in  $Vars$ ).  $aggr\_func(a)$  returns the SQL aggregation statement corresponding to the aggregate function of  $a$ . Function  $removeFromBody(r, a)$  (resp.,  $addToBody(r, a)$ ) removes (resp., adds) the atom  $a$  from (resp., to) the rule  $r$ . Finally,  $aux\_atom\_supp(a)$  yields in output the name of the atom corresponding to the just created auxiliary view, whereas  $guards(a)$  converts the guard of  $a$  in a logic statement between attributes in the rule.

**Example 3.5.4** Consider the following rule computing the departments which spend for the salaries of their employees, an amount greater than a certain threshold, say 100000:

$$\begin{aligned} costlyDep(Dept) & \quad :- \quad department(Dept, -), \\ & \quad \quad \quad \#sum\{Salary, Ename : employee(Ename, Salary, Dept, -)\} > \\ & \quad \quad \quad 100000. \end{aligned}$$

---

```

Function TranslateAggregateRule(VAR r: DLPA rule): SQL statement
begin
  for each a in aggr_atom(r) do begin
    aux:=aux_atom(a);
    SQL:="CREATE VIEW " + aux + "_supp" +
      "AS (SELECT " + bound_attr(a) + ", " +
        aggr_func(a) + "(" + aggr_attr(a) + ")" +
      "FROM " + aux
      "GROUP BY " + bound_attr(a) + ")";
    removeFromBody(r, a);
    addToBody(r, aux_atom_supp(a));
    addToBody(r, guards(a));
  end;
return SQL;
end.

```

---

Figure 3.7: Function TranslateAggregateRule

The optimizer automatically rewrites this rule as follows:

$$\begin{aligned}
 aux\_emp(Salary, Ename, Dep) &:- department(Dep, -), \\
 &\quad employee(Ename, Salary, Dep, -). \\
 costlyDep(Dep) &:- department(Dep, -), \\
 &\quad \#sum\{Salary, Ename : aux\_emp(Salary, Ename, Dep)\} > 100000.
 \end{aligned}$$

The first rule is treated as a standard positive rule and is translated into:

```

INSERT INTO aux_emp (
  SELECT employee.att2, employee.att1, department.att1
  FROM department, employee
  WHERE department.att1 = employee.att3
  EXCEPT
  (SELECT * FROM aux_emp))

```

The second rule is translated into:

```

CREATE VIEW aux_emp_supp AS (
  SELECT aux_emp.att3, SUM(aux_emp.att1)
  FROM aux_emp
  GROUP BY aux_emp.att3)

INSERT INTO costlyDep (
  SELECT department.att1
  FROM department, aux_emp_supp
  WHERE department.att1 = aux_emp_supp.att1
  AND aux_emp_supp.att2 > 100000
  EXCEPT
  (SELECT * FROM costlyDep))

```

---

```

Function TranslateRecursiveRule( $r$ : DLPA rule): SQL statement
begin
  SQL:= "";
  if(hasAggregate( $r$ )) then
    SQL:=TranslateAggregateRule( $r$ );
   $n := 2^{RecursivePredicates(r)}$ 
  SQL:=SQL+"INSERT INTO " +  $\Delta head(r)$  + "(";
  for  $i:=1$  to  $n$  do begin
    Let  $r'$  be a rule;
    setHead( $r'$ ,  $\Delta head(r)$ );
    for each non recursive predicate  $q_j$  in body( $r$ ) do
      addToBody( $r'$ ,  $q_j$ );
    for each recursive predicate  $p_j$  in body( $r$ ) do
      if ( $bit(j,i)=0$ ) then addToBody( $r'$ ,  $p_j^{k-2}$ );
      else addToBody( $r'$ ,  $\Delta p_j^{k-1}$ );
    if ( $i \neq 1$ ) SQL:=SQL+"UNION ";
    SQL:=SQL + TranslateNonRecursiveRule( $r'$ );
  end;
  SQL:=SQL + ")";
  return SQL;
end.

```

---

Figure 3.8: Function TranslateRecursiveRule

### 3.5.2 Translating recursive rules

Recursive rules are translated into non recursive SQL statements operating alternatively on standard and differential versions of the tables associated with recursive predicates. Each time one of such statements is executed by the algorithm, it must compute just the new values for the predicate in the head that can be obtained from the values computed in the last two iterations of the fixpoint.

The translation algorithm *TranslateRecursiveRule*, shown in Figure 3.8, first selects the proper combinations of standard and differential relations from the rule  $r$  under consideration; then, for each of these combinations, rewrites  $r$  into a corresponding rule  $r'$ . Each  $r'$  is non recursive and, consequently, it can be handled by Function *TranslateNonRecursiveRule* defined above.

Functions *hasAggregate*, *TranslateAggregateRule* and *TranslateNonRecursiveRule* have been introduced previously. Function *RecursivePredicates*( $r$ ) returns the number of occurrences of recursive predicates in the body of  $r$ ;  $\Delta head(r)$  returns the differential version of the relation corresponding to the head of  $r$ . Function *setHead*( $r'$ ,  $p$ ) sets the head of the rule  $r'$  to the predicate  $p$ ; analogously, function *addToBody*( $r'$ ,  $p$ ) adds to the body of  $r'$  a conjunction with the predicate  $p$ . Finally, function *bit*( $j,i$ ) returns the  $j$ -th bit of the binary representation of  $i$ .

**Example 3.5.5** Consider the situation in which we need to know whether the employee  $e_1$  is the boss of the employee  $e_n$  either directly or by means of a number of employees  $e_2, \dots, e_n$  such that  $e_1$  is the boss of  $e_2$ ,  $e_2$  is the boss of  $e_3$ , etc. Then, we have to evaluate the program:

$$\begin{aligned}
 q_2(E_1, E_2) & :- employee(E_1, Salary, Dep, E_2). \\
 q_2(E_1, E_3) & :- q_2(E_1, E_2), q_2(E_2, E_3).
 \end{aligned}$$

containing a recursive rule. This program cannot be evaluated in one single iteration of the Semi-Naive computation. In fact, the SQL statement corresponding to the recursive rule must be

executed until no new values can be derived for  $q_2$ . The SQL statement obtained by Function TranslateRecursiveRule for the second rule of this example is:

```

INSERT INTO  $\Delta q_2^k$  (
  SELECT  $q_2^{k-2}.att_1, \Delta q_2^{k-1}.att_2$ 
  FROM  $q_2^{k-2}, \Delta q_2^{k-1}$ 
  WHERE ( $q_2^{k-1}.att_2 = \Delta q_2^{k-1}.att_1$ )
  EXCEPT (SELECT * FROM  $\Delta q_2^k$ )
  UNION
  SELECT  $\Delta q_2^{k-1}.att_1, q_2^{k-2}.att_2$ 
  FROM  $\Delta q_2^{k-1}, q_2^{k-2}$ 
  WHERE ( $\Delta q_2^{k-1}.att_2 = q_2^{k-1}.att_1$ )
  EXCEPT (SELECT * FROM  $\Delta q_2^k$ )
  UNION
  SELECT  $\Delta q_2^{k-1}.att_1, \Delta q_2^{k-1}_1.att_2$ 
  FROM  $\Delta q_2^{k-1}, \Delta q_2^{k-1}$  AS  $\Delta q_2^{k-1}_1$ 
  WHERE ( $\Delta q_2^{k-1}.att_2 = \Delta q_2^{k-1}_1.att_1$ )
  EXCEPT (SELECT * FROM  $\Delta q_2^k$ ))

```

Actually, the real implementation of this function adds, for performance reasons, also the following parts to the statement above:

```

EXCEPT (SELECT * FROM  $\Delta q^{k-1}$ )
EXCEPT (SELECT * FROM  $q^{k-2}$ )

```

# Chapter 4

## Related Work

---

This chapter analyzes and compares the most recent ASP systems [44, 17] that support unstratified negation and other advanced constructs like disjunction, and various forms of constraints. Such systems are often referred to as *declarative* computational logic systems, since they support a fully declarative programming style, in contrast to Prolog in which the result depends on the ordering of the rules in the program, and also on the ordering of the goals in the bodies of the rules.

The ASP systems analyzed in this chapter are compared on the basis of two main aspects: (i) expressiveness of the supported language, and (ii) efficiency to answer a query. The first aspect concerns the kinds of features supported by the language, such as the ability to express views, recursive rules, integrity constraints, and nonmonotonic queries. The second aspects can be characterized on the basis of two main parameters: (i) the quantity of data to be analyzed for answering a query, and (ii) the intrinsic complexity of the query itself.

### 4.1 Relevant features of ASP systems

This section introduces the main features of ASP systems that will be used for the analysis and comparison presented in Section 4.2.

Since one of the main features of  $DLV^{DB}$  is answering queries involving several data sources rather than one single database, this is one of the aspects we focused our analysis upon. As an example, the set of information of interest for a user might be distributed over a network. It is necessary to bring data distributed over a network to a user's machine so that the data may be manipulated to answer user queries. In a distributed environment, it is likely that one will want to save answers to queries in the local machine's cached database for answers, rather than to have to access data over the network to answer the query. In this situation the resources are the cached relations (i.e. views on the source data) and the use of these resources is an important aspect of query answering.

In some systems the database relations are views themselves (either virtual or materialized) on other resources; thus, it is necessary to follow chains of views in order to reach the original source data. In this context, the expressiveness of the language supported by the ASP system plays a central role.

Another important feature of an ASP system is the availability of constructs in the query language allowing to express complex queries. Indeed, the possibility to express computationally

hard queries is a crucial property that ASP systems should provide in order to deal with complex scenarios. We recall that the ASP language (function-free disjunctive logic programming) is very expressive, and allows to represent even problems of high complexity (every problem in the complexity class  $\Sigma_2^P = NP^{NP}$  [18]). However, some ASP systems do not allow to represent problems belonging to this complexity class, while other systems allow to represent even more complex problems. Therefore, this is one of the features that will be considered to compare the ASP systems analyzed in the remainder of this chapter.

Finally, we point out that answering queries is often a hard task. Hardness may arise either because of the huge amounts of data to be analyzed or because of the complexity of the queries to be processed. The efficiency of the ASP system in answering queries is an important measure for its characterization.

Therefore, the ASP systems will be analyzed and compared on the basis of two main aspects: (i) language expressiveness and (ii) optimizations in the evaluation. In the remainder of this section, we specify in more detail the features that will be considered for each of the above aspects.

#### 4.1.1 Language expressiveness

To characterize the expressiveness of logic languages of ASP systems, we consider the following properties that a logic language should provide:

- *Ability to express views.* As outlined above, views play a relevant role because in some systems the database relations are views themselves on other resources.
- *Ability to express recursive queries.* Recursion allows to model easily even complex problems. Therefore, the ability to express recursive queries is a fundamental property to assess the expressiveness of a logic language.
- *Ability to express integrity constraints.* When querying data from different sources, differences and inconsistencies of the data must be taken into account. The ability to express integrity constraints allows to avoid most cases of data inconsistencies.
- *Ability to express nonmonotonic queries.* This property is necessary in order to define appropriate methods dealing with complex scenarios involving incomplete or inconsistent data sources, or in presence of dynamic knowledge.
- *Ability to deal with computationally hard queries.* As stated above, the possibility to express computationally hard queries is a crucial property that ASP systems should provide in order to deal with complex scenarios.

#### 4.1.2 Optimizations

The efficiency of an ASP system can be measured by analyzing its optimization strategies in query answering. In order to analyze the optimization strategies adopted by the considered systems, we consider their behavior in answering two particular classes of queries:

- simple queries (i.e., queries that can be answered in polynomial time);
- complex queries (i.e., whose complexity goes beyond polynomial-time).

Table 4.1 summarizes the expressiveness and optimization features described above. The table will be used as a template for the analysis of the systems that will be described in the remainder of the chapter.

<b>Language expressiveness</b>		
Ability to express views		
Ability to express recursive queries		
Ability to express integrity constraints		
Ability to express nonmonotonic queries		
Ability to deal with computationally hard queries		
Other		
<b>Optimization</b>		
Optimizations in simple but data intensive queries		
Optimizations in computationally hard queries		

Table 4.1: Template table for describing systems' features

## 4.2 ASP systems: analysis and comparison

ASP is a declarative approach to programming, alternative to SAT-based programming, which is successful and widely used in the area of Artificial Intelligence [37]. In SAT-based programming, a given computational problem  $P$  is encoded as a propositional CNF formula whose models correspond to solutions of  $P$ . A SAT solver is then used to find such models (and thus solutions of  $P$ ). In ASP, on the contrary, a problem  $P$  is represented by a program whose answer sets correspond to solutions; an ASP system is then used to find such solutions [44].

The main advantage of ASP over SAT-based programming is its higher language expressiveness, which enjoys the knowledge modeling power of logic programming features like variables, negation as failure, and disjunction. Indeed, the knowledge representation language of ASP consists of function-free logic programs with classical negation where disjunction is allowed in the heads and negation as failure may occur in the bodies of the rules.

As mentioned earlier, the ASP language supports the representation of problems of high computational complexity (specifically, all problems in the complexity class  $\Sigma_2^P = \text{NP}^{\text{NP}}$ ). Furthermore, the ASP encoding of a large variety of problems is often very concise, simple, and elegant [17].

In the remainder of this section we analyzed several ASP systems: DLV, SMOBELS, Cmodels, ASSAT, noMoRe, SLG, DeReS, XSB and claspD.

### 4.2.1 DLV

The development of the DLV system (datalog plus vel, i.e., disjunction) [6, 12, 14] has started as a research project funded by FWF (the Austrian Science Funds) in 1996, and has evolved into an international collaboration over the years. Currently, the University of Calabria and TU Wien

participate in the project, supported by a scientific-technological collaboration between Italy and Austria.

The system is based on disjunctive logic programming without function symbols under the consistent answer set semantics [30] and has the following important features:

**Advanced Knowledge Modeling Capabilities.** DLV provides support for declarative problem solving in several respects:

- High expressiveness in a formally precise sense ( $\Sigma_2^P$ ), so any such problem can be uniformly solved by a fixed program over varying input.
- Declarative problem solving following a “Guess/Check/Optimize ” paradigm where a solution to a problem is guessed by one part of a program and then verified through another part of the program.
- Capability to express hard and weak constraints.
- A number of front-ends for dealing with specific AI applications.

**Solid Implementation.** Much effort has been spent on sophisticated algorithms and techniques for improving the performance, including

- database optimization techniques [14, 19], and
- non-monotonic reasoning optimization techniques.

**Database Interfaces.** The DLV system provides an experimental interface to an object-oriented database management system (Objectivity), by means of a special query tool, which is useful for the integration of specific problem solvers developed in DLV into more complex systems.

The architecture of DLV is illustrated in Figure 4.1. The general flow in this picture is top-down. The principal User Interface is command-line oriented, but also a Graphical User Interface (GUI) for the core systems and most front-ends is available. Subsequently, front-end transformations might be performed. Input data can be supplied by regular files, and also by Objectivity databases. The DLV kernel (the shaded part in the figure) then produces answer sets one at a time, and each time an answer set is found, “Filtering” is invoked, which performs post-processing (dependent on the active front-ends) and controls continuation or abortion of the computation.

The DLV kernel consists of three major components: the “Intelligent Grounding,” “Model Generator,” and “Model Checker” modules share a principal data structure, the “Ground Program”. It is created by the Intelligent Grounding using differential (and other advanced) database techniques together with suitable data structures, and used by the Model Generator and the Model Checker. The Ground Program is guaranteed to have exactly the same answer sets as the original program. For some syntactically restricted classes of programs (e.g. stratified programs), the Intelligent Grounding module already computes the corresponding answer sets.

For harder problems, most of the computation is performed by the Model Generator and the Model Checker. Roughly, the former produces some “candidate” answer sets (models) [20, 21], the stability of which are subsequently verified by the latter.

