

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXIII CICLO

Settore Disciplinare INF/01 – INFORMATICA

TESI DI DOTTORATO

DYNAMIC MAGIC SETS

MARIO ALVIANO

Supervisor

Prof. Wolfgang Faber

Prof. Nicola Leone

Coordinatore

Prof. Nicola Leone

A.A. 2009 – 2010

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica


Dottorato di Ricerca in Matematica ed Informatica

XXIII CICLO

Settore Disciplinare INF/01 – INFORMATICA

TESI DI DOTTORATO

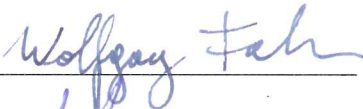
DYNAMIC MAGIC SETS



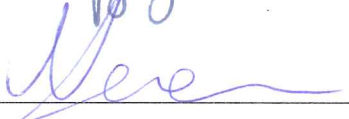
Mario Alviano

Supervisor

Prof. Wolfgang Faber



Prof. Nicola Leone



Coordinatore

Prof. Nicola Leone



A.A. 2009 – 2010

*Dedicated to my parents,
Vincenzo and Natalina,
for their loving support*

Acknowledgments

I am thankful to my supervisor Wolfgang Faber, whose encouragement, guidance and support have made possible to achieve the results described in this thesis. It is an honor for me to be his student and I am indebted to him for his meticulous proofreading of this thesis. I am also thankful to my supervisor Nicola Leone for his always interesting insights that contributed to achieve the results presented in this thesis. And I am grateful to Thomas Eiter for having kindly hosted me at the Institute of Information Systems of Vienna University of Technology.

I would like to thank also all the colleagues who made these three years pleasant, especially Lucantonio Ghionna and Marco Sirianni, who shared the office with me, and Annamaria Bria and Marco Manna, with whom I attended the ESSLLI summer school in Bordeaux.

Finally, I owe my deepest gratitude to my parents for their support, and to my brother, Leonardo, for keeping the threat of terrorism away from Italy.

This research has been partly carried out within the PIA project of DLVSYS-TEM s.r.l., supported by Regione Calabria and EU under POR Calabria FESR 2007-2013, and within the PRIN project LoDeN supported by MIUR.

Abstract

Disjunctive Datalog with stable model semantics is a rule-based language for knowledge representation and common sense reasoning that also allows to use queries for checking the presence of specific atoms in stable models. Expressiveness is a strength of the language, which indeed captures the second level of the polynomial hierarchy. However, because of this high expressive power, evaluating Disjunctive Datalog programs and queries is inherently nondeterministic. In fact, Disjunctive Datalog computations are typically characterized by two distinct phases. The first phase, referred to as *program instantiation*, is deterministic and associates input programs with equivalent ground programs; only deterministic knowledge is inferred in this phase. The second phase, referred to as *stable model search*, is nondeterministic and computes stable models of instantiated programs.

Many query optimization techniques have been proposed in the literature. Among them are Magic Sets, originally introduced for standard Datalog programs. Program instantiation is sufficient for computing the semantics of standard Datalog programs because only deterministic knowledge can be represented in this case. For this reason, the original Magic Set technique is only focused on the optimization of program instantiation. Dynamic Magic Sets are an extension of the technique that takes into account the nondeterministic knowledge encoded into Disjunctive Datalog programs. In fact, in addition to the standard optimization of program instantiation, Dynamic Magic Sets provide further optimization potential to the subsequent stable model search.

In this thesis, Dynamic Magic Sets are proved to be sound and complete for stratified and super-coherent programs. To this end, a strong relationship between magic atoms and unfounded sets is highlighted. Dynamic Magic Sets are also used for proving decidability of reasoning for a class of programs with uninterpreted function symbols. In particular, it is shown that the application of Dynamic Magic Sets to finitely recursive queries generates finitely ground programs, for which decidability of reasoning has been established in the literature.

Dynamic Magic Sets have been implemented in a prototype extending DLV, a state-of-the-art system for Disjunctive Datalog programs and queries. The effectiveness of Dynamic Magic Sets has been assessed by experimenting with the prototype system. Experimental results confirm that Dynamic Magic Sets can provide significant, possibly exponential, performance gains.

Sommario

Datalog disgiuntivo con semantica dei modelli stabili è un linguaggio a regole per la rappresentazione della conoscenza e il ragionamento di senso comune. In Datalog disgiuntivo la presenza di specifici atomi nei modelli stabili di un programma può essere verificata facendo uso di query. Uno dei punti di forza del linguaggio è la sua alta espressività: tutte le proprietà nel secondo livello della gerarchia polinomiale possono infatti essere rappresentate in Datalog disgiuntivo. Tuttavia, a causa di questa alta espressività, la valutazione di programmi e query in Datalog disgiuntivo è inerentemente non-deterministica. In Datalog disgiuntivo, infatti, le computazioni sono tipicamente caratterizzate da due fasi distinte. Nella prima fase, che è denominata *istanziazione* ed è deterministica, i programmi in input vengono associati con programmi equivalenti senza variabili; solo conoscenza deterministica può essere inferita in questa fase. I modelli stabili dei programmi istanziati vengono quindi computati durante la seconda fase, che è denominata *ricerca dei modelli stabili* ed è tipicamente non-deterministica.

Molte tecniche per l'ottimizzazione di query in programmazione logica sono state proposte in letteratura. Fra queste, una delle più note è la tecnica Magic Set, originariamente introdotta per programmi Datalog standard. Poiché solo conoscenza deterministica può essere rappresentata in programmi Datalog standard, i modelli stabili di questi programmi possono essere computati durante la fase di istanziazione. Dynamic Magic Set è un'estensione della tecnica che si avvale della natura non-deterministica di Datalog disgiuntivo per ottimizzare anche la successiva fase di ricerca dei modelli stabili.

In questa tesi, la tecnica Dynamic Magic Set è dimostrata essere corretta e completa per programmi stratificati e super-coerenti. Nel provare questi risultati viene evidenziata una interessante relazione fra atomi magic e insiemi infondati. Inoltre, la tecnica Dynamic Magic Set viene utilizzata per dimostrare la decidibilità della valutazione per una classe di programmi con simboli di funzione non-interpretati. Più precisamente, nella tesi viene dimostrato che l'applicazione di Dynamic Magic Set su query *finitamente ricorsive* produce programmi *finitamente istanziabili*, per i quali la decidibilità della valutazione è stata stabilita in letteratura. Inoltre, l'efficacia della tecnica Dynamic Magic Set è stata valutata sperimentalmente. A questo scopo, è stata implementata un prototipo estendendo DLV, un noto sistema per Datalog disgiuntivo. I risultati della sperimentazione confermano che la tecnica Dynamic Magic Set può fornire guadagni prestazionali esponenziali.

Contents

1	Introduction	1
2	Disjunctive Datalog	5
2.1	Syntax	5
2.1.1	Disjunctive Datalog Programs	5
2.1.2	Syntactically Restricted Classes	7
2.2	Semantics	9
2.2.1	Stable Models	9
2.2.2	Query Answering	12
2.3	Bottom-Up Computations	14
2.4	Expressive Power	17
2.5	Knowledge Representation and Reasoning	18
3	Magic Set Techniques	23
3.1	Magic Sets for Datalog	23
3.1.1	Sideways Information Passing	24
3.1.2	Classic Magic Sets	27
3.2	Magic Sets for Disjunctive Datalog	33
3.2.1	SIPS for Disjunctive Datalog Rules	33
3.2.2	Static Magic Sets	37
3.2.3	Dynamic Magic Sets	45
3.2.4	Query Equivalence Theorem	51
3.3	Dynamic Magic Sets and Super-Coherent Disjunctive Datalog Programs	65
3.3.1	Super-Coherent Disjunctive Datalog Programs	66
3.3.2	Running Example	66
3.3.3	Query Equivalence Theorem	69
4	Application: Decidability for Datalog with Functions	73
4.1	Datalog with Function Symbols	74
4.1.1	Preliminaries	74
4.1.2	Finitely Ground Programs	75
4.1.3	Finitely Recursive Queries and Programs	77
4.2	Dynamic Magic Sets for Finitely Recursive Queries	78
4.2.1	The DMS Algorithm Revised	78
4.2.2	Query Equivalence Theorem	80
4.3	Decidability Theorem	82
4.4	Expressive Power of Finitely Recursive Programs	84

5	Implementation and Experiments	89
5.1	System Architecture	89
5.2	Compared Methods, Benchmark Problems and Data	90
5.3	Results and Discussion	95
6	Application to Data Integration	103
6.1	Data Integration Systems	103
6.2	Consistent Query Answering	104
6.3	Experimental Results	107
7	Related Work	119
8	Conclusion	121
	Bibliography	123

List of Tables

5.1	Average execution time for Strategic Companies	97
5.2	Average execution time for Simple Path	98
5.3	Average execution time for Related	99
5.4	Average execution time for Conformant Plan Checking	100
5.5	Average execution time for Related — Super-Coherent Encoding	101
5.6	Average execution time for Conformant Plan Checking — Super-Coherent Encoding	102
6.1	Average execution time for Query 1 — INFOMIX benchmark . .	109
6.2	Average execution time for Query 2 — INFOMIX benchmark . .	111
6.3	Average execution time for Query 3 — INFOMIX benchmark . .	113
6.4	Average execution time for Query 4 — INFOMIX benchmark . .	115
6.5	Average execution time for Query 5 — INFOMIX benchmark . .	117

List of Figures

2.1	Containment relationships between Datalog ^{V,¬} and its subclasses	8
2.2	Dependency graphs for programs from Examples 2.1.4 and 2.1.7	9
2.3	ProgramInstantiation algorithm	15
2.4	StableModels algorithm	16
2.5	QueryAnswering algorithm	17
3.1	Graphical representation of EDB \mathcal{F}_1 for program \mathcal{P}_3	24
3.2	Classic Magic Sets algorithm for Datalog programs	28
3.3	ProcessQuery function for Classic Magic Sets	29
3.4	Adorn function for Classic Magic Sets	30
3.5	Generate function for Classic Magic Sets	31
3.6	Modify function for Classic Magic Sets	32
3.7	Graphical representation of EDB \mathcal{F}_2 for program \mathcal{P}_5	35
3.8	Static Magic Sets algorithm for Disjunctive Datalog programs	39
3.9	ProcessQuery function for Static and Dynamic Magic Sets	40
3.10	Adorn function for Static and Dynamic Magic Sets	41
3.11	Generate function for Static Magic Sets	42
3.12	Modify function for Static Magic Sets	44
3.13	Dynamic Magic Sets algorithm for Disjunctive Datalog programs	46
3.14	Generate function for Dynamic Magic Sets	47
3.15	Modify function for Dynamic Magic Sets	48
4.1	Dynamic Magic Sets algorithm for finitely recursive queries	79
5.1	DLV prototype system architecture	90
5.2	Structure of Simple Path and Related instances	92
5.3	Structure of Conformant Plan Checking instances	94
5.4	Average execution time for Strategic Companies	97
5.5	Average execution time for Simple Path	98
5.6	Average execution time for Related	99
5.7	Average execution time for Conformant Plan Checking	100
5.8	Average execution time for Related — Super-Coherent Encoding	101
5.9	Average execution time for Conformant Plan Checking — Super-Coherent Encoding	102
6.1	Encoding of Query 1 — INFOMIX benchmark	108
6.2	Average execution time for Query 1 — INFOMIX benchmark	109
6.3	Encoding of Query 2 — INFOMIX benchmark	110

6.4	Average execution time for Query 2 — INFOMIX benchmark . .	111
6.5	Encoding of Query 3 — INFOMIX benchmark	112
6.6	Average execution time for Query 3 — INFOMIX benchmark . .	113
6.7	Encoding of Query 4 — INFOMIX benchmark	114
6.8	Average execution time for Query 4 — INFOMIX benchmark . .	115
6.9	Encoding of Query 5 — INFOMIX benchmark	116
6.10	Average execution time for Query 5 — INFOMIX benchmark . .	117

Chapter 1

Introduction

Datalog [65] and its extensions (most notably negation and disjunction, see for instance [24, 59, 54, 53, 34, 67, 51]) are rule-based languages for knowledge representation and common sense reasoning originally introduced in the context of deductive databases [58]. In particular, Disjunctive Datalog programs [51] are characterized by the presence of disjunction in rule heads and nonmonotonic negation in bodies. In this setting the language allows for expressing all properties in the second level of the polynomial hierarchy [27], that is, all problems belonging to the complexity class Σ_2^P ($= \text{NP}^{\text{NP}}$). The high expressive power of Disjunctive Datalog, which is guaranteed also if recursive definitions involving negation are forbidden, allows for modeling problems that cannot be polynomially encoded as instances of the satisfiability problem (unless the polynomial hierarchy collapses).

The semantics of a (Disjunctive) Datalog program is defined by the set of its stable models [34, 48], that is, minimal models of the program which satisfy an additional condition specifically designed for nonmonotonic negation. Stable models are usually defined for ground programs, i.e., programs without variables. The semantics of non-ground programs is then given by the stable models of associated, equivalent ground programs. For this reason, many systems for computing stable models of Disjunctive Datalog programs implement two distinct phases, namely *program instantiation* and *stable model search*. Program instantiation usually consists of a bottom-up algorithm which iteratively derives new rules by matching bodies with atoms encountered in previous iterations. Stable model search, instead, is generally implemented by a backtracking algorithm which explores the search space efficiently. Among the systems adopting these two distinct phases are DLV [47], GnT [40], Cmodels [49], and ClaspD [26].

The framework of Disjunctive Datalog also supports query answering for retrieving all substitution answers matching a given query. Answers to a query have to be witnessed by some or all stable models of the input program, depending on whether brave or cautious reasoning is adopted. However, even if query answering depends on stable models, their complete computation is often not required. In fact, substantial parts of the stable models could be irrelevant for answering a query. For this reason, sometimes queries over Datalog programs are better handled by top-down strategies, similar to those adopted by Prolog systems [60, 44]. A typical top-down strategy looks for a rule from which some

answers to the input query might be derived; if a rule of this kind is found, its body atoms are considered as subqueries and the procedure is iterated. In this way, a top-down evaluation of a query over a standard Datalog program only considers parts of the program which are relevant for answering the query. However, in contrast to bottom-up computations, termination is not guaranteed per se in top-down evaluations, and extra effort is required for avoiding loops in case of recursive definitions.

Magic Sets [65, 7, 10], one of the best known technique in logic programming for the optimization of query answering, aim at combining the benefits of the two strategies. In fact, given a program and a query, the Magic Set technique rewrites the program in order to simulate a top-down evaluation of the query by means of a bottom-up computation of the stable models of the rewritten program. This feature of Magic Sets is mostly evident in the original technique, referred to as Classic Magic Sets in this thesis. Classic Magic Sets have been defined for standard Datalog programs, where disjunction and nonmonotonic negation are not allowed. Under these restrictions, the semantics of a program is given by exactly one stable model, which contains all and only deterministic consequences of the program. Therefore, the unique stable model of a standard Datalog program can be completely computed during program instantiation, and only this model has to be checked for answering a given query. In this setting, Classic Magic Sets introduce rules defining additional atoms, named *magic atoms*, in order to identify *relevant atoms* for answering the input query. Relevant atoms are those atoms which are reachable by a top-down evaluation of the query. Program instantiation is then limited by adding magic atoms in the bodies of the original rules. In fact, only rules which would be considered in a top-down evaluation of the query are produced in this case. It is important to observe that magic atoms have deterministic definitions according to Classic Magic Sets, and can be completely determined during program instantiation.

Extending Classic Magic Sets to Disjunctive Datalog, the objective of the work presented in this thesis, has to face with additional difficulties. A first difficulty is that the semantics of Disjunctive Datalog programs is given by many stable models in general. A second difficulty, which is eventually due to the first, is extending the notion of relevant atoms: Is an atom still relevant when it is reachable in a hypothetical top-down evaluation of the input query? Or should an ad-hoc definition be introduced? The problem of extending Classic Magic Sets to Disjunctive Datalog has been first addressed in [38], where the standard notion of relevant atoms has been adopted. As a consequence of this choice, when the technique of [38] is applied to a Disjunctive Datalog program, magic atoms represent a sizable superset of all the atoms required for answering the input query. Moreover, these magic atoms have deterministic definitions and can be completely determined during program instantiation. It follows that only program instantiation is directly optimized by the technique, and no further optimization potential is provided to the subsequent stable model search. For this reason, the technique of [38] is referred to as Static Magic Sets in this thesis.

Dynamic Magic Sets [2] are another proposal for extending Classic Magic Sets to Disjunctive Datalog. In particular, Dynamic Magic Sets are the main topic of the work reported in this thesis. What distinguishes Dynamic Magic Sets from all previously proposed techniques is the notion of relevant atoms. Indeed, while Classic Magic Sets and Static Magic Sets define relevant atoms

as the atoms reachable by a hypothetical top-down evaluation, Dynamic Magic Sets also consider a notion of *conditional relevance*. More specifically, following Dynamic Magic Sets, the relevant atoms are the atoms which can be reached by a hypothetical top-down evaluation according to all previously done assumptions. In this way Dynamic Magic Sets enlarge the optimization provided by magic atoms to the stable model search: parts of the program are dynamically disabled on the base of previously done assumptions.

Contribution

The main contribution of the work reported in this thesis are summarized in the following. First of all, a new Magic Set technique for the optimization of query answering over Disjunctive Datalog programs is presented. The new technique, referred to as Dynamic Magic Sets, introduces many interesting novelties with respect to Static Magic Sets, a previously proposed technique for Disjunctive Datalog. Indeed, while the optimization provided by Static Magic Sets is limited to program instantiation, Dynamic Magic Sets enlarge their influence to the subsequent stable model search phase by introducing a concept of conditional relevance.

A second relevant contribution is the establishment of the correctness of Dynamic Magic Sets for the class of Disjunctive Datalog programs with stratified negation. This result is also extended to a larger class of programs named super-coherent. Notably, super-coherent programs include all *odd-cycle-free* programs.

A third contribution is the identification of a strong relationship between magic sets and unfounded sets [67, 48].¹ More specifically, it has been shown that the relevant atoms for answering a given query are either true or belong to some unfounded set. It is the relationship with unfounded sets that eventually allows for proving the correctness of Dynamic Magic Sets.

Another theoretical contribution is the application of Dynamic Magic Sets for proving the decidability of the reasoning over finitely recursive programs [9], a class of Datalog programs with uninterpreted function symbols. More specifically, finitely recursive programs are mapped to finitely ground programs [20], for which decidability of the reasoning has already been established in the literature. Moreover, expressive power of finitely recursive programs is analyzed, showing that the restrictions that guarantee decidability of the reasoning do not limit the set of computable functions which can be expressed by the class.

A practical contribution is the implementation of Dynamic Magic Sets in a prototype system obtained by extending DLV, a state-of-the-art solver for Disjunctive Datalog. In particular, a new module has been added to the core of DLV. Input queries and programs are processed by the new module, which implements Dynamic Magic Sets. The rewritten program generated by the new module is then processed by the standard procedures of DLV for program instantiation and stable model search.

Finally, the implemented prototype system has been tested for assessing the impact of Dynamic Magic Sets on query answering over Disjunctive Datalog programs. The results of the experiments, involving synthetic as well as real data,

¹Unfounded sets have been used in the literature to characterize stable models and to define the well-founded semantics.

highlight that in many cases Dynamic Magic Sets provide significant performance gains and often outperform Static Magic Sets. Notably, even when most of the program is relevant for answering a given query, the overhead introduced by Dynamic Magic Sets is negligible.

Organization

The remainder of the thesis is structured as follows. First, syntax and semantics of Disjunctive Datalog programs are introduced in Chapter 2, where expressive power of the language is also discussed and some examples of knowledge representation and reasoning are provided. Then, Magic Set techniques are introduced in Chapter 3. In particular, Dynamic Magic Sets are proved to be sound and complete for the classes of stratified and super-coherent Disjunctive Datalog programs. After that, finitely recursive programs, and the applicability of Dynamic Magic Sets to this class, are discussed in Chapter 4. Afterwards, experimental results and an application to data integration are presented in Chapters 5–6. Finally, related work and conclusion are reported in Chapters 7–8.

Chapter 2

Disjunctive Datalog

In this chapter we introduce Disjunctive Datalog with nonmonotonic negation under the stable model semantics ($\text{Datalog}^{\vee, \neg}$), also known as Answer Set Programming. The chapter is structured as follows. In Section 2.1 we present the syntax of the language and relevant syntactically restricted fragments. Then, in Section 2.2 we define stable model semantics and query answering. After that, in Section 2.3 we present a typical bottom-up evaluation strategy for Disjunctive Datalog programs. Finally, in Section 2.4 we briefly discuss the expressive power of the language, and in Section 2.5 we provide examples of knowledge representation and reasoning by means of $\text{Datalog}^{\vee, \neg}$ programs.

2.1 Syntax

In this section we present the basis of Disjunctive Datalog and introduce some syntactically restricted subclasses.

2.1.1 Disjunctive Datalog Programs

Let \mathcal{C} be a set of *constants*, \mathcal{V} a set of *variables* and \mathcal{S} a set of *predicate symbols* (or simply *predicates*). In this thesis we use the convention that variables are denoted by strings starting with upper case letters, predicate symbols by strings starting with lower case letters, and constants either by natural numbers or by strings starting with lower case letters.

Example 2.1.1. Examples of constants, variables and predicates are:

- Constants: 1, 2, 341, a, b, c, alice, bob;
- Variables: X, Y, Z, Person, Man;
- Predicates: p, q, fatherOf, parentOf.

□

A *term* is either a variable or a constant, i.e., an element in $\mathcal{V} \cup \mathcal{C}$. Predicates are associated with non-negative arities and combined with terms to obtain

atoms and *literals*. An atom is a structure of the form

$$p(\mathfrak{t}_1, \dots, \mathfrak{t}_k),^1$$

where:

- p is a predicate, i.e., p belongs to \mathcal{S} ;
- $\mathfrak{t}_1, \dots, \mathfrak{t}_k$ are terms (named *arguments*), i.e., $\mathfrak{t}_1, \dots, \mathfrak{t}_k$ belong to $\mathcal{V} \cup \mathcal{C}$;
- the disequality $k \geq 0$ is satisfied.

The arity of p is k ; if $k = 0$, parenthesis are omitted and the simpler notation p is used. An atom $p(\mathfrak{t}_1, \dots, \mathfrak{t}_k)$ is *ground* (i.e., variable-free) if all its arguments are constants (i.e., if $\mathfrak{t}_1, \dots, \mathfrak{t}_k$ belong to \mathcal{C}). A positive literal is an atom $p(\bar{\mathfrak{t}})$, while a negative literal is an atom preceded by the *negation as failure* symbol **not**. A literal is ground if its atom is ground.

Example 2.1.2. Examples of atoms and literals are $p(1)$, **fatherOf**(X, bob), **succ**, **not fatherOf**(alice, bob), **not fail**. \square

A Datalog ^{\vee, \neg} rule r is a structure of the form

$$\begin{aligned} h_1(\bar{\mathfrak{u}}_1) \vee \dots \vee h_m(\bar{\mathfrak{u}}_m) &:- b_1(\bar{\mathfrak{v}}_1), \dots, b_k(\bar{\mathfrak{v}}_k), \\ &\text{not } b_{k+1}(\bar{\mathfrak{v}}_{k+1}), \dots, \text{not } b_n(\bar{\mathfrak{v}}_n). \end{aligned} \quad (2.1)$$

where:

- $h_1(\bar{\mathfrak{u}}_1), \dots, h_m(\bar{\mathfrak{u}}_m), b_1(\bar{\mathfrak{v}}_1), \dots, b_n(\bar{\mathfrak{v}}_n)$ are atoms;
- the disequalities $m \geq 1$ and $n \geq k \geq 0$ are satisfied.

The disjunction

$$h_1(\bar{\mathfrak{u}}_1) \vee \dots \vee h_m(\bar{\mathfrak{u}}_m)$$

is the *head* of r , while the conjunction

$$b_1(\bar{\mathfrak{v}}_1), \dots, b_k(\bar{\mathfrak{v}}_k), \text{not } b_{k+1}(\bar{\mathfrak{v}}_{k+1}), \dots, \text{not } b_n(\bar{\mathfrak{v}}_n)$$

is the *body* of r . The set of head atoms is denoted by $H(r)$, while the set of body literals is denoted by $B(r)$. We also use $B^+(r)$ and $B^-(r)$ for denoting the set of atoms appearing in positive and negative body literals, respectively, and $\text{ATOMS}(r)$ for the set of all atoms appearing in r . Hence, the following sets are associated with a rule r of the form (2.1):

- $H(r) = \{h_1(\bar{\mathfrak{u}}_1), \dots, h_m(\bar{\mathfrak{u}}_m)\}$;
- $B(r) = \{b_1(\bar{\mathfrak{v}}_1), \dots, b_k(\bar{\mathfrak{v}}_k), \text{not } b_{k+1}(\bar{\mathfrak{v}}_{k+1}), \dots, \text{not } b_n(\bar{\mathfrak{v}}_n)\}$;
- $B^+(r) = \{b_1(\bar{\mathfrak{v}}_1), \dots, b_k(\bar{\mathfrak{v}}_k)\}$;
- $B^-(r) = \{b_{k+1}(\bar{\mathfrak{v}}_{k+1}), \dots, b_n(\bar{\mathfrak{v}}_n)\}$;
- $\text{ATOMS}(r) = \{h_1(\bar{\mathfrak{u}}_1), \dots, h_m(\bar{\mathfrak{u}}_m), b_1(\bar{\mathfrak{v}}_1), \dots, b_n(\bar{\mathfrak{v}}_n)\}$.

¹We use the notation $\bar{\mathfrak{t}}$ for a sequence of terms. Thus, an atom with predicate p is usually referred to as $p(\bar{\mathfrak{t}})$ in this thesis.

A rule r is ground if all atoms in $\text{ATOMS}(r)$ are ground.

Datalog^{∨,¬} rules are constrained to be *safe*, that is, all variables appearing in a rule r must also appear in $B^+(r)$. A consequence of the safety condition is that rules with empty bodies have to be ground (i.e., $B(r) = \emptyset$ implies that r is ground).² Indeed, any variable would violate the safety condition in this case. Particular attention is deserved to rules with empty bodies and atomic heads: These rules constitute certain knowledge, as we will see in Section 2.2, and for this reason are named *facts*. In other words, a rule r is a fact if $|H(r)| = 1$ and $B(r) = \emptyset$.

Example 2.1.3. Consider the following rules:

$$\begin{aligned} r_1 &: \text{person}(X) \text{ :- } \text{parentOf}(X, Y). \\ r_2 &: \text{male}(Z) \text{ :- } \text{not female}(Z). \end{aligned}$$

Rule r_1 is safe thanks to $\text{parentOf}(X, Y)$, while r_2 is *unsafe* because of Z . \square

A Datalog^{∨,¬} program \mathcal{P} is a set of Datalog^{∨,¬} rules. A program \mathcal{P} is ground if all its rules are ground. In a similar way, a program \mathcal{P} is safe if all its rules are safe. As mentioned earlier, all programs are assumed to be safe unless otherwise specified.

Example 2.1.4. The following is a Datalog^{∨,¬} program, referred to as \mathcal{P}_1 hereinafter:

$$\begin{aligned} & \text{person(a). person(b). parentOf(a, b).} \\ r_3 &: \text{ancestorOf}(X, Y) \text{ :- } \text{parentOf}(X, Y). \\ r_4 &: \text{ancestorOf}(X, Y) \text{ :- } \text{parentOf}(X, Z), \text{ ancestorOf}(Z, Y). \\ r_5 &: \text{nonAncestorOf}(X, Y) \text{ :- } \text{person}(X), \text{ person}(Y), \text{ not ancestorOf}(X, Y). \\ r_6 &: \text{fatherOf}(X, Y) \vee \text{motherOf}(X, Y) \text{ :- } \text{parentOf}(X, Y). \end{aligned}$$

\square

2.1.2 Syntactically Restricted Classes

Disjunctive Datalog is a large class of programs which comprises many interesting subclasses. Some of these subclasses are introduced in this section. Containment relationships between these subclasses are shown in Figure 2.1.

We start by introducing *Standard*, *Normal* and *Positive Disjunctive Datalog*. Standard Datalog (simply denoted Datalog) is the language that has given rise to all other formalisms discussed in this thesis. In Datalog, disjunction and nonmonotonic negation are not allowed. Hence, a Datalog rule r is a rule of the form (2.1) such that $|H(r)| = 1$ and $B^-(r) = \emptyset$. Normal Datalog (denoted Datalog[¬]), instead, is characterized by atomic heads and the possibility of nonmonotonic negation in rule bodies. Formally, a Datalog[¬] rule r is a rule of the form (2.1) such that $|H(r)| = 1$. Finally, Positive Disjunctive Datalog (denoted Datalog[∨]) is characterized by rules with disjunctive heads and positive bodies. Therefore, a Datalog[∨] rule r is a rule of the form (2.1) such that $B^-(r) = \emptyset$.

Example 2.1.5. Consider the rules from Example 2.1.4:

²In these rules the symbol “:-” is omitted.

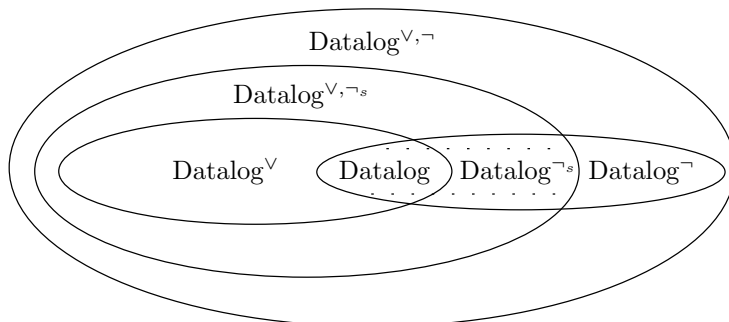


Figure 2.1: Containment relationships between $\text{Datalog}^{V, \neg}$ and its subclasses

- r_3 and r_4 are Datalog rules;
- r_3, r_4 and r_5 are Datalog^{\neg} rules;
- r_3, r_4 and r_6 are Datalog^V rules.

□

Two other relevant subclasses are obtained from $\text{Datalog}^{V, \neg}$ and Datalog^{\neg} by disabling the use of nonmonotonic negation in recursive definitions. The definition of these subclasses is based on the notion of dependency graph.

Definition 2.1.6 (Dependency Graph). Let \mathcal{P} be a $\text{Datalog}^{V, \neg}$ program. The dependency graph of \mathcal{P} , denoted by $\mathcal{G}(\mathcal{P})$, is a labeled directed graph having:

- a node for each predicate appearing in \mathcal{P} ;
- an arc $\mathbf{h} \rightarrow^{\ell} \mathbf{b}$ if there are a rule $r \in \mathcal{P}$ and two atoms $\mathbf{h}(\bar{\mathbf{u}}) \in H(r)$, $\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r) \cup B^-(r)$, where
 - ℓ is ϵ (the empty string) if $\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r)$ (in this case, \mathbf{h} depends positively on \mathbf{b}), or
 - ℓ is “ \neg ” if $\mathbf{b}(\bar{\mathbf{v}}) \in B^-(r)$ (i.e., \mathbf{h} depends negatively on \mathbf{b}).

A program \mathcal{P} is stratified with respect to negation if no cycles in $\mathcal{G}(\mathcal{P})$ involve negative dependencies (i.e., arcs labeled “ \neg ”). We can then define the class of *Stratified* $\text{Datalog}^{V, \neg}$ programs (denoted $\text{Datalog}^{V, \neg_s}$) and the class of *Stratified* Datalog^{\neg} programs (denoted Datalog^{\neg_s}).

Example 2.1.7. Program \mathcal{P}_1 in Example 2.1.4 is a $\text{Datalog}^{V, \neg_s}$ program. Indeed, its dependency graph $\mathcal{G}(\mathcal{P}_1)$ contains just one cycle (actually, a self-loop) which involves no negative dependencies; $\mathcal{G}(\mathcal{P}_1)$ is shown in Figure 2.2(a). An unstratified program \mathcal{P}_2 can be achieved from \mathcal{P}_1 by substituting r_5 with the following rules:

$$\begin{aligned} r_7 : \text{fatherOf}(X, Y) & :- \text{parentOf}(X, Y), \text{not motherOf}(X, Y). \\ r_8 : \text{motherOf}(X, Y) & :- \text{parentOf}(X, Y), \text{not fatherOf}(X, Y). \end{aligned}$$

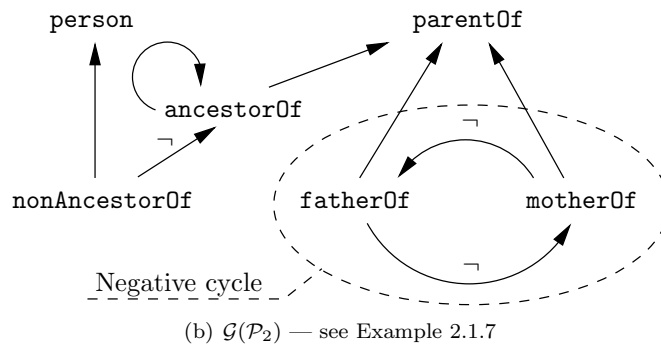
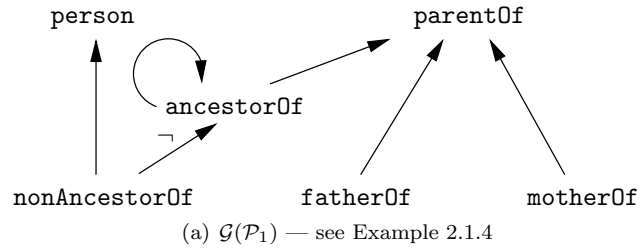


Figure 2.2: Dependency graphs for programs from Examples 2.1.4 and 2.1.7

The dependency graph $\mathcal{G}(\mathcal{P}_2)$ of \mathcal{P}_2 , shown in Figure 2.2(b), contains one cycle with negative dependencies which involves `fatherOf` and `motherOf`. \square

2.2 Semantics

The semantics of a Disjunctive Datalog program is given by the set of its stable models. Each stable model represents a plausible world or scenario according to the knowledge encoded in the program. Stable models and query answering under stable model semantics are introduced in this section.

2.2.1 Stable Models

Stable models are defined for ground programs only, while the semantics of programs with variables is given by considering equivalent ground programs. The process which associates a ground program with every $\text{Datalog}^{\vee, \neg}$ program is referred to as *program instantiation*. Essentially, rules of a non-ground program are interpreted as a schema: All variables are considered universally quantified and ranging over the set of all constants of the program. This concept is formalized below.

Let \mathcal{P} be a $\text{Datalog}^{\vee, \neg}$ program. The set of constants appearing in \mathcal{P} is the

universe of \mathcal{P} , denoted by $\mathcal{U}_{\mathcal{P}}$; if \mathcal{P} contains no constants, an arbitrary constant ξ is added to $\mathcal{U}_{\mathcal{P}}$. The set of ground atoms constructible from predicates in \mathcal{P} with elements of $\mathcal{U}_{\mathcal{P}}$ is the *base* of \mathcal{P} , denoted by $\mathcal{B}_{\mathcal{P}}$. A substitution ϑ is a function from variables to elements of $\mathcal{U}_{\mathcal{P}}$. For a structure S (atom, literal, rule), by $S\vartheta$ we denote the structure obtained from S by substituting all occurrences of each variable X in S with $\vartheta(X)$. A ground atom $p(\bar{t}')$ is an instance of an atom $p(\bar{t})$ if there is a substitution ϑ from the variables in $p(\bar{t})$ to $\mathcal{U}_{\mathcal{P}}$ such that $p(\bar{t}') = p(\bar{t})\vartheta$, that is, the application of ϑ to the variables of $p(\bar{t})$ generates $p(\bar{t}')$. In a similar way, a ground rule r' is an instance of a rule r if there is a substitution ϑ from the variables in r to $\mathcal{U}_{\mathcal{P}}$ such that $r' = r\vartheta$. The program instantiation of \mathcal{P} is the set of all instances of the rules in \mathcal{P} , denoted by $Ground(\mathcal{P})$.

Example 2.2.1. Consider again program \mathcal{P}_1 from Example 2.1.4:

- $\mathcal{U}_{\mathcal{P}_1} = \{a, b\}$;
- $\mathcal{B}_{\mathcal{P}_1} = \{\text{person}(a), \text{person}(b), \text{parentOf}(a, a), \text{parentOf}(a, b), \text{parentOf}(b, a), \text{parentOf}(b, b), \text{ancestorOf}(a, a), \text{ancestorOf}(a, b), \text{ancestorOf}(b, a), \text{ancestorOf}(b, b), \text{nonAncestorOf}(a, a), \text{nonAncestorOf}(a, b), \text{nonAncestorOf}(b, a), \text{nonAncestorOf}(b, b), \text{fatherOf}(a, a), \text{fatherOf}(a, b), \text{fatherOf}(b, a), \text{fatherOf}(b, b), \text{motherOf}(a, a), \text{motherOf}(a, b), \text{motherOf}(b, a), \text{motherOf}(b, b)\}$.

The program instantiation $Ground(\mathcal{P}_1)$ of \mathcal{P}_1 consists of 20 ground rules (plus facts): 4 ground rules for each of r_3 , r_5 and r_6 , and 8 ground rules obtained from r_4 . We avoid to enumerate all of these rules and just report the instances of r_6 to give an example:

```

fatherOf(a, a) v motherOf(a, a) :- parentOf(a, a).
fatherOf(a, b) v motherOf(a, b) :- parentOf(a, b).
fatherOf(b, a) v motherOf(b, a) :- parentOf(b, a).
fatherOf(b, b) v motherOf(b, b) :- parentOf(b, b).

```

□

We point out that predicates are used for defining relations. Relations (and predicates) can be either extensional or intensional: Extensional relations are defined by an enumeration of facts, while intensional relations are defined by rules, not all of which are facts. More formally, let \mathcal{P} be a Datalog^{v,¬} program. A *defining rule* for a predicate p is a rule $r \in \mathcal{P}$ such that some atom $p(\bar{t})$ belongs to $H(r)$. If all defining rules for a predicate p are facts, p is an *extensional database* predicate (EDB predicate); otherwise, p is an *intensional database* predicate (IDB predicate). The set of rules defining IDB predicates is denoted by $IDB(\mathcal{P})$, while $EDB(\mathcal{P})$ is used for referring to the set of the remaining rules. Moreover, the set of all facts in \mathcal{P} is denoted by $FACTS(\mathcal{P})$.

Example 2.2.2. Consider again program \mathcal{P}_1 from Example 2.1.4:

- `person` and `parentOf` are EDB predicates;
- all other predicates are IDB predicates;

- r_3 and r_4 are defining rules for `ancestorOf`;
- r_5 is a defining rule for `nonAncestorOf`;
- r_6 is a defining rule for both `fatherOf` and `motherOf`.

□

We are now ready for introducing the semantics of Disjunctive Datalog. We start by defining *interpretations* and *models*. An interpretation for a Datalog^{∨,¬} program \mathcal{P} is a function which associates each atom in $\mathcal{B}_{\mathcal{P}}$ with a truth value, i.e., a value between *true* and *false*. An interpretation I is usually represented by the set of atoms interpreted as true. An interpretation also assigns a truth value to ground literals: A positive ground literal $p(\bar{c})$ is true with respect to an interpretation I if and only if $p(\bar{c}) \in I$, while a negative ground literal `not` $p(\bar{c})$ is true if and only if $p(\bar{c}) \notin I$ (or, equivalently, if and only if $p(\bar{c})$ is false with respect to I). An interpretation I *satisfies* a ground rule r if at least one atom in $H(r)$ is true whenever all body literals of r are true, that is, if $H(r) \cap I \neq \emptyset$ holds whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I \neq \emptyset$. Finally, an interpretation M is a *model* of \mathcal{P} if M satisfies all rules in $Ground(\mathcal{P})$.

Example 2.2.3. Consider again program \mathcal{P}_1 from Example 2.1.4. The set of all interpretations of \mathcal{P}_1 is given by the power set of $\mathcal{B}_{\mathcal{P}_1}$, which accounts for a total of 2^{22} possible interpretations (more than 4 millions). Many of these interpretations are not models of \mathcal{P}_1 . For example, every interpretation not containing `person(a)` or `person(b)`, which are facts of \mathcal{P}_1 . Other interpretations are considered below.

- $FACTS(\mathcal{P}_1)$ is not a model because, for instance, rule

$$\text{ancestorOf}(a, b) \text{ :- } \text{parentOf}(a, b).$$

is not satisfied in this case (the head is false while the body is true).

- Adding `ancestorOf(a, b)` does not result in a model because, for instance, rule

$$\text{nonAncestorOf}(a, a) \text{ :- } \text{person}(a), \text{person}(a), \text{not ancestorOf}(a, a).$$

is not satisfied.

- A model is neither obtained if `nonAncestorOf(a, a)`, `nonAncestorOf(b, a)` and `nonAncestorOf(b, b)` are added to the previous interpretation; indeed, rule

$$\text{fatherOf}(a, b) \vee \text{motherOf}(a, b) \text{ :- } \text{parentOf}(a, b).$$

is not satisfied because `parentOf(a, b)` is true, but both `fatherOf(a, b)` and `motherOf(a, b)` are false.

- A model is finally achieved from the previous interpretation by adding either `fatherOf(a, b)` or `motherOf(a, b)`.
- All supersets of these two models are models as well.

□

The definition of stable model is based on the notion of program reduct. Let \mathcal{P} be a Datalog^{∨,∩} program and I an interpretation. The reduct of \mathcal{P} with respect to I , denoted by $Ground(\mathcal{P})^I$, is obtained from $Ground(\mathcal{P})$ in two steps:

1. Rules with false negative literals are deleted, i.e., each rule $r \in Ground(\mathcal{P})$ such that $B^-(r) \cap I \neq \emptyset$;
2. All negative literals are removed from the remaining rules, that is, if r has not been deleted in the previous step, $Ground(\mathcal{P})^I$ contains a rule r' such that $H(r') = H(r)$ and $B(r') = B^+(r)$.

An interpretation M is a stable model of \mathcal{P} if and only if M is a subset-minimal model of $Ground(\mathcal{P})^M$. The set of all stable models of a Datalog^{∨,∩} program \mathcal{P} is denoted by $\mathcal{SM}(\mathcal{P})$.

Example 2.2.4. Consider again program \mathcal{P}_1 from Example 2.1.4 and the following interpretations:

- $I_1 = \text{FACTS}(\mathcal{P}_1) \cup \{\text{ancestorOf}(a, b), \text{nonAncestorOf}(a, a), \text{nonAncestorOf}(b, a), \text{nonAncestorOf}(b, b)\}$;
- $I_2 = I_1 \cup \{\text{fatherOf}(a, b)\}$;
- $I_3 = I_1 \cup \{\text{motherOf}(a, b)\}$;
- $I_4 = I_1 \cup \{\text{fatherOf}(a, b), \text{motherOf}(a, b)\}$.

I_1 is not a stable model because the following rule in the reduct is not satisfied:

$$\text{fatherOf}(a, b) \vee \text{motherOf}(a, b) \text{ :- parentOf}(a, b).$$

On the other hand, I_2 and I_3 are minimal models of the respective reducts, and so stable models of \mathcal{P}_1 . The interpretation I_4 , instead, is not a stable model because it is not a subset-minimal model of the reduct with respect to I_4 . In fact, it can be checked that $\mathcal{SM}(\mathcal{P}_1) = \{I_2, I_3\}$. □

Stable models possess many interesting properties. Among them are *minimality* and *support*. In fact, stable models are incomparable with respect to set inclusion, which means that no superfluous information is contained in stable models. Stable models are also supported models, that is, models containing only atoms which are “supported” by some rule with a true body. Finally, in Datalog^{∨,∩} we distinguish *coherent* and *incoherent* programs: Coherent programs admit at least one stable model, while incoherent programs have no stable models. Datalog[∩] programs are either coherent or incoherent in general, while Datalog, Datalog[∨], Datalog^{∩s}, Datalog^{∨,∩s} programs are guaranteed to be coherent. In particular, each Datalog or Datalog^{∩s} program is characterized by a unique stable model.

2.2.2 Query Answering

Stable models of a Datalog^{∨,∩} program represent all plausible scenarios according to the knowledge encoded in the program. Then, stable models can be used

for checking if a property of interest holds in some or in all possible scenarios. Queries are used for expressing this interest. A formal definition of a query is provided in this section.

We start by introducing the notions of brave and cautious consequence of a program, which are used in the definition of query answering. Let \mathcal{P} be a Datalog ^{\forall, \neg} program. A ground atom $\mathbf{p}(\bar{\mathbf{t}})$ is a *brave consequence* of \mathcal{P} if $\mathbf{p}(\bar{\mathbf{t}})$ belongs to some stable model of \mathcal{P} , i.e., if there is $M \in \mathcal{SM}(\mathcal{P})$ such that $\mathbf{p}(\bar{\mathbf{t}}) \in M$. In this case we also say that \mathcal{P} bravely entails $\mathbf{p}(\bar{\mathbf{t}})$, denoted by $\mathcal{P} \models_b \mathbf{p}(\bar{\mathbf{t}})$. Cautious consequences are defined in a similar way, but all stable models are considered in this case. Thus, a ground atom $\mathbf{p}(\bar{\mathbf{t}})$ is a *cautious consequence* of \mathcal{P} if $\mathbf{p}(\bar{\mathbf{t}})$ belongs to all stable models of \mathcal{P} , i.e., if $\mathbf{p}(\bar{\mathbf{t}}) \in M$ for all $M \in \mathcal{SM}(\mathcal{P})$. In this case we also say that \mathcal{P} cautiously entails $\mathbf{p}(\bar{\mathbf{t}})$, denoted by $\mathcal{P} \models_c \mathbf{p}(\bar{\mathbf{t}})$.

Example 2.2.5. Examples of brave and cautious consequences of program \mathcal{P}_1 from Example 2.1.4 are the following:

- $\mathbf{fatherOf}(\mathbf{a}, \mathbf{b})$ and $\mathbf{motherOf}(\mathbf{a}, \mathbf{b})$ are brave consequences of \mathcal{P}_1 ;
- $\mathbf{ancestorOf}(\mathbf{a}, \mathbf{b})$ is both a brave and a cautious consequence of \mathcal{P}_1 .

□

Queries are expressed by atoms, but more complex queries are expressible by using additional rules (see Example 2.2.6 below). Hence, a query \mathcal{Q} is an atom $\mathbf{p}(\bar{\mathbf{t}})$ followed by a question mark:³

$$\mathcal{Q} = \mathbf{p}(\bar{\mathbf{t}})?$$

We assume that each constant appearing in \mathcal{Q} also appears in \mathcal{P} ; if this is not the case, a fact $\mathbf{q}(\bar{\mathbf{s}})$ is added to \mathcal{P} , where \mathbf{q} is a fresh predicate and $\bar{\mathbf{s}}$ is the list of constants occurring in \mathcal{Q} . The set of substitutions ϑ for the variables of $\mathbf{p}(\bar{\mathbf{t}})$ such that $\mathbf{p}(\bar{\mathbf{t}})\vartheta$ is a brave consequence of \mathcal{P} is denoted by $\mathit{Ans}_b(\mathcal{Q}, \mathcal{P})$. In a similar way, the set of substitutions ϑ for the variables of $\mathbf{p}(\bar{\mathbf{t}})$ such that $\mathbf{p}(\bar{\mathbf{t}})\vartheta$ is a cautious consequence of \mathcal{P} is denoted by $\mathit{Ans}_c(\mathcal{Q}, \mathcal{P})$. Thus:

- $\mathit{Ans}_b(\mathcal{Q}, \mathcal{P}) = \{\vartheta \mid \mathcal{P} \models_b \mathbf{p}(\bar{\mathbf{t}})\vartheta\}$ (brave reasoning);
- $\mathit{Ans}_c(\mathcal{Q}, \mathcal{P}) = \{\vartheta \mid \mathcal{P} \models_c \mathbf{p}(\bar{\mathbf{t}})\vartheta\}$ (cautious reasoning).

For ground queries, the sets $\mathit{Ans}_b(\mathcal{Q}, \mathcal{P})$ and $\mathit{Ans}_c(\mathcal{Q}, \mathcal{P})$ are either empty or just contain ϵ (the empty substitution).

Example 2.2.6. Let $\mathcal{Q}_1 = \mathbf{fatherOf}(\mathbf{X}, \mathbf{b})$ and $\mathcal{Q}_2 = \mathbf{ancestorOf}(\mathbf{a}, \mathbf{Y})$ be two queries for program \mathcal{P}_1 from Example 2.1.4. Query \mathcal{Q}_1 is asking for the father of \mathbf{b} , while \mathcal{Q}_2 is interested in the descendants of \mathbf{a} . Hence:

- $\mathit{Ans}_b(\mathcal{Q}_1, \mathcal{P}_1) = \{\mathbf{a}\}$ and $\mathit{Ans}_c(\mathcal{Q}_1, \mathcal{P}_1) = \emptyset$;
- $\mathit{Ans}_b(\mathcal{Q}_2, \mathcal{P}_1) = \{\mathbf{b}\} = \mathit{Ans}_c(\mathcal{Q}_2, \mathcal{P}_1)$.

³For simplifying the reading, question marks of queries are omitted when referring to them in the text.

Now suppose that we want to check that the relation expressed by `parentOf` is anti-reflexive, that is, `parentOf(x, y)` true implies that `parentOf(y, x)` is false. In other words, we have to verify that there is no substitution ϑ such that the conjunction

$$\text{parentOf}(X, Y)\vartheta, \text{parentOf}(Y, X)\vartheta$$

is true. In this case a query `inconsistent` and the following auxiliary rule can be considered:

$$\text{inconsistent} \text{ :- } \text{parentOf}(X, Y), \text{parentOf}(Y, X).$$

If the query is not a brave or a cautious consequence, the relation expressed by `parentOf` is anti-reflexive. \square

For Datalog and Datalog^{¬s} programs, brave and cautious reasoning coincide. Indeed, these programs have unique stable models. For Datalog[∨] and Datalog^{∨,¬s} programs, instead, it can be observed that cautious consequences are also brave consequences. Indeed, these programs admit at least one stable model. This is not true in general for Datalog[¬] and Datalog^{∨,¬} programs. In fact, a Datalog[¬] or Datalog^{∨,¬} program \mathcal{P} may be incoherent, in which case all atoms in $\mathcal{B}_{\mathcal{P}}$ are cautious consequences of \mathcal{P} , but no atom is a brave consequence of \mathcal{P} .

Two programs are said to be *equivalent* with respect to a given query if they provide the same answers to the query. Formally, let \mathcal{P} and \mathcal{P}' be Datalog^{∨,¬} programs, and \mathcal{Q} a query. The programs \mathcal{P} and \mathcal{P}' are brave equivalent with respect to \mathcal{Q} , denoted by $\mathcal{P} \equiv_{\mathcal{Q}}^b \mathcal{P}'$, if $\text{Ans}_b(\mathcal{Q}, \mathcal{P} \cup \mathcal{F}) = \text{Ans}_b(\mathcal{Q}, \mathcal{P}' \cup \mathcal{F})$ is guaranteed for each set of facts \mathcal{F} defined over the EDB predicates of \mathcal{P} and \mathcal{P}' . Similarly, \mathcal{P} and \mathcal{P}' are cautious equivalent with respect to \mathcal{Q} , denoted by $\mathcal{P} \equiv_{\mathcal{Q}}^c \mathcal{P}'$, if $\text{Ans}_c(\mathcal{Q}, \mathcal{P} \cup \mathcal{F}) = \text{Ans}_c(\mathcal{Q}, \mathcal{P}' \cup \mathcal{F})$ is guaranteed for each set of facts \mathcal{F} defined over the EDB predicates of \mathcal{P} and \mathcal{P}' .

Example 2.2.7. Program \mathcal{P}_1 from Example 2.1.4 and program \mathcal{P}_2 from Example 2.1.7 are brave and cautious equivalent with respect to any query. In fact, $\mathcal{SM}(\mathcal{P}_1)$ and $\mathcal{SM}(\mathcal{P}_2)$ coincide for all possible EDB. \square

Equivalence is fundamental for query optimization. In Chapter 3 we will present some techniques for associating input programs with equivalent programs which allow for potentially optimized query answering.

2.3 Bottom-Up Computations

Many Datalog^{∨,¬} systems implement a two-phase bottom-up strategy. The first phase is named *program instantiation*. For an input program \mathcal{P} , it produces a ground program which is equivalent to $\text{Ground}(\mathcal{P})$, but significantly smaller. Most of the techniques used in this phase stem from bottom-up methods developed for classic and deductive databases; see for example [1] or [33, 47] for details. A fact which is used in these techniques is that the truth of an atom $\text{p}(\bar{\tau})$ has to be supported by some rule having $\text{p}(\bar{\tau})$ in the head and such that all body literals are true. If no rule of this kind exists, $\text{p}(\bar{\tau})$ is guaranteed to be false in all stable models. This is the case, for instance, of all instances of

```

Algorithm ProgramInstantiation( $\mathcal{P}$ )
Input: A Datalog $\forall, \neg$  program  $\mathcal{P}$ ;
Output: An equivalent ground program;
var
   $R, R'$  : set of rules;
begin
  1.  $R := \text{FACTS}(\mathcal{P})$ ;
  2. repeat
  3.    $R' := R$ ;
  4.   Let  $\mathcal{H}$  be the set of atoms occurring in the head of some rule in  $R'$ ;
  5.   for each rule  $r \in \mathcal{P}$  do
  6.     for each substitution  $\vartheta$  for all variables of  $r$  do
  7.       if  $B^+(r)\vartheta \subseteq \mathcal{H}$  do add  $r\vartheta$  to  $R$ ; end if
  8.     end for
  9.   end for
  10. until  $R = R'$ ;
  11. return  $R$ ;
end.

```

Figure 2.3: ProgramInstantiation algorithm

EDB atoms which do not occur in $\text{EDB}(\mathcal{P})$. Also rules having some positive literals which are known to be false cannot support any atom, which in turn could imply new false atoms. We point out that no assumption is made during program instantiation, only deterministic knowledge is derived. An elementary algorithm implementing program instantiation is reported in Figure 2.3. The algorithm uses two sets of rules, namely R and R' . Initially, R contains all facts of \mathcal{P} (line 1). Then, a copy of R is stored in R' (line 3) and the set \mathcal{H} of all atoms occurring in the head of some rule in r' is computed (line 4). After that, each rule r of \mathcal{P} is instantiated with respect to all substitutions ϑ such that $B^+(r)\vartheta$ is a subset of \mathcal{H} (lines 5–9). All new instances are added to R (line 7). The process is repeated until no new rules are generated (lines 2–10). Finally, the algorithm terminates and the set R is returned in output (line 11).

Example 2.3.1. Consider again program \mathcal{P}_1 from Example 2.1.4. By invoking **ProgramInstantiation**, the following ground program is generated:

```

person(a). person(b). parentOf(a,b).
ancestorOf(a,b) :- parentOf(a,b).
nonAncestorOf(a,a) :- person(a), person(a), not ancestorOf(a,a).
nonAncestorOf(a,b) :- person(a), person(b), not ancestorOf(a,b).

nonAncestorOf(b,a) :- person(b), person(a), not ancestorOf(b,a).
nonAncestorOf(b,b) :- person(b), person(b), not ancestorOf(b,b).
fatherOf(a,b) v motherOf(a,b) :- parentOf(a,b).

```

□

The second phase is often referred to as *stable model search* and takes care

```

Algorithm StableModels( $\mathcal{P}$ )
Input: A DatalogV,∇ program  $\mathcal{P}$ ;
Output: The set of stable models of  $\mathcal{P}$ ;
var
  SM: set of stable models;
begin
  1.  $SM := \emptyset$ ;
  2. StableModelsAux(ProgramInstantiation( $\mathcal{P}$ ),  $\emptyset, \emptyset, \mathcal{B}_{\mathcal{P}}, SM$ );
  3. return  $SM$ ;
end.

```

```

Procedure StableModelsAux( $\mathcal{P}, T, F, U, SM$ )
Input
   $\mathcal{P}$ : program;
   $T, F, U$ : set of ground atoms;
   $SM$ : set of stable models;
Output: none;
begin
  1. ComputeDeterministicConsequences( $\mathcal{P}, T, F, U$ );
  2. if  $T \cap F = \emptyset$  then
  3.   if  $U \neq \emptyset$  then
  4.    take an element  $p(\bar{c})$  from  $U$ ;
  5.    StableModelsAux( $\mathcal{P}, T \cup \{p(\bar{c})\}, F, U \setminus \{p(\bar{c})\}, SM$ );
  6.    StableModelsAux( $\mathcal{P}, T, F \cup \{p(\bar{c})\}, U \setminus \{p(\bar{c})\}, SM$ );
  7.   else if StabilityCheck( $T$ ) then
  8.    add  $T$  to  $SM$ ;
  9.   end if
  10. end if
end.

```

Figure 2.4: StableModels algorithm

of the non-deterministic computation. Essentially, one undefined atom is selected and its truth or falsity is assumed. The assumption might imply truth or falsity of other undefined atoms. Hence, the process is repeated until either an inconsistency is derived or all atoms have been interpreted. In the latter case an additional check is performed to ensure stability of the model. Details on this process can be found for example in [29]. An elementary algorithm implementing stable model search is reported in Figure 2.4. The algorithm uses a recursive procedure for computing all stable models of the input program. The recursive procedure starts by computing all deterministic consequences (line 1). After that, if an inconsistency has been introduced (line 2), the procedure returns. Otherwise, recursive invocations are made assuming the truth or falsity of some undefined atom. Finally, when all atoms are true or false, the constructed model is checked for stability and eventually added to the set of stable models (line 7). The procedures used in Figure 2.4 for computing deterministic consequences and checking stability of models are out of the scope of this work and have been omitted for simplicity.

Query answering is typically handled by storing all admissible answer substitutions as stable models are computed. For brave reasoning, each stable model can contribute substitutions to the set of answers. In this case the set of an-

```

Algorithm QueryAnswering( $\mathcal{Q}, \mathcal{P}, \textit{brave-reasoning}$ )
Input: A query  $\mathcal{Q}$  over a DatalogV,∇ program  $\mathcal{P}$  and a boolean brave-reasoning;
Output: The set  $Ans_b(\mathcal{Q}, \mathcal{P})$  if brave-reasoning, the set  $Ans_c(\mathcal{Q}, \mathcal{P})$  otherwise;
var
   $Ans$ : set of substitutions;
begin
  1. if brave-reasoning then
  2.    $Ans := \emptyset$ ;
  3.   for each stable model  $M$  in StableModels( $\mathcal{P}$ ) do
  4.      $Ans := Ans \cup \{\vartheta \mid \mathcal{Q}\vartheta \in M\}$ ;
  5.   end for
  6. else
  7.    $Ans := \{\vartheta \mid \mathcal{Q}\vartheta \in \mathcal{B}_{\mathcal{P}}\}$ ;
  8.   for each stable model  $M$  in StableModels( $\mathcal{P}$ ) do
  9.      $Ans := Ans \setminus \{\vartheta \mid \mathcal{Q}\vartheta \notin M\}$ ;
  10.  end for
  11. end if
  12. return  $Ans$ ;
end.

```

Figure 2.5: QueryAnswering algorithm

swers is initially empty. For cautious reasoning, instead, each stable model may eliminate some substitutions from the set of admissible answers. Therefore, in this case all possible substitutions for the input query are initially contained in the set of answers. An elementary algorithm implementing this strategy is shown in Figure 2.5. Admissible answers are stored in Ans , which initially is empty (in case of brave reasoning; line 2) or contains all possible substitutions (in case of cautious reasoning; line 7). Then, for each stable model produced by the **StableModels** algorithm, some substitutions are added (brave reasoning; lines 3–5) or removed (cautious reasoning; lines 8–10) from Ans . Finally, the algorithm terminates and the set Ans is returned in output.

2.4 Expressive Power

Disjunctive Datalog is a powerful formalism for knowledge representation and common sense reasoning. It is particularly suitable for representing incomplete knowledge and nonmonotonic reasoning, and it is also used in Artificial Intelligence for applications in diagnosis and planning. Its expressiveness is powerful in a precise mathematical sense, and in particular it allows for expressing all problems in the complexity class Σ_2^P in a uniform way [27]. This high expressiveness is relevant in many contexts. For instance, important problems in Artificial Intelligence are not solvable with polynomial reductions to SAT⁴ (unless the polynomial hierarchy collapses), while they are directly representable in Disjunctive Datalog [47]. It is important to observe that Disjunctive Datalog captures the second level of the polynomial hierarchy thanks to disjunction. Indeed, without disjunction, the expressive power of the language decreases down to the first level of the polynomial hierarchy, that is, only NP queries can be

⁴A formal definition of SAT, the *satisfiability* problem, is provided in Section 2.5.

represented by Datalog[¬] programs. On the other hand, nonmonotonic negation only afford a marginal contribution to the expressive power of Disjunctive Datalog. Indeed, stratified negation is enough for expressing all properties in the complexity class Σ_2^P [27], which means that Datalog^{∨,¬^s} and Datalog^{∨,¬} have the same expressive power. Below, we present a modular and polynomial translation of Datalog^{∨,¬} programs to Datalog^{∨,¬^s} programs.