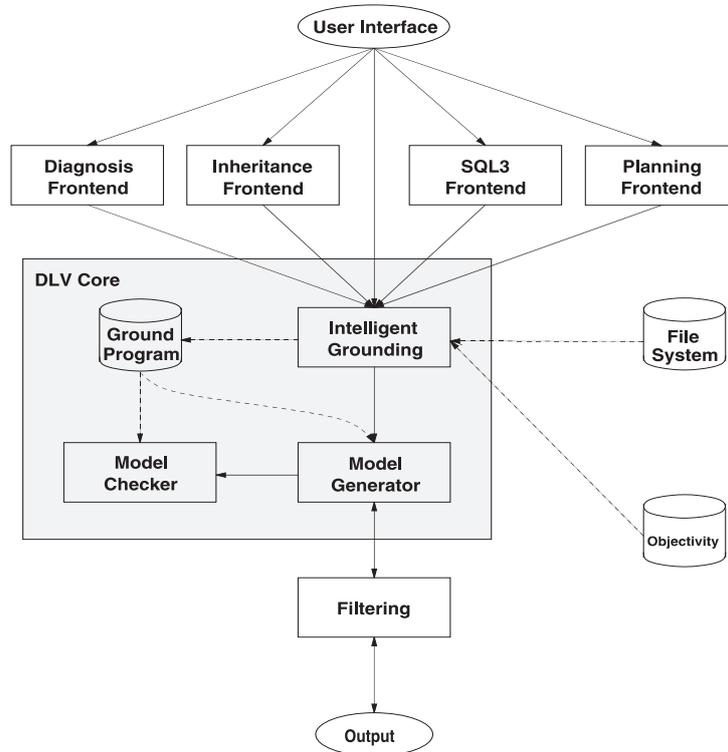


Figure 4.1: The System Architecture of DLV

The Model Checker (MC) verifies whether the model at hand is an answer set. This task is very hard in general, because checking the stability of a model is known to be co-NP-complete. However, MC exploits the fact that minimal model checking — the hardest part — can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs.

In Table 4.2 we summarize the features of the DLV system w.r.t. the properties we have pointed out in Section 4.1.

## 4.2.2 SMOBELS

The SMOBELS system [54, 53] implements the answer set semantics for normal logic programs extended by built-in functions as well as cardinality and weight constraints for domain-restricted programs.

As input, the SMOBELS system takes logic program rules basically in Prolog style syntax. However, in order to support efficient implementation techniques and extensions the programs are required to be *domain-restricted* where the idea is the following. The predicate symbols in the program are divided into two classes, *domain predicates* and *non-domain predicates*. Domain predicates are predicates that are defined non-recursively.

The main intuition of domain predicates is that they are used to define the set of terms over which the variable range in each rule of a program  $P$ . All rules of  $P$  have to be domain-restricted in the sense that every variable in a rule must appear in a domain predicate which appears positively in the rule body.

DLV	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes, both hard and weak constraints (desiderata) are expressible
Ability to express nonmonotonic queries	Yes, supports unstratified negation and disjunction
Ability to deal with computationally hard queries	Expresses very hard queries, up to $\Delta_3^P$
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	Some database optimization techniques are incorporated, but magic sets are not implemented
Optimizations in computationally hard queries	Heuristics and optimization techniques from the field of SAT programming have been implemented

Table 4.2: Features of DLV

In addition to normal logic program rules, SMOELS supports rules with cardinality and weight constraints. The idea is that, e.g., a cardinality constraint

$$1 \{a,b,\text{not } c\} 2$$

holds in an answer set if at least 1 but at most 2 of the literals in the constraint are satisfied in the model and a weight constraint

$$10 [ a=10,b=10,\text{not } c=10 ] 20$$

holds if the sum of weights of the literals satisfied in the model is between 10 and 20 (inclusive). With built-in functions for integer arithmetic (included in the system), these kind of rules allow compact and fairly straightforward encodings of many interesting problems.

Answer sets of a domain-restricted logic program with variables are computed in three stages:

- First the program is transformed into a ground program without variables.
- Second, the rules of the ground program are translated into primitive rules,

- Third, an answer set is computed using a Davis-Putnam like procedure [60].

The first two stages have been implemented in a program called `lparse`, which functions as a front end to `smodels` which in turn implements the third stage.

In the first stage `lparse` automatically determines the domain predicates and then using database techniques evaluates the domain predicates and creates a ground program which has exactly the same answer sets as the original program with variables. Then the rules are compiled into primitive rules.

The `smodels` procedure is a Davis-Putnam like backtracking search procedure that finds the answer sets of a set of primitive rules by assigning truth values to the atoms of the program. Moreover it uses the properties of the answer set semantics to infer and propagate additional truth values. Since the procedure is in effect traversing a binary search tree, the number of nodes in the search space is in the worst case on the order of  $2^n$ , where  $n$  can be taken from the number of atoms that appear in a constraint in a head of a rule or that appear as a negative literal in a recursive loop of the program.

Hence, in order to compute answer sets, one uses the two programs `lparse`, which translates logic programs into an internal format, and `smodels`, which computes the models.

It is worthwhile noting that, even if `SMODELS` kernel does not support disjunction, this feature has been implemented on top of `SMODELS` by a suitable rewriting technique. The resulting system is called `GnT` [35]. Such a rewriting-based implementation of disjunction, however, can obviously not provide the same performance than a built-in implementation. And indeed, `GnT` turns out to be sensibly slower than `DLV` on  $\Sigma_2^P$ -hard problems.

In Table 4.3 we summarize the features of `SMODELS` w.r.t. the properties relevant to data integration we have pointed out in Section 4.1.

### 4.2.3 Cmodels

`Cmodels` is a system that computes answer sets for either disjunctive logic programs or logic programs containing choice rules. Answer set solver `Cmodels` uses SAT solvers as a search engine for enumerating models of the logic program – possible solutions, in case of disjunctive programs SAT solver `zChaff` is also used for verifying the minimality of found models.

The system `Cmodels` is based on the relation between two semantics: the answer set and the completion semantics for logic programs. For big class of programs called *tight*, the answer set semantics is equivalent to the completion semantics, so that the answer sets for such a program can be enumerated by a SAT solver.

On the other hand for nontight programs [46], and [38] introduced the concept of the loop formulas, and showed that models of completion extended by all the loop formulas of the program are equivalent to the answer sets of the program. Unfortunately number of loop formulas might be large, therefore computing all of them may become computationally expensive. This led to the adoption of the algorithm that computes loop formulas “as needed” for finding answer sets of a program.

`Cmodels` is similar to `Smodels` or `GnT` in that its input is a grounded logic program that can be generated by the front-end called `Lparse`. The input of `Cmodels` may contain weight constraints, but optimize statements are not allowed. The representation of weight constraints by propositional formulas used in `Cmodels` is based on [24].

Table 4.4 summarizes the features of `Cmodels` relevant to data integration tasks.

SMODELS	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes, supports unstratified negation
Ability to deal with computationally hard queries	Up to NP thanks to unstratified negation and “choice” rules $\Sigma_2^P$ problems can be solved through a disjunctive extension (GnT) implemented by a rewriting technique
Other	Supports cardinality and weight constraints; handles only domain-restricted programs
<b>Optimization</b>	
Optimizations in simple but data intensive queries	Inefficiency caused by the domain-restriction requirement on program variables. No database optimization technique has been implemented
Optimizations in computationally hard queries	Heuristics and optimization techniques from the field of SAT programming have been implemented

Table 4.3: Features of SMOLELS

#### 4.2.4 ASSAT

ASSAT (Answer Sets by SAT solvers) [45] is a recently developed system for computing answer sets of a logic program by using SAT solvers. Given a ground logic program  $P$  and a SAT solver  $X$ , ASSAT( $X$ ) works as follows:

- Computes the completion of  $P$  and converts it into a set  $C$  of clauses.
- Repeats the following steps
  - Calls  $X$  on  $C$  to get a model  $M$  (terminates with failure if no such  $M$  exists).
  - Returns  $M$  if  $M$  is an answer set.

Cmodels	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes, supports unstratified negation and disjunction
Ability to deal with computationally hard queries	Up to $\Sigma_2^P$
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	No database optimization technique has been implemented
Optimizations in computationally hard queries	Heuristics and optimization techniques from the field of SAT programming have been implemented

Table 4.4: Features of Cmodels

- Otherwise, finds some loops in  $P$  whose loop formulas are not satisfied by  $M$  and adds their corresponding clauses to  $C$ .

As shown in [45], this procedure is sound and complete, assuming that  $X$  is a sound and complete SAT solver.

ASSAT exploits `lparse`, the grounding system of `SMODELS`, to instantiate a given program. Then, for each loop in the program which is found during the computation, a corresponding loop formula is added to the program's completion. In this way, a one-to-one correspondence between the answer sets of the program and the models of the resulting propositional theory is obtained. In the worst case, this process requires computing an exponential number of loop formulas.

In Table 4.5 we summarize the features of ASSAT w.r.t. the properties relevant to data integration we have pointed out in Section 4.1.

#### 4.2.5 noMoRe

The **non-monotonic reasoning** system `noMoRe` [2] implements answer set semantics for normal logic programs. It realizes a novel, rule-based paradigm to obtain answer sets by computing non-standard graph colorings of the *block graph* associated with a given logic program (see [47, 48] for details). These non-standard graph colorings are called *a-colorings* or *application-colorings*

ASSAT	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes, supports unstratified negation
Ability to deal with computationally hard queries	Up to NP thanks to unstratified negation
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	No database optimization technique has been implemented
Optimizations in computationally hard queries	By the use of well-assessed SAT solvers

Table 4.5: Features of ASSAT

since they reflect the set of generating rules (applied rules) for an answer set. Hence `noMoRe` is rule-based and not atom -based like most of the other known systems. It handles *backward propagation* of partial a-colorings and exploit a technique called *jumping* in order to ensure full (backward) propagation [48]. Both techniques improve the search space pruning of `noMoRe`.

The `noMoRe`-system is implemented in the programming language Prolog; it has been developed under the `ECLiPSe` Constraint Logic Programming System [1] and it was also successfully tested with `SWI-Prolog` [66].

`noMoRe` uses a compilation technique to compute answer sets of a logic program  $P$  in three steps. At first, the block graph  $\Gamma_P$  is computed. Secondly,  $\Gamma_P$  is compiled into Prolog code in order to obtain an efficient coloring procedure. Users may choose between two different kinds of compilation, one which is fast but which gives a lot of compiled code and another one which is a little bit slower but which produces less compiled code than the other. The second way of compiling has to be used with large logic programs, depending on the memory management of the underlying Prolog system. The compiled Prolog code (together with the example-independent code) is then used to actually compute the answer sets. To read logic programs it is used a parser (eventually after running a grounder, e.g. `lparse` or `dlv`) and there is a separate part for interpretation of a-colorings into answer sets. Additionally, `noMoRe` comes with an interface to the graph drawing tool `DaVinci` [52] for visualization of block graphs. This allows for a structural analysis of programs.

The `noMoRe` system is used for purposes of research on the underlying paradigm. But even in this early state, usability for anybody familiar with the logic programming paradigm is given. The syntax accepted by `noMoRe` is Prolog-like, like that of `DLV` and `smodels`. Furthermore, `noMoRe` is able to deal with integrity constraints as well as weight and cardinality constraints.

In Table 4.6 we summarize the features of noMoRe w.r.t. the properties relevant to data integration we have pointed out in Section 4.1.

noMoRe	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Unstratified negation is supported
Ability to deal with computationally hard queries	Up to NP through unstratified negation
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	No database technique has been implemented
Optimizations in computationally hard queries	It is a research prototype implemented in Prolog, which needs some engineering to become efficient

Table 4.6: Features of noMoRe

#### 4.2.6 SLG

The SLG system [10] is a research-oriented system for deductive databases and nonmonotonic reasoning. It is built as a meta interpreter on top of existing Prolog systems. In addition to all the functionalities of Prolog, SLG contains several features not usually found in logic programming systems, including:

- Query evaluation under the well-founded semantics using SLG resolution.
- Query evaluation of deductive databases whose rules may have explicit universal quantifiers in the body.
- Query answering under stable models.
- Abductive reasoning with integrity constraints.
- Skeptical reasoning with respect to the intersection of stable models.

The SLG meta interpreter employs an efficient algorithm for incremental maintenance of dependencies among subgoals so that subgoals that are completely evaluated or are possibly involved in recursion through negation can be detected by inspecting the dependency information of a single subgoal.

The SLG- $\forall$  meta interpreter augments the SLG meta interpreter with the handling of universal rules. Traditionally universal quantification is eliminated by conversion into the negation of an existential quantification. This, however, may introduce extra recursion through negation, and does not preserve the alternating fixpoint semantics of general logic programs. SLG- $\forall$  computes the alternating fixpoint semantics by processing universal rules directly.

A profiling of the SLG- $\forall$  meta interpreter shows that two major factors of overhead are meta interpretation and the lack of destructive assignment for managing tables of subgoals and their answers.

In Table 4.7 we summarize the features of SLG w.r.t. the properties relevant to data integration we have pointed out in Section 4.1.

SLG	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes
Ability to deal with computationally hard queries	Up to NP
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	No
Optimizations in computationally hard queries	It is a research prototype which needs some engineering to become efficient

Table 4.7: Features of SLG

#### 4.2.7 DeReS

The system DeReS [11] supports basic automated reasoning tasks for default logic and for logic programming under the answer set semantics. It is shown that a normal logic program  $P$  can be represented by a suitable default theory  $D$ , such that the answer sets of  $P$  correspond to the so-called *extensions* of  $D$ . DeReS uses *relaxed stratification* as a primary mechanism for pruning the search-space. A default theory  $D$  is partitioned into several smaller subtheories, called *strata* and the extensions of  $D$  are constructed from the extensions of its strata. The approach taken by DeReS is somehow orthogonal to the one taken by SMOBELS, and it is argued in [11] that

next generation implementations of nonmonotonic systems must combine techniques developed in both projects in order to be effective in a large range of different applications.

The features of the system DeReS are summarized in Table 4.8.

DeReS	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Unstratified negation is supported
Ability to deal with computationally hard queries	Up to NP in Datalog, up to $\Sigma_2^P$ in Default Logic
Other	Also Default Logic is supported
<b>Optimization</b>	
Optimizations in simple but data intensive queries	No database optimization technique is implemented
Optimizations in computationally hard queries	An engineering phase is needed

Table 4.8: Features of DeReS

#### 4.2.8 XSB

The XSB system is an inmemory deductive database engine based on a Prolog/SLD resolution strategy. Clearly, the traditional Prolog systems are known to have serious deficiencies when used as database systems. Indeed, the SLD computational mechanism, which well serves the needs of a programming language, is clearly inadequate as a database computation strategy. Its most serious drawback is that it does not terminate for the datalog language. Datalog is a decidable language (one reason that makes it a reasonable candidate for a database language) but SLD refutation is not finite on it. The deductive database community has adopted datalog as a leading database query language, identified these problems, and rectified them. Rewriting techniques have been developed to introduce goaldirectedness into a bottomup, setatime evaluation strategy. These techniques solve SLD's problems of lack of finiteness and redundant computation. XSB offers an alternative approach to creating a deductive database system. Rather than depending on rewriting techniques, it extends Prolog's SLD resolution in two ways: 1) adding tabling to make evaluations finite and nonredundant on datalog, and 2) adding a scheduling strategy and delay mechanisms to treat general negation efficiently. The resulting strategy is called SLG resolution, which is complete and finite for nonfloundering programs with finite models, whether they are stratified or not.

The system XSB [58] can compute most cases of the well-founded semantics for normal logic programs with functions symbols. The inference engine, which is called the SLG-WAM, consists

of an efficient tabling engine for definite logic programs, which is extended by mechanisms for handling cycles through negation. These mechanisms are negative loop detection, delay and simplification. They serve for detecting, breaking and resolving cycles through negation. It is worth pointing out that XSB can work only in main memory and, consequently, it could not evaluate programs working on huge amounts of data.

Table 4.9 summarizes the features of XSB relevant to data integration tasks.

XSB	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	No
Ability to express nonmonotonic queries	Only “well-founded” queries
Ability to deal with computationally hard queries	Only polynomial-time queries
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	The system is at an advanced engineering state. However, the “tuple-oriented”, top-down computational model limits the efficiency on database queries
Optimizations in computationally hard queries	Not applicable

Table 4.9: Features of XSB

### 4.2.9 claspD

`claspD` is an answer set programming (ASP) solver for (extended) normal and disjunctive logic programs. It is able to deal with problems at the second level of the polynomial hierarchy. `claspD` deploys a generate and test approach, both tasks implemented by way of `clasp`'s core technology [26]; consequently, it combines the high-level modeling capacities of Answer Set Programming with state-of-the-art techniques from the area of Boolean constraint solving.

Unlike existing ASP solvers, `claspD` is originally designed and optimized for conflict-driven ASP solving [28, 27], centered around the concept of a nogood from the area of constraint processing (CSP). Rather than applying a SAT(isfiability checking) solver to a CNF conversion, `clasp` directly incorporates suitable data structures, particularly fitting backjumping and learning.

Such techniques include:

- conflict analysis via the First-UIP scheme;

- nogood recording and deletion;
- backjumping;
- restarts;
- conflict-driven decision heuristics;
- unit propagation via watched literals;
- dedicated propagation of binary and ternary nogoods;

However, claspD is a genuine ASP solver. Its basic propagation engine includes advanced unfounded set checking based on source pointers. In fact, claspD is the first disjunctive ASP solver whose propagation engine is able (but not guaranteed) to detect non-singleton unfounded sets within non-head-cycle-free strongly connected components of a program's (positive) atom dependency graph. Furthermore, non-polynomial unfounded set checks are only applied if necessary, that is, when an exhaustive test of a non-head-cycle-free strongly connected component is needed.

Table 4.10 summarizes the features of claspD relevant to data integration tasks.

claspD	
<b>Language expressiveness</b>	
Ability to express views	Yes
Ability to express recursive queries	Yes
Ability to express integrity constraints	Yes
Ability to express nonmonotonic queries	Yes, supports unstratified negation and disjunction
Ability to deal with computationally hard queries	Up to $\Sigma_2^P$
Other	
<b>Optimization</b>	
Optimizations in simple but data intensive queries	No database optimization technique has been implemented
Optimizations in computationally hard queries	Heuristics and optimization techniques from the field of SAT programming have been implemented

Table 4.10: Features of claspD

#### 4.2.10 Other systems

Among other ASP systems we cite the system *near Horn* [49, 51] which has been implemented in PROLOG; in [59] the system DisLog is described, which incorporates different disjunctive theories and strategies including the semantics introduced in [50]; DisLog tries to eliminate redundant computations by using a breadth-first approach. The system DisLoP [4, 5] which aims at extending the *restart model elimination* and *hyper tableau calculi*, for disjunctive logic programming under the D-WFS and stable semantics. The *aspps* system is an answer-set programming system based on the extended logic of propositional schemes [16], which allows variables but not function symbols in the language. The GnT system is an implementation of the stable model semantics for disjunctive logic programs constructs on top of Smodels system; this implementation is based on an architecture consisting of two interacting Smodels solvers for non-disjunctive programs, one of the them is responsible for generating as good as possible model candidates while the other checks for minimality, as required from disjunctive stable models.

# Chapter 5

## DLV<sup>DB</sup> Extensions

---

The DLV<sup>DB</sup> system, described in Chapter 3, is a database-oriented variant of DLV that carries out all of its tasks in mass memory to enable data intensive applications. In its early implementation, DLV<sup>DB</sup> did not support external predicates, list terms, and function symbols, which are of fundamental importance to enhance knowledge representation and reasoning capabilities of a DLP language. In particular, as highlighted earlier in this thesis, function symbols and lists terms allow the aggregation of atomic data, the manipulation of complex data structures and the generation of new symbols. Finally, external predicates allow the isolation of units of a procedural program for calling them within declarative logic programs.

The goal of this work is to extend the DLV<sup>DB</sup> system to let it support the above mentioned language constructs, in order to improve its knowledge modelling power. The remainder of this chapter is organized as follows. First, we recall the basic notions of database stored functions, which are used in our approach to implement the external predicates in DLV<sup>DB</sup>. Then, we describe the evaluation strategies used for supporting external predicates, lists, and functional terms.

Part of the material presented in this chapter appeared in [61, 62].

### 5.1 Database stored functions

Most ASP systems implement external predicates by means of calls to external functions defined with procedural languages, such as C or C++. For example, DLV-EX [9] extends DLV by external predicates with the aim of enabling ASP to deal with external sources of computation. This feature is obtained by the introduction of parametric external predicates, that are not specified by means of a logic program but implicitly computed through external code provided in a dynamic library.

Since DLV<sup>DB</sup> transforms logical programs into SQL statements to enable database-oriented processing, we implement external predicates by exploiting database-oriented solutions. In particular, we implement external predicates by calls to database stored functions.

A stored function is a general mechanism used in database systems to provide procedural capabilities within declarative SQL statements. The use of stored functions in an SQL statement allows to execute some parts of the application directly within the database system process space.

For the sake of completeness we mention that, according to the SQL specifications, two alternative mechanisms may be used to introduce procedural capabilities in SQL:

- SQL-invoke-procedures, also referred to as *stored procedures*;
- SQL-invoke-functions, or simply *stored functions*.

A stored procedure receives a set of parameters, where each parameter may be of input or of output type, but does not return any result. A stored function can receive only input parameters, and returns a single result in a functional way.

Once defined, a stored procedure is invoked, outside of an SQL statement, as follows:

```
CALL procedure_name(value,...,value)
```

Conversely, a stored function can be directly invoked within an SQL statement, either in the SELECT or in the WHERE part, as in the following example:

```
SELECT table.field FROM table WHERE function_name(field)=value;
```

Given the possibility to invoke stored functions directly in SQL, in DLV<sup>DB</sup> we use stored functions instead of stored procedures to implement external predicates.

Stored functions are defined by the SQL/PSM specification<sup>1</sup> introduced with SQL:1999. However, with the exception of few database systems (e.g., Informix, RDBMS), most DBMSs use proprietary languages to define stored functions that exceed the SQL/PSM specifications. Table 5.1 lists the languages used by some database systems to implement stored functions.

Table 5.1: Languages used by some database systems to implement stored functions

Database	Stored function languages
Microsoft SQL Server	Transact-SQL; various .NET Framework languages
Oracle	PL/SQL; Java
DB2	SQL/PL; Java
Postgres	PL/pgSQL; pl/perl; pl/php
MySQL	language close to SQL:2003 standard
Firebird	PSQL

The following example shows a stored function *factorial* that returns the factorial of a number which is created in MySQL:

```
CREATE FUNCTION 'factorial' (n int) RETURNS INT
BEGIN
  DECLARE res int;
  DECLARE i int;
  SET i=2;
  SET res=1;
  WHILE (i<=n) do
    SET res=res*i;
    SET i=i+1;
  END WHILE;
  RETURN res;
END
```

<sup>1</sup>PSM stands for Persistent Stored Module

To define a mapping between a stored function, created as above, and an external predicate name to be used in a DLP program, we have introduced an auxiliary directive named USEFUNCTION, in addition to the other auxiliary directives already introduced in Section 3.3.

The syntax of USEFUNCTION is as follows:

```
USEFUNCTION FunctionName MAPTO ExternalPredName.
```

For example, assuming the existence of the *factorial* stored function, the following directive should be used to map that function onto an external predicate *fact*:

```
USEFUNCTION factorial MAPTO fact.
```

## 5.2 Evaluation of external predicates

Recall that, by convention, given an external atom  $\#f(X_1, \dots, X_n, O)$  used in a rule  $r$ , only the last variable  $O$  can be considered as an output parameter, while all the other variables must be intended as input for  $f$ . This corresponds to the stored function call  $f(X_1, \dots, X_n) = O$  on the DLV<sup>DB</sup> working database. Moreover,  $O$  can be: (i) bound to other variables in  $r$ 's body, (ii) bound to a constant, (iii) bound to a variable of  $r$ 's head, (iv) bound to an input parameter of another external function.

To cope with the four cases listed above, different rules are followed to derive the SQL statement corresponding to  $r$ . The translation rules have been implemented by extending the *DLV to SQL Translator* module represented in Figure 3.1. In particular, the *TranslateNonRecursiveRule* function, already introduced in the previous chapter, has been extended as shown in Figure 5.1.

---

```
Function TranslateNonRecursiveRule( $r$ : DLPA rule): SQL statement
begin
   $SQL := ""$ ;
  if (hasAggregate( $r$ )) then
     $SQL := SQL + \text{TranslateAggregateRule}(\mathbf{r})$ ;
   $SQL := SQL + \text{"INSERT INTO " + head}(\mathbf{r}) + \text{"("}$ ;
  if (isPositive( $r$ )) then
     $SQL := SQL + \text{TranslatePositiveRule}(\mathbf{r})$ ;
  else if (hasNegation( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithNegation}(\mathbf{r})$ ;
  else if (hasBuilt-In( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithBuilt}(\mathbf{r})$ ;
  else if (hasExternalPredicate( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithExternalPredicates}(\mathbf{r})$ ;
  else if (hasBuilt-InAndExternalPredicates( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithBuilt-InAndExternalPredicates}(\mathbf{r})$ ;
  else if (hasNegationAndBuilt-In( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithNegationAndBuilt-In}(\mathbf{r})$ ;
  else if (hasNegationAndBuilt-InAndExternalPredicates( $r$ )) then
     $SQL := SQL + \text{TranslateRuleWithNegationAndBuilt-InAndExternalPredicates}(\mathbf{r})$ ;
   $SQL := SQL + \text{"})"$ ;
  return  $SQL$ ;
end.
```

---

Figure 5.1: Extended version of the *TranslateNonRecursiveRule* function

The new version of *TranslateNonRecursiveRule*, differently from the previous version, checks whether the rule includes external predicates by the call of *hasExternalPredicates* function; if so, it calls the *TranslateRuleWithExternalPredicates* function shown in Figure 5.2.

---

```

Function TranslateRuleWithExternalPredicates(r: DLPA rule): SQL statement
begin
  SELECT:String:="SELECT " + head attr(r);
  FROM:String:="FROM " + body+(r);
  WHERE:String:="WHERE " + joinConditions(r) + "AND " + bodyConstantConditions(r);
  for each a in externalAtom(r) do begin
    if (extAtomBoundInHead(a,r)) then begin
      SELECT:=SELECT+", "" + getExtFunctionDef(a);
    end.
    if (extAtomBoundInBody(a,r)) then begin
      if (isNegative(a)) then begin
        WHERE:=WHERE+ "AND ("
          + getExtFunctionDef(a) + "<>" + getVarBoundToExtAtom(a,r);
        WHERE:=WHERE+ "OR" + getExtFunctionDef(a) + " IS NULL");
      end
      else begin
        WHERE:=WHERE+ "AND "
          + getExtFunctionDef(a) + " " + getVarBoundToExtAtom(a,r);
        WHERE:=WHERE+ "AND" + getExtFunctionDef(a) + " IS NOT NULL";
      end.
    end.
    if (extAtomBoundToConstant(a,r)) then begin
      if (isNegative(a)) then begin
        WHERE:=WHERE+ "AND ("
          + getExtFunctionDef(a) + "i" + getConstantBoundToExtAtom(a,r);
        WHERE:=WHERE+ "OR" + getExtFunctionDef(a) + " IS NULL");
      end.
      else begin
        WHERE:=WHERE+ "AND"
          + getExtFunctionDef(a) + " =" + getConstantBoundToExtAtom(a,r);
        WHERE:=WHERE+ "AND" + getExtFunctionDef(a) + " IS NOT NULL";
      end.
    end.
    if (extAtomBoundInputOtherExtAtom(a,r)) then begin
      var:String:=getVarBoundToExtAtom(a,r);
      extFunDef:String:=getExtFunctionDef(a);
      replaceVarWithExtFunctionDef(SELECT, var, extFunDef);
      replaceVarWithExtFunctionDef(WHERE, var, extFunDef);
      WHERE:=WHERE+ "AND" + getExtFunctionDef(a) + " IS NOT NULL";
    end.
    SQL:=SELECT + FROM + WHERE + "EXCEPT (SELECT * FROM " + head(r) + " )";
  return SQL;
end.

```

---

Figure 5.2: Function TranslateRuleWithExternalPredicates

*TranslateRuleWithExternalPredicates* receives a rule *r* as input and returns the corresponding SQL statement. Basically, it first initializes the SQL statement in the same way as function *TranslatePositiveRule* in Figure 3.5; then for each external atom, it creates an appropriate SELECT or WHERE part, based on whether the output variable of the external atom is

bound (i) to other variables in the rule's body, (ii) to a constant, (iii) to a variable of the rule's head, or (iv) to an input parameter of another external function, as already mentioned above. This check is performed by function *extAtomBoundInHead* (resp., *extAtomBoundInBody*, *extAtomBoundToConstant*, *extAtomBoundInputOtherExtAtom*) which receives a rule  $r$  and an external atom  $a$  and returns true if  $a$  is bound to other variables in  $r$ 's body (resp., to a constant, to a variable of  $r$ ' head, to an input parameter of another external function), false otherwise. In the following we describe, with the help of some examples, how the four cases are managed by *TranslateRuleWithExternalPredicates*.

### 5.2.1 Output variable bound to other variables in the rule's body

Given a rule containing an external atom having its output variable bound to other variables in the rule's body, the corresponding SQL statement is obtained by introducing a stored function call (associated with the external predicate) in the WHERE part. The WHERE clause includes a condition dictating that the stored function output is equal to the database mapping of the predicate variable to which the output variable is bound.

As an example, let us consider the following rule:

$$\textit{nounStartsWithVowel}(N) :- \textit{noun}(N), \textit{vowel}(V), \#characterAt(N, 1, V).$$

The rule includes an external predicate, *#characterAt*, which takes in input a string  $N$  and the position of a character (1, in this case), and returns in  $V$  the character at that position. The output variable  $V$  is bound to the variable of the *vowel* atom; therefore, by applying the strategy described above, the rule is translated into the following SQL statement:

```
INSERT INTO nounStartsWithVowel
(SELECT noun.n FROM noun, vowel WHERE characterAt(noun.n,1)=vowel.v AND
characterAt(noun.n,1) is NOT NULL);
```

As shown in the SQL statement above, a stored function *characterAt* is used to perform the operation associated with the external predicate *#characterAt*. The stored function is introduced into the WHERE part, with the condition that its output is equal to *vowel.v* and not NULL. A stored function returns NULL when it is not applicable to the provided input parameters<sup>2</sup>

Note that, since the grounding phase instantiates all the variables, there is no need to invoke again the functions associated with external predicates after the grounding (this is true even for disjunctive or non stratified programs). As a consequence, the handling of external predicates can be carried out completely during the grounding and, hence, within the SQL statements generated from the logic rules.

We consider now a variant of the example above containing a negated predicate:

$$\textit{nounDoesNotStartWithVowel}(N) :- \textit{noun}(N), \textit{vowel}(V), \textit{not} \#characterAt(N, 1, V).$$

The rule is translated into the following SQL statement:

```
INSERT INTO nounDoesNotStartWithVowel
(SELECT noun.n FROM noun, vowel WHERE characterAt(noun.n,1)<>vowel.v) OR
characterAt(noun.n,1) is NULL;
```

<sup>2</sup>Indeed, it is mandatory that a stored function returns NULL if the input parameters cannot be processed due to incompatible types or out-of-range values.

In this example the WHERE clause includes a condition dictating that the output of the stored function, *characterAt*, is not equal to the database mapping of the predicate variable, *vowel.v*, or it is NULL. For the sake of clarity, in the following examples we omit to explicitly check whether the output of the stored function is (or is not) NULL.

### 5.2.2 Output variable bound to a constant

If the rule contains an external function having its output variable bound to a constant, the corresponding SQL statement is constructed by introducing a stored function call in the WHERE part, with the condition that its output is equal to the constant value. This can be considered as a special case of the scenario described in the previous subsection, in which the output variable is bound to other variables in the rule's body.