Let \mathcal{P} be a ground Datalog^{∨,¬} program. Moreover, let **fail** be an atom not occurring in $\mathcal{B}_{\mathcal{P}}$ which will be used to discard some interpretations. The Datalog^{∨,¬^s} program associated with \mathcal{P} is the program \mathcal{P}' obtained from \mathcal{P} by applying the following changes:

1. Each occurrence of a negative literal **not a** is replaced by \mathbf{a}_F , where \mathbf{a}_F is an atom not occurring in $\mathcal{B}_{\mathcal{P}}$;
2. For each fresh atom \mathbf{a}_F introduced in the previous step, another fresh atom \mathbf{a}_T and the following rules are added to \mathcal{P}' :

$$\begin{aligned} r_9 &: \mathbf{a}_T \vee \mathbf{a}_F. \\ r_{10} &: \mathbf{a}_T :- \mathbf{a}. \\ r_{11} &: \mathbf{fail} :- \mathbf{a}_T, \mathbf{not} \mathbf{a}. \end{aligned}$$

We start by observing that \mathcal{P}' is a stratified program. Indeed, negative literals appear only in r_{11} , and the only head atom in r_{11} is **fail**, which does not appear elsewhere in \mathcal{P}' . Then, we note that one among \mathbf{a}_T and \mathbf{a}_F is guessed by r_9 , but \mathbf{a}_T must be chosen if \mathbf{a} is true because of r_{10} , and only in this case because of r_{11} . Therefore, since \mathbf{a}_T and \mathbf{a}_F do not appear in other heads in \mathcal{P}' , \mathbf{a}_F is derived if and only if \mathbf{a} is false, that is, if and only if **not a** is true. Hence, there is a one-to-one mapping between the stable models of \mathcal{P} and \mathcal{P}' , and the following equality is established:

$$SM(\mathcal{P}) = \{M \cap \mathcal{B}_{\mathcal{P}} \mid M \in SM(\mathcal{P}') \text{ and } \mathbf{fail} \notin M\}.$$

2.5 Knowledge Representation and Reasoning

Examples of knowledge representation and reasoning are reported in this section. We point out that all problems discussed in this section are encoded by stratified programs. Further examples are discussed in the remainder of the thesis. In particular, some problems falling in the complexity class Σ_2^P , like for instance *strategic companies* and *consistent query answering*, are presented in Chapters 5–6.

Satisfiability

Satisfiability (SAT) is among the most studied problems in computer science. It consists of deciding whether a given propositional formula is satisfiable. SAT was the first problem which was proved to be NP-complete and it is a well-studied problem, for both theoretical and practical aspects. Concerning theoretical aspects, SAT has been used for proving NP-hardness of many other problems. Concerning practical aspects, SAT is widely used for solving difficult problems

belonging to the complexity class NP, such as verification of circuits. More specifically, instances of problems in NP can be translated to SAT instances, which can be efficiently handled by SAT solvers. A well-known result about SAT is that the problem remains NP-complete even if boolean formulas are constrained to have a particular form, referred to as 3-CNF (3 conjunctive normal form). In this case the problem is referred to as 3-SAT and can be formulated as follows:

Given a propositional 3-CNF formula Φ over the variables x_1, \dots, x_n , is there a truth assignment for the variables x_1, \dots, x_n satisfying the formula Φ ? Let Φ be of the form

$$\Phi = C_1 \wedge \dots \wedge C_m, \quad (2.2)$$

where each clause C_i is a disjunction $\ell_i^1 \vee \ell_i^2 \vee \ell_i^3$, and each ℓ_i^j is a positive or negative literal (note that in the context of SAT the term “literal” denotes a propositional variable x_k or a propositional variable preceded by the negation symbol $\neg x_k$; clauses and literals are unordered).

A formula Φ of form (2.2) can be encoded as follows:

- `variable(x)`, for each propositional variable x appearing in Φ ;
- `clause0(x, y, z)`, for each clause $\neg x \vee \neg y \vee \neg z$ in Φ ;
- `clause1(x, y, z)`, for each clause $x \vee \neg y \vee \neg z$ in Φ ;
- `clause2(x, y, z)`, for each clause $x \vee y \vee \neg z$ in Φ ;
- `clause3(x, y, z)`, for each clause $x \vee y \vee z$ in Φ .

Then, SAT can be encoded by the program \mathcal{P}_{SAT} and the query below:

```

true(X) v false(X) :- variable(X).
unsat :- clause0(X, Y, Z), true(X), true(Y), true(Z).
unsat :- clause1(X, Y, Z), false(X), true(Y), true(Z).
unsat :- clause2(X, Y, Z), false(X), false(Y), true(Z).
unsat :- clause3(X, Y, Z), false(X), false(Y), false(Z).
sat :- not unsat.
sat?

```

The query `sat` is a brave consequence of \mathcal{P}_{SAT} if and only if Φ is satisfiable.

A Non-Decisional Variant

Suppose we are now interested in the following problem:

Given a proposition formula Φ , retrieve all variables that are required to be true (resp. false) in order to satisfy Φ .

This problem can be handled by slightly modifying the program \mathcal{P}_{SAT} . In particular, for retrieving these variables and the associated values, it is enough to replace the query by $\text{value}(X, V)$, add the rules

```

value(X, true) :- true(X).
value(X, false) :- false(X).
value(X, true) :- variable(X), unsat.
value(X, false) :- variable(X), unsat.

```

and perform cautious reasoning. Note that no substitution is removed from the set of possible answers if **unsat** is derived to be true.

Tautology

Tautology is a coNP-complete problem which is strictly related to SAT, as it is its complement. The problem can be defined as follows:

Given a propositional 3-CNF formula Φ over the variables x_1, \dots, x_n , is Φ satisfied by all possible truth assignments for the variables x_1, \dots, x_n ?

The formula Φ can be encoded as described above for SAT, and it can be shown that the query **sat** is a cautious consequence of the program \mathcal{P}_{SAT} if and only if Φ is a tautology.

Not-All-Equal-SAT

Not-all-equal-SAT (NAE-SAT) is a variant of SAT which remains NP-complete even if all literals are positive. Essentially, the problem can be stated as follows:

Given a propositional 3-CNF formula Φ over the variables x_1, \dots, x_n , is there a truth assignment for the variables x_1, \dots, x_n satisfying the formula Φ and such that no clause has all literals true?

Assuming the further restriction that formula Φ is positive (i.e., all literals in Φ are positive), and that Φ is encoded by the predicates `clause3` and `variable`, NAE-SAT can be handled by performing brave reasoning over the following program and query:

```

true(X) v false(X) :- variable(X).
unsat :- clause3(X, Y, Z), true(X), true(Y), true(Z).
unsat :- clause3(X, Y, Z), false(X), false(Y), false(Z).
sat :- not unsat.
sat?

```

Colorability

Colorability is an NP-complete problem in graph theory which can be formulated as follows:

Given an undirected graph G and k colors c_1, \dots, c_k , is there an assignment of colors for the vertices of G such that no two adjacent vertices share the same color?

Here we assume that only three colors are available, namely *red*, *green* and *blue*. Under this restriction, the problem is still NP-complete and referred to as 3-colorability. Graphs can be encoded by the predicates **vertex** and **edge**, where **edge** is assumed to define a symmetric relation. In this case the problem can be encoded by the program \mathcal{P}_{COL} and the query below:

```
color(X,red) v color(X,green) v color(X,blue) :- vertex(X).
nocol :- edge(X,Y), color(X,C), color(Y,C).
col :- not nocol.
col?
```

Intuitively, a color is guessed for each vertex by the first rule. Here we are implicitly taking advantage of the minimality of the stable model semantics. The second rule, instead, checks that no pair of adjacent vertices share the same color. Therefore, the original graph G is a yes-instance of 3-colorability if and only if **col** is a brave consequence of the program above.

A Non-Decisional Variant

Suppose that we are now interested in the following problem:

Given an undirected graph G , retrieve all pairs of vertices sharing the same color in all 3-coloring of G .

This problem can be handled by slightly modifying program \mathcal{P}_{COL} . In particular, for retrieving these pairs of vertices, it is enough to replace the query by **sameColor(X,Y)**, add the rules

```
sameColor(X,Y) :- color(X,C), color(Y,C).
sameColor(X,Y) :- vertex(X), vertex(Y), nocol.
```

and perform cautious reasoning. Note that no substitution is removed from the set of possible answers if **nocol** is derived to be true.

k -Clique

We conclude this chapter by presenting another NP-complete problem in graph theory, usually referred to as k -Clique. It can be stated as follows:

Given an undirected graph G and an integer k , does G contain a subgraph which is a k -clique, i.e., a complete graph with k vertices?

Instances of the problem can be encoded by means of the predicates **vertex** and **edge** in an intuitive way (**edge** is assumed to define a symmetric relation). In addition, there is a fact **k(i)** for each $i = 1, \dots, k$. The problem can be encoded

by the following program and query:

```
in(I,X) v out(I,X) :- k(I), vertex(X).
noclique :- in(I,X), in(I,Y), X ≠ Y.
inclique(X) :- in(I,X).
noclique :- inclique(X), inclique(Y), X ≠ Y, not edge(X,Y).
clique :- not noclique.
clique?
```

The first three rules above guess k vertices or more, while the last two rules check whether each pair of guessed vertices is connected. Therefore, the original graph G contains a k clique if and only if the query `clique` is a brave consequence of the program above.

Chapter 3

Magic Set Techniques

In this chapter we describe three Magic Set techniques for query optimization, the main topic of the work described in this thesis. In particular, we discuss the original Magic Set method for Datalog programs, herein referred to as Classic Magic Sets, and two extensions of the technique to Disjunctive Datalog, namely Static and Dynamic Magic Sets. Static Magic Sets are the first proposed extension of Classic Magic Sets to Disjunctive Datalog programs, while Dynamic Magic Sets are the technique developed and analyzed in this thesis.

Classic Magic Sets aim at identifying parts of a program which are relevant for answering a given query. In this way Classic Magic Sets allow for discarding a sensible number of ground rules during the instantiation of a program. The same notion of relevance, adapted to the disjunctive case, is used by Static Magic Sets. However, there is a crucial difference between Datalog and Disjunctive Datalog which is not addressed by Static Magic Sets: Datalog programs are characterized by the uniqueness of the intended model, while Disjunctive Datalog programs possess many stable models in general. We point out that what is relevant for a query in one stable model may be completely irrelevant in another one, or also in all others. Indeed, every stable model represents a different, plausible scenario according to the knowledge represented in a program. Dynamic Magic Sets take advantage of this aspect, which can allow for exponential performance gains with respect to the original program as well as with respect to the program generated by Static Magic Sets.

The chapter is structured as follows. Classic Magic Sets are presented in Section 3.1, while Static and Dynamic Magic Sets are introduced in Section 3.2. Dynamic Magic Sets are proved to be sound and complete for Datalog^{V,¬s} programs in Section 3.2.4, where we also show a strong relationship between magic sets and unfounded sets which was not previously recognized in the literature. After that, in Section 3.3 the correctness of Dynamic Magic Sets is enlarged to super-coherent programs, which include all odd-cycle-free programs.

3.1 Magic Sets for Datalog

In this section we introduce Classic Magic Sets (CMS) for standard Datalog programs. In particular, in Section 3.1.1 we introduce the concept of binding and of sideways information passing. After that, in Section 3.1.2 we provide a

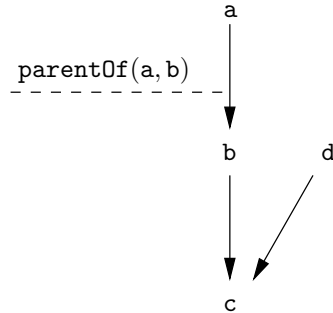


Figure 3.1: Graphical representation of EDB \mathcal{F}_1 for program \mathcal{P}_3

detailed description of an algorithm implementing CMS.

3.1.1 Sideways Information Passing

Magic Sets aim to simulate a top-down evaluation of a query Q , like SLD-resolution [43] adopted by Prolog. According to this kind of evaluation, all rules r such that $Q = p(\bar{t})\vartheta$, for an atom $p(\bar{t}) \in H(r)$ and a substitution ϑ , are considered in a first step. Then, the atoms in $B^+(r)\vartheta$ are taken as subqueries,¹ and the procedure is iterated. According to this process, if a (sub)query has some arguments *bound* to constant values, this information is “passed” to the atoms in the body. Moreover, bodies are processed in a certain sequence, and processing a body atom may bind some of its arguments for subsequently considered body atoms, thus “generating” and “passing” bindings within the body. Whenever a body atom is processed, each of its arguments is therefore considered to be either *bound* or *free*. We illustrate this mechanism by means of an example.

Example 3.1.1. Consider a database consisting of a relation `parentOf(X, Y)`, where X is a parent of Y . In particular, assume the following EDB, graphically represented in Figure 3.1:

$$\mathcal{F}_1 = \{\text{parentOf}(a, b), \text{parentOf}(d, c), \text{parentOf}(b, c)\}.$$

A relation `ancestorOf` defining the transitive closure of `parentOf` can be achieved by means of the program \mathcal{P}_3 reported below:

$$\begin{aligned} r_{12} : \text{ancestorOf}(X, Y) &:- \text{parentOf}(X, Y). \\ r_{13} : \text{ancestorOf}(X, Y) &:- \text{parentOf}(X, Z), \text{ancestorOf}(Z, Y). \end{aligned}$$

Now assume we are just interested in the descendants of a person a given in input. Thus, we are interested in answering the query Q_3 reported below:

$$\text{ancestorOf}(a, Y)?$$

¹We recall that standard Datalog rules have empty negative body.

A typical top-down evaluation scheme would consider r_{12} and r_{13} with X bound to a and Y free. In particular, when considering r_{12} , the information about the binding of variable X is passed to the atom `parentOf(X, Y)`, which is indeed the only subquery atom occurring in r_{12} . Hence, the subquery `parentOf(a, Y)` is considered. The only atom in \mathcal{F}_1 which matches the subquery is `parentOf(a, b)`, and so the answer b is obtained from the rule instance

$$\text{ancestorOf}(a, b) \text{ :- } \text{parentOf}(a, b).$$

When considering r_{13} , instead, the binding information can be passed either to `parentOf(X, Z)` or to `ancestorOf(Z, Y)`. Suppose that subqueries are evaluated according to their order in rules, from left to right, so that `parentOf(X, Z)` is considered before `ancestorOf(Z, Y)` in r_{13} . In particular, \mathcal{F}_1 contains the atom `parentOf(a, b)` which binds Z to b . This binding information might be propagated to the remaining subquery `ancestorOf(Z, Y)`, which becomes `ancestorOf(b, Y)` in this case. The process is then repeated by looking for an answer to the subquery `ancestorOf(b, Y)`. Again, the rule r_{12} is considered, from which `ancestorOf(b, c)` is derived because `parentOf(b, c)` belongs to \mathcal{F}_1 . Then, since all subqueries in the rule

$$\text{ancestorOf}(a, c) \text{ :- } \text{parentOf}(a, b), \text{ancestorOf}(b, c).$$

are true, c is another answer for \mathcal{Q}_3 . It can be checked that no other substitutions are answers for \mathcal{Q}_3 . Indeed, the only other person occurring in \mathcal{F}_1 is d , which however only appears as a parent. \square

In the example above we had two degrees of freedom in the specification of the top-down evaluation scheme. The first one concerns the order used for processing body atoms. In fact, the declarative semantics of Datalog allows for an arbitrary order to be adopted. The second degree of freedom is slightly more subtle and concerns the selection of the terms to be considered bound to constants from previous evaluations. Indeed, while we have considered the propagation of all binding information that originates from previously processed body atoms, it is in general possible to restrict the top-down evaluation by partially propagating this information. For instance, one may desire to propagate only information generated from the evaluation of EDB predicates, or even just the information that is passed on via head atoms.

The specific propagation strategy adopted in a top-down evaluation scheme is called *sideways information passing strategy* (SIPS). Roughly, a SIPS is a partial order over the atoms of each rule which also specifies how the bindings originate and propagate [10, 38]. In order to properly formalize this concept we first introduce adornment strings, a convenient way for representing binding information for IDB predicates.

Definition 3.1.2 (Adornment String). Consider an atom $p(\mathbf{t}_1, \dots, \mathbf{t}_k)$. An adornment string for $p(\mathbf{t}_1, \dots, \mathbf{t}_k)$ is a string

$$\alpha = a_1 \cdots a_k$$

defined over the alphabet $\{b, f\}$. For each $i \in \{1, \dots, k\}$ the argument \mathbf{t}_i is either bound (if $a_i = b$) or free (if $a_i = f$).

A formal definition of SIPS for Datalog rules is provided below.

Definition 3.1.3 (SIPS for Datalog Rules). A SIPS for a Datalog rule r with respect to a binding α for the atom $\mathbf{p}(\bar{\mathbf{t}}) \in H(r)$ is a pair

$$(\prec_r^\alpha, f_r^\alpha),$$

where:

- \prec_r^α is a strict partial order over $\text{ATOMS}(r)$; \prec_r^α is such that

$$\mathbf{p}(\bar{\mathbf{t}}) \prec_r^\alpha \mathbf{q}(\bar{\mathbf{s}})$$

holds for all atoms $\mathbf{q}(\bar{\mathbf{s}}) \in \text{ATOMS}(r)$ different from $\mathbf{p}(\bar{\mathbf{t}})$;

- f_r^α is a function assigning to each atom $\mathbf{q}(\bar{\mathbf{s}}) \in \text{ATOMS}(r)$ a subset of the variables in $\bar{\mathbf{s}}$ — intuitively, those made bound after processing $\mathbf{q}(\bar{\mathbf{s}})$; f_r^α must guarantee that $f_r^\alpha(\mathbf{p}(\bar{\mathbf{t}}))$ (the head atom of r) contains all and only the variables of $\mathbf{p}(\bar{\mathbf{t}})$ corresponding to bound arguments in \mathbf{p}^α .

Examples of adornment strings and SIPS are provided below.

Example 3.1.4. Resume from Example 3.1.1. The adornment string associated with the query `ancestorOf(a, Y)` is *bf*. Indeed, only the first argument of `ancestorOf` is bound to a constant in this case. The SIPS $(\prec_{r_{12}}^{bf}, f_{r_{12}}^{bf})$ adopted in Example 3.1.1 for the rule r_{12} and the binding *bf* can be formalized as follows:

- `ancestorOf(X, Y) $\prec_{r_{12}}^{bf}$ parentOf(X, Y)`;
- $f_{r_{12}}^{bf}(\text{ancestorOf}(X, Y)) = \{X\}$;
- $f_{r_{12}}^{bf}(\text{parentOf}(X, Y)) = \{X, Y\}$.

The SIPS $(\prec_{r_{13}}^{bf}, f_{r_{13}}^{bf})$ for r_{13} and *bf*, instead, can be formalized as follows:

- `ancestorOf(X, Y) $\prec_{r_{13}}^{bf}$ parentOf(X, Z) $\prec_{r_{13}}^{bf}$ ancestorOf(Z, Y)`;
- $f_{r_{13}}^{bf}(\text{ancestorOf}(X, Y)) = \{X\}$;
- $f_{r_{12}}^{bf}(\text{parentOf}(X, Z)) = \{X, Z\}$;
- $f_{r_{12}}^{bf}(\text{ancestorOf}(Z, Y)) = \{Z, Y\}$.

A different SIPS for r_{13} and *bf* could be $(\prec_{r_{13}}^{bf}, \hat{f}_{r_{13}}^{bf})$, where $\hat{f}_{r_{13}}^{bf}$ is as follows:

- $\hat{f}_{r_{13}}^{bf}(\text{ancestorOf}(X, Y)) = \{X\}$;
- $\hat{f}_{r_{13}}^{bf}(\text{parentOf}(X, Z)) = \{X\}$;
- $\hat{f}_{r_{13}}^{bf}(\text{ancestorOf}(Z, Y)) = \{Z, Y\}$.

According to this SIPS the atom `parentOf(X, Z)` does not provide a binding for the variable *Z*. \square

Given a query over a Datalog program, the answers associated with the query do not depend on the adopted SIPS, but different SIPS could imply very different computations. Top-down systems like Prolog have to pay quite attention to the choice of SIPS, as adopting some SIPS could result in infinite

computations. For instance, consider program \mathcal{P}_4 below, obtained from \mathcal{P}_3 by slightly modifying the rule r_{13} :

$$\begin{aligned} r_{12} : \text{ancestorOf}(X, Y) &:- \text{parentOf}(X, Y). \\ r_{14} : \text{ancestorOf}(X, Y) &:- \text{ancestorOf}(X, Z), \text{parentOf}(Z, Y). \end{aligned}$$

Even if \mathcal{P}_3 and \mathcal{P}_4 are equivalent, a top-down evaluation of \mathcal{P}_4 following a left-to-right selection strategy would not terminate because of r_{14} . Indeed, to answer $\text{ancestorOf}(a, Y)$, the subquery $\text{ancestorOf}(a, Z)$ has to be processed first. But $\text{ancestorOf}(a, Z)$ and $\text{ancestorOf}(a, Y)$ are actually the same query, which means that the computation starts an infinite loop. Usually, top-down systems adopt special techniques, like for instance tabling, to overcome these problems. These techniques are out of the scope of this thesis, as termination is guaranteed for bottom-up systems. All the algorithms and techniques we shall develop and discuss in this thesis are orthogonal with respect to the underlying SIPS to be used in hypothetical top-down evaluations. Therefore, hereinafter Datalog programs are assumed to be provided in input together with some arbitrarily defined SIPS $(\prec_r^\alpha, f_r^\alpha)$, for each rule r and for each possible adornment string α of the atom in $H(r)$.

3.1.2 Classic Magic Sets

The original Magic Set technique, referred to as Classic Magic Sets in this thesis, has been defined in the context of Datalog. In this section we present the details of the algorithm implementing Classic Magic Sets. Minor changes to the original algorithm have been made for allowing a better comparison with the other Magic Set techniques presented in this thesis. The reader is referred to [65] for a detailed presentation of the original algorithm. Additional definitions and notations that will be used in the presentation of Classic Magic Sets are given below.

Definition 3.1.5 (Adorned Atom). Let α be an adornment string. Therefore, p^α and $p^\alpha(\bar{t})$ are, respectively, the adorned predicate and atom associated with $p(\bar{t})$ and α . The arguments of $p^\alpha(\bar{t})$ are considered either bound or free according to the adornment string α .

Adorned atoms are associated with magic atoms, which are used for identifying relevant atoms for answering an input query.

Definition 3.1.6 (Magic Atom). Let $p^\alpha(\bar{t})$ be an adorned atom. The *magic version* of $p^\alpha(\bar{t})$, denoted by $\text{magic}(p^\alpha(\bar{t}))$, is an atom $\text{magic}_p^\alpha(\bar{t}')$ such that:

- \bar{t}' is obtained from \bar{t} by eliminating all arguments corresponding to an f label in α ;
- magic_p^α is a new predicate symbol, for simplicity denoted by attaching the prefix “magic_” to the predicate symbol p^α (we assume that no standard predicate in \mathcal{P} has the prefix “magic_”).

We are now ready to describe the CMS algorithm, reported in Figure 3.2. CMS starts with a query Q over a Datalog program \mathcal{P} and outputs a rewritten program $\text{CMS}(Q, \mathcal{P})$. The method uses two sets, S and D , to store adorned

```

Algorithm CMS( $\mathcal{Q}, \mathcal{P}$ )
Input: A query  $\mathcal{Q}$  and a Datalog program  $\mathcal{P}$ 
Output: A rewritten Datalog program
var
   $S, D$ : set of adorned predicates;
   $R_{\mathcal{Q}, \mathcal{P}}^{mgc}, R_{\mathcal{Q}, \mathcal{P}}^{mod}$ : set of rules;
   $r^a$ : adorned rule;
begin
  1.  $S := \emptyset; D := \emptyset; R_{\mathcal{Q}, \mathcal{P}}^{mgc} := \emptyset; R_{\mathcal{Q}, \mathcal{P}}^{mod} := \emptyset;$ 
  2. ProcessQuery( $\mathcal{Q}, S, R_{\mathcal{Q}, \mathcal{P}}^{mod}, R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ );
  3. while  $S \neq \emptyset$  do
  4.   take an element  $\mathbf{p}^\alpha$  from  $S$ ; remove  $\mathbf{p}^\alpha$  from  $S$ ; add  $\mathbf{p}^\alpha$  to  $D$ ;
  5.   for each rule  $r \in \mathcal{P}$  such that  $H(r) = \{\mathbf{p}(\bar{\tau})\}$  do
  6.      $r^a := \mathbf{Adorn}(r, \alpha, S, D)$ ;
  7.      $R_{\mathcal{Q}, \mathcal{P}}^{mgc} := R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup \mathbf{Generate}(r, \alpha, r^a)$ ;
  8.     add Modify( $r^a$ ) to  $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ ;
  9.   end for
  10. end while
  11. return  $R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup R_{\mathcal{Q}, \mathcal{P}}^{mod} \cup \text{EDB}(\mathcal{P})$ ;
end.

```

Figure 3.2: Classic Magic Sets algorithm for Datalog programs

predicates to be propagated and already processed, respectively. Magic rules are stored in the set $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, modified rules in $R_{\mathcal{Q}, \mathcal{P}}^{mod}$. Initially, all sets S , D , $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ and $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ are empty (line 1). The algorithm starts by processing the query (line 2), which includes putting the adorned version of the query predicate into S . After that, the main loop of the algorithm is repeated until S is empty (lines 3–10). More specifically, an adorned predicate \mathbf{p}^α is moved from S to D (line 4) and each rule r having an atom $\mathbf{p}(\bar{\tau})$ in the head is considered (lines 5–9). The adorned version r^a of the rule r is computed (this step may also put new adorned predicates into S ; line 6), from which magic rules (line 7) and a modified rule r' are generated (line 8). Finally, the algorithm terminates returning the program obtained by the union of $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ and $\text{EDB}(\mathcal{P})$ (line 11). The details of the four auxiliary functions in CMS are discussed in the remainder of this section. The query and the program from Example 3.1.1 and the associated SIPS presented in Example 3.1.4 are used as running example.

Function 1: ProcessQuery

The first function of CMS, **ProcessQuery**, is reported in Figure 3.3. For a query $\mathcal{Q} = \mathbf{p}(\bar{\tau})$, **ProcessQuery** builds an adornment string α for the predicate \mathbf{p} (lines 2–5). The element in position i of α is b if the i -th argument of $\mathbf{p}(\bar{\tau})$ is a constant, otherwise the element in position i of α is f . After that, the adorned predicate \mathbf{p}^α is added to S (line 6) in order to be subsequently processed for binding propagation. Moreover, the function builds a query seed and a rule for

```

Function ProcessQuery( $\mathcal{Q}$ ,  $S$ ,  $R_{\mathcal{Q},\mathcal{P}}^{mod}$ ,  $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ )
Input
   $\mathcal{Q}$ : query;
   $S$ : set of adorned predicates;
   $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ ,  $R_{\mathcal{Q},\mathcal{P}}^{mod}$ : set of rules;
Output: none;
var
   $\alpha$ : adornment string;
begin
  1. Let  $p(\bar{t})$  be the atom in  $\mathcal{Q}$ ;
  2.  $\alpha := \epsilon$ ;
  3. for each argument  $t$  in  $\bar{t}$  do
  4.   if  $t$  is a constant then  $\alpha := \alpha b$ ; else  $\alpha := \alpha f$ ; end if
  5. end for
  6. add  $p^\alpha$  to  $S$ ;
  7. add magic( $p^\alpha(\bar{t})$ ) to  $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ ;
  8. add  $p(\bar{X}) :- p^\alpha(\bar{X})$  to  $R_{\mathcal{Q},\mathcal{P}}^{mod}$ ;
end.

```

Figure 3.3: ProcessQuery function for Classic Magic Sets

gathering answers:

$$\begin{aligned} & \text{magic}(p^\alpha(\bar{t})). \\ & p(\bar{X}) :- p^\alpha(\bar{X}). \end{aligned}$$

where \bar{X} is a list of variables. The query seed is stored into the set $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ (line 7), while the gathering rule into the set $R_{\mathcal{Q},\mathcal{P}}^{mod}$ (line 8).

Example 3.1.7. For the query \mathcal{Q}_3 from Example 3.1.1 the function **ProcessQuery** builds:

- the adorned predicate **ancestorOf**^{bf};
- the query seed **magic_ancestorOf**^{bf}(a);
- the gathering rule **ancestorOf**(X, Y) :- **ancestorOf**^{bf}(X, Y).

□

Function 2: Adorn

The key idea of Magic Sets is to materialize the binding information for IDB predicates that would be propagated during a top-down computation. Hence, after processing the query, each adorned predicate is eventually used to propagate its information into the body of the rules defining it. This kind of propagation is performed according to an appropriate SIPS, thereby simulating a top-down evaluation of the query. In particular, if a binding α has to be propagated into a rule r with head $p(\bar{t})$, the associated SIPS $(\prec_r^\alpha, f_r^\alpha)$ determines which variables are bound in the evaluation of each atom of r . More specifically, a variable X of an atom $q(\bar{s})$ in $\text{ATOMS}(r)$ is bound if and only if either

```

Function Adorn( $r, \alpha, S, D$ )
Input
   $r$ : rule;
   $\alpha$ : adornment string;
   $S, D$ : set of rules;
Output: an adorned rule;
var
   $r^a$ : adorned rule;
   $\alpha_i$ : adornment string;
begin
  1. Let  $p(\bar{t})$  be the atom in  $H(r)$ .
  2. Let  $(\prec_r^\alpha, f_r^\alpha)$  be the SIPS associated with  $r$  and  $\alpha$ .
  3.  $r^a := r$ ;
  4. for each IDB atom  $p_i(\bar{t}_i)$  in  $ATOMS(r)$  do
  5.    $\alpha_i := \epsilon$ ;
  6.   for each argument  $t$  in  $\bar{t}$  do
  7.     if  $t$  is a constant then
  8.        $\alpha_i := \alpha_i b$ ;
  9.     else
  10.      Argument  $t$  is a variable. Let  $X$  be such a variable.
  11.      if  $X \in f_r^\alpha(p(\bar{t}))$  or there is  $b(\bar{v})$  in  $B^+(r)$  such that
  12.         $b(\bar{v}) \prec_r^\alpha q(\bar{s})$  and  $X \in f_r^\alpha(b(\bar{v}))$  then
  13.           $\alpha_i := \alpha_i b$ ;
  14.        else
  15.           $\alpha_i := \alpha_i f$ ;
  16.        end if
  17.      end if
  18.    end for
  19.    substitute  $p_i(\bar{t}_i)$  in  $r^a$  with  $p_i^{\alpha_i}(\bar{t}_i)$ ;
  20.    if set  $D$  does not contain  $p_i^{\alpha_i}$  then add  $p_i^{\alpha_i}$  to  $S$ ; end if
  21.  end for
  22. return  $r^a$ ;
end.

```

Figure 3.4: Adorn function for Classic Magic Sets

-
- $X \in f_r^\alpha(p(\bar{t}))$, or
 - there is an atom $b(\bar{v}) \in B^+(r)$ such that $b(\bar{v}) \prec_r^\alpha q(\bar{s})$ and $X \in f_r^\alpha(b(\bar{v}))$.

The function **Adorn**, reported in Figure 3.4, takes care of the materialization of the binding information for IDB predicates. Given a rule r and a binding α , the function builds an adorned rule r^a starting from r (line 3) and substituting all IDB predicates by adorned predicates obtained according to the SIPS $(\prec_r^\alpha, f_r^\alpha)$ associated with r and α (lines 4–21). Each new adorned predicate generated in this phase is added to S unless it has been produced previously (that is, unless the adorned predicate does belong to D ; line 20). Finally, the function terminates, returning the adorned rule r^a (line 22).