As an example, let us consider the following rule:

$$\text{multiple}(X, Y) :- \text{number}(X), \text{number}(Y), \#remainder(X, Y, 0).$$

The external predicate *#remainder* receives two numbers *X* and *Y* and returns the integer remainder of a division between them. The output is bound to the constant 0 to check whether the first number is a multiple of the second one. According to the translation strategy described above, the following SQL statement is derived:

```
INSERT INTO multiple
(SELECT n1.n, n2.n FROM number as n1, number as n2 WHERE remainder(n1.n,
n2.n)=0);
```

As shown above, the stored function *remainder* is introduced into the WHERE part, with the condition that its output is equal to the constant value 0.

The following rule is obtained by negating the *remainder* predicate used in the previous rule:

$$\text{notMultiple}(X, Y) :- \text{number}(X), \text{number}(Y), \text{not } \#remainder(X, Y, 0).$$

The rule is translated into the following SQL statement, in which the condition is that the stored function output is not equal to the constant value 0:

```
INSERT INTO multiple
(SELECT n1.n, n2.n FROM number as n1, number as n2 WHERE remainder(n1.n,
n2.n)<>0);
```

### 5.2.3 Output variable bound to a variable in the rule's head

This case comprises those rules having an external predicate with the output variable bound to a head variable. The translation is performed by introducing a stored function call into the SELECT part of the SQL statement.

As an example, consider the following rule:

$$\text{factorial}(X, Y) :- \text{number}(X), \#fact(X, Y).$$

The rule includes an external predicate, *#fact*, which receives a number *X* and returns its factorial *Y*. The output variable *Y* is bound to a variable of the head. Therefore, according to the strategy above, the rule is translated into the following SQL statement:

```
INSERT INTO factorial
(SELECT number.n, fact(number.n) FROM number);
```

The SQL statement shows that, differently from the previous two scenarios, in this case the stored function *fact* is introduced into the SELECT part. In more detail, this statement feeds the extensional database with new facts, having predicate *factorial*, which store the factorial for each number in the *number* table.

Note that a rule that contains the output variable of a negated external atom bound to a variable in the rule's head is unsafe. For instance, the following rule is unsafe:

$$\text{notFactorial}(X, Y) :- \text{number}(X), \text{not } \# \text{fact}(X, Y).$$

### 5.2.4 Output variable bound to an input parameter of another external predicate

We conclude this section with the case in which a rule contains a non negative external predicate  $ep_1$  having the output variable  $O$  bound to an input parameter  $I$  of another external predicate  $ep_2$ . In this case, the SQL statement is obtained by substituting the variable  $I$  with a call to the stored function associated with  $ep_1$ . This is independent from the fact that  $ep_2$  has its variable bound to other variables in the rule's body (which requires a stored function in the WHERE part), or bound to a variable in the rule's head (which requires a stored function in the SELECT part).

As an example, we consider the rule:

$$\text{palindromic}(Z) :- \text{word}(X), \# \text{reverse}(X, Y), \# \text{concat}(X, Y, Z).$$

The rule includes two external predicates:  $\# \text{reverse}$  that receives a string  $X$  and returns its reversed version  $Y$ , and  $\# \text{concat}$  that receives two strings,  $X$  and  $Y$ , and returns their concatenation  $Z$ . The output variable of the first external predicate is bound to an input parameter of the second external predicate; therefore, the rule is translated into the following SQL statement:

```
INSERT INTO palindromic
(SELECT concat(word.w, reverse(word.w)) FROM word);
```

The stored function *concat* is included into the SELECT part because the output variable of the corresponding external function is bound to an input parameter of the rule's head. Moreover, as the second input parameter of *concat*, we use the output of the stored function *reverse*, according to the translation strategy described above.

Note that a rule that contains the output variable of a negated atom bound to an input parameter of another external atom is unsafe. For instance, the following rule is unsafe:

$$\text{notPalindromic}(Z) :- \text{word}(X), \text{not } \# \text{reverse}(X, Y), \# \text{concat}(X, Y, Z).$$

## 5.3 Evaluation of list terms

List terms are handled through a rewriting of the rule using suitable external predicates. In particular, programs containing list terms are automatically rewritten to contain only terms and

external predicates. Three basic operations can be singled out to handle lists: (i) initialization, (ii) packing of list terms, (iii) unpacking of list terms.

Lists are internally handled as strings, starting (resp., ending) with a '[' (resp., ']') where terms are separated by a ','. Initialization is then implicitly implemented by the transformation of the list into a string.

Packing and unpacking are the core operations to handle list terms. We implemented an additional software module in DLV<sup>DB</sup> to manage such operations. The new module, called *LFT Rewriter* (Lists and Functional Terms Rewriter), is located between the *Parser* and the *Optimizer* modules represented in Figure 3.1. Moreover, for each supported DBMS, we provide a set of stored functions associated with the external predicates used to handle list terms. The source code of such functions is included in Appendix A.

Recall, from Section 2.1, that a list term can be defined using one of following forms:

1.  $[H|T]$  where  $H$  (the head of the list) is a term, and  $T$  (the tail of the list) is a list term.
2.  $[T_1, \dots, T_n]$  where  $T_1, \dots, T_n$  are terms;

We use the term *closed lists* to indicate lists defined using the first form, and *open lists* to indicate lists of the second form. We assume that lists contain homogeneous data, i.e., we manage lists of integers or list of strings, but not lists containing both integers and strings.

Figure 5.3 shows the *RewriteRuleWithListTerms* function that, given a rule with list terms, returns that rule rewritten by replacing each list definition with calls to external predicates. In particular, for each *list\_term* that appear in the rule head, the function checks, using a function *isOpenListTerm* (resp. *isClosedListTerm*), if that *list\_term* is in open (resp., closed) list form. If a *list\_term* is in open list form, it is replaced with a set of *pack* external atoms; otherwise, it is replaced with a set of *cat* external atoms. Similarly, for each *list\_term* that appear in the body, the function uses *isOpenListTerm* (resp. *isClosedListTerm*) to check whether it is in open (resp., closed) list form. If so, it is replaced with a set of *unpack* external atoms; otherwise, it is replaced with a set of *memberNth* external atoms.

In the following we describe through some examples how the packing and unpacking operations are performed by the *RewriteRuleWithListTerms* function, based on which form is used to define a list term.

### 5.3.1 Packing operation

Packing is the operation that builds a list starting from its basic elements. Note that the packing is needed only if the list term is located into the head of the rule. Here we show how the packing is performed, based on which form (closed or open) the list belongs to.

#### Closed lists

The packing of a closed list is carried out by an external predicate *#pack* that receives a term  $H$  and a list  $T$  and returns the list  $L = [H|T]$ .

For example, the rule:

$$p([H|T]) :- dom(H), list(T).$$

is translated into the following rule:

$$p(L) :- dom(H), list(T), \#pack(H, T, L).$$

---

```

Function RewriteRuleWithListTerms(r: DLPA rule): DLPA rule
begin
  newRule:DLPA rule:=r;
  for each list_term in head(r) do begin
    if (isOpenListTerm(list_term)) then begin
      packAtom:DLPA atom ARRAY:=buildPackExtAtoms(list_term);
      addAtomToRuleBody(newRule,packAtom);
      replaceListTermInHead(newRule,list_term,packAtom);
    end.
    elseif (isClosedListTerm(list_term)) then begin
      catAtoms:DLPA atom ARRAY:=buildCatExtAtoms(list_term);
      addAtomToRuleBody(newRule,catAtoms);
      replaceFuncTermInHead(newRule,list_term,catAtoms[catAtoms.length()-1]);
    end.
  end.
  for each list_term in body(r) do begin
    if (isOpenListTerm(list_term)) then begin
      unpackAtoms:DLPA atom ARRAY:=buildUnpackExtAtoms(a);
      addAtomToRuleBody(newRule,unpackAtoms);
      replaceListTermInBody(newRule,list_term,unpackAtoms[0]);
    else if (isClosedListTerm(list_term)) then begin
      memberNthAtoms:DLPA atom ARRAY:=buildMemberNthExtAtoms(a);
      addAtomToRuleBody(newRule,memberNthAtoms);
      replaceListTermInBody(newRule,list_term,memberNthAtoms[0]);
    end.
  end.
  return newRule;
end.

```

---

Figure 5.3: Function RewriteRuleWithListTerms

The corresponding SQL statement, according to the rules described in Section 5.2, is the following:

```

INSERT INTO p
(SELECT pack(dom.h, list.t) FROM q, list);

```

As another example, consider the following rule:

$$p([H2|[H1|T]]) :- s(H1), q(H2), list(T).$$

In this case, the tail of the list is in turn composed by another closed list. The packing is performed by building a list  $L0 = [H1|T]$  first; then a list corresponding to  $[H2|L0]$  is derived. Therefore, the final rule is as follows:

$$p(L) :- s(H1), q(H), list(T), \#pack(H1, T, L0), \#pack(H, L0, L).$$

### Open lists

The packing of an open list is based on the use of an external predicate  $\#cat$  that concatenates two strings.

As mentioned earlier, lists are handled as strings, starting with a '[' , ending with a ']' , and with terms separated by a ',' . In brief, the packing algorithm iterates on the string that represents

the list (from the starting parenthesis to the ending parenthesis), and introduces a set of external predicates *#cat* to build different portions of the list.

To illustrate the process, consider the rule:

$$l([A, B, c]) :- p(A), q(B).$$

The list includes three terms: two variables (*A* and *B*) and a constant (*c*). A first external predicate *#cat* builds a first portion of the list (*L1*) by concatenating the starting parenthesis with the first variable (*A*). A second predicate *#cat* builds another portion of the list (*L2*) that is obtained by concatenating the first portion (*L1*) with a ','. Again, a third *#cat* builds a portion *L3* that concatenates *L2* with variable *B*. Finally, the last external predicate *#cat* builds the overall list *L* by concatenating *L3* with string ',c]'. Note that, the last *#cat* takes both a ',' and and 'c]', because *c* is a constant. Overall, the rule above is translated into the following rule:

$$l(L) :- p(A), q(B), \#cat('[' , A, L1), \#cat(L1, ' , L2), \#cat(L2, B, L3), \#cat(L3, ' , c]', L).$$

Finally, the corresponding SQL statement is derived by applying the translation rules described earlier in this chapter:

```
INSERT INTO l
(SELECT cat(cat(cat(cat('[' , p.a), ','), q.b), 'c]')) FROM p, q)
```

### 5.3.2 Unpacking operation

Unpacking is the operation that returns the basic elements of a list. The unpacking operation must be performed only if the list term is located into the body of the rule. As for the packing operation, the unpacking is performed differently if the list term is defined in closed or open form. In the following we describe both procedures with the help of some examples. Additionally, we describe how rules with negated atoms containing list terms are rewritten.

#### Closed lists

The unpacking of a closed list would require a function that returns two values (the head and the tail). However, database stored functions can return one value only and cannot have side effects on existing tables. Our solution is to perform the unpacking by using two different calls to external predicates *#head* and *#tail* that, given a string representing a list, return its head element and its tail, respectively. Since a list can be empty and this cannot be unpacked, we use a third external predicate, *#unpack*, which returns true only if the input string represents a non-empty list.

As an example, the rule:

$$p(T) :- q(H), list([H|T]).$$

is translated into the following rule:

$$p(T) :- q(H), list(L), \#head(L, H), \#tail(L, T), \#unpack(L, true).$$

The corresponding SQL statement is:

```
INSERT INTO p
(SELECT tail(list.l) FROM q, list WHERE (q.h = head(list.l))
AND ('true' = unpack(list.l)) .
```

Note that availability of external predicates `#head` and `#tail` allows also the manipulation of nested lists, similarly to the example shown for the packing of closed lists.

### Open lists

The unpacking of an open list is based on the use of two external predicates: `#memberNth` that receives a list  $L$  and an index  $I$  and returns the list element at position  $I$ ; `#length` that receives a list and returns the number of its elements. Given a list containing  $n$  terms, we use  $n$  instances of `#memberNth`, each one to extract one of the list terms. Moreover, we use one instance of `#length` to check whether the length of the list is  $n$ .

For example, the rule:

$$q(B) :- l([A, B, c]), p(A).$$

is translated into the following rule:

$$q(B) :- l(L), p(A), \#memberNth(L, 1, A), \#memberNth(L, 2, B), \\ \#memberNth(L, 3, 'c'), \#length(L, 3).$$

Note that the first instance of `#memberNth` takes the first element of  $L$  and binds it to variable  $A$ . In the same way, the second `#memberNth` takes the second element of  $L$  (bound to variable  $B$ ), and another `#memberNth` takes the last element of  $L$  (bound to constant  $c$ ). Finally, external predicate `#length` checks that the length of the list is 3.

The corresponding SQL statement is then obtained as follows:

```
INSERT INTO q
(SELECT memberNth(l.a1, 2) FROM l, p WHERE (p.a1 = memberNth(l.a1, 1)) AND
('c' = memberNth(l.a1, 3)) AND (3 = length(l.a1))).
```

### Lists in negated atoms

Finally, we consider the case of negated atoms containing list terms. The unpacking in this case is performed in two phases: first, an auxiliary rule is derived; then, another rule is derived from the original one, by substituting the negated atom containing the list with the predicate generated from the auxiliary rule. The auxiliary rule contains in the head all the list terms (variables and constants); in the body it contains the predicate containing the list term (without negation), and the unpacking of the list term itself, which in turn depends on its type (closed or open).

Let us consider the following example:

$$p(H) :- \text{not } q([H|T]), a(H), b(T).$$

Here, the closed list  $[H|T]$  appears in a negated atom (`not q`). According to the procedure defined above, the unpacking generates the following rules:

$$\text{aux}(H, T) :- q(L), \#head(L, H), \#tail(L, T), \#unpack(L, \text{true}). \\ p(H) :- a(H), b(T), \text{not } \text{aux}(H, T).$$

The auxiliary rule generates an atom  $\text{aux}(H, T)$ . The second rule is obtained from the original one by replacing  $q$  with  $\text{aux}$ .

The SQL statement corresponding to the auxiliary rule is obtained following the general procedure defined for the external predicates, as defined in Section 5.2. For the second rule, the corresponding SQL statement is generated following the procedure described in Section 3.5.

## 5.4 Evaluation of functional terms

Functional terms are handled similarly to open lists. Hence, the rewriting includes a packing operation and an unpacking operation. The packing is performed when the functional term appears in one of the atoms of the rule's head, while the unpacking is needed when it appears in one of the atoms of the rule's body.

As for the lists, also functional terms are handled as strings. Such strings start with a functor (e.g.,  $f$ ), followed by an opening parenthesis '(', a set of terms separated by a ',', and are terminated by a closing parenthesis ')

Packing and unpacking are managed by the *LFT Rewriter* module, as mentioned in Section 5.3. To this end, we defined the *RewriteRuleWithFunctionalTerms* function shown in Figure 5.4 that, given a rule with functional terms, returns that rule rewritten by replacing each functional term definition with calls to external predicates. In particular, each *funct\_term* that appears in the rule head is replaced with a set of *cat* external atoms, whereas each *funct\_term* that appears in the rule body is replaced with a set of *memberNthFunct* external atoms.

---

```

Function RewriteRuleWithFunctionalTerms( $r$ : DLPA rule): DLPA rule
begin
   $newRule$  : DLPA rule:= $r$ ;
  for each  $funct\_term$  in head( $r$ ) do begin
     $catAtoms$ :DLPA atom ARRAY:=buildCatFunctExtAtoms( $funct\_term$ );
    addAtomToRuleBody( $r$ , $catAtoms$ );
    replaceFunctTermInHead( $r$ , $funct\_term$ , $catAtoms$ [ $catAtoms.length()-1$ ]);
  end.
  for each  $funct\_term$  in body( $r$ ) do begin
     $memberNthFunctAtoms$ :DLPA atom ARRAY:=buildMemberNthFunctExtAtoms( $funct\_term$ );
    addAtomToRuleBody( $r$ , $memberNthFunctAtoms$ );
    replaceListTermInBody( $r$ , $funct\_term$ , $memberNthFunctAtoms$ [0]);
  end.
  return  $newRule$ ;
end.

```

---

Figure 5.4: Function RewriteRuleWithFunctionalTerms

In the following we describe, with the help of some examples, how the packing and unpacking operations are performed by the *RewriteRuleWithFunctionalTerms* function.

### 5.4.1 Packing operation

The packing operation builds a functional term starting from its basic elements, which include functor, parentheses, terms, and separation commas. As for the open lists, the external predicate *#cat* is used.

For example, the following rule:

$$p(f(A)) :- a(A).$$

is translated into the following one:

$$p(F) :- a(A), \#cat('f(', A, F1), \#cat(F1, ')', F).$$

The first *#cat* generates a portion  $F1$  of the functional term that is obtained as the concatenation of 'f(' with variable  $A$ . Then,  $F1$  is concatenated with the closing parenthesis to derive  $F$ . The corresponding SQL statement is then obtained as follows:

```
INSERT INTO p
(SELECT cat(cat('f', a.a1), '')) FROM a );
```

### 5.4.2 Unpacking operation

The unpacking operation returns the basic elements of a functional term. We use three external predicates to perform this operation:

- *#getFunctionName* that receives a functional term and returns its functor;
- *#memberNthFun* that receives a functional term  $F$  and an index  $i$ , and returns the  $i^{\text{th}}$  term of  $F$
- *#lengthFun* that returns the number of terms in a functional term received as input.

As an example, the following rule:

$$q(A) :- p(f(A, B)), a(A), b(B).$$

is translated into:

$$q(A) :- p(F), a(A), b(B), \#getFunctionName(F, f), \#memberNthFun(F, 1, A), \#memberNthFun(F, 2, B), \#lengthFun(F, 2).$$

Note that *#memberNthFun* is used twice, because the functional term includes two terms  $A$  and  $B$ . The corresponding SQL statement is then obtained as follows:

```
INSERT INTO p
(SELECT a.a1 FROM p, a, b WHERE ('f' = getFunctionName(p.a1)) AND (a.a1 =
memberNthFun(p.a1, 1)) AND b.a1 = memberNthFun(p.a1, 2)) AND (2 = lengthFun(p.a1))
);
```

# Chapter 6

## Evaluation

---

This chapter presents an evaluation of our extensions to the  $DLV^{DB}$  system. We recall that three main extensions have been implemented: *i*) support to external predicates, *ii*) support to list terms, and *iii*) support to functional terms. All three extensions significantly improve the expressiveness of the supported language, since the possibility to use external predicates, list terms, and functional terms greatly simplify the writing of logical programs. Moreover, we will show that such extensions also significantly reduce the execution times as compared to  $DLV^{DB}$  programs that do not support them.

The following approach will be followed in this chapter: for each extension (i.e., external predicates, list terms, functional terms) a set of test cases is presented; each test case represents a logic program that is solved both with and without the use of the extension under evaluation; then, the two versions of each program (with and without extensions) are compared in terms of execution time to assess their relative merits. Besides performance considerations, we will show that, in many cases, the solutions that make use of our extensions are easier to write and more flexible than those that do not use extensions.

### 6.1 External predicates

We tested the capability to improve usability and efficiency of  $DLV^{DB}$  via external predicates for two real-world problems: number conversion and string similarity computation. All tests presented in the remainder of the chapter have been carried out on a 64-bit Intel Core 2 Duo processor with 4 GB of RAM. The working database of  $DLV^{DB}$  was defined on Microsoft SQL Server 2005.

#### 6.1.1 Number conversion

As a first test case, we considered the problem of transforming integer numbers into their binary representation (*Integer to Binary*). We solve this problem by writing two alternative versions of *Integer to Binary* in  $DLV^{DB}$ : one with and one without external predicates. In both versions, *Integer to Binary* transforms integers stored in an input table to binaries.

The version without external predicates is the following:

```
#maxint=31.  
digit(0).
```

```

digit(1).
binary(V,DT4,DT3,DT2,DT1,DT0) :- digit(DT4), digit(DT3), digit(DT2), digit(DT1), digit(DT0),
                                   N = DT4*16, O = DT3*8, P = DT2*4,
                                   Q = DT1*2, R = DT0*1, V = N+PART3,
                                   PART3 = PART2+PART1, PART2 = O+P, PART1 = Q+R.
int2bin(V,DT4,DT3,DT2,DT1,DT0) :- integer(V), binary(V,DT4,DT3,DT2,DT1,DT0).

```

The version shown above converts numbers with 5 bits only. To convert numbers with a greater number of bits, a different (yet similar) version must be implemented, as we did to perform the tests discussed later in this section.

The version with external predicates is as follows:

```
int2bin(GR,BI) :- integer(GR), #IntToBin(GR, 5, BI).
```

In this case, the number of bits is specified as a input parameter of a stored function *IntToBin*. Therefore, no program rewriting is needed when the number of bits changes. The code of the stored function *IntToBin* implemented on the *SqlServer* database set as working database for *DLV<sup>DB</sup>* is shown below:

```

SET ANSI_NULLS ON
SET QUOTED_IDENTIFIER ON
GO
CREATE FUNCTION [dbo].[IntToBin]
(
    @dec INT,
    @nbit INT
)
RETURNS VARCHAR (20)
AS
BEGIN
    DECLARE @result VARCHAR (20)
    DECLARE @tmp VARCHAR (1)
    DECLARE @quotient INT
    DECLARE @base INT
    DECLARE @remainder INT
    DECLARE @count INT
    SET @quotient=@dec;
    SET @base=2;
    SET @remainder=0;
    SET @count=0;
    SET @result='';
    WHILE @quotient <> 0 AND @count < @nbit
    BEGIN
        SET @remainder=@quotient%@base
        SET @quotient=@quotient/@base
        SET @tmp=CAST(@remainder AS VARCHAR(1))

```

```

        SET @result=@tmp+@result
        SET @count=@count+1
    END;
    IF @count < @nbit
    BEGIN
        WHILE @nbit-@count <> 0
        BEGIN
            SET @result=`0'+@result
            SET @count=@count+1
        END;
    END;
    RETURN @result;
END;

```

In order to measure the scalability of  $DLV^{DB}$  in this test, we considered output binary numbers having from 5 to 16 bits. The obtained results are shown in Figure 6.1, where the  $x$  axis indicates the number of bits and the  $y$  axis represents the execution times expressed in seconds.

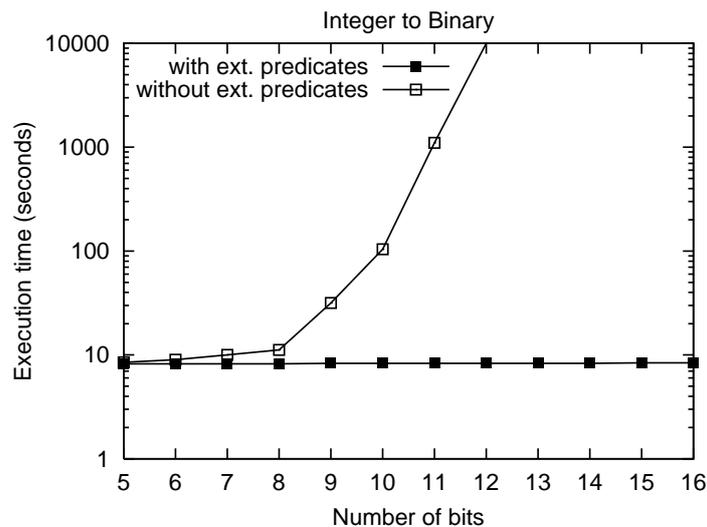


Figure 6.1: Comparison of the execution times of the *Integer to Binary* program with and without external predicates

The graph clearly shows the significant advantage of using external predicates in this context. In fact, the execution time of *Integer to Binary* with external predicates is almost constant because it requires a fixed number of function calls (one for each mark to convert), independently of the number of bits. To the contrary, the version without external predicates must generate all the binary numbers in the admissible range; this explains the exponential growth of the response time.

### 6.1.2 String similarity computation

String similarity computation is an important task in several application areas. In particular, in bioinformatics, it is essential for measuring several parameters between portions of DNA or proteins and to identify frequently repeated patterns. ASP (with some extensions) has already been exploited also in this context, see for example [55].

In this test, we considered the computation of the Hamming distance between pairs of strings, which is at the basis of several similarity measures. It is defined as the number of positions in which the corresponding symbols of two strings of the same length are different. This problem is inherently procedural and, even if a declarative solution for it is possible, this is quite unnatural.

In particular, we considered the following problem, referred to as *Hamming Distances* in the following: given a set of strings, compute the Hamming distance between each string pair. Note that, in classical ASP, in order to properly compute the Hamming distance, each string must be represented as a set of pairs (CHAR, POS); to the contrary, a solution based on external predicates can directly handle the whole string.

We designed two encodings for the problem, one using external predicates and one not; specifically, in the former case input strings are represented as `string(ID, S)`, while in the latter case strings are expressed as `string(ID, CHAR, POS)`. Note that we did not count the time for converting the strings from one format to the other in our tests. In both cases, the output has the form `hd(ID1, ID2, H)`.

The version without external predicates is the following:

```
hd(ID1, ID2, H) :- string(ID1, C1, P1), string(ID2, C2, P2), ID1 < ID2,
                  #count{POS : string(ID1, CHAR1, POS), string(ID2, CHAR2, POS),
                  CHAR1 != CHAR2} = H.
```

The version with external predicates is as follows:

```
hd(ID1, ID2, H) :- string(ID1, S1), string(ID2, S2), ID1 < ID2,
                  #hamming(S1, S2, H).
```

The code of the stored function *hamming* implemented on the *SqlServer* database set as working DB for  $DLV^{DB}$  is reported below:

```
SET ANSI_NULLS ON
SET QUOTED_IDENTIFIER ON
GO
CREATE FUNCTION [dbo].[hamming]
(
    @s1 VARCHAR (250),
    @s2 VARCHAR (250)
)
RETURNS INT
AS
BEGIN
    DECLARE @count INT,
```

```

DECLARE @index INT,
DECLARE @len INT,
DECLARE @s1_sub VARCHAR (250),
DECLARE @s2_sub VARCHAR (250)
IF DATALENGTH(@s1) = DATALENGTH(@s2)
BEGIN
    SET @len=DATALENGTH(@s1)
    SET @index=1
    SET @count=0
    WHILE @index <= @len
    BEGIN
        SET @s1_sub=SUBSTRING(@s1,@index, @index)
        SET @s2_sub=SUBSTRING(@s2,@index, @index)
        SET @index=@index+1
        IF ASCII(@s1_sub) != ASCII(@s2_sub)
        BEGIN
            SET @count=@count+1
        END
    END
END
ELSE
BEGIN
    SET @count=-1
END
RETURN @count
END

```

The experimental results are shown in Figure 6.2 for increasing numbers of input strings. The gain provided by the program that uses external predicates is similar to that we have observed in the integer-to-binary conversion problem, thus confirming the advantage of providing access to stored functions to solve procedural sub-tasks.

As mentioned above, the Hamming distance is at the basis of several similarity measures and is used for the identification of periodic structures in words, which is a fundamental algorithmic task in many practical applications such as DNA sequence analysis [36]. In the following we provide an example focusing on  $K$ -repetitions, a common type of periodic structures that could be identified in a word.

Let  $h(\cdot, \cdot)$  denote the Hamming distance between two words of equal length. A word  $r[1..n]$  is called a  $K$ -repetition of period  $p$ ,  $p \leq n/2$ , iff  $h(r[1..n-p], r[p+1..n]) \leq K$ . In other terms, a word  $r[1..n]$  is a  $K$ -repetition of period  $p$ , if the number of mismatches, i.e. the number of  $i$  such that  $r[i] \neq r[i+p]$ , is at most  $K$ . For example, *ataaattacttact* is a 2-repetition of period 4.

A solution that uses external predicates is the following:

```

substrings(P,W,S1,S2) :- word(W), #substring(W,I,F,S1), L=F+P,
                        #substring(W,I,L,S2), I=P+1, #stringLength(W,L), period(P).
repetition(W,H,P) :- substrings(P,W,S1,S2), #hamming(S1,S2,H), k(K), H <= K.

```

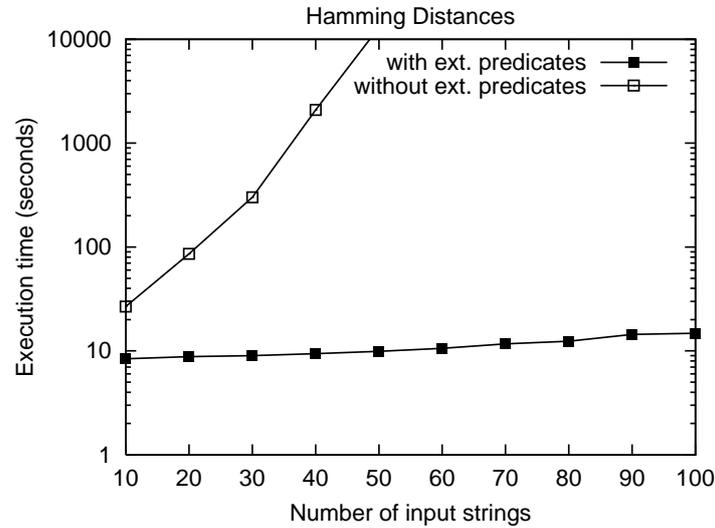


Figure 6.2: Comparison of the execution times of the *Hamming Distances* program with and without external predicates

The first rule derives the two substrings  $S1=r[1..n-p]$  and  $S2=r[p+1..n]$ ; the second rule calculates the Hamming distance between  $S1$  and  $S2$ , to check whether it is lower than or equal to  $K$ .

Without external predicates we cannot extract the two substrings of the word. Therefore, we assume to provide them as input, using a representation similar to that used for the Hamming Distances problem. In particular, a predicate string( $W,P,ID,POS,CHAR$ ) is used for each character of each substring of word  $W$  of period  $P$ . We assume to provide a complete set of string predicates as input, since it is impossible to generate such a set without relying on external predicates.

The solution to the  $K$ -repetition problem, without external predicates, can therefore be written as follows:

```
repetition(W,H,P) :- string(W,P,ID1,P1,C1), string(W,P,ID2,P2,C2),
                    ID1 < ID2, #count{POS : string(W,P,ID1,POS,CHAR1),
                    string(W,P,ID2,POS,CHAR2), CHAR1 != CHAR2} = H, k(K),
                    H <= K.
```

Figure 6.3 compares the execution times of the two versions. As expected, the result is similar to that shown in Figure 6.2, because the execution times of the two programs depend almost entirely on the computation of the Hamming distance, which is faster using external predicates.

Besides the better performance that can be achieved using external predicates, in this last example is even more important the advantage deriving from the possibility to generate the input by program (using the substring predicates) rather than requiring the user to provide it manually.

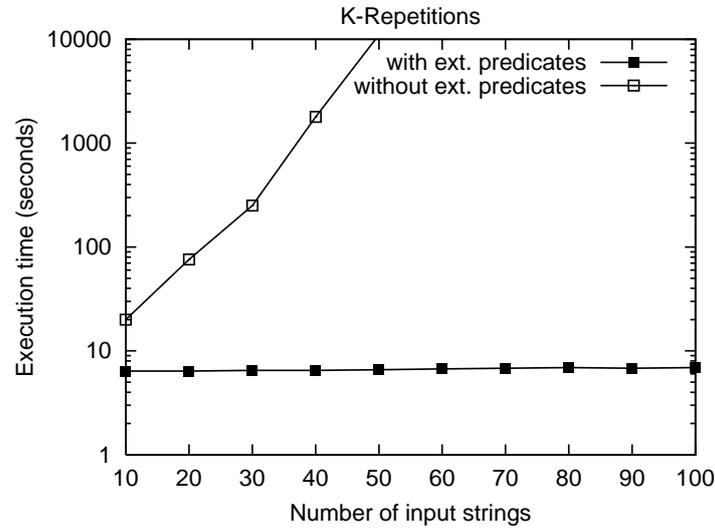


Figure 6.3: Comparison of the execution times of the *K-Repetitions* program with and without external predicates

## 6.2 List terms

In this section we focus on some classical problems (the Towers of Hanoi puzzle, and the graph reachability problem) to show how the use of list terms greatly simplify the writing of logical programs in  $DLV^{DB}$ . Moreover, we will show that the use of list terms allows in many cases to reduce the execution times in a significant way.