Example 3.1.8. The adorned predicate `ancestorOfbf` has been added to S while processing the query `ancestorOf(a, Y)` in Example 3.1.7. After that, `ancestorOfbf` is moved from S to D (line 4 of Figure 3.2) and its binding information bf is propagated in all rules defining the predicate `ancestorOf` (lines 5–9 of Figure 3.2). In this case the for loop is repeated twice, for r_{12} and r_{13} , and

```

Function Generate( $r, \alpha, r^a$ )
Input
   $r$ : rule;
   $\alpha$ : adornment string;
   $r^a$ : adorned rule;
Output: a set of magic rules;
var
   $R$ : set of rules;
   $r^*$ : rule;
begin
  1. Let  $p^\alpha(\bar{t})$  be the atom in  $H(r^a)$ .
  2. Let  $(\prec_r^\alpha, f_r^\alpha)$  be the SIPS associated with  $r$  and  $\alpha$ .
  3.  $R := \emptyset$ ;
  4. for each atom  $p_i^{\alpha_i}(\bar{t}_i)$  in  $B^+(r^a)$  do
  5.   if  $\alpha_i \neq \epsilon$  then
  6.      $r^* := \text{magic}(p_i^{\alpha_i}(\bar{t}_i)) :- \text{magic}(p^\alpha(\bar{t}))$ ;
  7.     for each atom  $p_j^{\alpha_j}(\bar{t}_j)$  in  $B^+(r^a)$  such that  $p_j(\bar{t}_j) \prec_r^\alpha p_i(\bar{t}_i)$  do
  8.       add atom  $p_j^{\alpha_j}(\bar{t}_j)$  to  $B^+(r^*)$ ;
  9.     end for
  10.    add  $r^*$  to  $R$ ;
  11.   end if
  12. end for
  13. return  $R$ ;
end.

```

Figure 3.5: Generate function for Classic Magic Sets

the adorned rules r_{12}^a, r_{13}^a (reported below) are generated.² The order in which rules are processed is negligible, so let us assume that r_{12} is processed first. According to the SIPS $(\prec_{r_{12}}^{bf}, f_{r_{12}}^{bf})$ defined in Example 3.1.4, when the binding information bf is propagated into r_{12} , the following adorned rule is produced:

$$r_{12}^a : \text{ancestorOf}^{bf}(X, Y) :- \text{parentOf}(X, Y).$$

Instead, when propagating bf into rule r_{13} according to the SIPS $(\prec_{r_{13}}^{bf}, f_{r_{13}}^{bf})$ defined in Example 3.1.4, the following adorned rule is obtained:

$$r_{13}^a : \text{ancestorOf}^{bf}(X, Y) :- \text{parentOf}(X, Z), \text{ancestorOf}^{bf}(Z, Y).$$

Note that adornment strings are only applied to IDB predicates; hence, the EDB predicate `parentOf` is not adorned in the rules r_{12}^a and r_{13}^a . Note also that no new adorned predicates are generated in this case. \square

Function 3: Generate

The function **Generate**, reported in Figure 3.5, is used for producing *magic rules* from each adorned rule produced by **Adorn**. When **Generate** is invoked for an adorned rule r^a with head atom $p^\alpha(\bar{t})$, which has been obtained by adorning a rule r with respect to the adornment string α , a magic rule r^* is

²We recall here that each adorned rule produced in this step of the algorithm is subsequently processed by the functions **Generate** and **Modify** (see Examples 3.1.9 and 3.1.10).

```

Function Modify( $r^a$ )
Input
   $r^a$ : adorned rule;
Output: a modified rule;
var
   $r'$ : rule;
begin
  1. Let  $p^\alpha(\bar{t})$  be the atom in  $H(r^a)$ .
  2.  $r' := r^a$ ;
  3. add  $\text{magic}(p^\alpha(\bar{t}))$  to  $B^+(r')$ ;
  4. return  $r'$ ;
end.

```

Figure 3.6: Modify function for Classic Magic Sets

produced for each atom $p_i^{\alpha_i}(\bar{t}_i)$ in the body of r^a such that α_i is not the empty string: The head atom of r^* is $\text{magic}(q_i^{\beta_i}(\bar{s}_i))$, while the body of r^* consists of $\text{magic}(p^\alpha(\bar{t}))$ and all atoms $q_j^{\beta_j}(\bar{s}_j)$ in $B^+(r)$ such that $q_j(\bar{s}_j) \prec_r^\alpha q_i(\bar{s}_i)$ holds.

Example 3.1.9. In Example 3.1.8 two adorned rules have been produced, namely r_{12}^a and r_{13}^a . When **Generate** is invoked for rule r_{12}^a , an empty set is returned. Indeed, in this case $B^+(r_{12}^a)$ does not contain IDB predicates. For rule r_{13}^a , instead, **Generate** returns the following magic rule:

$$r_{13}^* : \text{magic_ancestorOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Z).$$

□

Function 4: Modify

Each adorned rule produced by **Adorn** is also processed by the function **Modify** reported in Figure 3.6. In particular, given an adorned rule r^a with head atom $p^\alpha(\bar{t})$, **Modify** returns a *modified rule* r' obtained from r^a by adding the magic atom $\text{magic}(p^\alpha(\bar{t}))$ to its body. This magic atom limits the range of the head variables during program instantiation.

Example 3.1.10. For the adorned rules r_{12}^a and r_{13}^a from Example 3.1.8 the following modified rules are generated:

$$r'_{12} : \text{ancestorOf}^{bf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y).$$

$$r'_{13} : \text{ancestorOf}^{bf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X),$$

$$\text{parentOf}(X, Z), \text{ancestorOf}^{bf}(Z, Y).$$

□

The correctness of Classic Magic Sets is well known for Datalog programs; see for instance [65].

Example 3.1.11. The complete program $\text{CMS}(\mathcal{Q}_3, \mathcal{P}_3)$ is reported below:³

```

magic_ancestorOfbf(a).
ancestorOf(X, Y) :- ancestorOfbf(X, Y).
r13* : magic_ancestorOfbf(Z) :- magic_ancestorOfbf(X), parentOf(X, Z).
r12' : ancestorOfbf(X, Y) :- magic_ancestorOfbf(X), parentOf(X, Y).
r13' : ancestorOfbf(X, Y) :- magic_ancestorOfbf(X),
                                parentOf(X, Z), ancestorOfbf(Z, Y).

```

The rewritten program above is equivalent to the original program with respect to the query $\text{ancestorOf}(\mathbf{a}, Y)$. Indeed, each answer ϑ for \mathcal{Q}_3 and \mathcal{P}_3 derived by means of r_{12} is such that $\text{parentOf}(\mathbf{a}, Y)\vartheta$ occurs as a fact. Answers of this kind are also derived by $\text{CMS}(\mathcal{Q}_3, \mathcal{P}_3)$ thanks to the magic seed $\text{magic_ancestorOf}^{\text{bf}}(\mathbf{a})$ and the rule r'_{12} . All other answers are obtained from r_{13} by combining instances of $\text{parentOf}(\mathbf{a}, Z)$ and $\text{ancestorOf}(Z, Y)$ matching on Z . All answers of this kind are also provided by $\text{CMS}(\mathcal{Q}_3, \mathcal{P}_3)$ thanks to the magic rule r_{13}^* . We also note that the program $\text{CMS}(\mathcal{Q}_3, \mathcal{P}_3)$ is an optimized version of \mathcal{P}_3 . Indeed, only rules defining relevant atoms are produced during program instantiation. For instance, no rule defining $\text{ancestorOf}(\mathbf{d}, \mathbf{c})$ is generated during the program instantiation of $\text{CMS}(\mathcal{Q}_3, \mathcal{P}_3)$ because the magic atom $\text{magic_ancestorOf}^{\text{bf}}(\mathbf{d})$ cannot be produced in this case. \square

3.2 Magic Sets for Disjunctive Datalog

Static and Dynamic Magic Sets are two extensions of Classic Magic Sets that allow for query optimization over Disjunctive Datalog programs. The details of the two techniques are discussed in this section. In particular, the section is structured as follows. In Section 3.2.1 we present the main ideas that have been used for enabling the Magic Set method to work on Disjunctive Datalog programs. After that, Static Magic Sets are introduced in Section 3.2.2 and Dynamic Magic Sets in Section 3.2.3. Finally, in Section 3.2.4 the correctness of Dynamic Magic Sets is established for $\text{Datalog}^{\vee, \neg, s}$ and $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ programs.

3.2.1 SIPS for Disjunctive Datalog Rules

Sideways information passing strategies (SIPS) for Datalog rules have been discussed in Section 3.1.1. In particular, SIPS have been presented as a convenient way for identifying relevant atoms for answering queries over Datalog programs. In this section we extend the concept of SIPS to the framework of Disjunctive Datalog.

As first observed in [38], while in Datalog bindings have to be propagated only from head atoms to body atoms, in Disjunctive Datalog bindings have to be propagated also from head atoms to head atoms. Indeed, since stable models are subset-minimal, a rule r can support the truth of an atom $\mathbf{p}(\bar{\mathbf{t}})$ in $H(r)$ only if all atoms in $H(r) \setminus \{\mathbf{p}(\bar{\mathbf{t}})\}$ are false. Hence, when processing a rule r and a

³EDB facts are usually omitted in examples because only IDB rules are rewritten by Magic Set techniques.

(sub)query $p(\bar{c}) \in H(r)$, all atoms in $ATOMS(r) \setminus \{p(\bar{c})\}$ have to be considered as subqueries, which means that instances of these atoms may be relevant for answering the (sub)query $p(\bar{c})$. An example is given below.

Example 3.2.1. Consider the following Datalog ^{\vee, \neg} program:

$$p(X) \vee q(Y) :- b(X, Y).$$

Suppose we are interested in answering the following query:

$$p(1)?$$

In this case the binding information b for predicate p is propagated from the query $p(1)$ to the disjunctive rule above. In particular, the atom $q(Y)$ has to be evaluated in order to answer the query $p(1)$. Let us assume that the adopted SIPS provides a binding for the variable Y of $q(Y)$ through the atom $b(X, Y)$, and that the EDB of the program consists of the facts $b(1, 2)$ and $b(2, 3)$. Thus, the atom $q(2)$ is considered as a subquery, while the atom $q(3)$ is not. \square

In order to properly extend the concept of SIPS to Disjunctive Datalog rules, the following aspects have to be taken into account:

- rule heads may have more than one atom; therefore, the SIPS depends on the specific head atom matching the (sub)query;
- the only head atom which can pass bindings is the one matching the (sub)query;
- negative literals cannot pass bindings to other atoms.

These aspects are formalized below.

Definition 3.2.2 (SIPS for Datalog ^{\vee, \neg} Rules). A SIPS for a Datalog ^{\vee, \neg} rule r with respect to a binding α for an atom $p(\bar{c}) \in H(r)$ is a pair

$$(\prec_r^{p^\alpha(\bar{c})}, f_r^{p^\alpha(\bar{c})}),$$

where:

- $\prec_r^{p^\alpha(\bar{c})}$ is a strict partial order over the atoms in $ATOMS(r)$; $\prec_r^{p^\alpha(\bar{c})}$ is such that

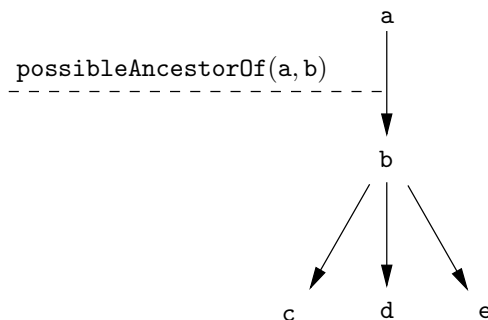
$$p(\bar{c}) \prec_r^{p^\alpha(\bar{c})} q(\bar{s})$$

holds for all atoms $q(\bar{s}) \in ATOMS(r)$ different from $p(\bar{c})$, and

$$q_i(\bar{s}_i) \prec_r^{p^\alpha(\bar{c})} q_j(\bar{s}_j)$$

implies that atom $q_i(\bar{s}_i)$ belongs to $B^+(r) \cup \{p(\bar{c})\}$;

- $f_r^{p^\alpha(\bar{c})}$ is a function assigning to each atom $q(\bar{s}) \in ATOMS(r)$ a subset of the variables in \bar{s} — intuitively, those made bound after processing $q(\bar{s})$; $f_r^{p^\alpha(\bar{c})}$ is such that $f_r^{p^\alpha(\bar{c})}(p(\bar{c}))$ contains all and only the variables of $p(\bar{c})$ corresponding to bound arguments in p^α .

Figure 3.7: Graphical representation of EDB \mathcal{F}_2 for program \mathcal{P}_5

Note that Definition 3.1.3 and Definition 3.2.2 coincide for Datalog programs because Datalog rules have atomic heads and positive bodies. Below is an example of SIPS for a Disjunctive Datalog program.

Example 3.2.3. In Example 3.1.1 we have considered a program defining an IDB predicate `ancestorOf` as the transitive closure of an EDB predicate `parentOf`. This program has been used to determine all descendants of a given person `a`. Let us now assume that only *uncertain* knowledge is contained in the database. In particular, assume that a superset of the relation `parentOf(X, Y)` is stored by means of the EDB predicate `possibleParentOf`. In this case the relation `parentOf(X, Y)` can be reconstructed by guessing a subset of the relation `possibleParentOf(X, Y)`. Therefore, the descendants of `a` can be retrieved by means of the query Q_4 and the program \mathcal{P}_5 below:

$$\begin{aligned} Q_4 &: \text{ancestorOf}(a, Y)? \\ r_{15} &: \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{possibleParentOf}(X, Y). \\ r_{16} &: \text{ancestorOf}(X, Y) :- \text{parentOf}(X, Y). \\ r_{17} &: \text{ancestorOf}(X, Y) :- \text{ancestorOf}(X, Z), \text{parentOf}(Z, Y). \end{aligned}$$

In particular, the relation `parentOf(X, Y)` is guessed by the rule r_{15} ,⁴ and its transitive closure is computed as before by the rules r_{16} and r_{17} . Assume now the following EDB, graphically represented in Figure 3.7:

$$\mathcal{F}_2 = \{\text{possibleParentOf}(a, b), \text{possibleParentOf}(b, c), \text{possibleParentOf}(b, d), \text{possibleParentOf}(b, e)\}.$$

In this case program \mathcal{P}_5 has sixteen stable models, one for each possible subset of \mathcal{F}_2 . Hence, the answers to the query Q_4 are the following sets:

⁴In order to avoid loops in relation `parentOf`, program \mathcal{P}_5 could be modified by adding the following rule:

$$\text{nonParentOf}(X, Y) :- \text{possibleParentOf}(X, Y), \text{ancestorOf}(Y, X).$$

For simplicity, in the example we assume that loops cannot occur, for instance by assuming that the relation `possibleParentOf(X, Y)` is acyclic.

- $Ans_b(\mathcal{Q}_4, \mathcal{P}_5) = \{\mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}$ (brave answers);
- $Ans_c(\mathcal{Q}_4, \mathcal{P}_5) = \emptyset$ (cautious answers).

We now describe the adornments produced and propagated in a hypothetical top-down evaluation of \mathcal{Q}_4 over \mathcal{P}_5 . The first binding information comes from the query and can be represented by an adorned predicate $\mathbf{ancestorOf}^{bf}$. This binding information is propagated in the rules r_{16} and r_{17} according to some properly defined SIPS. Let us assume that the SIPS

$$(\prec_{r_{16}}^{\mathbf{ancestorOf}^{bf}(X,Y)}, f_{r_{16}}^{\mathbf{ancestorOf}^{bf}(X,Y)}) \text{ and } (\prec_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)}, f_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)})$$

are defined as follows:

- $\mathbf{ancestorOf}(X, Y) \prec_{r_{16}}^{\mathbf{ancestorOf}^{bf}(X,Y)} \mathbf{parentOf}(X, Y)$;
- $f_{r_{16}}^{\mathbf{ancestorOf}^{bf}(X,Y)}(\mathbf{ancestorOf}(X, Y)) = \{X\}$;
- $f_{r_{16}}^{\mathbf{ancestorOf}^{bf}(X,Y)}(\mathbf{parentOf}(X, Y)) = \{X, Y\}$;
- $\mathbf{ancestorOf}(X, Y) \prec_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)} \mathbf{ancestorOf}(X, Z)$;
- $\mathbf{ancestorOf}(X, Y) \prec_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)} \mathbf{parentOf}(Z, Y)$;
- $\mathbf{ancestorOf}(X, Z) \prec_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)} \mathbf{parentOf}(Z, Y)$;
- $f_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)}(\mathbf{ancestorOf}(X, Y)) = \{X\}$;
- $f_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)}(\mathbf{ancestorOf}(X, Z)) = \{X, Z\}$;
- $f_{r_{17}}^{\mathbf{ancestorOf}^{bf}(X,Y)}(\mathbf{parentOf}(Z, Y)) = \{Z, Y\}$.

Hence, a new binding bf for the predicate $\mathbf{parentOf}$ is produced, and $\mathbf{parentOf}^{bf}$ is propagated in the rule r_{15} . Let us assume that the SIPS

$$(\prec_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)}, f_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)})$$

is as follows:

- $\mathbf{parentOf}(X, Y) \prec_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)} \mathbf{possibleParentOf}(X, Y)$;
- $\mathbf{parentOf}(X, Y) \prec_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)} \mathbf{nonParentOf}(X, Y)$;
- $f_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)}(\mathbf{parentOf}(X, Y)) = f_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)}(\mathbf{nonParentOf}(X, Y)) = \{X\}$;
- $f_{r_{15}}^{\mathbf{parentOf}^{bf}(X,Y)}(\mathbf{possibleParentOf}(X, Y)) = \{X, Y\}$.

Thus, a new binding bf for predicate $\mathbf{nonParentOf}$ is produced and propagated in the rule r_{15} . Let assume that the SIPS

$$(\prec_{r_{15}}^{\mathbf{nonParentOf}^{bf}(X,Y)}, f_{r_{15}}^{\mathbf{nonParentOf}^{bf}(X,Y)})$$

is defined as follows:

- $\text{nonParentOf}(X, Y) \prec_{r_{15}}^{\text{nonParentOf}^{bf}(X, Y)} \text{possibleParentOf}(X, Y);$
- $\text{nonParentOf}(X, Y) \prec_{r_{15}}^{\text{nonParentOf}^{bf}(X, Y)} \text{parentOf}(X, Y);$
- $f_{r_{15}}^{\text{nonParentOf}^{bf}(X, Y)}(\text{nonParentOf}(X, Y)) = \{X\};$
- $f_{r_{15}}^{\text{nonParentOf}^{bf}(X, Y)}(\text{parentOf}(X)) = \{X\};$
- $f_{r_{15}}^{\text{nonParentOf}^{bf}(X, Y)}(\text{possibleParentOf}(X, Y)) = \{X, Y\}.$

Note that the choice of SIPS for all other pairs of rules and adorned predicates is negligible, as these combinations do not occur while Q_4 is evaluated over \mathcal{P}_5 according to the SIPS reported above. \square

We recall that all the algorithms and techniques discussed in this thesis are orthogonal with respect to the underlying SIPS to be used in a hypothetical top-down evaluation. Therefore, hereinafter Datalog $^{\vee, \neg}$ programs are assumed to be provided in input together with some arbitrarily defined SIPS

$$(\prec_r^{\mathbf{p}^\alpha(\bar{\mathbf{t}})}, f_r^{\mathbf{p}^\alpha(\bar{\mathbf{t}})}),$$

for each rule r , for each atom $\mathbf{p}(\bar{\mathbf{t}}) \in H(r)$, and for each possible adornment string α for $\mathbf{p}(\bar{\mathbf{t}})$.

3.2.2 Static Magic Sets

The new notion of SIPS is not enough for extending Classic Magic Sets to Disjunctive Datalog. In fact, adorning a rule may give rise to multiple rules with different adornments for head atoms. While this is not a problem for standard Datalog programs, the semantics of a disjunctive program may be affected. For example, let us consider the following query and program:

$$\begin{aligned} & \mathbf{g}(\mathbf{a})? \\ & \text{edb}(\mathbf{a}, \mathbf{a}). \\ r_{18} : & \mathbf{g}(X) :- \mathbf{p}(X, Y), \mathbf{q}(Z, X). \\ r_{19} : & \mathbf{p}(X, Y) \vee \mathbf{q}(X, Y) :- \text{edb}(X, Y). \end{aligned}$$

Since r_{19} is the only rule defining \mathbf{p} and \mathbf{q} , for each substitution ϑ such that the atom $\text{edb}(X, Y)\vartheta$ occurs in the EDB, every stable model of the program above has to contain exactly one of $\mathbf{p}(X, Y)\vartheta$ and $\mathbf{q}(X, Y)\vartheta$ (because of minimality). In particular, the program above has two stable models, one containing $\mathbf{p}(\mathbf{a}, \mathbf{a})$ and the other $\mathbf{q}(\mathbf{a}, \mathbf{a})$. In these two stable models the query atom $\mathbf{g}(\mathbf{a})$ is false, which implies that the query is neither a brave nor a cautious consequence of the program above. Now let us try to apply the CMS algorithm in a naive way, by just changing the kind of SIPS for dealing with disjunctive heads. When processing the query, the adorned predicate \mathbf{g}^b is produced. After that, this binding information is propagated in the rule r_{18} and the following adorned

rule is produced:⁵

$$r_{18}^a : \mathbf{g}^b(\mathbf{X}) :- \mathbf{p}^{bf}(\mathbf{X}, \mathbf{Y}), \mathbf{q}^{fb}(\mathbf{Z}, \mathbf{X}).$$

Two new adorned predicates have been produced, namely \mathbf{p}^{bf} and \mathbf{q}^{fb} . If the atom $\mathbf{edb}(\mathbf{X}, \mathbf{Y})$ does not provide any binding for the variables \mathbf{X} and \mathbf{Y} , when propagating \mathbf{p}^{bf} and \mathbf{q}^{fb} in r_{19} the following adorned rules are generated:

$$r_{19,1}^a : \mathbf{p}^{bf}(\mathbf{X}, \mathbf{Y}) \vee \mathbf{q}^{bf}(\mathbf{X}, \mathbf{Y}) :- \mathbf{edb}(\mathbf{X}, \mathbf{Y}).$$

$$r_{19,2}^a : \mathbf{p}^{fb}(\mathbf{X}, \mathbf{Y}) \vee \mathbf{q}^{fb}(\mathbf{X}, \mathbf{Y}) :- \mathbf{edb}(\mathbf{X}, \mathbf{Y}).$$

These rules might support two atoms $\mathbf{p}^{bf}(\mathbf{a}, \mathbf{a})$ and $\mathbf{q}^{fb}(\mathbf{a}, \mathbf{a})$, thus giving a chance to the query atom $\mathbf{g}(\mathbf{a})$ to be true. Indeed, the application of CMS would generate the following program:

$$\begin{aligned} & \mathbf{magic_g}^b(\mathbf{a}). \\ & \mathbf{g}(\mathbf{X}) :- \mathbf{g}^b(\mathbf{X}). \\ r'_{18} : & \mathbf{g}^b(\mathbf{X}) :- \mathbf{magic_g}^b(\mathbf{X}), \mathbf{p}^{bf}(\mathbf{X}, \mathbf{Y}), \mathbf{q}^{fb}(\mathbf{Z}, \mathbf{X}). \\ r'_{19,1} : & \mathbf{p}^{bf}(\mathbf{X}, \mathbf{Y}) \vee \mathbf{q}^{bf}(\mathbf{X}, \mathbf{Y}) :- \mathbf{magic_p}^{bf}(\mathbf{X}), \mathbf{magic_q}^{bf}(\mathbf{X}), \mathbf{edb}(\mathbf{X}, \mathbf{Y}). \\ r'_{19,2} : & \mathbf{p}^{fb}(\mathbf{X}, \mathbf{Y}) \vee \mathbf{q}^{fb}(\mathbf{X}, \mathbf{Y}) :- \mathbf{magic_p}^{fb}(\mathbf{Y}), \mathbf{magic_q}^{fb}(\mathbf{Y}), \mathbf{edb}(\mathbf{X}, \mathbf{Y}). \\ \\ r^*_{18} : & \mathbf{magic_p}^{bf}(\mathbf{X}) :- \mathbf{magic_g}^b(\mathbf{X}). \\ r^*_{18} : & \mathbf{magic_q}^{fb}(\mathbf{X}) :- \mathbf{magic_g}^b(\mathbf{X}). \\ r^*_{19,1} : & \mathbf{magic_q}^{bf}(\mathbf{X}) :- \mathbf{magic_p}^{bf}(\mathbf{X}). \\ r^*_{19,2} : & \mathbf{magic_p}^{bf}(\mathbf{X}) :- \mathbf{magic_q}^{bf}(\mathbf{X}). \\ r^*_{19,3} : & \mathbf{magic_q}^{fb}(\mathbf{X}) :- \mathbf{magic_p}^{fb}(\mathbf{X}). \\ r^*_{19,4} : & \mathbf{magic_p}^{fb}(\mathbf{X}) :- \mathbf{magic_q}^{fb}(\mathbf{X}). \end{aligned}$$

We recall that the original EDB consists only of the atom $\mathbf{edb}(\mathbf{a}, \mathbf{a})$. Hence, the program above admits a stable model containing both $\mathbf{p}^{bf}(\mathbf{a}, \mathbf{a})$ and $\mathbf{q}^{fb}(\mathbf{a}, \mathbf{a})$. Thus, this stable model also contains $\mathbf{g}^b(\mathbf{a})$ because of r'_{18} . The truth of $\mathbf{g}^b(\mathbf{a})$ in turn implies that the query atom $\mathbf{g}(\mathbf{a})$ is true, which means that $\mathbf{g}(\mathbf{a})$ is a brave consequence of the rewritten program (while it was not a brave consequence of the original program).

Static Magic Sets (SMS) circumvents this problem by using some auxiliary predicates that collect all facts coming from different adornments. For instance, for the predicate \mathbf{p} in the previous example, the following collector rules are introduced by SMS:

$$\mathbf{collect_p}(\mathbf{X}, \mathbf{Y}) :- \mathbf{p}^{bf}(\mathbf{X}, \mathbf{Y}).$$

$$\mathbf{collect_p}(\mathbf{X}, \mathbf{Y}) :- \mathbf{p}^{fb}(\mathbf{X}, \mathbf{Y}).$$

Collector predicates of SMS store a sizable superset of the atoms which are

⁵Note that in this specific case the rule produced by CMS does not depend on the chosen SIPS. In fact, every valid SIPS would give rise to the production of this rule.

```

Algorithm SMS( $\mathcal{Q}, \mathcal{P}$ )
Input: A query  $\mathcal{Q}$  and a Datalog $\vee, \neg$  program  $\mathcal{P}$ 
Output: A rewritten Datalog $\vee, \neg$  program
var
   $S, D$ : set of adorned predicates;
   $R_{\mathcal{Q}, \mathcal{P}}^{mgc}, R_{\mathcal{Q}, \mathcal{P}}^{mod}$ : set of rules;
   $r^a$ : adorned rule;
begin
  1.  $S := \emptyset$ ;  $D := \emptyset$ ;  $R_{\mathcal{Q}, \mathcal{P}}^{mgc} := \emptyset$ ;  $R_{\mathcal{Q}, \mathcal{P}}^{mod} := \emptyset$ ;
  2. ProcessQuery $\vee$ ( $\mathcal{Q}, S, R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ );
  3. while  $S \neq \emptyset$  do
  4.   take an element  $p^\alpha$  from  $S$ ; remove  $p^\alpha$  from  $S$ ; add  $p^\alpha$  to  $D$ ;
  5.   for each rule  $r$  in  $\mathcal{P}$  and for each atom  $p(\bar{t})$  in  $H(r)$  do
  6.      $r^a := \mathbf{Adorn}^\vee(r, \alpha, S, D)$ ;
  7.      $R_{\mathcal{Q}, \mathcal{P}}^{mgc} := R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup \mathbf{Generate}^{\text{SMS}}(r, \alpha, r^a)$ ;
  8.   end for
  9.   add collect- $p(\bar{X}) :- p^\alpha(\bar{X})$  to  $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ ;
  10. end while
  11.  $R_{\mathcal{Q}, \mathcal{P}}^{mod} := \mathbf{Modify}^{\text{SMS}}(\mathcal{P})$ ;
  12. return  $R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup R_{\mathcal{Q}, \mathcal{P}}^{mod} \cup \text{EDB}(\mathcal{P})$ ;
end.

```

Figure 3.8: Static Magic Sets algorithm for Disjunctive Datalog programs

relevant for answering a given query. It is important to note that this set is defined deterministically, which means that the assumptions made during the computation cannot be used to restrict the relevant part of the program. In terms of bottom-up systems, this implies that the magic atoms of SMS can provide a direct optimization only to program instantiation.

The SMS algorithm is reported in Figure 3.8 (the algorithm has been restated in order to be compared with Dynamic Magic Sets; we refer to [38] for the original description). SMS starts with a query \mathcal{Q} over a Datalog ^{\vee, \neg} program \mathcal{P} and outputs a rewritten program $\text{SMS}(\mathcal{Q}, \mathcal{P})$. As CMS, the method uses two sets, S and D , to store adorned predicates to be propagated and already processed, respectively. Rules defining magic and “collector” predicates are stored in the set $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, modified rules in $R_{\mathcal{Q}, \mathcal{P}}^{mod}$. Initially, all sets S , D , $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ and $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ are empty (line 1). The algorithm starts by processing the query (line 2), which also puts the adorned version of the query predicate into S . After that, the main loop of the algorithm is repeated until S is empty (lines 3–10). More specifically, an adorned predicate p^α is moved from S to D (line 4) and each rule r having an atom $p(\bar{t})$ in its head is processed (lines 5–8). In particular, the adorned version r^a of the rule r is computed (line 6), from which magic rules are generated (note that this process is repeated for each atom $p(\bar{t})$ in $H(r)$ matching the predicate p ; line 7). After having processed all defining rules for $p(\bar{t})$, a rule for collecting all facts coming from p^α is stored in $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ (line 9). Just before the end, the original rules of \mathcal{P} are modified by adding “collector” atoms to their bodies (line 11). Finally, the algorithm terminates returning the program obtained by the union of $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ and $\text{EDB}(\mathcal{P})$ (line 12). The details of the four auxiliary functions in SMS are discussed in the

```

Function ProcessQueryv( $\mathcal{Q}$ ,  $S$ ,  $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ )
Input
   $\mathcal{Q}$ : query;
   $S$ : set of adorned predicates;
   $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ ,  $R_{\mathcal{Q},\mathcal{P}}^{mod}$ : set of rules;
Output: none;
var
   $\alpha$ : adornment string;
begin
  1. Let  $p(\bar{t})$  be the atom in  $\mathcal{Q}$ .
  2.  $\alpha := \epsilon$ ;
  3. for each argument  $t$  in  $\bar{t}$  do
  4.   if  $t$  is a constant then  $\alpha := \alpha b$ ; else  $\alpha := \alpha f$ ; end if
  5. end for
  6. add  $p^\alpha$  to  $S$ ;
  7. add  $\text{magic}(p^\alpha(\bar{t}))$  to  $R_{\mathcal{Q},\mathcal{P}}^{mgc}$ ;
end.

```

Figure 3.9: ProcessQuery function for Static and Dynamic Magic Sets

remainder of this section. The program and SIPS from Example 3.2.3 are used as running example.

Function 1: ProcessQuery^v

The first function of SMS, **ProcessQuery**^v, is reported in Figure 3.9. Essentially, the only difference with respect to the original **ProcessQuery** is that rules for gathering answers are not generated by **ProcessQuery**^v.

Example 3.2.4. For the query \mathcal{Q}_4 from Example 3.2.3, **ProcessQuery**^v builds:

- the adorned predicate ancestorOf^{bf} ;
- the query seed $\text{magic_ancestorOf}^{bf}(a)$.

□

Function 2: Adorn^v

The second function of SMS, **Adorn**^v, is reported in Figure 3.10. For each adorned predicate p^α produced by SMS, its binding information is propagated into all rules defining the predicate p . The propagation is performed according to a given SIPS. In particular, for a binding α associated with an atom $p(\bar{t})$ in the head of a rule r , the associated SIPS

$$(\prec_r^{p^\alpha(\bar{t})}, f_r^{p^\alpha(\bar{t})})$$

determines which variables are bound in the evaluation of each atom of r : A variable X of an atom $q(\bar{s})$ in $\text{ATOMS}(r)$ is bound if and only if either

- $X \in f_r^{p^\alpha(\bar{t})}(p(\bar{t}))$, or

```

Function Adornv ( $r, p^\alpha(\bar{t}), S, D$ )
Input
   $r$ : rule;
   $p^\alpha(\bar{t})$ : adorned atom;
   $S, D$ : set of rules;
Output: an adorned rule;
var
   $r^a$ : adorned rule;
   $\alpha_i$ : adornment string;
begin
  1. Let  $(\prec_r^{p^\alpha(\bar{t})}, f_r^{p^\alpha(\bar{t})})$  be the SIPS associated with  $r$  and  $p^\alpha(\bar{t})$ .
  2.  $r^a := r$ ;
  3. for each IDB atom  $p_i(\bar{t}_i)$  in  $\text{ATOMS}(r)$  do
  4.    $\alpha_i := \epsilon$ ;
  5.   for each argument  $t$  in  $\bar{t}$  do
  6.     if  $t$  is a constant then
  7.        $\alpha_i := \alpha_i b$ ;
  8.     else
  9.       Argument  $t$  is a variable. Let  $X$  be such a variable.
 10.      if  $X \in f_r^{p^\alpha(\bar{t})}(p(\bar{t}))$  or there is  $b(\bar{v})$  in  $B^+(r)$  such that
 11.         $b(\bar{v}) \prec_r^{p^\alpha(\bar{t})} p_i(\bar{t}_i)$  and  $X \in f_r^{p^\alpha(\bar{t})}(b(\bar{v}))$  then
 12.           $\alpha_i := \alpha_i b$ ;
 13.        else
 14.           $\alpha_i := \alpha_i f$ ;
 15.        end if
 16.      end if
 17.    end for
 18.    substitute  $p_i(\bar{t}_i)$  in  $r^a$  with  $p_i^{\alpha_i}(\bar{t}_i)$ ;
 19.    if set  $D$  does not contain  $p_i^{\alpha_i}$  then add  $p_i^{\alpha_i}$  to  $S$ ; end if
 20.  end for
 21.  return  $r^a$ ;
end.

```

Figure 3.10: Adorn function for Static and Dynamic Magic Sets

- there exists $b(\bar{v}) \in B^+(r)$ such that $b(\bar{v}) \prec_r^{p^\alpha(\bar{t})} q(\bar{s})$ and $X \in f_r^{p^\alpha(\bar{t})}(b(\bar{v}))$.

Example 3.2.5. The adorned predicate `ancestorOfbf` has been added to S while processing the query atom `ancestorOf(a, Y)` in Example 3.2.4. Then, `ancestorOfbf` is moved from S to D (line 4 of Figure 3.8) and its binding information bf is propagated in all rules defining the predicate `ancestorOf` (lines 5–9 of Figure 3.8). In this case the for loop is repeated twice, for r_{16} and r_{17} , and the following adorned rules are generated:

$$r_{16}^a : \text{ancestorOf}^{bf}(X, Y) :- \text{parentOf}^{bf}(X, Y).$$

$$r_{17}^a : \text{ancestorOf}^{bf}(X, Y) :- \text{ancestorOf}^{bf}(X, Z), \text{parentOf}^{bf}(Z, Y).$$

A new adorned predicate, `parentOfbf`, is added to S in order to be processed in a subsequent iteration. When `Adornv` is invoked for `parentOfbf`, the following adorned rule is produced:

$$r_{15}^a : \text{parentOf}^{bf}(X, Y) \vee \text{nonParentOf}^{bf}(X, Y) :- \text{possibleParentOf}(X, Y).$$

```

Function GenerateSMS( $r, p^\alpha(\bar{t}), r^a$ )
Input
 $r$ : rule;
 $p^\alpha(\bar{t})$ : adorned atom;
 $r^a$ : adorned rule;
Output: a set of magic rules;
var
 $R$ : set of rules;
 $r^*$ : rule;
begin
1. Let  $(\prec_r^{p^\alpha(\bar{t})}, f_r^{p^\alpha(\bar{t})})$  be the SIPS associated with  $r$  and  $p^\alpha(\bar{t})$ .
2.  $R := \emptyset$ ;
3. for each atom  $p_i^{\alpha_i}(\bar{t}_i)$  in  $\text{ATOMS}(r^a)$  different from  $p^\alpha(\bar{t})$  do
4.   if  $\alpha_i \neq \epsilon$  do
5.      $r^* := \text{magic}(p_i^{\alpha_i}(\bar{t}_i)) :- \text{magic}(p^\alpha(\bar{t}))$ ;
6.     for each  $p_j^{\alpha_j}(\bar{t}_j)$  in  $B^+(r^a)$  such that  $p_j(\bar{t}_j) \prec_r^{p^\alpha(\bar{t})} p_i(\bar{t}_i)$  do
7.       add atom  $p_j^{\alpha_j}(\bar{t}_j)$  to  $B^+(r^*)$ ;
8.     end for
9.     add rule  $r^*$  to  $R$ ;
10.  end if
11. end for
12.  $r^* := p^\alpha(\bar{t}) :- \text{magic}(p^\alpha(\bar{t}))$ ;
13. for each atom  $p_j^{\alpha_j}(\bar{t}_j)$  in  $B^+(r^a)$  do
14.   add atom  $p_j^{\alpha_j}(\bar{t}_j)$  to  $B^+(r^*)$ ;
15. end for
16. add rule  $r^*$  to  $R$ ;
17. return  $R$ ;
end.

```

Figure 3.11: Generate function for Static Magic Sets

Another new adorned predicate, nonParentOf^{bf} , is added to S . Finally, when Adorn^ν is invoked for nonParentOf^{bf} , the rule r_{15}^a is produced again. \square

Function 3: Generate^{SMS}

The third function of SMS, **Generate**^{SMS}, is reported in Figure 3.11. Given an adorned rule r^a , obtained by adorning a rule r with respect to an adorned atom $p^\alpha(\bar{t})$, **Generate**^{SMS} produces one magic rule for each adorned atom in r^a (lines 3–11). Moreover, the function generates a rule having $p^\alpha(\bar{t})$ in its head such that the body contains $\text{magic}(p^\alpha(\bar{t}))$ and all literals in $B^+(r^a)$ (lines 12–16). Note also that the SMS algorithm enriches the set of magic rules by introducing a *collecting* rule

$$\text{collect_p}(\bar{t}) :- p^\alpha(\bar{t}).$$

for each adorned predicate p^α (line 9 of Figure 3.8).

Example 3.2.6. In Example 3.2.5 the following adorned rules have been produced: r_{16}^a , r_{17}^a and r_{15}^a (the latter is produced twice). When the function

Generate^{SMS} is invoked for r_{16}^a , the following rules are produced:

$$\begin{aligned} r_{16,1}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\ r_{16,2}^* &: \text{ancestorOf}^{bf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}^{bf}(X, Y). \end{aligned}$$

Note that the second rule above is considered a magic rule because, as we will see later, its purpose is to restrict the range of the variables during program instantiation. When **Generate**^{SMS} is invoked for r_{17}^a , instead, the following magic rules are generated:

$$\begin{aligned} r_{17,1}^* &: \text{magic_ancestorOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\ r_{17,2}^* &: \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}^{bf}(X, Z). \\ r_{17,3}^* &: \text{ancestorOf}^{bf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}^{bf}(X, Z), \\ &\quad \text{parentOf}^{bf}(Z, Y), \end{aligned}$$

Finally, **Generate**^{SMS} is invoked for r_{17}^a with $\text{parentOf}^{bf}(X, Y)$, and for r_{17}^a with $\text{nonParentOf}^{bf}(X, Y)$. The following magic rules are generated:

$$\begin{aligned} r_{15,1}^* &: \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X). \\ r_{15,2}^* &: \text{parentOf}^{bf}(X, Y) :- \text{magic_parentOf}^{bf}(X), \text{possibleParentOf}(X, Y). \\ r_{15,3}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X). \\ r_{15,4}^* &: \text{nonParentOf}^{bf}(X, Y) :- \text{magic_nonParentOf}^{bf}(X), \\ &\quad \text{possibleParentOf}(X, Y). \end{aligned}$$

Additional rules are introduced by the SMS algorithm (line 9 of Figure 3.8) for each generated adorned predicate. In our example, the following additional rules are added:

$$\begin{aligned} \text{collect_ancestorOf}(X, Y) &:- \text{ancestorOf}^{bf}(X, Y). \\ \text{collect_parentOf}(X, Y) &:- \text{parentOf}^{bf}(X, Y). \\ \text{collect_nonParentOf}(X, Y) &:- \text{nonParentOf}^{bf}(X, Y). \end{aligned}$$

□

Function 4: **Modify**^{SMS}

The last function of SMS, **Modify**^{SMS}, is reported in Figure 3.12. Given a Datalog^{∨,¬} program \mathcal{P} , the function returns a *modified program* containing a rule r' for each rule r of \mathcal{P} . The rule r' is obtained from r by adding to $B^+(r)$ an atom $\text{collect_p}(\bar{c})$ for each atom $\text{p}(\bar{c})$ in $H(r)$.

Example 3.2.7. Consider again the program \mathcal{P}_5 from Example 3.2.3. When **Modify**^{SMS} is invoked, the following modified rules are generated:

$$\begin{aligned} r'_{15} &: \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{collect_parentOf}(X, Y), \\ &\quad \text{collect_nonParentOf}(X, Y), \text{possibleParentOf}(X, Y). \\ r'_{16} &: \text{ancestorOf}(X, Y) :- \text{collect_ancestorOf}(X, Y), \text{parentOf}(X, Y). \end{aligned}$$

```

Function ModifySMS( $\mathcal{P}$ )
Input
   $\mathcal{P}$ : program;
Output: a set of modified rules;
var
   $R$ : set of rules;
   $r'$ : rule;
begin
  1.  $R := \emptyset$ ;
  2. for each rule  $r$  in  $\mathcal{P}$  do
  3.    $r' := r$ ;
  4.   for each atom  $p(\bar{t})$  in  $H(r)$  do
  5.     add collect_p $\alpha$ ( $\bar{t}$ ) to  $B^+(r')$ ;
  6.   end for
  7.   add rule  $r'$  to  $R$ ;
  8. end for
  9. return  $R$ ;
end.

```

Figure 3.12: Modify function for Static Magic Sets

```

 $r'_{17}$ : ancestorOf( $X, Y$ ) :- collect_ancestorOf( $X, Y$ ), ancestorOf( $X, Z$ ),
                             parentOf( $Z, Y$ ).

```

□

The correctness of Static Magic Sets has been established for Datalog ^{\vee} programs [38], while the applicability to Datalog ^{\vee, \neg_s} has just been hinted at in [38].

Example 3.2.8. The complete program SMS($\mathcal{Q}_4, \mathcal{P}_5$) is reported below:

```

 $r'_{15}$ : parentOf( $X, Y$ ) v nonParentOf( $X, Y$ ) :- collect_parentOf( $X, Y$ ),
                                                collect_nonParentOf( $X, Y$ ), possibleParentOf( $X, Y$ ).
 $r'_{16}$ : ancestorOf( $X, Y$ ) :- collect_ancestorOf( $X, Y$ ), parentOf( $X, Y$ ).
 $r'_{17}$ : ancestorOf( $X, Y$ ) :- collect_ancestorOf( $X, Y$ ), ancestorOf( $X, Z$ ),
                             parentOf( $Z, Y$ ).

```

```

collect_ancestorOf( $X, Y$ ) :- ancestorOfbf( $X, Y$ ).

```

```

collect_parentOf( $X, Y$ ) :- parentOfbf( $X, Y$ ).

```

```

collect_nonParentOf( $X, Y$ ) :- nonParentOfbf( $X, Y$ ).

```

```

 $r_{\mathcal{Q}_4}$ : magic_ancestorOfbf( $a$ ).

```

```

 $r^*_{16,1}$ : magic_parentOfbf( $X$ ) :- magic_ancestorOfbf( $X$ ).

```

```

 $r^*_{17,1}$ : magic_ancestorOfbf( $X$ ) :- magic_ancestorOfbf( $X$ ).

```

```

 $r^*_{17,2}$ : magic_parentOfbf( $Z$ ) :- magic_ancestorOfbf( $X$ ), ancestorOfbf( $X, Z$ ).

```

$$\begin{aligned}
r_{15,1}^* &: \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X). \\
r_{15,3}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X). \\
r_{16,2}^* &: \text{ancestorOf}^{bf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}^{bf}(X, Y). \\
r_{17,3}^* &: \text{ancestorOf}^{bf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}^{bf}(X, Z), \\
&\quad \text{parentOf}^{bf}(Z, Y), \\
r_{15,2}^* &: \text{parentOf}^{bf}(X, Y) :- \text{magic_parentOf}^{bf}(X), \text{possibleParentOf}(X, Y). \\
r_{15,4}^* &: \text{nonParentOf}^{bf}(X, Y) :- \text{magic_nonParentOf}^{bf}(X), \\
&\quad \text{possibleParentOf}(X, Y).
\end{aligned}$$

The rewritten program above is query equivalent to the original program \mathcal{P}_5 with respect to the query \mathcal{Q}_4 , but allows for a more succinct instantiation. Indeed, if the EDB contains many facts which are not related to the query, the magic rules in $\text{SMS}(\mathcal{Q}_4, \mathcal{P}_5)$ allow for discarding rules related to these atoms during program instantiation. \square

3.2.3 Dynamic Magic Sets

Dynamic Magic Sets (DMS) are our proposal to extend Classic Magic Sets to Disjunctive Datalog programs. While Static Magic Sets introduce collecting predicates for preserving the semantics of input programs, Dynamic Magic Sets achieve the same result by stripping off adornments from non-magic predicates. In this way, Dynamic Magic Sets allow for more performant instantiations. Moreover, Dynamic Magic Sets introduce the concept of conditional relevance, also referred to as dynamic relevance, which allows for further optimization potential in stable model search.

The DMS algorithm is reported in Figure 3.13. DMS starts with a query \mathcal{Q} over a Datalog ^{\vee, \neg} program \mathcal{P} and outputs a rewritten program $\text{DMS}(\mathcal{Q}, \mathcal{P})$. As CMS and SMS, the method uses two sets, S and D , to store adorned predicates to be propagated and already processed, respectively. Magic rules are stored in the set $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, modified rules in $R_{\mathcal{Q}, \mathcal{P}}^{mod}$. No “collector” predicates or rules are produced by DMS. Initially, all sets S , D , $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ and $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ are empty (line 1). The algorithm starts by processing the query (line 2), also putting the adorned version of the query predicate into S . The main loop of the algorithm is then repeated until S is empty (lines 3–10). In particular, an adorned predicate p^α is moved from S to D (line 4) and each rule r having an atom $p(\bar{c})$ in head is considered (lines 5–9). The adorned version r^a of the rule r is computed (line 6), from which magic rules are generated (line 7) and a modified rule r' is obtained (line 8). Finally, the algorithm terminates returning the program obtained by the union of $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ and $\text{EDB}(\mathcal{P})$ (line 11). The details of the four auxiliary functions in Figure 3.13 are discussed below by using the query, the program and the SIPS from Example 3.2.3 as running example.

Function 1: ProcessQuery ^{\vee}

The first function of DMS, **ProcessQuery** ^{\vee} , is the same as for SMS, reported in Figure 3.9 and discussed in Section 3.2.2. We recall here that invoking **ProcessQuery** ^{\vee} for \mathcal{Q}_4 generates:

```

Algorithm DMS( $\mathcal{Q}, \mathcal{P}$ )
Input: A query  $\mathcal{Q}$  and a Datalogv,∇ program  $\mathcal{P}$ 
Output: A rewritten Datalogv,∇ program
var
   $S, D$ : set of adorned predicates;
   $R_{\mathcal{Q}, \mathcal{P}}^{mgc}, R_{\mathcal{Q}, \mathcal{P}}^{mod}$ : set of rules;
   $r^a$ : adorned rule;
begin
  1.  $S := \emptyset; D := \emptyset; R_{\mathcal{Q}, \mathcal{P}}^{mgc} := \emptyset; R_{\mathcal{Q}, \mathcal{P}}^{mod} := \emptyset;$ 
  2. ProcessQueryv( $\mathcal{Q}, S, R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ );
  3. while  $S \neq \emptyset$  do
  4.   take an element  $\mathbf{p}^\alpha$  from  $S$ ; remove  $\mathbf{p}^\alpha$  from  $S$ ; add  $\mathbf{p}^\alpha$  to  $D$ ;
  5.   for each rule  $r$  in  $\mathcal{P}$  and for each atom  $\mathbf{p}(\bar{e})$  in  $H(r)$  do
  6.      $r^a := \mathbf{Adorn}^v(r, \alpha, S, D)$ ;
  7.      $R_{\mathcal{Q}, \mathcal{P}}^{mgc} := R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup \mathbf{Generate}^{\text{DMS}}(r, \alpha, r^a)$ ;
  8.      $R_{\mathcal{Q}, \mathcal{P}}^{mod} := R_{\mathcal{Q}, \mathcal{P}}^{mod} \cup \{\mathbf{Modify}^{\text{DMS}}(r, r^a)\}$ ;
  9.   end for
  10. end while
  11. return  $R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup R_{\mathcal{Q}, \mathcal{P}}^{mod} \cup \text{EDB}(\mathcal{P})$ ;
end.

```

Figure 3.13: Dynamic Magic Sets algorithm for Disjunctive Datalog programs

- the adorned predicate `ancestorOfbf`;
- the query seed `magic_ancestorOfbf(a)`.

Function 2: \mathbf{Adorn}^v

The second function of DMS, \mathbf{Adorn}^v , is the same as for SMS, reported in Figure 3.10 and discussed in Section 3.2.2. We report below the adorned rules produced for the query \mathcal{Q}_4 and the program \mathcal{P}_5 :

$$\begin{aligned}
 r_{16}^a &: \text{ancestorOf}^{bf}(X, Y) :- \text{parentOf}^{bf}(X, Y). \\
 r_{17}^a &: \text{ancestorOf}^{bf}(X, Y) :- \text{ancestorOf}^{bf}(X, Z), \text{parentOf}^{bf}(Z, Y). \\
 r_{15}^a &: \text{parentOf}^{bf}(X, Y) \vee \text{nonParentOf}^{bf}(X, Y) :- \text{possibleParentOf}(X, Y).
 \end{aligned}$$

We recall that the rule r_{15}^a is produced twice, for `parentOfbf` and `nonParentOfbf`.

Function 3: $\mathbf{Generate}^{\text{DMS}}$

The third function of DMS, $\mathbf{Generate}^{\text{DMS}}$, is reported in Figure 3.14. The function is similar to $\mathbf{Generate}$, used by CMS, but SIPS for Disjunctive Datalog rules are considered in this case. We observe that only magic atoms are adorned in the rewritten program produced by DMS, while adornments are stripped off standard predicates (lines 6–8). Stripping off adornments allows to avoid the introduction of “collector” predicates, which are instead required by SMS for preserving the correctness of the technique.

Example 3.2.9. In this example, we will re-use rule labels of Example 3.2.6 for denoting some generated rules. While processing \mathcal{Q}_4 and \mathcal{P}_5 , $\mathbf{Generate}^{\text{DMS}}$ is

```

Function GenerateDMS( $r, p^\alpha(\bar{t}), r^a$ )
Input
   $r$ : rule;
   $p^\alpha(\bar{t})$ : adorned atom;
   $r^a$ : adorned rule;
Output: a set of magic rules;
var
   $R$ : set of rules;
   $r^*$ : rule;
begin
  1. Let  $(\prec_r^{p^\alpha(\bar{t})}, f_r^{p^\alpha(\bar{t})})$  be the SIPS associated with  $r$  and  $p^\alpha(\bar{t})$ .
  2.  $R := \emptyset$ ;
  3. for each atom  $p_i^{\alpha_i}(\bar{t}_i)$  in  $\text{ATOMS}(r^a)$  different from  $p^\alpha(\bar{t})$  do
  4.   if  $\alpha_i \neq \epsilon$  then
  5.      $r^* := \text{magic}(p_i^{\alpha_i}(\bar{t}_i)) :- \text{magic}(p^\alpha(\bar{t}))$ ;
  6.     for each atom  $p_j(\bar{t}_j)$  in  $B^+(r)$  such that  $p_j(\bar{t}_j) \prec_r^{p^\alpha(\bar{t})} p_i(\bar{t}_i)$  do
  7.       add atom  $p_j(\bar{t}_j)$  to  $B^+(r^*)$ ;
  8.     end for
  9.     add  $r^*$  to  $R$ ;
  10.  end if
  11. end for
  12. return  $R$ ;
end.

```

Figure 3.14: Generate function for Dynamic Magic Sets

invoked four times. When **Generate**^{DMS} is invoked for rule r_{16}^a and the adorned atom $\text{ancestorOf}^{bf}(X, Y)$, the following magic rule is produced:

$$r_{16,1}^* : \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X).$$

When **Generate**^{DMS} is invoked for r_{17}^a and $\text{ancestorOf}^{bf}(X, Y)$, the following magic rules are generated:

$$r_{17,1}^* : \text{magic_ancestorOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X).$$

$$r_{17,2}^{**} : \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z).$$

When **Generate**^{DMS} is invoked for rule r_{15}^a and $\text{parentOf}^{bf}(X, Y)$, the following magic rule is produced:

$$r_{15,1}^* : \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X).$$

When **Generate**^{DMS} is invoked for rule r_{15}^a and $\text{nonParentOf}^{bf}(X, Y)$, instead, the following magic rule is generated:

$$r_{15,3}^* : \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X).$$

Note that DMS does not produce any collector rule. Moreover, note the

```

Function  $\text{Modify}^{\text{DMS}}(r, r^a)$ 
Input
   $r$ : rule;
   $r^a$ : adorned rule;
Output: a modified rule;
var
   $r'$ : rule;
begin
  1.  $r' := r$ ;
  2. for each atom  $p^\alpha(\bar{t})$  in  $H(r^a)$  do
  3.   add  $\text{magic}(p^\alpha(\bar{t}))$  to  $B^+(r')$ ;
  4. end for
  5. return  $r'$ ;
end.

```

Figure 3.15: Modify function for Dynamic Magic Sets

difference between the rule $r_{17,2}^{**}$ above and the rule $r_{17,2}^*$ produced by SMS:

$$r_{17,2}^* : \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}^{bf}(X, Z).$$

The predicate `ancestorOf` is adorned in $r_{17,2}^*$, while it is not in $r_{17,2}^{**}$. \square

Function 4: $\text{Modify}^{\text{DMS}}$

The last function of DMS, $\text{Modify}^{\text{DMS}}$, is reported in Figure 3.15. Given an adorned rule r^a , obtained from a rule r , the function builds and returns a modified rule r' . The modified rule r' is obtained from r by adding to its body a magic atom $\text{magic}(p^\alpha(\bar{t}))$ for each adorned atom $p^\alpha(\bar{t})$ occurring in $H(r^a)$. $\text{Modify}^{\text{DMS}}$ is similar to the function Modify used by CMS, but in this case more than one magic atom is possibly added to rule bodies. Moreover, while adornments of non-magic predicates are preserved by CMS, they are stripped off in DMS.

Example 3.2.10. While processing \mathcal{Q}_4 and \mathcal{P}_5 , the following modified rules are produced:

$$r_{16}'' : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y).$$

$$r_{17}'' : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z), \\ \text{parentOf}(Z, Y).$$

$$r_{15}'' : \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{magic_parentOf}^{bf}(X), \\ \text{magic_nonParentOf}^{bf}(X), \text{possibleParentOf}(X, Y).$$

To sum up, the complete program $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_5)$ comprises the following

rules:

$$\begin{aligned}
r''_{16} &: \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y). \\
r''_{17} &: \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z), \\
&\quad \text{parentOf}(Z, Y). \\
r''_{15} &: \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{magic_parentOf}^{bf}(X), \\
&\quad \text{magic_nonParentOf}^{bf}(X), \text{possibleParentOf}(X, Y).
\end{aligned}$$

$$\begin{aligned}
r_{Q_4} &: \text{magic_ancestorOf}^{bf}(a). \\
r^*_{16,1} &: \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\
r^*_{17,1} &: \text{magic_ancestorOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\
r^{**}_{17,2} &: \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z). \\
r^*_{15,1} &: \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X). \\
r^*_{15,3} &: \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X).
\end{aligned}$$

We recall below the original EDB:

$$\mathcal{F}_2 = \{\text{possibleParentOf}(a, b), \text{possibleParentOf}(b, c), \\
\text{possibleParentOf}(b, d), \text{possibleParentOf}(b, e)\},$$

During program instantiation the following atoms are derived to be true:

- $\text{magic_ancestorOf}^{bf}(a)$, because of r_{Q_4} ;
- $\text{magic_parentOf}^{bf}(a)$, because of $r^*_{16,1}$ and $\text{magic_ancestorOf}^{bf}(a)$;
- $\text{magic_nonParentOf}^{bf}(a)$, because of $r^*_{15,1}$ and $\text{magic_parentOf}^{bf}(a)$.

Thus, program instantiation produces the following rules:⁶

$$\begin{aligned}
\underline{\text{magic_parentOf}^{bf}(b)} &: \underline{\text{magic_ancestorOf}^{bf}(a)}, \underline{\text{ancestorOf}(a, b)}. \\
\underline{\text{magic_nonParentOf}^{bf}(b)} &: \underline{\text{magic_parentOf}^{bf}(b)}. \\
\underline{\text{ancestorOf}(a, b)} &: \underline{\text{magic_ancestorOf}^{bf}(a)}, \underline{\text{parentOf}(a, b)}. \\
\underline{\text{ancestorOf}(a, c)} &: \underline{\text{magic_ancestorOf}^{bf}(a)}, \underline{\text{ancestorOf}(a, b)}, \\
&\quad \underline{\text{parentOf}(b, c)}. \\
\underline{\text{ancestorOf}(a, d)} &: \underline{\text{magic_ancestorOf}^{bf}(a)}, \underline{\text{ancestorOf}(a, b)}, \\
&\quad \underline{\text{parentOf}(b, d)}. \\
\underline{\text{ancestorOf}(a, e)} &: \underline{\text{magic_ancestorOf}^{bf}(a)}, \underline{\text{ancestorOf}(a, b)}, \\
&\quad \underline{\text{parentOf}(b, e)}.
\end{aligned}$$

⁶Note that only underlined atoms are actually present in the program generated by program instantiation. Atoms already known to be true have been included to simplify the identification of rules.

$$\begin{aligned}
\underline{\text{parentOf}}(a, b) \vee \underline{\text{nonParentOf}}(a, b) &:- \underline{\text{magic_parentOf}}^{bf}(a), \\
&\quad \underline{\text{magic_nonParentOf}}^{bf}(a), \text{possibleParentOf}(a, b). \\
\underline{\text{parentOf}}(b, c) \vee \underline{\text{nonParentOf}}(b, c) &:- \underline{\text{magic_parentOf}}^{bf}(b), \\
&\quad \underline{\text{magic_nonParentOf}}^{bf}(b), \text{possibleParentOf}(b, c). \\
\underline{\text{parentOf}}(b, d) \vee \underline{\text{nonParentOf}}(b, c) &:- \underline{\text{magic_parentOf}}^{bf}(b), \\
&\quad \underline{\text{magic_nonParentOf}}^{bf}(b), \text{possibleParentOf}(b, d). \\
\underline{\text{parentOf}}(b, e) \vee \underline{\text{nonParentOf}}(b, c) &:- \underline{\text{magic_parentOf}}^{bf}(b), \\
&\quad \underline{\text{magic_nonParentOf}}^{bf}(b), \text{possibleParentOf}(b, e).
\end{aligned}$$

This program is then processed by the stable model search phase. We note that there is just one rule with a true body, from which either `parentOf(a, b)` or `nonParentOf(a, b)` may be inferred. Note also that the other disjunctive rules are not active because of `magic_parentOfbf(b)` and `magic_nonParentOfbf(b)`. We point out that in this case magic sets provide a dynamic optimization to stable model search by deactivating rules which are not relevant in the considered partial model. Moreover, it can be checked that \mathcal{P}_5 and $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_5)$ are query equivalent with respect to \mathcal{Q}_4 . \square

Below we give another example of application of Dynamic Magic Sets. This time a Datalog^{V, \neg s} program is considered.

Example 3.2.11. Suppose we are interested in retrieving all people which *are not* ancestors of a person **a** in a given genealogy graph like the one shown in Figure 3.7. Assuming the presence of an EDB predicate `person`, the scenario above can be modeled by means of the query \mathcal{Q}_5 and the program \mathcal{P}_6 reported below:

$$\begin{aligned}
\mathcal{Q}_5 &: \text{nonAncestorOf}(a, Y)? \\
r_{15} &: \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{possibleParentOf}(X, Y). \\
r_{16} &: \text{ancestorOf}(X, Y) :- \text{parentOf}(X, Y). \\
r_{17} &: \text{ancestorOf}(X, Y) :- \text{ancestorOf}(X, Z), \text{parentOf}(Z, Y). \\
r_{20} &: \text{nonAncestorOf}(X, Y) :- \text{person}(X), \text{person}(Y), \text{not ancestorOf}(X, Y).
\end{aligned}$$

Note that \mathcal{P}_6 has been obtained by adding the rule r_{20} to the program \mathcal{P}_5 . We now describe the application of DMS to \mathcal{Q}_5 and \mathcal{P}_6 . While processing the query, the following magic seed is produced:

$$r_{\mathcal{Q}_5} : \underline{\text{magic_nonAncestorOf}}^{bf}(a).$$

Thus, the adorned predicate `nonAncestorOfbf` has to be propagated into r_{20} , assuming the SIPS

$$(\prec_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}, f_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)})$$

defined below:

- `nonAncestorOf(X, Y)` $\prec_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}$ `person(X)`;
- `nonAncestorOf(X, Y)` $\prec_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}$ `person(Y)`;

- $\text{nonAncestorOf}(X, Y) \prec_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)} \text{ancestorOf}(X, Y)$;
- $f_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}(\text{nonAncestorOf}(X, Y)) = \{X\}$;
- $f_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}(\text{person}(X)) = \{X\}$;
- $f_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}(\text{person}(Y)) = \{Y\}$;
- $f_{r_{20}}^{\text{nonAncestorOf}^{bf}(X, Y)}(\text{ancestorOf}(X, Y)) = \{X\}$.