### 6.2.1 Towers of Hanoi

The Tower of Hanoi (ToH) problem is a famous puzzle, already introduced in Section 2.3.3. In the ToH problem there are three stacks (or pegs) and  $n$  disks. Initially, all  $n$  disks are on the left-most stack. The goal is to move all  $n$  disks to the right-most stack with the help of the middle stack, by respecting the following rules:

1. it is possible to move one disk at a time;
2. only the top disk on a stack can be moved;
3. a larger disk cannot be placed on top of a smaller one.

It is well known that, for a classic ToH problem with  $n$  disks, the plan of moving all  $n$  disks from the left-most stack to the right-most stack consists of  $2^{n-1}$  moves.

We solve the ToH problem by writing two alternative logical programs in  $DLV^{DB}$ : one with lists and one without lists. The goal is twofold: showing that lists allow to write a more intuitive solution and demonstrating that the version with lists is faster than the one without lists.

We start with the list-based solution. The input of the program is of the form:

```

largest_disc(4).
number_of_moves(15).
initial_state([4,3,2,1],[[],[]]).
goal([],[],[4,3,2,1]).

```

As shown above, the input includes: the identifier of the largest disc (this corresponds to  $n$  since disks are numbered starting from 1); the number of moves needed to reach the goal (as stated above, its value is given by  $2^{n-1}$ ); the initial state of stacks (three lists, each one containing the identifiers of the disks that are placed on a given stack); the goal (i.e., the final state of stacks).

The list-based version of the program is the following:

```

% —— all discs involved ——
disc(X) :- largest_disc(X).
disc(X) :- disc(Y), #succ(X,Y), X != 0.

% —— legal stacks ——
legalStack([]).
legalStack([T]) :- disc(T).
legalStack([T | L]) :- legalStack(L), #head(L,T1), disc(T), T > T1.

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

% From stack one to stack two.
possible_move(I1,[X | S1],S2,S3,S1,[X | S2],S3) :- possible_state(I,[X | S1],S2,S3),
    legalMoveNumber(I), #succ(I,I1),
    legalStack([X | S2]).

% From stack one to stack three.
possible_move(I1,[X | S1],S2,S3,S1,S2,[X | S3]) :- possible_state(I,[X | S1],S2,S3),
    legalMoveNumber(I),#succ(I,I1),
    legalStack([X | S3]).

% From stack two to stack one.
possible_move(I1,S1,[X | S2],S3,[X | S1],S2,S3) :- possible_state(I,S1,[X | S2],S3),
    legalMoveNumber(I), #succ(I,I1),
    legalStack([X | S1]).

% From stack two to stack three.
possible_move(I1,S1,[X | S2],S3,S1,S2,[X | S3]) :- possible_state(I,S1,[X | S2],S3),
    legalMoveNumber(I), #succ(I,I1),
    legalStack([X | S3]).

% From stack three to stack one.
possible_move(I1,S1,S2,[X | S3],[X | S1],S2,S3) :- possible_state(I,S1,S2,[X | S3]),

```

```

legalMoveNumber(I, #succ(I,I1),
legalStack([X | S1]).

% From stack three to stack two.
possible_move(I1,S1,S2,[X | S3],S1,[X | S2],S3) :- possible_state(I,S1,S2,[X | S3]),
legalMoveNumber(I, #succ(I,I1),
legalStack([X | S2]).

%—— actual moves ——

% Choose from the possible moves.
move(I,S1,S2,S3) :- goal(S1,S2,S3), possible_state(I,S1,S2,S3).
move(I,S1,S2,S3) ∨ nomove(I,S1,S2,S3) :-
move(I1,A1,A2,A3), #succ(I,I1),
possible_move(I1,S1,S2,S3,A1,A2,A3).

%—— precisely one move at each step ——
moveStepI(I) :- move(I,_,_,_).
:- legalMoveNumber(I), not moveStepI(I).
:- legalMoveNumber(I), move(I,T1,T2,T3), move(I,TT1,TT2,TT3), T1 != TT1.
:- legalMoveNumber(I), move(I,T1,T2,T3), move(I,TT1,TT2,TT3), T2 != TT2.
:- legalMoveNumber(I), move(I,T1,T2,T3), move(I,TT1,TT2,TT3), T3 != TT3.
legalMoveNumber(0).
legalMoveNumber(I1) :- legalMoveNumber(I), #succ(I,I1), number_of_moves(J), I < J.

```

As shown above, a possible move is represented by a predicate *possible\_move*, featuring seven attributes: the first one represents the move number; the second, third, and fourth attributes represent the states of the three stacks before applying the current move; the fifth, sixth, and seventh attributes represent the last state of the three stacks after the move has been applied. Hence, this predicate is used in the program to model a possible move between any couple of stacks (from stack one to stack two, from stack one to stack three, and so on).

For example, a possible move from the first stack to the second stack is encoded by means of the following rule:

```

possible_move(I1,[X | S1],S2,S3,S1,[X | S2],S3) :- possible_state(I,[X | S1],S2,S3),
legalMoveNumber(I, #succ(I,I1),
legalStack([X | S2]).

```

Roughly, the top element of the first stack can be moved on top of the second stack if: (i) the current stack is admissible, i.e. this state can be reached after applying a sequence of "I" moves (*possible\_state(I, [X | S1], S2, S3)*); (ii) the number "I" is in the range of allowed move numbers (*legalMoveNumber(I)*); (iii) the new resulting configuration for the second stack is legal, i.e. there is no larger disc on top of the smaller one (*legalStack([X | S2])*).

A solution exists if and only if there is a possible move leading to the goal. In this case, starting from the goal, the program proceeds backward to the initial state to single out the full set of moves.

The version without lists is similar, but the status of each stack is not modeled with a list but using an integer. For example, the status [1,2,3,4] is represented with number 1234. This requires to implement some manipulations to access the single cyphers of an integer.

According to the integer representation of stack states, the input of the program without lists is of the form:

```
largest_disc(4).
number_of_moves(15).
initial_state(4321,0,0).
goal(0,0,4321).
```

The program without lists is as follows:

```
% —— all discs involved ——
disc(X) :- largest_disc(X).
disc(X) :- disc(Y), #succ(X,Y), X != 0.

% – legal non-empty stacks, their top element and the rest stack ——
legalstack_top_rest(D,D,0) :- disc(D).
legalstack_top_rest(S,T,B) :- legalstack_top_rest(B,T1,R1), disc(T), T < T1,
    AUX = B * 10, S = AUX + T.

% —— possible moves ——

% From stack one to stack two.
possible_move(I1,S1,S2,L3) :- move(I,L1,L2,L3),
    number_of_moves(J), I < J, #succ(I,I1),
    legalstack_top_rest(L1,X,S1),
    legalstack_top_rest(S2,X,L2).

% From stack one to stack three.
possible_move(I1,S1,L2,S3) :- move(I,L1,L2,L3),
    number_of_moves(J), I < J, #succ(I,I1),
    legalstack_top_rest(L1,X,S1),
    legalstack_top_rest(S3,X,L3).

% From stack two to stack one.
possible_move(I1,S1,S2,L3) :- move(I,L1,L2,L3),
    number_of_moves(J), I < J, #succ(I,I1),
    legalstack_top_rest(L2,X,S2),
    legalstack_top_rest(S1,X,L1).

% From stack two to stack three.
possible_move(I1,L1,S2,S3) :- move(I,L1,L2,L3),
    number_of_moves(J), I < J, #succ(I,I1),
    legalstack_top_rest(L2,X,S2),
```

```

legalstack_top_rest(S3,X,L3).

% From stack three to stack one.
possible_move(I1,S1,L2,S3) :- move(I,L1,L2,L3),
    number_of_moves(J), I < J, #succ(I,I1),
    legalstack_top_rest(L3,X,S3),
    legalstack_top_rest(S1,X,L1).

% From stack three to stack two.
possible_move(I1,L1,S2,S3) :- move(I,L1,L2,L3),
    number_of_moves(J), I < J, #succ(I,I1),
    legalstack_top_rest(L3,X,S3),
    legalstack_top_rest(S2,X,L2).

%—— actual moves ——

% Choose from the possible moves.
move(I,A1,A2,A3) v nomove(I,A1,A2,A3) :- possible_move(I,A1,A2,A3).

%—— one configuration at each step ——
:- move(I,L1,L2,L3), move(I,M1,M2,M3), L1 != M1.
:- move(I,L1,L2,L3), move(I,M1,M2,M3), L2 != M2.
:- move(I,L1,L2,L3), move(I,M1,M2,M3), L3 != M3.

```

We compared the execution times of the two versions of the program by varying the number of disks from 3 to 6. Figure 6.4 reports the results of such comparison, where the  $x$  axis indicates the number of disks, and the  $y$  axis indicates the execution times expressed in seconds.

The graph shows that the execution time of the version with lists is always lower than that without lists. In particular, with 3 disks the execution time passes from about 19 seconds with lists, to about 36 seconds without lists; with 4 disks the execution time with lists is 29 seconds, without lists is 53; with 5 disks the execution time passes from 140 to 227 seconds; finally, with 6 disks the execution time of the list-based version is 2240 seconds, while the version without lists failed to complete the execution due to out of memory.

The results above demonstrate that the list-based version is more efficient compared to the version without lists. This is due to the fact that in the version without lists the status of each stack is modeled using an integer, which requires to perform several manipulations each time we need to access the single parts (integer cyphers) of a stack status. Besides performance considerations, it is worth noticing that the list-based version is more intuitive, as it allows programmers to express the solution without relying on tricky representations of stack states.

## 6.2.2 Graph reachability

Graph reachability is a classic problem in computer science with many applications in a lot of relevant real world applications, ranging from databases to product configurations and networks. Given a directed graph  $G = (V, E)$ , a couple of vertices  $i$  and  $j$  of  $V$  belongs to the *Reachability* relation, also called the transitive closure relation, if there is a non-empty sequence of edges in  $E$

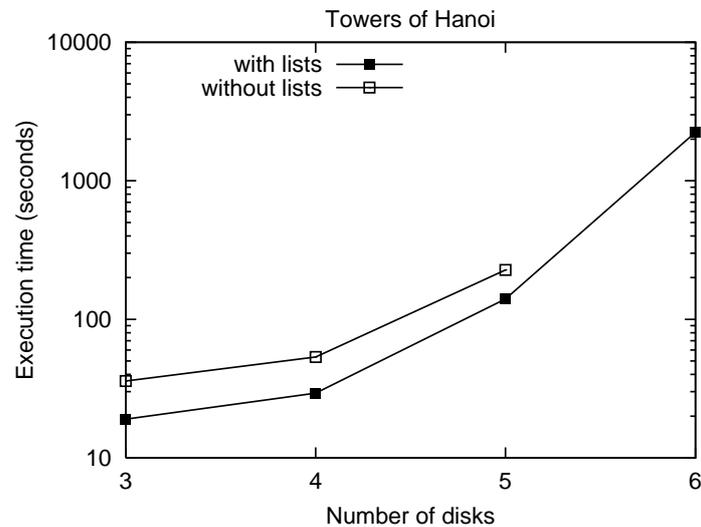


Figure 6.4: Comparison of the execution times of the *Towers of Hanoi* program with list and without lists, by varying the number of disks from 3 to 6; the execution time for 6 disks is shown only for the version with lists because the version without lists failed to complete due to out of memory

that forms a path from  $i$  to  $j$ . We consider here the problem of determining all pairs of reachable nodes in  $G$ . Such problem is solved first using  $DLV^{DB}$  without lists, and then using  $DLV^{DB}$  with lists.

The code of the reachability program without lists is as follows:

```
reaches(X,Y) :- edge(X,Y).
reaches(X,Y) :- reaches(X,Z), edge(Z,Y).
```

The program solves the problem recursively. The first rule states that a vertex  $Y$  is reachable from a vertex  $X$  if there is an edge from  $X$  to  $Y$ . The second rule, states that  $Y$  is reachable from  $X$  if there is a vertex  $Z$  that is reachable from  $X$ , and there is an edge from  $Z$  to  $Y$ . The input graph is represented by the set of its edges. The predicate used to define edges is a binary predicate  $edge$  where  $edge(i, j)$  means that there is a directed edge going from  $i$  to  $j$ , as in the following example:

```
edge(1,3). edge(3,4). edge(3,5). edge(4,2). edge(2,5).
```

The output of the program is an atom  $reaches(i, j)$  for each vertex  $j$  that is reachable from a vertex  $i$ . Given the set of  $edge$  facts of the example above, the output of the program is:

```
{reaches(1,2), reaches(1,3), reaches(1,4), reaches(1,5), reaches(2,5), reaches(3,2), reaches(3,4),
reaches(3,5), reaches(4,2), reaches(4,5)}
```

The same problem can be solved using lists as follows:

```

path([X,Z]) :- edge(X,Z).
path([X | L]) :- edge(X,Y), path(L), #head(L,Y), #member(X,L,false).
reaches(X,Y) :- path(L), #head(L,X), #last(L,Y).

```

Also this version solves the problem recursively. The first rule builds a simple path as a list of two vertices 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` 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. The third rule selects the head and the tail of a path, where the tail represents a node reachable from the head. It is easy to check that the output of this program is the same of the program that does not use lists.

Clearly, in general the number of paths in a graph grows exponentially with the number of vertices. Here we are just interested in assessing the overhead (if any) of using lists for solving a problem. In fact, we use only graphs having a particular structure: each vertex has exactly one outgoing edge which points to the next vertex in clockwise direction; in other terms, the graph is a circle having  $n$  vertices and  $n - 1$  directed edges. We use this particular graph structure for two reasons: (i) we can obtain a uniform measurement with increasing input size; (ii) the circle structure implies that there are long paths (the maximum length being  $n - 1$ ) and thus long lists to represent them, which is useful to stress the system.

We then compared the execution times of the two versions of the program by varying the number of vertices of the input graph from 10 to 70. The results are reported in Figure 6.5, where the  $x$  axis indicates the number of vertices, and the  $y$  axis indicates the execution times.

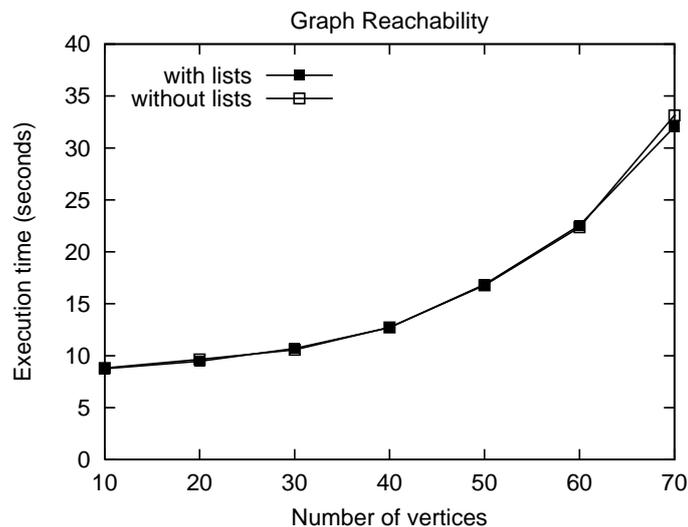


Figure 6.5: Comparison of the execution times of the *Graph Reachability* program with list and without lists, by varying the number of vertices of the input graph from 10 to 70

The graph shows that the execution time of the two versions are about the same for every number of vertices in the input graph, ranging from about 9 seconds for 10 vertices to about 32

seconds for 70 vertices. In this case the list-based version of the program does not produce overhead in execution time, but allows to obtain additional results compared to the non-list version. Indeed, the list-version permits to calculate not only the reachability, but also the path (exactly one for the considered graphs) that links a vertex with one another. Given the same input as in the example above, we obtain the following reachability paths by filtering the answer set generated by the program using the *path* predicate:

```
{path([1,3,4,2,5]), path([1,3,4,2]), path([1,3,4]), path([1,3,5]), path([1,3]), path([2,5]), path([3,4]),
path([3,4,2]), path([3,4,2,5]), path([3,5]), path([4,2,5]), path([4,2])}
```

Under the strict assumption that all vertices are named using non-zero integers having the same number of cyphers (e.g., vertices are named from 1 to 9, or from 10 to 99), it is possible to write a non-list-based version of the reachability program that also calculates the path linking a vertex with one another (*Path Calculator*). The approach is similar to that used to implement the non-list-based version of the Towers of Hanoi problem: we represent each path as an integer like 1234, to indicate the presence of a path formed by vertices 1, 2, 3 and 4. As for the Towers of Hanoi problem, this requires to implement some manipulations to access the single cyphers of an integer.

Assuming that vertices are named from 1 to 9, the code of the path calculator program without lists is as follows:

```
path(S,X,Y) :- edge(X,Y), AUX = X * 10, S = AUX + Y.
path(S,X,Y) :- path(B,X,Z), edge(Z,Y), AUX = B * 10, S = AUX + Y.
```

We compared the execution times of the non-list-based version of the path calculator with the list-based one, which is shown below for the reader convenience:

```
path([X,Z]) :- edge(X,Z).
path([X | L]) :- edge(X,Y), path(L), #head(L,Y), #member(X,L,false).
```

This time comparison has been done for input graphs having a number of vertices varying from 3 to 9. Input graphs have been randomly generated and do not contain cycles. The results are shown in Figure 6.6.

The results above demonstrate that the list-based version is more efficient compared to the version without lists. As for the Towers of Hanoi problem, this is due to the fact that the version without lists performs some arithmetic operations that are not required by the list-based one. We finally remind that the list-based version works with any input, while the non-list-based one works only under the strict assumptions discussed above.

### 6.3 Functional terms

Support to functional terms represents an important added value to  $DLV^{DB}$ , since they are a very convenient means for generating domains and objects, allowing a more natural representation of problems in such domains. Moreover, functional terms allow logical programs to significantly reduce execution times as compared to  $DLV^{DB}$  programs that do not use them. We focus on the

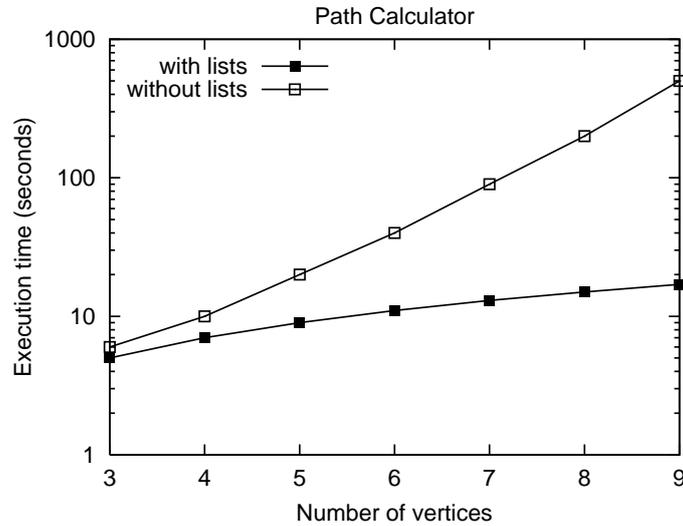


Figure 6.6: Comparison of the execution times of the *Path Calculator* program with list and without lists, by varying the number of vertices of the input graph from 3 to 9

performance aspect by discussing a test program written in the context of database repair.

### 6.3.1 Database repair

In the relational data model, an inclusion dependency  $R1(X) \subseteq R2(Y)$  is a dependency between an attribute set  $X$  in a relation schema  $R1$  and an attribute set  $Y$  in a relation schema  $R2$ . In other words, an inclusion dependency is a statement saying that some columns of a relation are contained in other columns. A foreign key constraint is an example of inclusion dependency.

If an inclusion dependency is violated, a *Database Repair* operation must be performed. For example, let's assume to have two relations  $d(X)$  and  $p(X, Y)$ , and that there is a foreign key constraint between the attributes  $X$  of  $d$  and  $p$ . The constraint is violated whenever there is an instance  $d(a)$  but there is not an instance  $p(a, -)$ . If this violation occurs, the repair is performed by generating an instance  $p(a, f(a))$ , where  $f(a)$  is a new value uniquely associated to  $a$ , that is, a value that is not already present in the domain of attribute  $Y$ .

Functional terms can be used to generate the new values that are needed for the repair, by writing a rule like the following one:

```
p(X,f(X)) :- d(X).
```

Without functional terms the creation of a new value is more expensive because, after guessing it, we must also ensure that it is unique in the attribute's domain:

```
p(X,Y) ∨ np(X,Y) :- d(X), u(Y).
:- u(Y), #count{X: p(X,Y)} > 1.
:- u(Y), #count{X: p(X,Y)} < 1.
:- p(X,Y1), p(X1,Y), X==X1, Y!=Y1.
:- p(X,Y1), p(X1,Y), X!=X1, Y==Y1.
```

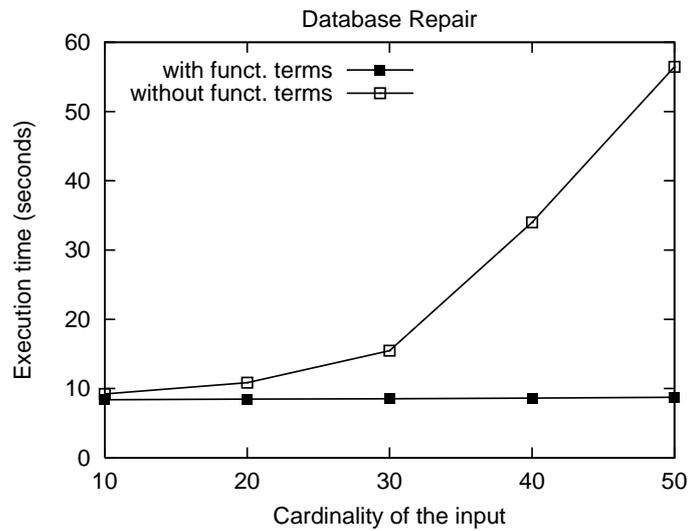


Figure 6.7: Comparison of the execution times of the *Database Repair* test with functional terms and without functional terms, by varying the cardinality of the input from 10 to 50

We have compared the execution times of the first version, which uses functional terms, with those of the second version, which does not use functional terms. The comparison, whose results are shown in Figure 6.7, has been done by varying the cardinality of the input, i.e., the number of instances of relation  $d$ , from 10 to 50.

As shown in the figure, the execution times of the program with functional terms are independent from the input size, hovering around 9 seconds in all cases. The execution times of the program without functional terms exponentially increase with the input size, going from 10 seconds when the input size is equal to 50, to 57 seconds when the size of the input is 100.

# Chapter 7

## Conclusions

---

This thesis focused on  $DLV^{DB}$ , an ASP system allowing the instantiation of logic programs directly on databases to combine the expressive power of DLP with the efficient data management features of DBMSs. In its early implementation,  $DLV^{DB}$  did not support external predicates, list terms, and function symbols, which are of fundamental importance to enhance knowledge representation and reasoning capabilities of a DLP language. Thus, the main goal of this thesis was to extend the  $DLV^{DB}$  system to let it support the above mentioned language constructs, in order to improve its knowledge modelling power.

In particular, the main results of this thesis can be summarized as follows:

- The  $DLV^{DB}$  system has been extended to provide full support to external predicates. Since  $DLV^{DB}$  transforms logical programs into SQL statements to enable database-oriented processing, we implemented external predicates by calls to database stored functions.
- $DLV^{DB}$  has been extended to support programs with list terms. This has been obtained through a rewriting of the rules using suitable external predicates, i.e., programs containing list terms are automatically rewritten to contain only terms and external predicates.
- $DLV^{DB}$  has been extended to support programs with functional symbols. Similarly to list terms, rules are rewritten by replacing each functional term definition with calls to external predicates.
- A library of database stored functions for lists manipulation has been realized to facilitate the use of list terms in  $DLV^{DB}$  through the use of external functions.
- Some experiments have been performed to evaluate our extensions to the  $DLV^{DB}$  system. The experimental results show that such extensions significantly reduce the execution times as compared to  $DLV^{DB}$  programs that do not exploit them.

As for future work, we plan to:

- provide support to additional language constructs, such a sets, whose evaluation strategy could be similar to that used for lists;
- introduce data typing features to allow the handling of complex domains and to enable the manipulation of objects;
- investigate strategies to allow not-finite domain rules.

# Appendix A

## List and Functional Terms Manipulation Library

---

This appendix provides the code of the database stored functions implemented for list and functional terms manipulation, whose use has been introduced in Chapter 5. We provide a detailed description only for SQLServer functions. Those implemented for the other DBMSs are similar and their description is avoided.

### HEAD

This function receives a list and returns its head.

```
CREATE FUNCTION [dbo].[head]
( @list varchar(max) )
RETURNS varchar(max)
AS
BEGIN
    DECLARE @tmpList varchar(max), @pos int, @bracketsOpen int,
            @bracketsClosed int, @length int, @found int
    IF (SUBSTRING(@list, 1, 1)='['
        AND SUBSTRING(@list, LEN(@list), 1)=']')
        SET @list=SUBSTRING(@list, 2, LEN(@list)-2)
    ELSE
        RETURN NULL
    SET @length=LEN(@list)
    SET @bracketsOpen=0
    SET @bracketsClosed=0
    SET @pos=1
    SET @found=0
    WHILE (@pos<=@length AND @found=0)
    BEGIN
        IF SUBSTRING(@list,@pos,1)='['
            BEGIN
                SET @bracketsOpen=@bracketsOpen+1
                SET @pos=@pos+1
            END
        ELSE
            IF SUBSTRING(@list,@pos,1)=']'
```

```

        BEGIN
            SET @bracketsClosed=@bracketsClosed+1
            SET @pos=@pos+1
        END
    ELSE
        IF ((SUBSTRING(@list,@pos,1)=',' )
            AND (@bracketsOpen-@bracketsClosed=0))
        BEGIN
            SET @found=1
            SET @pos=@pos-1
        END
    ELSE
        SET @pos=@pos+1
END
IF (@bracketsOpen-@bracketsClosed<>0) RETURN NULL
IF (LEN(SUBSTRING ( @list, @pos+1, @lung-@pos+1))<>0)
BEGIN
    SET @tmpList=' '+SUBSTRING ( @list, @pos+2, @lung-@pos+2)+' ]'
    IF (dbo.[isList](@tmpList)=0)
        RETURN NULL
END
SET @list = SUBSTRING ( @list, 1 , @pos)
RETURN @list
END

```

## TAIL

This function receives a list and returns its tail.

```

CREATE FUNCTION [dbo].[tail]
( @list varchar(max) )
RETURNS varchar(max)
AS
BEGIN
    DECLARE @pos int, @bracketsOpen int, @bracketsClosed int,
            @length int, @found int
    IF (SUBSTRING(@list,1,1)=' [' AND SUBSTRING(@list,LEN(@list),1)=' ]')
        SET @list=SUBSTRING (@list,2,LEN(@list)-2)
    ELSE RETURN NULL
    SET @length=LEN(@list)
    SET @bracketsOpen=0
    SET @bracketsClosed=0
    SET @pos=1
    SET @found=0
    WHILE (@pos<=@length AND @found=0)
    BEGIN
        IF SUBSTRING(@list,@pos,1)=' ['
        BEGIN
            IF (@bracketsOpen-@bracketsClosed=0)
                SET @bracketsOpen=@bracketsOpen+1

```

```

        ELSE RETURN NULL;
        SET @pos=@pos+1
    END
    ELSE
        IF SUBSTRING(@list,@pos,1)=']'
        BEGIN
            SET @bracketsClosed=@bracketsClosed+1
            SET @pos=@pos+1
        END
        ELSE
            IF ((SUBSTRING(@list,@pos,1)=',' )
                AND (@bracketsOpen-@bracketsClosed=0))
            BEGIN
                SET @found=1
                SET @pos=@pos+1
            END
            ELSE SET @pos=@pos+1
    END
    SET @length=@length-@pos+1
    IF (@length>0)
        SET @list=' '+SUBSTRING(@list,@pos,@length)+' ]'
    ELSE SET @list=' []'
    IF (dbo.[isList](@list)=0)
        RETURN NULL
    RETURN @list
END

```

## LAST

This function receives a list and returns its last element.

```

CREATE FUNCTION [dbo].[last]
( @list varchar(max) )
RETURNS varchar(max)
AS
BEGIN
    DECLARE @pos int, @brackets int, @length int, @found int
    IF (SUBSTRING(@list, 1, 1)='['
        AND SUBSTRING(@list, LEN(@list), 1)=']')
        SET @list=SUBSTRING(@list, 2, LEN(@list)-2)
    ELSE
        RETURN NULL
    SET @length=LEN(@list)
    SET @brackets=0
    SET @pos=@length
    SET @found=0
    WHILE (@pos>=1 AND @found=0)
    BEGIN
        IF SUBSTRING(@list,@pos,1)='['
        BEGIN

```

```

        SET @brackets=@brackets-1
        SET @pos=@pos-1
    END
    ELSE
        IF (SUBSTRING(@list,@pos,1)=']')
        BEGIN
            SET @brackets=@brackets+1
            SET @pos=@pos-1
        END
        ELSE
            IF ((SUBSTRING(@list,@pos,1)=',' ) AND (@brackets=0))
            BEGIN
                SET @found=1
                SET @pos=@pos+1
            END
            ELSE SET @pos=@pos-1
        END
    IF (@brackets<>0) RETURN NULL
    SET @list=SUBSTRING(@list,@pos,@length-@pos+1)
    RETURN @list
END

```

## MEMBER

This function receives a list  $L$  and an element  $E$ , and checks whether  $L$  contains  $E$ .

```

CREATE FUNCTION [dbo].[member]
(
    @element varchar(max),
    @list varchar(max)
)
RETURNS varchar(5)
AS
BEGIN
    DECLARE @pos int,@startPos int, @bracketsOpen int,
            @bracketsClosed int, @length int
    IF (SUBSTRING(@list,1,1)='[' AND SUBSTRING(@list,LEN(@list),1)=']')
        SET @list=SUBSTRING(@list,2,LEN(@list)-2)
    ELSE RETURN NULL
    SET @startPos=1
    SET @length=LEN(@list)
    SET @bracketsOpen=0
    SET @bracketsClosed=0
    SET @pos=1
    WHILE (@pos<=@length)
    BEGIN
        IF SUBSTRING(@list,@pos,1)='['
        BEGIN
            IF (@bracketsOpen-@bracketsClosed=0)
                SET @bracketsOpen=@bracketsOpen+1

```

```

        ELSE RETURN NULL;
    END
    ELSE
        IF SUBSTRING(@list,@pos,1)=']'
            SET @bracketsClosed=@bracketsClosed+1
        ELSE
            IF ((SUBSTRING(@list,@pos,1)=',' )
                AND (@bracketsOpen-@bracketsClosed=0))
            BEGIN
                IF (SUBSTRING(@list,@startPos,
                    @pos-@startPos)=@element)
                    RETURN 'true'
                SET @startPos=@pos+1
            END
            SET @pos=@pos+1
        END
    END
    IF (@bracketsOpen-@bracketsClosed<>0) RETURN NULL
    IF (SUBSTRING(@list,@startPos,@pos-@startPos)=@element)
        RETURN 'true'
    RETURN 'false'
END

```

## MEMBERNTH

This function receives a list  $L$  and an index  $I$ , and returns the  $I$ th element of  $L$ .

```

CREATE FUNCTION [dbo].[memberNth]
(
    @list varchar(max),
    @index int
)
RETURNS varchar(max)
AS
BEGIN
    DECLARE @element varchar(255), @car varchar(255), @nTok int,
            @brackets int, @length int, @inConstant int,
            @inTok int, @sPos int, @ePos int, @pos int,
    IF([dbo].[isList](@list)=1)
        SET @list=SUBSTRING (@list, 2 , LEN(@list)-2)
        ELSE RETURN NULL;
    set @length=LEN(@list)
    IF (@length=0) RETURN @list
    SET @nTok=1
    SET @sPos=@lung+1
    SET @ePos=@lung
    SET @pos=1
    SET @brackets=0
    SET @inConstant=0
    WHILE(@nTok <= @index)
    BEGIN