The following modified and magic rules are produced according to this SIPS:

- $$r'_{20} : \text{nonAncestorOf}(X, Y) :- \text{magic_nonAncestorOf}^{bf}(X), \text{person}(X), \\ \text{person}(Y), \text{not ancestorOf}(X, Y).$$
- $$r^*_{20} : \text{magic_ancestorOf}^{bf}(X) :- \text{magic_nonAncestorOf}^{bf}(X).$$

A new adorned predicate is then produced, namely ancestorOf^{bf} , at which point the algorithm continues in the same way as in Examples 3.2.4–3.2.10. If the SIPS described in Example 3.2.3 are adopted for all other rules, the rewritten program $\text{DMS}(\mathcal{Q}_5, \mathcal{P}_6)$ comprises the following rules:

- $$r'_{20} : \text{nonAncestorOf}(X, Y) :- \text{magic_nonAncestorOf}^{bf}(X), \text{person}(X), \\ \text{person}(Y), \text{not ancestorOf}(X, Y).$$
- $$r''_{16} : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y).$$
- $$r''_{17} : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z), \\ \text{parentOf}(Z, Y).$$
- $$r''_{15} : \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{magic_parentOf}^{bf}(X), \\ \text{magic_nonParentOf}^{bf}(X), \text{possibleParentOf}(X, Y).$$
- $$r_{\mathcal{Q}_5} : \text{magic_nonAncestorOf}^{bf}(a).$$
- $$r^*_{20} : \text{magic_ancestorOf}^{bf}(X) :- \text{magic_nonAncestorOf}^{bf}(X).$$
- $$r^*_{16,1} : \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X).$$
- $$r^*_{17,1} : \text{magic_ancestorOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X).$$
- $$r^{**}_{17,2} : \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z).$$
- $$r^*_{15,1} : \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X).$$
- $$r^*_{15,3} : \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X).$$

It can be checked that the program above is equivalent to the original program with respect to the query \mathcal{Q}_5 . \square

3.2.4 Query Equivalence Theorem

In this section correctness of Dynamic Magic Sets for Disjunctive Datalog programs with stratified negation is formally established. In particular, the following theorem is proved.

Theorem 3.2.12 (Query Equivalence Theorem). *Let \mathcal{Q} be a query over a $\text{Datalog}^{\vee, \neg_s}$ program \mathcal{P} . Then, the following equivalences are established:*

- $\mathcal{P} \equiv_{\mathcal{Q}}^b \text{DMS}(\mathcal{Q}, \mathcal{P})$; and
- $\mathcal{P} \equiv_{\mathcal{Q}}^c \text{DMS}(\mathcal{Q}, \mathcal{P})$.

The proof of the theorem above covers the remainder of this section. In particular, we first highlight syntactic relationships between original and rewritten programs, and show an interesting relationship between magic sets and unfounded sets. After that, we prove the soundness and the completeness of stable model correspondence for Dynamic Magic Sets for $\text{Datalog}^{\vee, \neg_s}$ programs by using the results obtained in the first part of this section. Based on this we can then show Theorem 3.2.12.

Preliminaries

We start by providing a syntactic relationship between transformed and original programs.

Proposition 3.2.13. *Consider a query \mathcal{Q} over a $\text{Datalog}^{\vee, \neg}$ program \mathcal{P} and a rule $r' \in \text{DMS}(\mathcal{Q}, \mathcal{P})$ of the form*

$$r' : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \cdots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{magic}(\mathbf{h}_1^{\alpha_1}(\bar{\mathbf{u}}_1)), \dots, \mathbf{magic}(\mathbf{h}_m^{\alpha_m}(\bar{\mathbf{u}}_m)), \\ \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

For each substitution ϑ such that $r'\vartheta \in \text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$, there is a rule $r\vartheta \in \text{Ground}(\mathcal{P})$ such that

$$r : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \cdots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \\ \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

Proof. The rule r' has been produced by a rule r^a obtained by adorning r . Hence, it is enough to note that the rules r and r' are defined over the same set of variables. \square

The proposition above allows for passing from a ground rule of the transformed program to an associated ground rule of the original program. We next provide a complementary result, connecting ground rules of the original program to ground rules of the rewritten program.

Proposition 3.2.14. *Consider a query \mathcal{Q} over a $\text{Datalog}^{\vee, \neg}$ program \mathcal{P} and a rule $r \in \mathcal{P}$ of the form*

$$r : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \cdots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \\ \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

For each substitution ϑ such that $r\vartheta \in \text{Ground}(\mathcal{P})$, and for each adornment string α_i such that $\mathbf{magic}(\mathbf{h}_1^{\alpha_i}(\bar{\mathbf{u}}_1))\vartheta \in \mathcal{B}_{\text{DMS}(\mathcal{Q}, \mathcal{P})}$, there are adornment strings $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_n$ such that $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$ contains the following rules:

1. A modified rule $r'\vartheta$, where:

$$r' : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \dots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{magic}(\mathbf{h}_1^{\alpha_i}(\bar{\mathbf{u}}_1)), \dots, \mathbf{magic}(\mathbf{h}_m^{\alpha_m}(\bar{\mathbf{u}}_m)), \\ \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

2. A magic rule $r_j^*\vartheta$, for each $j \in \{1, \dots, m\}$ different from i , where:

- $H(r_j^*) = \{\mathbf{magic}(\mathbf{h}_j^{\alpha_j}(\bar{\mathbf{u}}_j))\}$, and
- $B(r_j^*) = \{\mathbf{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\} \cup \{\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r) \mid \mathbf{b}(\bar{\mathbf{v}}) \prec_r^{\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i)} \mathbf{h}_j(\bar{\mathbf{u}}_j)\}$.

3. A magic rule $r_j^*\vartheta$, for each $j \in \{1, \dots, n\}$ such that \mathbf{b}_j is an IDB predicate, where:

- $H(r_j^*) = \{\mathbf{magic}(\mathbf{b}_j^{\beta_j}(\bar{\mathbf{v}}_j))\}$, and
- $B(r_j^*) = \{\mathbf{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\} \cup \{\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r) \mid \mathbf{b}(\bar{\mathbf{v}}) \prec_r^{\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i)} \mathbf{b}_j(\bar{\mathbf{v}}_j)\}$.

Proof. Since $\mathbf{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\vartheta$ is a ground atom in $\mathcal{B}_{\text{DMS}(\mathcal{Q}, \mathcal{P})}$, the adorned predicate $\mathbf{h}_i^{\alpha_i}$ has been added to S at some point of the DMS algorithm, and it has eventually been used to adorn r , thereby producing an adorned rule r^a . Let $\alpha_1, \dots, \alpha_m, \beta_1, \dots, \beta_n$ be the adornment strings in r^a . Hence, invoking **Modify**^{DMS} for r and r^a generates r' , which then belongs to $\text{DMS}(\mathcal{Q}, \mathcal{P})$. Since r and r' are defined over the same set of variables, we can conclude that $r'\vartheta \in \text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$. Moreover, the adorned rule r^a has been used by **Generate**^{DMS} for generating magic rules according to the adopted SIPS. The structure of these magic rules is as stated in 2. and 3. and their variables are a subset of the variables of r . We can thus conclude that the claim holds. \square

In order to prove the Query Equivalence Theorem we will use the well established notion of unfounded set for Datalog ^{\vee, \neg} programs introduced in [48]. Before introducing unfounded sets, however, we have to define partial interpretations, that is, interpretations for which some atoms may be undefined.

Definition 3.2.15 (Partial Interpretation). Let \mathcal{P} be a Datalog ^{\vee, \neg} program. A partial interpretation for \mathcal{P} is a pair $\langle T, U \rangle$ such that $T \subseteq U \subseteq \mathcal{B}_{\mathcal{P}}$. The atoms in T are interpreted as true, while the atoms in $U \setminus T$ are undefined. All other atoms are false.

We can then formalize the notion of unfounded set.

Definition 3.2.16 (Unfounded Set). Let $\langle T, U \rangle$ be a partial interpretation for a Datalog ^{\vee, \neg} program \mathcal{P} , and $X \subseteq \mathcal{B}_{\mathcal{P}}$ be a set of atoms. Then, X is an *unfounded set* for \mathcal{P} with respect to $\langle T, U \rangle$ if and only if, for each ground rule $r \in \text{Ground}(\mathcal{P})$ with $X \cap H(r) \neq \emptyset$, at least one of the following conditions holds:

- the body of r is false with respect to $\langle T, U \rangle$, i.e., either

$$B^+(r) \not\subseteq U, \text{ or} \tag{3.1}$$

$$B^-(r) \cap T \neq \emptyset; \tag{3.2}$$

- some positive body literal of r occurs in X , i.e.,

$$B^+(r) \cap X \neq \emptyset; \quad (3.3)$$

- some head atom of r not occurring in X is true with respect to I , i.e.,

$$H(r) \cap (T \setminus X) \neq \emptyset. \quad (3.4)$$

Intuitively, (3.1), (3.2) and (3.4) check if the rule is satisfied by $\langle T, U \rangle$ regardless to the atoms in X , while condition (3.3) ensures that the rule can be satisfied by forcing all atoms in X to be false. The following is an adaptation of the theorem in [48] to our notation.

Theorem 3.2.17 ([48]). *Let $\langle T, U \rangle$ be a partial interpretation for a Datalog ^{\vee, \neg} program \mathcal{P} . Then, for any stable model M of \mathcal{P} such that $T \subseteq M \subseteq U$, and for each unfounded set X of \mathcal{P} with respect to $\langle T, U \rangle$, $M \cap X = \emptyset$ holds.*

We now highlight an interesting property of unfounded sets, providing a link with magic sets which has not been previously considered in the literature. The link between unfounded sets and magic sets is given by the following set of “killed” atoms.

Definition 3.2.18 (Killed Atoms). Let \mathcal{Q} be a query over a Datalog ^{\vee, \neg} program \mathcal{P} , M' a model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$, and $N' \subseteq M'$ a model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$. The set

$$\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$$

of the *killed atoms* for $\text{DMS}(\mathcal{Q}, \mathcal{P})$ with respect to M' and N' is defined as follows:

$$\{\mathbf{p}(\bar{\mathbf{t}}) \in \mathcal{B}_{\mathcal{P}} \setminus N' \mid \text{either } \mathbf{p} \text{ is an EDB predicate, or} \\ \text{there is a binding } \alpha \text{ such that } \text{magic}(\mathbf{p}^\alpha(\bar{\mathbf{t}})) \in N'\}.$$

Intuitively, killed atoms are either false ground instances of some EDB predicate, or false atoms which are relevant with respect to \mathcal{Q} (they have associated magic atoms in the model N'). In terms of a hypothetical top-down evaluation of \mathcal{Q} this means that killed atoms would be considered as subqueries but discovered to be false. Note that, if M' is stable model, the set $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ is well-defined. Indeed, a stable model is both a model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$ and a minimal model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ trivially satisfying $M' \subseteq M'$.

Example 3.2.19. Consider again the program $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_5)$ from Example 3.2.10:

$$r''_{16} : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y).$$

$$r''_{17} : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z), \\ \text{parentOf}(Z, Y).$$

$$r''_{15} : \text{parentOf}(X, Y) \vee \text{nonParentOf}(X, Y) :- \text{magic_parentOf}^{bf}(X), \\ \text{magic_nonParentOf}^{bf}(X), \text{possibleParentOf}(X, Y).$$

$$\begin{aligned}
r_{\mathcal{Q}_4} &: \text{magic_ancestorOf}^{bf}(a). \\
r_{16,1}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\
r_{17,1}^* &: \text{magic_ancestorOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\
r_{17,2}^* &: \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z). \\
r_{15,1}^* &: \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X). \\
r_{15,3}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X).
\end{aligned}$$

The original EDB \mathcal{F}_2 is reported below:

$$\mathcal{F}_2 = \{\text{possibleParentOf}(a, b), \text{possibleParentOf}(b, c), \\
\text{possibleParentOf}(b, d), \text{possibleParentOf}(b, e)\}.$$

Hence, the following is a stable model of $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_5)$:

$$M'_1 = \mathcal{F}_2 \cup \{\text{magic_ancestorOf}^{bf}(a), \text{magic_parentOf}^{bf}(a), \\
\text{magic_nonParentOf}^{bf}(a), \text{nonParentOf}(a, b)\}.$$

Therefore, the set $\text{killed}_{\mathcal{Q}_4, \mathcal{P}_5}^{M'_1}(M'_1)$ contains the following ground atoms:

- $\text{ancestorOf}(a, Y)$, for each $Y \in \{a, b, c, d, e\}$;
- $\text{parentOf}(a, Y)$, for each $Y \in \{a, b, c, d, e\}$;
- $\text{nonParentOf}(a, Y)$, for each $Y \in \{a, c, d, e\}$;
- $\text{possibleParentOf}(X, Y)$, for each $X, Y \in \{a, b, c, d, e\}$ such that the atom $\text{possibleParentOf}(X, Y)$ does not occur in \mathcal{F}_2 .

□

We observe that the set of killed atoms in the example above is an unfounded set for \mathcal{P}_5 with respect to $\langle M'_1 \cap \mathcal{B}_{\mathcal{P}_5}, \mathcal{B}_{\mathcal{P}_5} \rangle$. This is not by chance, as formalized in the next theorem.

Theorem 3.2.20. *Consider a query \mathcal{Q} over a Datalog ^{\vee, \neg} program \mathcal{P} . Let M' be a model for $\text{DMS}(\mathcal{Q}, \mathcal{P})$, and $N' \subseteq M'$ a model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$. Then, $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$ is an unfounded set for \mathcal{P} with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$.*

Proof. Let X be the set $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$. According to Definition 3.2.16, for each rule $r\vartheta \in \text{Ground}(\mathcal{P})$ (ϑ a substitution) such that $H(r)\vartheta \cap X \neq \emptyset$, we have to show that at least one of the following conditions holds:

- (3.1) $B^+(r)\vartheta \not\subseteq \mathcal{B}_{\mathcal{P}}$,⁷ or
- (3.2) $B^-(r)\vartheta \cap (M' \cap \mathcal{B}_{\mathcal{P}}) \neq \emptyset$, or
- (3.3) $B^+(r)\vartheta \cap X \neq \emptyset$, or
- (3.4) $H(r)\vartheta \cap ((M' \cap \mathcal{B}_{\mathcal{P}}) \setminus X) \neq \emptyset$.

⁷Note that this condition cannot hold.

Let $r\vartheta$ be such that

$$r : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \cdots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \\ \text{not } \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \text{not } \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

and let $\mathbf{h}_i(\bar{\mathbf{u}}_i)\vartheta$ be an atom in $H(r)\vartheta \cap X$. By Definition 3.2.18 of $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$, $\mathbf{h}_i(\bar{\mathbf{u}}_i)\vartheta \in X$ implies that there is an adornment string α_i such that the magic atom $\text{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\vartheta$ belongs to N' . Thus, $\text{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\vartheta \in \mathcal{B}_{\text{DMS}(\mathcal{Q}, \mathcal{P})}$, and by Proposition 3.2.14 there is a ground rule $r'\vartheta \in \text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$ such that

$$r' : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \cdots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \text{magic}(\mathbf{h}_1^{\alpha_1}(\bar{\mathbf{u}}_1)), \dots, \text{magic}(\mathbf{h}_m^{\alpha_m}(\bar{\mathbf{u}}_m)), \\ \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \text{not } \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \text{not } \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

Since M' is a model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$, when considering the rule $r'\vartheta$ three cases may occur.

Case 1: $B^-(r')\vartheta \cap M' \neq \emptyset$, that is, the negative body of $r'\vartheta$ is false with respect to M' . In this case, since $B^-(r) = B^-(r')$ and $B^-(r)\vartheta \subseteq \mathcal{B}_{\mathcal{P}}$, from $B^-(r')\vartheta \cap M' \neq \emptyset$ we immediately conclude

$$B^-(r)\vartheta \cap (M' \cap \mathcal{B}_{\mathcal{P}}) \neq \emptyset,$$

that is, (3.2) holds.

Case 2: $B^+(r')\vartheta \not\subseteq M'$, that is, the positive body of $r'\vartheta$ is false with respect to M' . In this case we shall show that (3.3) holds, i.e., that $B^+(r)\vartheta \cap X \neq \emptyset$. Since $N' \subseteq M'$, $B^+(r')\vartheta \not\subseteq M'$ implies $B^+(r')\vartheta \not\subseteq N'$. Therefore, there is an atom $\mathbf{b}_j(\bar{\mathbf{v}}_j) \in B^+(r)$ such that $\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta \notin N'$ and, for any $\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r)$, $\mathbf{b}(\bar{\mathbf{v}}) \prec_r^{\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i)} \mathbf{b}_j(\bar{\mathbf{v}}_j)$ implies $\mathbf{b}(\bar{\mathbf{v}})\vartheta \in N'$. If \mathbf{b}_j is an EDB predicate, the atom $\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta$ belongs to X by the definition of killed atoms. Otherwise, \mathbf{b}_j is an IDB predicate, and so from Proposition 3.2.14 (item 3) there is a magic rule $r_j^*\vartheta$ in $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$ such that:

- $H(r_j^*) = \{\text{magic}(\mathbf{b}_j^{\beta_j}(\bar{\mathbf{v}}_j))\};$
- $B(r_j^*) = \{\text{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\} \cup \{\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r) \mid \mathbf{b}(\bar{\mathbf{v}}) \prec_r^{\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i)} \mathbf{b}_j(\bar{\mathbf{v}}_j)\}.$

Thus, $B^+(r_j^*)\vartheta \subseteq N'$ holds because $\text{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\vartheta$ belongs to N' and by the properties of $\mathbf{b}_j(\bar{\mathbf{v}}_j)$. Therefore, since N' is a model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$, $\text{magic}(\mathbf{b}_j^{\beta_j}(\bar{\mathbf{v}}_j))$ belongs to N' , from which $\mathbf{b}_j(\bar{\mathbf{v}}_j) \in X$ follows from the definition of killed atoms. Thus, independently of the type (EDB, IDB) of \mathbf{b}_j , (3.3) holds.

Case 3: $H(r')\vartheta \cap M' \neq \emptyset$, that is, the head of $r'\vartheta$ is true with respect to M' . In this case we can assume that $B^-(r')\vartheta \cap M' = \emptyset$ (otherwise, Case 1 can be applied) and show that either

$$(3.3) \quad B^+(r)\vartheta \cap X \neq \emptyset, \text{ or}$$

$$(3.4) \quad H(r)\vartheta \cap ((M' \cap \mathcal{B}_{\mathcal{P}}) \setminus X) \neq \emptyset$$

hold. From the assumption $B^-(r')\vartheta \cap M' = \emptyset$ we can conclude that there is a rule in $Ground(DMS(\mathcal{Q}, \mathcal{P}))^{M'}$ obtained from $r'\vartheta$ by removing its negative body literals. Consider now the literals $\mathbf{b}_1(\bar{\mathbf{v}}_1)\vartheta, \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k)\vartheta$ in $B^+(r)\vartheta$ and assume that there is a $j \in \{1, \dots, k\}$ such that:

- atom $\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta$ does not belong to N' ; and
- for each atom $\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r)$ such that

$$\mathbf{b}(\bar{\mathbf{v}}) \prec_r^{\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i)} \mathbf{b}_j(\bar{\mathbf{v}}_j),$$

atom $\mathbf{b}(\bar{\mathbf{v}})\vartheta$ belongs to N' .

In this case we can show that $\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta$ is a killed atom, from which condition (3.3) immediately follows. We have to consider two cases. If \mathbf{b}_j is an EDB predicate, $\mathbf{b}_j(\bar{\mathbf{v}}_j)$ belongs to $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$ because it is a false instance of an EDB atom (see Definition 3.2.18). Otherwise, if \mathbf{b}_j is an IDB predicate, we consider the rule r_j^* from Proposition 3.2.14 (item 3). In particular, note that

$$B^+(r_j^*)\vartheta \subseteq \{\mathbf{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i)), \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k)\}\vartheta \subseteq N';$$

hence, $\mathbf{magic}(\mathbf{b}_j^{\beta_j}(\bar{\mathbf{v}}_j))\vartheta \in H(r_j^*)$ belongs to N' because N' is a model of the program $Ground(DMS(\mathcal{Q}, \mathcal{P}))^{M'}$ by assumption. We can then conclude that, also in this case, $\mathbf{b}_j(\bar{\mathbf{v}}_j)$ belongs to $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$ by definition.

It remains to consider the case in which such a j does not exist. Then we know that $\mathbf{b}_j(\bar{\mathbf{v}}_j) \in N'$ holds for all $j \in \{1, \dots, k\}$. Moreover, recall that N' also contains $\mathbf{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))$. Hence, from the magic rules in Proposition 3.2.14 (item 2), we can conclude that $\mathbf{magic}(\mathbf{h}_j^{\alpha_j}(\bar{\mathbf{u}}_j)) \in N'$ holds for all $j \in \{1, \dots, n\}$ (we recall that N' is a model of $Ground(DMS(\mathcal{Q}, \mathcal{P}))^{M'}$ by assumption). Hence, we have $B^+(r')\vartheta \subseteq N'$, which in turn implies that

$$H(r')\vartheta \cap N' \neq \emptyset. \quad (3.5)$$

Note that $H(r')\vartheta = H(r)\vartheta$ and $H(r)\vartheta \subseteq \mathcal{B}_{\mathcal{P}}$; thus, from (3.5) we conclude

$$H(r)\vartheta \cap (N' \cap \mathcal{B}_{\mathcal{P}}) \neq \emptyset. \quad (3.6)$$

Now observe that $X = \mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$ is a subset of $\mathcal{B}_{\mathcal{P}} \setminus N'$ by Definition 3.2.18, so $N' \cap \mathcal{B}_{\mathcal{P}} = (N' \cap \mathcal{B}_{\mathcal{P}}) \setminus X$ and thus (3.6) implies

$$H(r)\vartheta \cap ((N' \cap \mathcal{B}_{\mathcal{P}}) \setminus X) \neq \emptyset. \quad (3.7)$$

Finally, remember that $N' \subseteq M'$ by assumption, therefore from (3.7) we conclude

$$H(r)\vartheta \cap ((M' \cap \mathcal{B}_{\mathcal{P}}) \setminus X) \neq \emptyset,$$

i.e., condition (3.4) holds in this case. \square

Soundness of Dynamic Magic Sets

Before proving the soundness of stable model correspondence for Dynamic Magic Sets for Datalog^{V, \neg *} programs, we show below a lemma which holds for Datalog^{V, \neg} programs in general.

Lemma 3.2.21. *Let \mathcal{Q} be a query over a Datalog^{v,¬} program \mathcal{P} , and M' a stable model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$. Moreover, let $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ be the program obtained by adding to \mathcal{P} a fact for each atom in $M' \cap \mathcal{B}_{\mathcal{P}}$. Then, each stable model M of $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ is in turn a stable model of \mathcal{P} containing $M' \cap \mathcal{B}_{\mathcal{P}}$.*

Proof. Let M be a stable model of $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$. Clearly enough, M is in turn a model of \mathcal{P} such that $M \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$. We shall show that M is in fact a stable model of \mathcal{P} . Assume, for the sake of contradiction, that M is not a stable model of \mathcal{P} , and let $N \subset M$ be a model of $\text{Ground}(\mathcal{P})^M$.

Consider the following interpretation:

$$N' = (N \cap (M' \cap \mathcal{B}_{\mathcal{P}})) \cup (M' \setminus \mathcal{B}_{\mathcal{P}}). \quad (3.8)$$

We can show $N' \subset M'$ in two steps.

1. First, $N' \subseteq M'$ can be proved by comparing (3.8) and

$$M' = (M' \cap \mathcal{B}_{\mathcal{P}}) \cup (M' \setminus \mathcal{B}_{\mathcal{P}}).$$

2. Then, we prove that $N' \neq M'$ must hold. Indeed, if $N' = M'$, $N \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$ holds because N' is the union of $N \cap (M' \cap \mathcal{B}_{\mathcal{P}})$ and $M' \setminus \mathcal{B}_{\mathcal{P}}$, which are disjoint. Since N is a model of $\text{Ground}(\mathcal{P})^M$ by assumption, $N \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$ in turn implies that N is a model of

$$\text{Ground}(\mathcal{P})^M \cup (M' \cap \mathcal{B}_{\mathcal{P}}) = \text{Ground}(\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}}))^M.$$

This is in contradiction with the assumption that M is a stable model of $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ because $N \subset M$ by assumption. Hence, $N' \subset M'$ holds.

Our aim is now to show that N' is a model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$, which would give rise to a contraction with the original assumption that M' is a stable model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$. Hence, we have to consider two types of rules in $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$, those obtained from modified rules and those obtained from magic rules, and show that they are satisfied by N' .

Rules of type 1 (modified rules): Consider first a rule obtained by removing negative literals from a ground modified rule $r'\vartheta \in \text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$ such that $B^+(r'\vartheta) \subseteq N'$. Our aim is then to show that

$$H(r'\vartheta) \cap N' \neq \emptyset. \quad (3.9)$$

Let r' be of the form

$$r' : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \dots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{magic}(\mathbf{h}_1^{\alpha_1}(\bar{\mathbf{u}}_1)), \dots, \mathbf{magic}(\mathbf{h}_m^{\alpha_m}(\bar{\mathbf{u}}_m)), \\ \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

By applying Proposition 3.2.13, we can conclude the existence of a rule $r\vartheta$ in $\text{Ground}(\mathcal{P})$ such that

$$r : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \dots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \\ \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

We can show that $B^-(r)\vartheta \cap M = \emptyset$, so that a rule obtained from $r\vartheta$ by removing its negative body literals belongs to $Ground(\mathcal{P})^M$. Note that $B^-(r')\vartheta \cap M' = \emptyset$ holds by definition of the reduct, and $B^-(r) = B^-(r')$. Hence, it is enough to show that $B^-(r)\vartheta \subseteq \mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$ and that $\mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$ is an unfounded set for $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$. Indeed, in this case $B^-(r)\vartheta \cap M = \emptyset$ would be a consequence of Theorem 3.2.17. Let us start by showing that $B^-(r)\vartheta \subseteq \mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$ holds. To this aim, consider the magic rules from Proposition 3.2.14 (item 3). In particular, for each $j \in \{k+1, \dots, n\}$ there is a magic rule $r_j^*\vartheta$ belonging to $Ground(DMS(\mathcal{Q}, \mathcal{P}))^{M'}$ (recall that magic rules have empty negative bodies) and such that:

- $H(r_j^*)$ consists of atom $\mathbf{magic}(b_j^{\beta_j}(\bar{v}_j))$, for some adornment β_j ; and
- $B(r_j^*)$ is a subset of $B^+(r')$.

Recall that N' is a model of $Ground(DMS(\mathcal{Q}, \mathcal{P}))^{M'}$ by assumption. Moreover, we assumed $B^+(r') \subseteq N'$. Hence, from these magic rules we can conclude that $\mathbf{magic}(b_j^{\beta_j}(\bar{v}_j))\vartheta \in N'$ for all $j \in \{k+1, \dots, n\}$. Thus, $B^-(r)\vartheta \subseteq \mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$ holds by Definition 3.2.18 of $\mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$.

We now prove that $\mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$ is an unfounded set for $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$. By Theorem 3.2.20, we already know that $\mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$ is an unfounded set for \mathcal{P} with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$. Hence, it is enough to note that the rules added to \mathcal{P} are facts corresponding to the atoms in $M' \cap \mathcal{B}_{\mathcal{P}}$, and

$$(M' \cap \mathcal{B}_{\mathcal{P}}) \cap \mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M') = \emptyset$$

holds by Definition 3.2.18 of $\mathbf{killed}_{\mathcal{Q},\mathcal{P}}^{M'}(M')$.

We are then ready to prove (3.9). To this aim consider $H(r')\vartheta \cap N'$. Since $H(r') = H(r)$, and by construction of N' ,

$$H(r')\vartheta \cap N' = H(r)\vartheta \cap (N \cap (M' \cap \mathcal{B}_{\mathcal{P}}))$$

holds. Hence, for proving (3.9) we just have to show that (3.10) and (3.11) below hold:

$$H(r)\vartheta \cap N \neq \emptyset; \tag{3.10}$$

$$H(r)\vartheta \cap N \subseteq H(r)\vartheta \cap (M' \cap \mathcal{B}_{\mathcal{P}}). \tag{3.11}$$

For proving (3.10) it is enough to observe that the rule obtained by removing the negative literals from $r\vartheta$ belongs to $Ground(\mathcal{P})^M$, and N is a model of $Ground(\mathcal{P})^M$. For proving (3.11), instead, note that

$$H(r)\vartheta \cap N \subseteq H(r)\vartheta \cap M$$

holds because $N \subset M$. Hence, we have just to show that

$$H(r)\vartheta \cap M = H(r)\vartheta \cap (M' \cap \mathcal{B}_{\mathcal{P}}). \tag{3.12}$$

To this aim, we consider the magic rules from Proposition 3.2.14 (item 2). From these rules, and by the same considerations already done for the magic rules of

item 3, there is an adornment string α_j such that $\text{magic}(\mathbf{h}_j^{\alpha_j}(\bar{u}_j)) \in M'$, for each $j \in \{1, \dots, m\}$. Hence, $H(r)\vartheta \setminus M'$ is a subset of $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$, which is disjoint from M . Therefore,

$$M \cap (H(r)\vartheta \setminus M') = \emptyset$$

holds, which combined with $M \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$ gives rise to (3.12), and thus (3.9) holds.

Rules of type 2 (magic rules): Consider now a ground magic rule $r^*\vartheta \in \text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ such that $B^+(r^*)\vartheta \subseteq N'$, and let $\text{magic}(\mathbf{p}^\alpha(\bar{\tau}))$ be the (only) atom in $H(r^*)\vartheta$. Since $N' \subset M'$ holds by assumption, $B^+(r^*)\vartheta \subseteq N'$ implies that $B^+(r^*)\vartheta \subset M'$, which in turn implies $\text{magic}(\mathbf{p}^\alpha(\bar{\tau}))\vartheta \in M'$ because M' is a model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$. Moreover, since $\mathcal{B}_{\mathcal{P}}$ does not contain any magic atom, $\text{magic}(\mathbf{p}^\alpha(\bar{\tau}))\vartheta$ is also contained in $M' \setminus \mathcal{B}_{\mathcal{P}}$. Thus, from (3.8) we can conclude

$$H(r^*)\vartheta \cap N' \neq \emptyset,$$

that is, the rule $r^*\vartheta$ is satisfied by N' . \square

For a query \mathcal{Q} over a $\text{Datalog}^{\vee, \neg}$ program \mathcal{P} , Lemma 3.2.21 above provides a link from each stable model M' of $\text{DMS}(\mathcal{Q}, \mathcal{P})$ to all stable models of \mathcal{P} containing $M' \cap \mathcal{B}_{\mathcal{P}}$, if any. Hence, in order to prove the correctness of stable model correspondence for Dynamic Magic Sets for $\text{Datalog}^{\vee, \neg_s}$ programs we just have to note that \mathcal{P} accepts at least one stable model of this kind. The next theorem formalizes this intuition.