```

```

SET @inTok=1
IF (@nTok=@index) SET @sPos=@pos
WHILE (@inTok=1 AND @pos<=@length)
BEGIN
    SET @car=substring(@list,@pos,1)
    IF (@car='') SET @inConstant= (@inConstant+1)
    ELSE IF (@car='[' AND @inConstant=0)
        SET @brackets=@brackets+1
    ELSE IF (@car=']' AND @inConstant=0)
        SET @brackets=@brackets-1
    ELSE IF (@car=',' AND @brackets=0 AND @inConstant=0)
        BEGIN
            IF (@nTok=@index) SET @ePos=@pos-1
            SET @inTok=0
            SET @nTok=@nTok+1
        END
    SET @pos=@pos+1
END
IF (@pos=@length+1) SET @nTok=@nTok+1
END
SET @element=rtrim(ltrim(substring(@list, @sPos, @ePos-@sPos+1)))
RETURN @element
END

```

## LENGTH

This function receives a list and returns its length.

```

CREATE FUNCTION [dbo].[length]
( @list varchar(max) )
RETURNS int
AS
BEGIN
    DECLARE @pos int, @bracketsOpen int, @bracketsClosed int,
            @length int, @count int
    SET @count=0
    IF (SUBSTRING(@list,1,1)='[' AND SUBSTRING(@list,LEN(@list),1)=']')
        SET @list=SUBSTRING (@list,2,LEN(@list)-2)
    ELSE RETURN NULL
    SET @length=LEN(@list)
    IF (@length<>0)
        SET @count=1
        SET @bracketsOpen=0
        SET @bracketsClosed=0
        SET @pos=1
        WHILE (@pos<=@length)
            BEGIN
                IF SUBSTRING(@list,@pos,1)='['
                    BEGIN
                        IF (@bracketsOpen-@bracketsClosed=0)

```

```

        SET @bracketsOpen=@bracketsOpen+1
    ELSE RETURN NULL;
END
ELSE
    IF SUBSTRING(@list,@pos,1)=']'
    BEGIN
        SET @bracketsClosed=@bracketsClosed+1
    END
    ELSE
        IF ((SUBSTRING(@list,@pos,1)=',' )
            AND (@bracketsOpen-@bracketsClosed=0))
        BEGIN
            SET @count=@count+1
        END
        SET @pos=@pos+1
    END
    IF (@bracketsOpen-@bracketsClosed<>0) RETURN NULL
    RETURN @count
END

```

## PACK

This function receives a term  $H$  and a list  $L$ , and returns the list  $[H|L]$ .

```

CREATE FUNCTION [dbo].[pack]
(
    @h varchar(max), @list varchar(max)
)
RETURNS varchar(max)
AS
BEGIN
    DECLARE @start varchar(max), @end varchar(max),
            @separator varchar(2)
    SET @start=SUBSTRING(@h, 1, 1)
    SET @end=SUBSTRING(@h, LEN(@h), 1)
    IF (isList(@list)=1)
    BEGIN
        SET @list=SUBSTRING(@list, 2, LEN(@list)-2)
        SET @separator=', '
        IF (@h='[]') SET @h=''
        IF ((len(@h)=0) OR (len(@list)=0)) SET @separator=''
        RETURN '['+@h+@separator+@list+']'
    END
    RETURN NULL
END

```

## UNPACK

This function receives a list and returns true if it is non-empty.

```
CREATE FUNCTION [dbo].[unpack]
(
    @list varchar(max)
)
RETURNS varchar(6)
AS
BEGIN
    IF (dbo.[isList](@list)=1)
        IF (@list=' []')
            RETURN 'false'
        ELSE RETURN 'true'
    ELSE RETURN NULL
END
```

## ISLIST

This function receives a string and checks whether it represents a list.

```
CREATE FUNCTION [dbo].[isList]
(
    @list varchar(max)
)
RETURNS int
AS
BEGIN
    DECLARE @pos int, @bracketsOpen int, @bracketsClosed int,
            @length int
    IF (SUBSTRING(@list,1,1)=' [' AND SUBSTRING(@list,LEN(@list),1)=' ]')
        SET @list=SUBSTRING(@list, 2, LEN(@list)-2)
    ELSE RETURN 0
    SET @length=LEN(@list)
    SET @bracketsOpen=0
    SET @bracketsClosed=0
    SET @pos=1
    WHILE (@pos<=@length)
    BEGIN
        IF SUBSTRING(@list,@pos,1)=' ['
            BEGIN
                IF (@bracketsOpen-@bracketsClosed=0)
                    SET @bracketsOpen=@bracketsOpen+1
                ELSE RETURN 0;
            END
        ELSE IF SUBSTRING(@list,@pos,1)=' ]'
            BEGIN
                SET @bracketsClosed=@bracketsClosed+1
            END
        SET @pos=@pos+1
    END
```

```

        END
        SET @pos=@pos+1
    END
    IF (@bracketsOpen-@bracketsClosed<>0) RETURN 0
    RETURN 1
END

```

## CAT

This function receives two strings and returns their concatenation.

```

CREATE FUNCTION [dbo].[cat]
(
    @args0 varchar(max),
    @args1 varchar(max)
)
RETURNS varchar(max)
AS
BEGIN
    RETURN @args0+@args1
END

```

## GETFUNCTIONNAME

This function receives a functional term and returns its functor.

```

CREATE FUNCTION [dbo].[getFunctionName]
(
    @args0 varchar(max),
)
RETURNS varchar(max)
AS
BEGIN
    DECLARE @startPos int
    SET @startPos = CHARINDEX('(', @args0, 1)
    IF (@startPos!=0)
    IF (SUBSTRING(@args0, @startPos, 1)='('
        AND SUBSTRING(@list, LEN(@args0), 1)='') )
        RETURN SUBSTRING(@args0, 1, @startPos-1)
    RETURN NULL
END

```

## LENGTHFUN

This function returns the number of terms in a functional term received as input.

```

CREATE FUNCTION [dbo].[lengthFun]
(
    @args0 varchar(max),
)
RETURNS varchar(max)
AS
BEGIN
    DECLARE @pos int, @brackets int, @length int,
            @n int, @startPos int
    SET @n=0
    SET @startPos = CHARINDEX('(', @list, 1)
    IF (@startPos!=0)
        IF (SUBSTRING(@list, @startPos, 1)='('
            AND SUBSTRING(@list, LEN(@list), 1)='')
            SET @list=SUBSTRING ( @list, 2 , LEN(@list)-2)
        ELSE
            RETURN NULL
    ELSE RETURN NULL
    SET @length=LEN(@list)
    IF (@length<>0)
    BEGIN
        SET @n=1
        SET @brackets=0
        SET @pos=@startPos
        WHILE (@pos<=@length)
        BEGIN
            IF (SUBSTRING(@list,@pos,1)='(')
            BEGIN
                SET @brackets=@brackets+1
            END
            ELSE IF (SUBSTRING(@list,@pos,1)='')
            BEGIN
                SET @brackets=@brackets-1
            END
            ELSE IF ((SUBSTRING(@list,@pos,1)=',' ) AND (@brackets=0))
                SET @n=@n+1
            SET @pos=@pos+1
        END
    END
    RETURN @n
END

```

## Bibliography

---

- [1] A. Aggoun, D. Chan, P. Dufresne, et al. Eclipse user manual release 5.0, 2000.
- [2] C. Anger, K. Konczak, and T. Linke. NoMoRe: A system for non-monotonic reasoning under answer set semantics. In W. F. T. Eiter and M. Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, pages 406–410. Springer, 2001.
- [3] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, California, 1988.
- [4] C. Aravidan, J. Dix, and I. Niemela. Dislop: A research project on disjunctive logic programming. *AICommunications*, 10(3/4):151–165, 1997.
- [5] C. Aravidan, J. Dix, and I. Niemela. Dislop: Towards a disjunctive logic programming system. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, pages 342–353. Springer, LNAI, 1997.
- [6] R. Bihlmeyer, W. Faber, C. Koch, N. Leone, C. Mateis, and G. Pfeifer. dl<sub>v</sub> – an overview. In U. Egly and H. Tompits, editors, *Proceedings of the 13th Workshop on Logic Programming (WLP'98)*, pages 65–67, Vienna, Austria, October 1998.
- [7] F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50:333–361, 2007.
- [8] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in asp: Theory and implementation. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, pages 407–424, 2008.
- [9] F. Calimeri and G. Ianni. External sources of computation for answer set solvers. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005)*, pages 105–118, 2005.
- [10] W. Chen and D. S. Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.
- [11] P. Cholewinski, A. Marek, V. Mikitiuk, and M. Truszczyński. Computing with default logic. *Journal of Artificial Intelligence*, 112(1/2):105–146, 1999.

- 
- [12] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dl<sub>v</sub> System: Model Generator and Application Frontends. In F. Bry, B. Freitag, and D. Seipel, editors, *Proceedings of the 12th Workshop on Logic Programming (WLP'97), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.
- [13] E. De Francesco, G. Di Santo, L. Palopoli, and S. Rombo. A summary of genomic databases: Overview and discussion. In A. Sidhu and T. Dillon, editors, *Biomedical Data and Applications*, volume 224 of *Studies in Computational Intelligence*, pages 37–54. Springer, 2009.
- [14] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System Description: DLV. In T. Eiter, W. Faber, and M. Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in *Lecture Notes in AI (LNAI)*, pages 409–412. Springer Verlag, September 2001.
- [15] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in DLV. In M. de Vos and A. Proveti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 274–288, Messina, Italy, Sept. 2003. Online at <http://CEUR-WS.org/Vol-78/>.
- [16] D. East and M. Truszczynski. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, pages 138–153. Springer Verlag, LNAI 2174, 2001.
- [17] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.
- [18] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.
- [19] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In I. O. Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDL'99)*, pages 135–139. Prolog Association of Japan, September 1999.
- [20] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in *Lecture Notes in AI (LNAI)*, pages 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.
- [21] W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, Aug. 2001. Morgan Kaufmann Publishers.

- 
- [22] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *In Proc. of JELIA 2004*, pages 200–212, 2004.
- [23] W. Faber and G. Pfeifer. DLV homepage, since 1996. <http://www.dbai.tuwien.ac.at/proj/dlv/>.
- [24] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *TPLP*, 5(1-2):45–74, 2005.
- [25] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.
- [26] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *Clasp* : A conflict-driven answer set solver. In *Int. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Tempe, AZ, USA*, pages 260–265, 2007.
- [27] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In *LPNMR*, pages 136–148, 2007.
- [28] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Jan. 2007. 386–392.
- [29] M. Gelfond and V. 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.
- [30] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
- [31] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Faithful and effective querying of RDF ontologies using DLV<sup>DB</sup>. In *Proc. of the 4th International Workshop on Answer Set Programming (ASP'07)*, Porto, Portugal, 2007.
- [32] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Some experiments on the usage of a deductive database for RDFS querying and reasoning. In *Proc. of the 4th Workshop on Semantic Web Applications and Perspectives (SWAP 2007)*, Bari, Italy, 2007.
- [33] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Efficiently querying RDF(S) ontologies with Answer Set Programming. *Journal of Logic and Computation (Special issue)*., To appear.
- [34] G. Ianni, C. Panetta, and F. Ricca. Specification of assessment-test criteria through ASP specifications. In *Proc. of Answer Set Programming: Advances in Theory and Implementation (ASP'05)*, Bath, UK, 2005.
- [35] T. Janhunen, I. Niemela, P. Simons, and J.-H. You. Partiality and disjunctions in stable model semantics. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.

- [36] M.-Y. Kao, editor. *Encyclopedia of Algorithms*. Springer, 2008.
- [37] H. Kautz and B. Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, 1992.
- [38] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *ICLP*, pages 451–465, 2003.
- [39] N. Leone, W. Faber, A. Bria, F. Calimeri, G. Catalano, S. Cozza, T. Dell'Armi, G. Greco, G. Ianni, G. Ielpa, M. Maratea, C. Panetta, S. Perri, F. Ricca, F. Scarcello, G. Terracina, G. Pfeifer, T. Eiter, and G. Gottlob. DLV: An Advanced System for Knowledge Representation and Reasoning. In *ALP Newsletter*, volume 20, n. 3-4. Editor: E. Pontelli, Area Editor: R. Bagnara, 2007.
- [40] N. Leone, V. Lio, C. Panetta, and G. Terracina. DLV<sup>DB</sup>: a system for the efficient evaluation of datalog programs directly on databases. *Intelligenza Artificiale*, 2006. To appear.
- [41] N. Leone, V. Lio, and G. Terracina. Dlvdb: Bridging the gap between asp systems and dbmss, 2004.
- [42] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, July 2006.
- [43] N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, 1997.
- [44] V. Lifschitz. Answer Set Planning. In D. D. Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, Nov. 1999. The MIT Press.
- [45] F. Lin and Y. Zhao. Assat: Computing answer sets of a logic program by sat solvers. *AAAI-02*, To appear, 2002.
- [46] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.
- [47] T. Linke. Graph theoretical characterization and computation of answer sets. In B. Nebel, editor, *International Joint Conference on Artificial Intelligence*, pages 641–645, 2001.
- [48] T. Linke, C. Anger, and K. Konczak. More on nomore. In G. Ianni and S. Flesca, editors, *Eighth European Workshop on Logics in Artificial Intelligence (JELIA'02)*, volume 2424, 2002.
- [49] J. Lloyd. *Foundation of Logic Programming*. Springer Verlag, New York, NY, 1987.
- [50] J. Lobo, J. Minker, and A. Rajasekar. *Foundation of Disjunctive Logic Programming*. Mit Press, 1992.
- [51] D. Loveland. Near-horn PROLOG. In *4th International Conference on Logic Programming (ICLP'87)*, pages 456–459, 1987.

- [52] M. Werner. *davinci v2.1.x* online documentation, 1998.
- [53] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.
- [54] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. In *Proc. of the 8th Int. Workshop on Non-Monotonic Reasoning (NMR'2000)*, Colorado, USA, April 2000.
- [55] L. Palopoli, S. Rombo, and G. Terracina. Flexible pattern discovery with (extended) disjunctive logic programming. In *Proc. of 15th International Symposium on Methodologies for Intelligent Systems (ISMIS 2005)*, pages 504–513, Saratoga Springs, New York, USA, 2005. Lecture Notes in Artificial Intelligence (3488), Springer-Verlag.
- [56] T. C. Przymusiński. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.
- [57] S. P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.
- [58] P. Rao, K. Sagonas, T. Swift, D. Warren, and J. Friere. XSB: a system for efficiently computing well-founded semantics. In *Proc. of 4th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR'97)*, pages 430–440. Springer, LNAI, 1997.
- [59] D. Seipel and H. Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In A. Olivé, editor, *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, pages 325–343. Universitat Politècnica de Catalunya (UPC), 1994.
- [60] P. Simons. Extending the Stable Model Semantics with More Expressive Rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in *Lecture Notes in AI (LNAI)*, pages 305–316, El Paso, Texas, USA, December 1999. Springer Verlag.
- [61] G. Terracina, E. De Francesco, C. Panetta, and N. Leone. Enhancing a DLP System for Advanced Database Applications. In *Proc. of International Conference on Web Reasoning and Rule Systems (RR 2008)*, pages 119–134, Karlsruhe, Germany, 2008. Lecture Notes in Computer Science, Springer.
- [62] G. Terracina, E. De Francesco, C. Panetta, and N. Leone. Experiencing ASP with real world applications. In *Inproc. of 15th Workshop on Knowledge Representation and Automated Reasoning (RCRA 2008)*, Udine, Italy, 2008.

- [63] G. Terracina, N. Leone, V. Lio, and C. Panetta. Adding Efficient Data Management to Logic Programming Systems. In *Proc. of 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006)*, pages 524–533, Bari, Italy, 2006. Lecture Notes in Artificial Intelligence (4203), Springer.
- [64] G. Terracina, N. Leone, V. Lio, and C. Panetta. Comparing Logic Programming Systems with DBMSs on Recursive Queries. In *Inproc. of 13th Workshop on Knowledge Representation and Automated Reasoning (RCRA 2006)*, Udine, Italy, 2008.
- [65] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming (TPLP)*, 8(2):129–165, 2008.
- [66] J. Wielemaker. Swi-prolog 3.4.3 reference manual, 1990–2000.