Theorem 3.2.22 (Soundness). *Let \mathcal{Q} be a query over a $\text{Datalog}^{\vee, \neg_s}$ program \mathcal{P} . Then, for each stable model M' of $\text{DMS}(\mathcal{Q}, \mathcal{P})$ and for each substitution ϑ , there is a stable model M of \mathcal{P} such that $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof. Since \mathcal{P} is a $\text{Datalog}^{\vee, \neg_s}$ program, $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ is also a $\text{Datalog}^{\vee, \neg_s}$ program (only facts are added to \mathcal{P}). Hence, the program $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ admits at least one stable model M . Therefore, by Lemma 3.2.21 such an M is a stable model of \mathcal{P} such that $M \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$. Thus, we trivially have that

$$\mathcal{Q}\vartheta \in M' \implies \mathcal{Q}\vartheta \in M$$

holds. The other direction is proved by considering the contrapositive, that is,

$$\mathcal{Q}\vartheta \notin M' \implies \mathcal{Q}\vartheta \notin M. \quad (3.13)$$

To this aim consider the set $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ and note that the magic seed is associated with each instance of \mathcal{Q} in M' . Hence, all instances of \mathcal{Q} which are false with respect to M' belong to $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$. Since $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ is an unfounded set for \mathcal{P} with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$ by Theorem 3.2.20, and M is a stable model of \mathcal{P} such that $M' \cap \mathcal{B}_{\mathcal{P}} \subseteq M \subseteq \mathcal{B}_{\mathcal{P}}$, from Theorem 3.2.17 we conclude (3.13). \square

Completeness of Dynamic Magic Sets

For proving the completeness of stable model correspondence for Dynamic Magic Sets we construct an interpretation for $\text{DMS}(\mathcal{Q}, \mathcal{P})$ based on one for \mathcal{P} . The

new interpretation is referred to as “magic variant” and is defined below.

Definition 3.2.23 (Magic Variant). Consider a query \mathcal{Q} over a Datalog^{V,∇} program \mathcal{P} . Let I be an interpretation for \mathcal{P} . We define an interpretation $\text{variant}_{\mathcal{Q},\mathcal{P}}^{\infty}(I)$ for $\text{DMS}(\mathcal{Q},\mathcal{P})$, called the magic variant of I with respect to \mathcal{Q} and \mathcal{P} , as the fixpoint of the following sequence:

$$\begin{aligned} \text{variant}_{\mathcal{Q},\mathcal{P}}^0(I) &= \text{EDB}(\mathcal{P}) \\ \text{variant}_{\mathcal{Q},\mathcal{P}}^{i+1}(I) &= \text{variant}_{\mathcal{Q},\mathcal{P}}^i(I) \cup \\ &\quad \{p(\bar{t}) \in I \mid \exists \alpha \text{ such that } \text{magic}(p^\alpha(\bar{t})) \in \text{variant}_{\mathcal{Q},\mathcal{P}}^i(I)\} \cup \\ &\quad \{\text{magic}(p^\alpha(\bar{t})) \mid \exists r^* \in \text{Ground}(\text{DMS}(\mathcal{Q},\mathcal{P})) \text{ such that} \\ &\quad \quad \text{magic}(p^\alpha(\bar{t})) \in H(r^*) \text{ and } B^+(r^*) \subseteq \text{variant}_{\mathcal{Q},\mathcal{P}}^i(I)\}, \quad \forall i \geq 0. \end{aligned}$$

Below is an example of magic variant.

Example 3.2.24. Consider again \mathcal{Q}_4 and \mathcal{P}_5 from Example 3.2.3, and the rewritten program $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_5)$ from Example 3.2.10. Moreover, consider the following stable model of \mathcal{P}_5 :

$$\begin{aligned} M_1 = \mathcal{F}_2 \cup \{ &\text{nonParentOf}(a, b), \text{parentOf}(b, c), \\ &\text{parentOf}(b, d), \text{parentOf}(b, e)\}. \end{aligned}$$

While computing the magic variant of M_1 , the following sequence of interpretations is built:

$$\begin{aligned} \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^0(M_1) &= \mathcal{F}_2 \\ \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^1(M_1) &= \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^0(M_1) \cup \{\text{magic_ancestorOf}^{bf}(a)\} \\ \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^2(M_1) &= \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^1(M_1) \cup \{\text{magic_parentOf}^{bf}(a)\} \\ \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^3(M_1) &= \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^2(M_1) \cup \{\text{magic_nonParentOf}^{bf}(a)\} \\ \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^4(M_1) &= \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^3(M_1) \cup \{\text{nonParentOf}(a, b)\} \\ \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^5(M_1) &= \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^4(M_1) = \text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^{\infty}(M_1) \end{aligned}$$

Note that $\text{variant}_{\mathcal{Q}_4, \mathcal{P}_5}^{\infty}(M_1)$ is a stable model $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_5)$. In particular, it coincides with the stable model M'_1 presented in Example 3.2.19. \square

In the example above, we started from a stable model of the original program \mathcal{P}_5 and computed its magic variant. The resulting interpretation was a stable model of the rewritten program. Below, we prove that this is true in general.

Lemma 3.2.25. *Let \mathcal{Q} be a query over a Datalog^{V,∇} program \mathcal{P} . For each stable model M of \mathcal{P} , there is a stable model M' of $\text{DMS}(\mathcal{Q}, \mathcal{P})$ (which is the magic variant of M) such that $M \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$.*

Proof. Let

$$M' = \text{variant}_{\mathcal{Q},\mathcal{P}}^{\infty}(M) \tag{3.14}$$

be the magic variant of the stable model M . Thus,

$$M \supseteq M' \cap \mathcal{B}_{\mathcal{P}} \tag{3.15}$$

holds by definition. Hence, it remains to prove that M' is a stable model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$. We start by showing that M' is a model for $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$. To this end, consider a rule in $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ having a true body, that is, a rule obtained by removing the negative body literals from a rule

$$r'\vartheta \in \text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P})) \quad (3.16)$$

such that the following relationships hold:

$$B^-(r'\vartheta) \cap M' = \emptyset; \quad (3.17)$$

$$B^+(r'\vartheta) \subseteq M'. \quad (3.18)$$

In particular, our aim is to show that

$$H(r'\vartheta) \cap M' \neq \emptyset. \quad (3.19)$$

We distinguish two cases.

Case 1: If r' is a magic rule, we prove that (3.19) holds by showing that the (only) atom in $H(r'\vartheta)$ belongs to M' . Indeed, this is the case because of Definition 3.2.23, (3.14) and (3.17).

Case 2: Otherwise, r' is a modified rule of the form

$$r' : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \dots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{magic}(\mathbf{h}_1^{\alpha_1}(\bar{\mathbf{u}}_1)), \dots, \mathbf{magic}(\mathbf{h}_m^{\alpha_m}(\bar{\mathbf{u}}_m)), \\ \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

For all $i \in \{1, \dots, m\}$ the atom $\mathbf{magic}(\mathbf{h}_i^{\alpha_i}(\bar{\mathbf{u}}_i))\vartheta$ belongs to M' because of (3.18). Hence, for proving (3.19) it is enough to show that

$$H(r'\vartheta) \cap M \neq \emptyset \quad (3.20)$$

because

$$H(r'\vartheta) \cap M' = H(r'\vartheta) \cap M$$

holds by Definition 3.2.23 of $\text{variant}_{\mathcal{Q}, \mathcal{P}}^{\infty}(M)$.

To prove (3.20) we apply Proposition 3.2.13 and conclude that there is a rule $r\vartheta \in \text{Ground}(\mathcal{P})$ such that

$$r : \mathbf{h}_1(\bar{\mathbf{u}}_1) \vee \dots \vee \mathbf{h}_m(\bar{\mathbf{u}}_m) :- \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k), \\ \mathbf{not} \mathbf{b}_{k+1}(\bar{\mathbf{v}}_{k+1}), \dots, \mathbf{not} \mathbf{b}_n(\bar{\mathbf{v}}_n).$$

In particular, note that the following equalities hold:

$$H(r) = H(r'); \quad (3.21)$$

$$B^-(r) = B^-(r'); \quad (3.22)$$

$$B^+(r)\vartheta = B^+(r')\vartheta \cap \mathcal{B}_{\mathcal{P}}. \quad (3.23)$$

From (3.21) we conclude that (3.20) is equivalent to

$$H(r)\vartheta \cap M \neq \emptyset. \quad (3.24)$$

From (3.15), (3.18) and (3.23) we conclude

$$B^+(r)\vartheta = B^+(r')\vartheta \cap \mathcal{B}_{\mathcal{P}} \subseteq M' \cap \mathcal{B}_{\mathcal{P}} \subseteq M. \quad (3.25)$$

Hence, since M is a model of \mathcal{P} , and the positive body of $r\vartheta$ is true with respect to M because of (3.25) above, for proving (3.24) it is enough to show that also the negative body is true with respect to M , i.e.,

$$B^-(r)\vartheta \cap M = \emptyset. \quad (3.26)$$

Assume, for the sake of contradiction, that this is not the case, and let $j \in \{k+1, \dots, n\}$ be such that

$$\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta \in M. \quad (3.27)$$

If \mathbf{b}_j is an EDB predicate, a contradiction with (3.17) arises. Indeed, by Definition 3.2.23, and because of (3.14), $\text{EDB}(\mathcal{P})$ is a subset of M' , which in turn implies that $\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta$ belongs to M' . Then, assume that \mathbf{b}_j is an IDB predicate. In this case we can apply Proposition 3.2.14 for each $\mathbf{h}_i(\bar{\mathbf{u}}_i)\vartheta \in H(r)\vartheta$. In particular, we consider the rule $r_j^*\vartheta$ from Proposition 3.2.14 (item 3) and note that

$$B(r_j^*)\vartheta \subseteq \{\text{magic}(\mathbf{h}_i^\alpha(\bar{\mathbf{u}}_i)), \mathbf{b}_1(\bar{\mathbf{v}}_1), \dots, \mathbf{b}_k(\bar{\mathbf{v}}_k)\}\vartheta \subseteq B^+(r')\vartheta. \quad (3.28)$$

Hence, by combining (3.14), (3.18) and (3.28) with Definition 3.2.23, we obtain

$$\text{magic}(\mathbf{b}_j^{\beta_j}(\bar{\mathbf{v}}_j))\vartheta \in M' \quad (3.29)$$

(β_j an adornment string). Finally, by combining (3.14), (3.27) and (3.29) with Definition 3.2.23, we conclude $\mathbf{b}_j(\bar{\mathbf{v}}_j)\vartheta \in M'$, which contradicts (3.17), and we have shown Case 2 for (3.19).

It remains to show that M' is also a minimal model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$. Let N' be a model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ such that $N' \subseteq M'$. We shall show that in this case also $M' \subseteq N'$ holds, from which the minimality of M' follows. We proceed by induction on the structure of $M' = \text{variant}_{\mathcal{Q}, \mathcal{P}}^\infty(M)$. The base case, i.e.,

$$\text{variant}_{\mathcal{Q}, \mathcal{P}}^0(M) \subseteq N',$$

is clearly true, since $\text{variant}_{\mathcal{Q}, \mathcal{P}}^0(M)$ contains only EDB facts which are also in $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ and N' is a model. Now suppose

$$\text{variant}_{\mathcal{Q}, \mathcal{P}}^i(M) \subseteq N'$$

in order to prove that

$$\text{variant}_{\mathcal{Q}, \mathcal{P}}^{i+1}(M) \subseteq N'$$

holds as well. Considering an atom ℓ in

$$\text{variant}_{\mathcal{Q}, \mathcal{P}}^{i+1}(M) \setminus \text{variant}_{\mathcal{Q}, \mathcal{P}}^i(M),$$

we distinguish two cases.

Case 1: Let $\ell = \mathbf{magic}(p^\alpha(\bar{t}))$ be a magic atom. In this case, by Definition 3.2.23, there must be a rule $r^* \in \mathit{Ground}(\mathit{DMS}(\mathcal{Q}, \mathcal{P}))$ having $\mathbf{magic}(p^\alpha(\bar{t}))$ in its head and such that $B^+(r^*) \subseteq \mathbf{variant}_{\mathcal{Q}, \mathcal{P}}^i(M)$. We can then apply the induction hypothesis and conclude that $B^+(r^*) \subseteq N'$ holds. Hence, since N' is a model of $\mathit{Ground}(\mathit{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$, the magic atom $\mathbf{magic}(p^\alpha(\bar{t}))$ belongs to N' (we recall that $r^* \in \mathit{Ground}(\mathit{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ because magic rules have empty negative bodies).

Case 2: Let $\ell = p(\bar{t})$ be a standard atom. By Definition 3.2.23, $p(\bar{t})$ belongs to M and there is an adornment string α such that $\mathbf{magic}(p^\alpha(\bar{t})) \in \mathbf{variant}_{\mathcal{Q}, \mathcal{P}}^i(M)$. Hence, $\mathbf{magic}(p^\alpha(\bar{t})) \in N'$ by the induction hypothesis. Since M' is a model of $\mathit{DMS}(\mathcal{Q}, \mathcal{P})$, and N' is a model of $\mathit{Ground}(\mathit{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$ such that $N' \subseteq M'$, we can compute the set $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$ (see Definition 3.2.18). Moreover, $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$ is an unfounded set for \mathcal{P} with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$ because of Theorem 3.2.20. Hence, since $M' \cap \mathcal{B}_{\mathcal{P}} \subseteq M \subseteq \mathcal{B}_{\mathcal{P}}$ holds because of (3.15), and M is a stable model of \mathcal{P} by assumption, by applying Theorem 3.2.17 we conclude that

$$M \cap \mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N') = \emptyset.$$

Finally, since $p(\bar{t}) \in M' \cap \mathcal{B}_{\mathcal{P}}$ and $M' \cap \mathcal{B}_{\mathcal{P}} \subseteq M$, the equality above implies that $p(\bar{t}) \notin \mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$, which in turn implies $p(\bar{t}) \in N'$. \square

We can now prove the completeness of stable model correspondence for Dynamic Magic Sets.

Theorem 3.2.26 (Completeness). *Let \mathcal{Q} be a query over a $\mathit{Datalog}^{\vee, \neg}$ program \mathcal{P} . Then, for each stable model M of \mathcal{P} and for each substitution ϑ , there is a stable model M' of $\mathit{DMS}(\mathcal{Q}, \mathcal{P})$ (which is the magic variant of M) such that $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof. Let M be a stable model of \mathcal{P} and $M' = \mathbf{variant}_{\mathcal{Q}, \mathcal{P}}^\infty(M)$ its magic variant. Because of Lemma 3.2.25, M' is a stable model of $\mathit{DMS}(\mathcal{Q}, \mathcal{P})$ such that $M \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$. Thus, we trivially have that

$$\mathcal{Q}\vartheta \in M' \implies \mathcal{Q}\vartheta \in M$$

holds. The other direction is proved by considering the contrapositive, that is,

$$\mathcal{Q}\vartheta \notin M' \implies \mathcal{Q}\vartheta \notin M. \quad (3.30)$$

To this aim, consider the set $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ and note that the magic seed is associated with each instance of \mathcal{Q} is in M' . Hence, all instances of \mathcal{Q} which are false with respect to M' belong to $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$. Since $\mathbf{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ is an unfounded set for \mathcal{P} with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$ by Theorem 3.2.20, and M is a stable model of \mathcal{P} such that $M' \cap \mathcal{B}_{\mathcal{P}} \subseteq M \subseteq \mathcal{B}_{\mathcal{P}}$, from Theorem 3.2.17 we conclude (3.30). \square

Note that the theorem above does not fully extend to the completeness of query answering for Dynamic Magic Sets for $\mathit{Datalog}^{\vee, \neg}$ programs. In fact, it only considers substitution answers achieved by means of some stable models.

This is sufficient for proving the completeness of brave reasoning in general, but it is not enough for cautious reasoning. Indeed, when cautious reasoning is performed on an incoherent program, the answer comprise all possible substitutions. These substitutions are not necessarily contained in the answer obtained for the rewritten program, which may be coherent. However, all $\text{Datalog}^{\vee, \neg_s}$ programs are coherent and we can then prove the correctness of query answering for Dynamic Magic Sets for this class.

Proof of Query Equivalence Theorem (3.2.12). Consider any set of facts \mathcal{F} defined over the EDB predicates of \mathcal{P} (we recall that \mathcal{P} and $\text{DMS}(\mathcal{Q}, \mathcal{P})$ have the same set of EDB predicates). Our aim is then to show the following equalities:

1. $\text{Ans}_b(\mathcal{Q}, \mathcal{P} \cup \mathcal{F}) = \text{Ans}_b(\mathcal{Q}, \text{DMS}(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F})$;
2. $\text{Ans}_c(\mathcal{Q}, \mathcal{P} \cup \mathcal{F}) = \text{Ans}_c(\mathcal{Q}, \text{DMS}(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F})$.

Hence, since \mathcal{P} is coherent, we are interested in proving that:

1. For all substitutions ϑ , atom $\mathcal{Q}\vartheta$ belongs to some stable model M of $\mathcal{P} \cup \mathcal{F}$ if and only if $\mathcal{Q}\vartheta$ belongs to some stable model M' of $\text{DMS}(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F}$;
2. For all substitutions ϑ , atom $\mathcal{Q}\vartheta$ belongs to all stable models M of $\mathcal{P} \cup \mathcal{F}$ if and only if $\mathcal{Q}\vartheta$ belongs to all stable models M' of $\text{DMS}(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F}$.

Note that the DMS algorithm does not depend on EDB facts; thus,

$$\text{DMS}(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F} = \text{DMS}(\mathcal{Q}, \mathcal{P} \cup \mathcal{F})$$

holds. Therefore, the two statements above are equivalent to:

1. For all substitutions ϑ , atom $\mathcal{Q}\vartheta$ belongs to some stable model M of $\mathcal{P} \cup \mathcal{F}$ if and only if $\mathcal{Q}\vartheta$ belongs to some stable model M' of $\text{DMS}(\mathcal{Q}, \mathcal{P} \cup \mathcal{F})$;
2. For all substitutions ϑ , atom $\mathcal{Q}\vartheta$ belongs to all stable models M of $\mathcal{P} \cup \mathcal{F}$ if and only if $\mathcal{Q}\vartheta$ belongs to all stable models M' of $\text{DMS}(\mathcal{Q}, \mathcal{P} \cup \mathcal{F})$.

We can then conclude the proof by observing that the last two statements above are direct consequences of Theorem 3.2.22 and Theorem 3.2.26. \square

3.3 Dynamic Magic Sets and Super-Coherent Disjunctive Datalog Programs

The correctness of Dynamic Magic Sets can be extended to the class of Super-Coherent Disjunctive Datalog programs, denoted by $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ and defined in the next section. In particular, we note that many definitions and claims in Section 3.2.4 have been introduced and proved for $\text{Datalog}^{\vee, \neg}$ programs in general. In fact, Theorem 3.2.22 is the only claim which relies on the properties of stratified programs. Thus, in the following we just have to extend this theorem to the class of $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ programs.

3.3.1 Super-Coherent Disjunctive Datalog Programs

The class of Super-Coherent Disjunctive Datalog programs is defined below.

Definition 3.3.1 (Datalog $_{SC}^{\vee, \neg}$ Programs). A Datalog $^{\vee, \neg}$ program \mathcal{P} is *super-coherent* if the program $\mathcal{P} \cup \mathcal{F}$ is coherent for every set of facts \mathcal{F} , i.e., if for every \mathcal{F} it holds that

$$SM(\mathcal{P} \cup \mathcal{F}) \neq \emptyset.$$

Let Datalog $_{SC}^{\vee, \neg}$ denote the set of all super-coherent programs.

Note that deciding whether a program \mathcal{P} belongs to Datalog $_{SC}^{\vee, \neg}$ is computable. If \mathcal{P} is not Datalog $_{SC}^{\vee, \neg}$, there is a set of facts \mathcal{F} such that $\mathcal{P} \cup \mathcal{F}$ is not coherent. However, we can restrict the sets of facts to be checked to a finite number. Assuming that different rules have different variable names, and that no constant ξ_X belongs to $\mathcal{U}_{\mathcal{P}}$, we can restrict \mathcal{F} to be among all possible sets of ground atoms constructible by combining predicates of \mathcal{P} with constants in

$$\mathcal{U}_{\mathcal{P}} \cup \{\xi_X \mid X \text{ is a variable of } \mathcal{P}\}.$$

In fact, if the incoherence is not due (only) to atoms in $\mathcal{B}_{\mathcal{P}}$, but if new constant symbols are required, the exact names of these symbols are irrelevant and the possibility of instantiating each variable with a different constant is sufficient to trigger the incoherence.

Datalog $_{SC}^{\vee, \neg}$ programs constitute an interesting class of programs. All stratified and all odd-cycle-free programs belong to Datalog $_{SC}^{\vee, \neg}$. Indeed, every stratified and every odd-cycle-free program is coherent, and coherence is maintained even if an arbitrary set of facts is added to its rules (no cycle of dependencies can be introduced in this way). On the other hand, there are Datalog $_{SC}^{\vee, \neg}$ programs which contain odd cycles. An example is shown below.

Example 3.3.2. Consider the following program:

```
a v b.
a :- not a, not b.
```

The program above is Datalog $_{SC}^{\vee, \neg}$ but not odd-cycle-free. Indeed, an odd cycle involving a is present, but the first rule ensures that the body of the second rule is false in all models. Thus, the odd cycle cannot be activated. \square

3.3.2 Running Example

In this section we show the applicability of Dynamic Magic Sets to Datalog $_{SC}^{\vee, \neg}$ programs by means of a running example similar to the one used in Section 3.2 for introducing SIPS and Magic Set techniques for Disjunctive Datalog programs. Consider the scenario from Example 3.2.3: We are interested in computing all descendants of a given person a in a database containing information about “possible” parent-child relationships. Program \mathcal{P}_5 in Example 3.2.3 guesses a subset of these possible relationships to be actually in the relation $\text{parentOf}(X, Y)$. The guess is obtained by means of the following disjunctive rule:

```
r15 : parentOf(X, Y) v nonParentOf(X, Y) :- possibleParentOf(X, Y).
```

Here, we consider an equivalent program \mathcal{P}_7 obtained from \mathcal{P}_5 by substituting r_{15} with the following rules:

$$\begin{aligned} r_{20} &: \text{parentOf}(X, Y) \text{ :- possibleParentOf}(X, Y), \text{ not nonParentOf}(X, Y). \\ r_{21} &: \text{nonParentOf}(X, Y) \text{ :- possibleParentOf}(X, Y), \text{ not parentOf}(X, Y). \end{aligned}$$

Hence, our running example consists of \mathcal{Q}_4 and the program \mathcal{P}_7 below:

$$\begin{aligned} \mathcal{Q}_4 &: \text{ancestorOf}(a, Y)? \\ r_{20} &: \text{parentOf}(X, Y) \text{ :- possibleParentOf}(X, Y), \text{ not nonParentOf}(X, Y). \\ r_{21} &: \text{nonParentOf}(X, Y) \text{ :- possibleParentOf}(X, Y), \text{ not parentOf}(X, Y). \\ r_{16} &: \text{ancestorOf}(X, Y) \text{ :- parentOf}(X, Y). \\ r_{17} &: \text{ancestorOf}(X, Y) \text{ :- ancestorOf}(X, Z), \text{ parentOf}(Z, Y). \end{aligned}$$

We consider the EDB \mathcal{F}_2 from Example 3.2.3:

$$\mathcal{F}_2 = \{\text{possibleParentOf}(a, b), \text{possibleParentOf}(b, c), \\ \text{possibleParentOf}(b, d), \text{possibleParentOf}(b, e)\}.$$

Therefore, we have the following sets of answers:

- $Ans_b(\mathcal{Q}_4, \mathcal{P}_7) = \{b, c, d, e\}$ (brave answers);
- $Ans_c(\mathcal{Q}_4, \mathcal{P}_7) = \emptyset$ (cautious answers).

A hypothetical top-down evaluation of \mathcal{Q}_4 over \mathcal{P}_7 would produce the same adornments discussed in Example 3.2.3. More specifically, the adopted SIPS are reported below.

- $(\prec_{r_{16}}^{\text{ancestorOf}^{bf}(X, Y)}, f_{r_{16}}^{\text{ancestorOf}^{bf}(X, Y)})$ is such that:
 - $\text{ancestorOf}(X, Y) \prec_{r_{16}}^{\text{ancestorOf}^{bf}(X, Y)} \text{parentOf}(X, Y)$;
 - $f_{r_{16}}^{\text{ancestorOf}^{bf}(X, Y)}(\text{ancestorOf}(X, Y)) = \{X\}$;
 - $f_{r_{16}}^{\text{ancestorOf}^{bf}(X, Y)}(\text{parentOf}(X, Y)) = \{X, Y\}$.
- $(\prec_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)}, f_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)})$ is such that:
 - $\text{ancestorOf}(X, Y) \prec_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)} \text{ancestorOf}(X, Z)$;
 - $\text{ancestorOf}(X, Y) \prec_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)} \text{parentOf}(Z, Y)$;
 - $\text{ancestorOf}(X, Z) \prec_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)} \text{parentOf}(Z, Y)$;
 - $f_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)}(\text{ancestorOf}(X, Y)) = \{X\}$;
 - $f_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)}(\text{ancestorOf}(X, Z)) = \{X, Z\}$;
 - $f_{r_{17}}^{\text{ancestorOf}^{bf}(X, Y)}(\text{parentOf}(Z, Y)) = \{Z, Y\}$.
- $(\prec_{r_{20}}^{\text{parentOf}^{bf}(X, Y)}, f_{r_{20}}^{\text{parentOf}^{bf}(X, Y)})$ is such that:
 - $\text{parentOf}(X, Y) \prec_{r_{20}}^{\text{parentOf}^{bf}(X, Y)} \text{possibleParentOf}(X, Y)$;

- $\text{parentOf}(X, Y) \prec_{r_{20}}^{\text{parentOf}^{bf}(X, Y)} \text{nonParentOf}(X, Y)$;
- $f_{r_{20}}^{\text{parentOf}^{bf}(X, Y)}(\text{parentOf}(X, Y)) = \{X\}$;
- $f_{r_{20}}^{\text{parentOf}^{bf}(X, Y)}(\text{nonParentOf}(X, Y)) = \{X\}$;
- $f_{r_{20}}^{\text{parentOf}^{bf}(X, Y)}(\text{possibleParentOf}(X, Y)) = \{X, Y\}$.
- $(\prec_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)}, f_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)})$ is such that:
 - $\text{nonParentOf}(X, Y) \prec_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)} \text{possibleParentOf}(X, Y)$;
 - $\text{nonParentOf}(X, Y) \prec_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)} \text{parentOf}(X, Y)$;
 - $f_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)}(\text{nonParentOf}(X, Y)) = \{X\}$;
 - $f_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)}(\text{parentOf}(X, Y)) = \{X\}$;
 - $f_{r_{21}}^{\text{nonParentOf}^{bf}(X, Y)}(\text{possibleParentOf}(X, Y)) = \{X, Y\}$.

The DMS algorithm starts by initializing S , D , $R_{\mathcal{Q}_4, \mathcal{P}_7}^{mgc}$ and $R_{\mathcal{Q}_4, \mathcal{P}_7}^{mod}$ to the empty set (see line 1 of Figure 3.13). Then, **ProcessQuery**^v is invoked for \mathcal{Q}_4 : The adorned atom ancestorOf^{bf} is stored into the set S and the query seed $\text{magic_ancestorOf}^{bf}(a)$ is added to $R_{\mathcal{Q}_4, \mathcal{P}_7}^{mgc}$ (reported in Figure 3.9). After that, the main loop of the algorithm is executed (lines 3–10 of Figure 3.13). In particular, ancestorOf^{bf} is moved from S to D and its binding information is propagated into the rules r_{16} and r_{17} . The following adorned rules are produced by **Adorn**^v (reported in Figure 3.10):

$$r_{16}^a : \text{ancestorOf}^{bf}(X, Y) :- \text{parentOf}^{bf}(X, Y).$$

$$r_{17}^a : \text{ancestorOf}^{bf}(X, Y) :- \text{ancestorOf}^{bf}(X, Z), \text{parentOf}^{bf}(Z, Y).$$

The new adorned predicate parentOf^{bf} is stored into S , and the adorned rules above are processed by **Generate**^{DMS} (reported in Figure 3.14) and **Modify**^{DMS} (reported in Figure 3.15), which produce the following rules:

$$r_{16,1}^* : \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X).$$

$$r_{17,1}^* : \text{magic_ancestorOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X).$$

$$r_{17,2}^{**} : \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z).$$

$$r_{16}'' : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y).$$

$$r_{17}'' : \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z), \\ \text{parentOf}(Z, Y).$$

The main loop is then repeated for parentOf^{bf} , which is moved from S to D . The rule r_{20} is considered, from which the following adorned rule is produced:

$$r_{20}^a : \text{parentOf}^{bf}(X, Y) :- \text{possibleParentOf}(X, Y), \text{not nonParentOf}^{bf}(X, Y).$$

Note that Y in nonParentOf remains free because possibleParentOf does not precede it in the considered SIPS. A new adorned predicate, nonParentOf^{bf} ,

is produced and stored into S . After that, from r_{20}^a the following rules are generated:

$$\begin{aligned} r_{20}^* &: \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X). \\ r_{20}' &: \text{parentOf}(X) :- \text{magic_parentOf}^{bf}(X), \text{possibleParentOf}(X, Y), \\ &\quad \text{not nonParentOf}(X, Y). \end{aligned}$$

The main loop is then repeated for nonParentOf^{bf} , which is moved from S to D . From r_{21} the following adorned rule is obtained:

$$r_{21}^a : \text{nonParentOf}^{bf}(X, Y) :- \text{possibleParentOf}(X, Y), \text{not parentOf}^{bf}(X, Y).$$

No new adorned predicate is added to S in this case. After that, from rule r_{21}^a the following rules are produced:

$$\begin{aligned} r_{21}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X). \\ r_{21}' &: \text{nonParentOf}(X, Y) :- \text{magic_nonParentOf}^{bf}(X), \\ &\quad \text{possibleParentOf}(X, Y), \text{not parentOf}^{bf}(X, Y). \end{aligned}$$

To sum up, the rewritten program $\text{DMS}(\mathcal{Q}_4, \mathcal{P}_7)$ consists of the following rules:

$$\begin{aligned} r_{\mathcal{Q}_4} &: \text{magic_ancestorOf}^{bf}(a). \\ r_{16,1}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_ancestorOf}^{bf}(X). \\ r_{17,2}^{**} &: \text{magic_parentOf}^{bf}(Z) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z). \\ r_{20}^* &: \text{magic_nonParentOf}^{bf}(X) :- \text{magic_parentOf}^{bf}(X). \\ r_{21}^* &: \text{magic_parentOf}^{bf}(X) :- \text{magic_nonParentOf}^{bf}(X). \\ \\ r_{20}' &: \text{parentOf}(X) :- \text{magic_parentOf}^{bf}(X), \text{possibleParentOf}(X, Y), \\ &\quad \text{not nonParentOf}(X, Y). \\ r_{21}' &: \text{nonParentOf}(X, Y) :- \text{magic_nonParentOf}^{bf}(X), \\ &\quad \text{possibleParentOf}(X, Y), \text{not parentOf}^{bf}(X, Y). \\ r_{16}'' &: \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{parentOf}(X, Y). \\ r_{17}'' &: \text{ancestorOf}(X, Y) :- \text{magic_ancestorOf}^{bf}(X), \text{ancestorOf}(X, Z), \\ &\quad \text{parentOf}(Z, Y). \end{aligned}$$

3.3.3 Query Equivalence Theorem

For proving the correctness of stable model correspondence for Dynamic Magic Sets for the class of $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ programs we first extend the soundness result. Hence, the following is an extension of Theorem 3.2.22 to $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ programs.

Theorem 3.3.3 (Soundness for $\text{Datalog}_{\text{SC}}^{\vee, \neg}$). *Let \mathcal{Q} be a query over a $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ program \mathcal{P} . Then, for each stable model M' of $\text{DMS}(\mathcal{Q}, \mathcal{P})$ and for each substitution ϑ , there is a stable model M of \mathcal{P} such that $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof. Since \mathcal{P} is a $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ program, $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ is coherent (only facts are added to \mathcal{P}). Hence, program $\mathcal{P} \cup (M' \cap \mathcal{B}_{\mathcal{P}})$ admits at least one stable model M . Therefore, by Lemma 3.2.21, such an M is a stable model of \mathcal{P} such that $M \supseteq M' \cap \mathcal{B}_{\mathcal{P}}$. Thus, we trivially have that

$$Q\vartheta \in M' \implies Q\vartheta \in M$$

holds. The other direction is proved by considering the contrapositive, that is,

$$Q\vartheta \notin M' \implies Q\vartheta \notin M. \quad (3.31)$$

To this aim consider the set $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ and note that the magic seed is associated with each instance of \mathcal{Q} is in M' . Hence, all instances of \mathcal{Q} which are false with respect to M' belongs to $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$. Since $\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(M')$ is an unfounded set for \mathcal{P} with respect to $\langle M' \cap \mathcal{B}_{\mathcal{P}}, \mathcal{B}_{\mathcal{P}} \rangle$ by Theorem 3.2.20, and M is a stable model of \mathcal{P} such that $M' \cap \mathcal{B}_{\mathcal{P}} \subseteq M \subseteq \mathcal{B}_{\mathcal{P}}$, from Theorem 3.2.17 we conclude (3.31). \square

In proving the soundness of stable model correspondence for Dynamic Magic Sets for super-coherent programs we used a key property of the class: Coherence of a $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ program is maintained even if an arbitrary set of facts is added to the program. In general, $\text{Datalog}^{\vee, \neg}$ programs do not have this property. In fact, we point out that the application of Dynamic Magic Sets to $\text{Datalog}^{\vee, \neg}$ programs which are not in $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ may result in unsound answers. Below is an example.

Example 3.3.4. Consider the following query and program:

```

q(a)?
edb(a).
q(X) v p(X) :- edb(X).
co(X) :- q(X), not co(X).

```

First of all, note that the program is not super-coherent: An incoherent program can be obtained by adding the fact $q(a)$. The rewritten program generated by DMS consists of the following rules:

```

magic_q^b(a).
edb(a).
q^b(X) v p^b(X) :- magic_q^b(X), magic_p^b(X), edb(X).
magic_p^b(X) :- magic_q^b(X).
magic_q^b(X) :- magic_p^b(X).

```

The rewritten program above has two stable models, namely

$$\{\text{edb}(a), \text{magic_q}^b(a), \text{magic_p}^b(a), p(a)\}$$

and

$$\{\text{edb}(a), \text{magic_q}^b(a), \text{magic_p}^b(a), q(a)\}.$$

Therefore, $q(a)$ is a brave consequence of the rewritten program. However, $q(a)$

is not a brave consequence of the original program, which admits only one stable model, namely $\{\mathbf{edb}(\mathbf{a}), \mathbf{p}(\mathbf{a})\}$. \square

We are then prove the correctness of query answering for Dynamic Magic Sets for the class of Super-Coherent Disjunctive Datalog programs.

Theorem 3.3.5 (Query Equivalence Theorem for $\text{Datalog}_{\text{SC}}^{\vee, \neg}$). *Let \mathcal{Q} be a query over a $\text{Datalog}_{\text{SC}}^{\vee, \neg}$ program \mathcal{P} . Then, the following equivalences hold:*

- $\mathcal{P} \equiv_{\mathcal{Q}}^b \text{DMS}(\mathcal{Q}, \mathcal{P})$; and
- $\mathcal{P} \equiv_{\mathcal{Q}}^c \text{DMS}(\mathcal{Q}, \mathcal{P})$.

Proof. Since \mathcal{P} is coherent, the claim is a consequence of Theorem 3.3.3 and Theorem 3.2.26. \square

Chapter 4

Application: Decidability for Datalog with Functions

Datalog and its extensions have been defined in Chapter 2 as rule-based languages allowing the use of variables and constants in atom arguments. In this chapter the definition of terms is enlarged for allowing the use of function symbols. When function symbols are permitted, the expressive power of the language increases considerably, up to the first level of the analytical hierarchy if disjunction or recursive negation are allowed [18]. However, this high expressive power implies that the common reasoning tasks became undecidable when function symbols are present. The possibility to identify large classes of programs with functions for which the reasoning tasks are still decidable has been intensively studied in the past decade by many authors; see for instance [20, 15, 50, 62, 28]. Among the identified classes, *finitely ground programs* are the largest class of Disjunctive Datalog programs with functions for which stable models are still computable by standard bottom-up strategies; however, it is not decidable whether a given program is finitely ground.

In this chapter we relate finitely ground programs to another class of programs with function symbols, namely the class of *finitely recursive queries and programs*. We prove the decidability of reasoning over finitely recursive queries and programs by providing a link with the class of finitely ground programs. In particular, we show that the application of Dynamic Magic Sets to a finitely recursive query generates a finitely ground program. We point out that finitely recursive queries constitute a natural formalization of queries over Disjunctive Datalog programs with function symbols which can be answered by means of a finite top-down evaluation. Note that our result does not imply a containment relationship between the two classes, indeed the class of finitely ground programs and the class of finitely recursive queries cannot be compared with respect to set inclusion. Finally, we show that each Turing machine defining a recursive function can be associated with a finitely recursive program, so that every computable function can still be expressed by the class.

The remainder of this chapter is structured as follows. In Section 4.1 Datalog is extended for allowing the use of function symbols and functional terms. In that section we also introduce the class of finitely ground programs and the class of finitely recursive queries and programs. Then, in Section 4.2 the cor-

rectness of Dynamic Magic Sets for finitely recursive queries is shown. After that, in Section 4.3 the decidability of reasoning over finitely recursive queries and programs is proved. Finally, in Section 4.4 the expressive power of finitely recursive programs is discussed.

4.1 Datalog with Function Symbols

The syntax of Disjunctive Datalog can be extended with function symbols and functional terms for allowing the representation of recursive structures. In this section we introduce some key concepts and provide some examples. Moreover, we present the class of finitely ground programs and the class of finitely recursive queries and programs.

4.1.1 Preliminaries

Let \mathcal{V} be a set of variables and \mathcal{S} be a set of predicate symbols, as introduced in Chapter 2. Moreover, let \mathcal{F} be a set of *function symbols* (or *functors*), each one associated with non-negative arity.¹ In this thesis, function symbols are denoted by strings starting with lower case letters or by numbers (function symbols representing numbers have arity zero).

A term is either a variable or a *functional term*, where the latter is obtained by combining functors and variables. Atoms, literals, rules and programs are defined as in the function-free case, yielding the language $\text{Datalog}_{\text{FS}}^{\vee, \neg}$.

Example 4.1.1. The following is a $\text{Datalog}_{\text{FS}}^{\vee, \neg}$ program:

```
int(0).
int(s(X)) :- int(X).
```

The program above contains two function symbols, namely s and 0 . The function symbol s has arity 1, while the function symbol 0 has arity zero (it is a constant). \square

Program instantiation and stable models of $\text{Datalog}_{\text{FS}}^{\vee, \neg}$ programs are defined as in the function-free case (see Section 2.2.1). The only difference is that for a $\text{Datalog}_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} the universe $\mathcal{U}_{\mathcal{P}}$ is obtained by combining functors in \mathcal{P} in all possible ways. Hence, if \mathcal{P} contains a functor of positive arity, $\mathcal{U}_{\mathcal{P}}$ contains infinitely many terms. Consequently, also the *base* $\mathcal{B}_{\mathcal{P}}$ and the instantiation $\text{Ground}(\mathcal{P})$ of a $\text{Datalog}_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} are infinite in general. For instance, the instantiation of the program in Example 4.1.1 is infinite and contains the following rules:

```
int(0).
int(s(0)) :- int(0).
int(s(s(0))) :- int(s(0)).
int(s(s(s(0)))) :- int(s(s(0))).
...
```

¹Function symbols of arity zero are constants.

Therefore, also stable models may contain infinitely many atoms. For instance, the program in Example 4.1.1 has a unique stable model which contains an infinite number of atoms:

$$\text{int}(0), \text{int}(s(0)), \text{int}(s(s(0))), \text{int}(s(s(s(0)))), \dots$$

Query answering over Datalog $_{\text{FS}}^{\vee, \neg}$ programs is undecidable in general. However, there are classes of programs for which the decidability of reasoning is guaranteed. For instance, finitely ground programs have equivalent and effectively computable finite ground programs, for which the stable model search phase can be performed as usual. This class and the class of finitely recursive programs and queries are presented in the next sections.

4.1.2 Finitely Ground Programs

The class of finitely ground (\mathcal{FG}) programs [20] constitutes a natural formalization of programs which can be finitely evaluated in a bottom-up way. We recall the key concepts in this section, and refer to [20] for details and examples.

Definition 4.1.2 (Positive Dependency Graph). Let \mathcal{P} be a Datalog $_{\text{FS}}^{\vee, \neg}$ program. The positive dependency graph of \mathcal{P} , denoted by $\mathcal{G}^+(\mathcal{P})$, is a directed graph having:

- a node for each IDB predicate of \mathcal{P} ;
- an edge $\mathbf{h} \rightarrow \mathbf{b}$ if there is a rule $r \in \mathcal{P}$ and two atoms $\mathbf{h}(\bar{\mathbf{u}}) \in H(r)$ and $\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r) \cup B^-(r)$.

Given a Datalog $_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} , a *component* C of \mathcal{P} is a maximal set of strongly connected nodes (predicates) in $\mathcal{G}^+(\mathcal{P})$, that is, two predicates \mathbf{p} and \mathbf{q} belong to the same component if and only if $\mathcal{G}^+(\mathcal{P})$ contains a path connecting \mathbf{p} to \mathbf{q} and a path connecting \mathbf{q} to \mathbf{p} .

Definition 4.1.3 (Component Graph). Let \mathcal{P} be a Datalog $_{\text{FS}}^{\vee, \neg}$ program. The component graph of \mathcal{P} , denoted by $\mathcal{G}^C(\mathcal{P})$, is a labeled directed graph having:

- a node for each component of $\mathcal{G}^+(\mathcal{P})$;
- an edge $C' \rightarrow^+ C$ if there is a rule $r \in \mathcal{P}$ and two atoms $\mathbf{h}(\bar{\mathbf{u}}) \in H(r)$ and $\mathbf{b}(\bar{\mathbf{v}}) \in B^+(r)$ such that $\mathbf{h} \in C$ and $\mathbf{b} \in C'$;
- an edge $C' \rightarrow^- C$ if
 - $C' \rightarrow^+ C$ is not an edge of $\mathcal{G}^C(\mathcal{P})$, and
 - there is a rule $r \in \mathcal{P}$ and two atoms $\mathbf{h}(\bar{\mathbf{u}}) \in H(r)$ and $\mathbf{b}(\bar{\mathbf{v}}) \in B^-(r)$ such that $\mathbf{h} \in C$ and $\mathbf{b} \in C'$.

In a component graph $\mathcal{G}^C(\mathcal{P})$ we distinguish between *weak paths* and *strong paths*: A path is weak if at least one of its edges is labeled with “−”, otherwise it is strong. A component ordering

$$\gamma = \langle C_1, \dots, C_n \rangle$$

is a total order for the components of \mathcal{P} such that the following conditions are satisfied for each pair of components C_i, C_j with $i < j$:

- there is no strong path from C_j to C_i in $\mathcal{G}^c(\mathcal{P})$;
- if there is a weak path from C_j to C_i , there must be a weak path also from C_i to C_j .

A component ordering γ for a Datalog $_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} allows for partitioning \mathcal{P} in modules. Each module is associated with a component C and contains only rules which define predicates in C . This concept is formalized below.

Definition 4.1.4 (Module). Let \mathcal{P} be a Datalog $_{\text{FS}}^{\vee, \neg}$ program,

$$\gamma = \langle C_1, \dots, C_n \rangle$$

a component ordering for \mathcal{P} , and C_i a component of \mathcal{P} . Then, the *module* of \mathcal{P} defining C_i , denoted by $P(C_i)$, comprises the rules of \mathcal{P} which define some predicate in C_i and no other predicate belonging to a component C_j such that $j < i$. In other words, each rule belongs to the module of the component which is minimal among all components to which head predicates of this rule belong.

Given a rule r and a set A of ground atoms, an instance $r\vartheta$ of r is an *A-restricted* instance of r if $B^+(r)\vartheta \subseteq A$. For a Datalog $_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} and a set A of ground atoms, the set of all *A-restricted* instances of the rules of \mathcal{P} is denoted by $Inst_{\mathcal{P}}(A)$. Note that

$$Inst_{\mathcal{P}}(A) \subseteq Ground(\mathcal{P})$$

holds for every $A \subseteq \mathcal{B}_{\mathcal{P}}$. Intuitively, the operator $Inst_{\mathcal{P}}(A)$ identifies those ground instances that may be *supported* by a given set A of ground atoms. Consider now a component ordering

$$\gamma = \langle C_1, \dots, C_n \rangle$$

for a Datalog $_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} , and two sets of ground rules T and R . Then, for every $i \in \{1, \dots, n\}$, the *simplification*

$$Simpl_i^\gamma(T, R)$$

of T with respect to R is obtained from T by performing the following operations:

1. Delete each rule r such that either the head or the negative body of r contains some atom $\mathbf{p}(\bar{\mathbf{t}})$ which is a fact in R , i.e., each rule r such that

$$(H(r) \cup B^-(r)) \cap \text{FACTS}(R) \neq \emptyset;$$

2. Eliminate from each remaining rule r the atoms in

$$B^+(r) \cap \text{FACTS}(R),$$

and each literal $\text{not } \mathbf{p}(\bar{\mathbf{t}})$ such that $\mathbf{p}(\bar{\mathbf{t}}) \in B^-(r)$ and \mathbf{p} belongs to a component C_j with $j < i$, and there is no rule $r' \in R$ with $\mathbf{p}(\bar{\mathbf{t}}) \in H(r')$.

Assuming that R contains all ground instances obtained from the modules preceding C_i , by invoking $Simpl_i^\gamma(T, R)$ we delete from T the rules which cannot contribute to the reasoning, either because of a “true” head atom or because

of a “false” body literal. The operator $Simpl_i^\gamma(T, R)$ simplifies the remaining rules by removing from all bodies those literals which are “true” with respect to R . We now define the operator Φ by combining $Inst$ and $Simpl$. Let \mathcal{P} be a $Datalog_{FS}^{\vee, \neg}$ program, C_i a component in a component ordering

$$\gamma = \langle C_1, \dots, C_n \rangle,$$

R and S two sets of ground rules. Then, the operator Φ is defined as the following set of ground rules:

$$\Phi_{\mathcal{P}, i, R}(S) = Simpl_i^\gamma(Inst_{P(C_i)}(\mathcal{H}), R),$$

where \mathcal{H} is the set of head atoms in $R \cup S$. Note that the operator Φ is monotone and thus always admit a least fixpoint

$$\Phi_{\mathcal{P}, i, R}^\infty(\emptyset).$$

We can then define the *intelligent instantiation* \mathcal{P}^γ of a $Datalog_{FS}^{\vee, \neg}$ program \mathcal{P} with respect to a component ordering

$$\gamma = \langle C_1, \dots, C_n \rangle$$

as the last element \mathcal{P}_n^γ of the following sequence:

- $\mathcal{P}_0^\gamma = EDB(\mathcal{P})$;
- $\mathcal{P}_i^\gamma = \mathcal{P}_{i-1}^\gamma \cup \Phi_{\mathcal{P}, i, \mathcal{P}_{i-1}^\gamma}^\infty(\emptyset)$, for each $i \in \{1, \dots, n\}$.

Definition 4.1.5 (Finitely Ground Program). Let \mathcal{P} be a $Datalog_{FS}^{\vee, \neg}$ program. If \mathcal{P}^γ is finite for every component ordering γ , the program \mathcal{P} is *finitely ground*. Let \mathcal{FG} denote the class of all finitely ground programs.

The main result for this class of programs is stated below.

Theorem 4.1.6 ([20]). *Cautious and brave reasoning over \mathcal{FG} programs are computable.*

4.1.3 Finitely Recursive Queries and Programs

The definition of finitely recursive queries and programs [9] are based on the notion of relevant atoms. Let \mathcal{P} be a program and \mathcal{Q} a ground query. The relevant atoms for \mathcal{Q} with respect to \mathcal{P} are:

- \mathcal{Q} itself;
- each atom in $ATOMS(r)$, where r is a rule in $Ground(\mathcal{P})$ such that some atom in $H(r)$ is relevant for \mathcal{Q} .

We can then define the class of finitely recursive queries and programs.

Definition 4.1.7 (Finitely Recursive Query and Program). Let \mathcal{Q} be a ground query over a $Datalog_{FS}^{\vee, \neg}$ program \mathcal{P} . Then,

- \mathcal{Q} is *finitely recursive* on \mathcal{P} if the set of relevant atoms for \mathcal{Q} with respect to \mathcal{P} is finite;

- \mathcal{P} is *finitely recursive* if every ground query is finitely recursive on \mathcal{P} .

Let $\text{Datalog}_{\text{FR}}^{\vee, \neg_s}$ denote the class of stratified finitely recursive $\text{Datalog}_{\text{FS}}^{\vee, \neg}$ queries and programs.

In the definition above, we have explicitly considered ground queries only. In fact, infinitely many atoms would be relevant for a query \mathcal{Q} with variables; for instance, all atoms $\mathcal{Q}\vartheta$ such that ϑ is a substitution for the variables of \mathcal{Q} . Therefore, in this chapter we only consider ground queries.

Example 4.1.8. Consider the query \mathcal{Q}_6 and the program \mathcal{P}_8 below:

```

 $\mathcal{Q}_6$  : greaterThan(s(s(0)), 0)?
 $r_{22}$  : int(0).
 $r_{23}$  : int(s(X)) :- int(X).
 $r_{24}$  : lessThan(X, s(X)) :- int(X).
 $r_{25}$  : lessThan(X, s(Y)) :- lessThan(X, Y).
 $r_{26}$  : greaterThan(s(X), Y) :- int(X), int(Y), not lessThan(X, Y).

```

The first observation is that \mathcal{P}_8 cautiously and bravely entails \mathcal{Q}_6 . Moreover, we note that \mathcal{Q}_6 is finitely recursive on \mathcal{P}_8 . Indeed, the number of relevant ground atoms is bounded by the term depth of the arguments, which themselves must be finite. \square

4.2 Dynamic Magic Sets for Finitely Recursive Queries

For queries over $\text{Datalog}_{\text{FR}}^{\vee, \neg_s}$ programs, the algorithm implementing Dynamic Magic Sets can be simplified by means of the restrictions in the definition of $\text{Datalog}_{\text{FR}}^{\vee, \neg_s}$. In this section we discuss these simplifications and extend the correctness result to $\text{Datalog}_{\text{FR}}^{\vee, \neg_s}$ queries.

4.2.1 The DMS Algorithm Revised

In this section we simplify the DMS algorithm reported in Figure 3.13 by means of the peculiarities of finitely recursive queries. These simplifications hold in general if at least one functor of positive arity is present; otherwise, if only constants and variables are used, the standard technique presented in Chapter 3 can be used.

For a finitely recursive query \mathcal{Q} over an $\text{Datalog}_{\text{FS}}^{\vee, \neg_s}$ program \mathcal{P} , the DMS algorithm can be simplified by means of the following observations:

- For each (sub)query $\mathbf{p}(\bar{\tau})$ and each rule r with an atom $\mathbf{p}(\bar{\tau}') \in H(r)$, all variables appearing in r must also appear in $\mathbf{p}(\bar{\tau}')$. Indeed, if this is not the case, infinitely many atoms would be relevant for \mathcal{Q} (i.e., the query would not be finitely recursive). Therefore, all adornment strings generated by the DMS algorithm contain only bs .
- For each (sub)query and for each processed rule r , all variables of r are bound by the (sub)query. Therefore, by properly limiting the adopted

```

Algorithm DMS( $\mathcal{Q}, \mathcal{P}$ )
Input: A query  $\mathcal{Q} = \mathbf{g}(\bar{\mathbf{z}})$  and a Datalog $_{\text{FR}}^{\vee, \neg^s}$  program  $\mathcal{P}$ 
Output: A rewritten program
var
   $S, D$ : set of predicates;
   $R_{\mathcal{Q}, \mathcal{P}}^{mgc}, R_{\mathcal{Q}, \mathcal{P}}^{mod}$ : set of rules;
   $r'$ : rule;
begin
  1.  $D := \emptyset$ ;  $R_{\mathcal{Q}, \mathcal{P}}^{mod} := \emptyset$ ;
  2.  $R_{\mathcal{Q}, \mathcal{P}}^{mgc} := \{\mathbf{magic\_g}(\bar{\mathbf{z}})\}$ ;  $S := \{\mathbf{g}\}$ ;
  3. while  $S \neq \emptyset$  do
  4.   take an element  $\mathbf{p}$  from  $S$ ; remove  $\mathbf{p}$  from  $S$ ; add  $\mathbf{p}$  to  $D$ ;
  5.   for each rule  $r$  in  $\mathcal{P}$  and for each atom  $\mathbf{p}(\bar{\mathbf{t}})$  in  $H(r)$  do
  6.      $r' := r$ ;
  7.     for each atom  $\mathbf{q}(\bar{\mathbf{s}}) \in H(r)$  do
  8.       add  $\mathbf{magic\_q}(\bar{\mathbf{s}})$  to  $B(r')$ ;
  9.     end for
  10.    add  $r'$  to  $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ ;
  11.    for each atom  $\mathbf{q}(\bar{\mathbf{s}}) \in \text{ATOMS}(r) \setminus \{\mathbf{p}(\bar{\mathbf{t}})\}$  s.t.  $\mathbf{q}$  is an IDB predicate do
  12.      add  $\mathbf{magic\_q}(\bar{\mathbf{s}}) :- \mathbf{magic\_p}(\bar{\mathbf{t}})$  to  $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ ;
  13.      if set  $D$  does not contain  $\mathbf{p}_i$  then add  $\mathbf{p}_i$  to  $S$ ; end if
  14.    end for
  15.  end while
  16. return  $R_{\mathcal{Q}, \mathcal{P}}^{mgc} \cup R_{\mathcal{Q}, \mathcal{P}}^{mod} \cup \text{EDB}(\mathcal{P})$ ;
end.

```

Figure 4.1: Dynamic Magic Sets algorithm for finitely recursive queries

SIPS, the bodies of the magic rules generated from r can contain only the magic version of the (sub)query.

The magic predicate associated with a predicate \mathbf{p} can then be denoted by $\mathbf{magic_p}$, meaning that all arguments of \mathbf{p} are considered bound. Thus, the magic atom associated with an atom $\mathbf{p}(\bar{\mathbf{t}})$ will be denoted by $\mathbf{magic_p}(\bar{\mathbf{t}})$.

The algorithm DMS implementing Dynamic Magic Sets for Datalog $_{\text{FR}}^{\vee, \neg^s}$ queries is reported in Figure 4.1. Given a program \mathcal{P} and a query $\mathcal{Q} = \mathbf{g}(\bar{\mathbf{z}})$, the algorithm outputs a rewritten program $\text{DMS}(\mathcal{Q}, \mathcal{P})$. The method exploits two sets, S and D , for storing predicates to be propagated and already processed, respectively. Magic rules are stored in the set $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$, while modified rules in $R_{\mathcal{Q}, \mathcal{P}}^{mod}$. Initially, D and $R_{\mathcal{Q}, \mathcal{P}}^{mod}$ are empty, S contains the predicate \mathbf{g} , and $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ contains the magic seed $\mathbf{magic_g}(\bar{\mathbf{z}})$ (lines 1–2). The main loop of the algorithm is then repeated until S becomes empty (lines 3–16). In the main loop, a predicate \mathbf{p} is moved from S to D (line 4) and each rule r having an atom $\mathbf{p}(\bar{\mathbf{t}})$ in its head is considered (lines 5–15). In particular, a modified rule r' is obtained from r by adding a positive body literal $\mathbf{magic_q}(\bar{\mathbf{s}})$ for each atom $\mathbf{q}(\bar{\mathbf{s}}) \in H(r)$ (lines 6–10). Moreover, a magic rule

$$r^* : \mathbf{magic_q}(\bar{\mathbf{s}}) :- \mathbf{magic_p}(\bar{\mathbf{t}}).$$

is generated for each atom $\mathbf{q}(\bar{\mathbf{s}}) \in \text{ATOMS}(r) \setminus \{\mathbf{p}(\bar{\mathbf{t}})\}$ such that \mathbf{q} is an IDB predicate. In this case, \mathbf{q} is also added to S if \mathbf{q} does not belong to D (lines

11–14). Finally, the algorithm terminates returning the program obtained by the union of $R_{\mathcal{Q},\mathcal{P}}^{mgc}$, $R_{\mathcal{Q},\mathcal{P}}^{mod}$ and $\text{EDB}(\mathcal{P})$ (line 17).

Example 4.2.1. The program obtained by applying DMS to the query and the program in Example 4.1.8 comprises the following rules:

```

 $r_{\mathcal{Q}_6}$  : magic_greaterThan(s(s(0)), 0).
 $r_{26,1}^*$  : magic_int(X) :- magic_greaterThan(s(X), Y).
 $r_{26,2}^*$  : magic_int(Y) :- magic_greaterThan(s(X), Y).
 $r_{26,3}^*$  : magic_lessThan(X, Y) :- magic_greaterThan(s(X), Y).
 $r_{24}^*$  : magic_int(X) :- magic_lessThan(X, s(X)).
 $r_{25}^*$  : magic_lessThan(X, Y) :- magic_lessThan(X, s(Y)).
 $r_{23}$  : magic_int(X) :- magic_int(s(X)).

 $r'_{22}$  : int(0) :- magic_int(0).
 $r'_{23}$  : int(s(X)) :- magic_int(s(X)), int(X).
 $r'_{24}$  : lessThan(X, s(X)) :- magic_lessThan(X, s(X)), int(X).
 $r'_{25}$  : lessThan(X, s(Y)) :- magic_lessThan(X, s(Y)), lessThan(X, Y).
 $r'_{26}$  : greaterThan(s(X), Y) :- magic_greaterThan(s(X), Y),
                                int(X), int(Y), not lessThan(X, Y).

```

In this case the unique stable model of $R_{\mathcal{Q}_6, \mathcal{P}_8}^{mgc}$ is

$$M^* = \{\text{magic_greaterThan}(s(s(0)), 0), \text{magic_int}(s(0)), \text{magic_int}(0), \text{magic_lessThan}(s(0), 0)\}.$$

The intelligent instantiation of $R_{\mathcal{Q}_6, \mathcal{P}_8}^{mod}$ is the program below:²

```

int(0) :- magic_int(0).
int(s(0)) :- magic_int(s(0)), int(0).
greaterThan(s(s(0)), 0) :- magic_greaterThan(s(s(0)), 0),
                                int(s(0)), int(0), not lessThan(s(0), 0).

```

Hence, \mathcal{P}_8 and $\text{DMS}(\mathcal{Q}_6, \mathcal{P}_8)$ are equivalent with respect to \mathcal{Q}_6 . \square

4.2.2 Query Equivalence Theorem

The correctness of Dynamic Magic Sets for the class of $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries can be proved by essentially following the proof presented in Section 3.2.4 for function-free programs. However, some definitions and proofs can be simplified because of the simplification of the DMS algorithm. A first observation is that Proposition 3.2.13 and Proposition 3.2.14 define syntactic properties which hold

²Note that only underlined atoms are actually present in the program generated by program instantiation. Atoms already known to be true have been included to simplify the identification of rules.

also in presence of function symbols. A second observation is that the definition of “killed atoms” can be simplified as follows.

Definition 4.2.2 (Killed Atoms for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$). Let \mathcal{Q} be a finitely recursive query over a $\text{Datalog}_{\text{FS}}^{\vee, \neg^s}$ program \mathcal{P} , M' a model of $\text{DMS}(\mathcal{Q}, \mathcal{P})$, and $N' \subseteq M'$ a model of $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))^{M'}$. The set

$$\text{killed}_{\mathcal{Q}, \mathcal{P}}^{M'}(N')$$

of the *killed atoms* for $\text{DMS}(\mathcal{Q}, \mathcal{P})$ with respect to M' and N' is the following set of ground atoms:

$$\{\mathbf{p}(\bar{\mathbf{t}}) \in \mathcal{B}_{\mathcal{P}} \setminus N' \mid \text{either } \mathbf{p} \text{ is an EDB predicate, or } \text{magic.p}(\bar{\mathbf{t}}) \in N'\}$$

Another simplification can be made in the definition of “magic variant”. Indeed, for a finitely recursive query \mathcal{Q} over a $\text{Datalog}_{\text{FS}}^{\vee, \neg^s}$ program \mathcal{P} , the program $R_{\mathcal{Q}, \mathcal{P}}^{\text{mgc}}$ is positive and disjunction-free. Therefore, $R_{\mathcal{Q}, \mathcal{P}}^{\text{mgc}}$ has a unique and finite stable model (see Lemma 4.3.3 in the next section for a detailed proof).

Definition 4.2.3 (Magic Variant for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$). Consider a finitely recursive query \mathcal{Q} over a $\text{Datalog}_{\text{FS}}^{\vee, \neg^s}$ program \mathcal{P} . Let I be an interpretation for \mathcal{P} . We define an interpretation $\text{variant}_{\mathcal{Q}, \mathcal{P}}(I)$ for $\text{DMS}(\mathcal{Q}, \mathcal{P})$, called the magic variant of I with respect to \mathcal{Q} and \mathcal{P} , as follows:

$$\text{variant}_{\mathcal{Q}, \mathcal{P}}(I) = \text{EDB}(\mathcal{P}) \cup M^* \cup \{\mathbf{p}(\bar{\mathbf{t}}) \in I \mid \text{magic.p}(\bar{\mathbf{t}}) \in M^*\},$$

where M^* is the unique stable model of $R_{\mathcal{Q}, \mathcal{P}}^{\text{mgc}}$.

We can then prove the correctness of Dynamic Magic Sets for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries.

Theorem 4.2.4 (Main Theorem for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$). *Let \mathcal{Q} be a finitely recursive query over a $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ program \mathcal{P} . Then, the following equivalences are established:*

- $\mathcal{P} \equiv_{\mathcal{Q}}^b \text{DMS}(\mathcal{Q}, \mathcal{P})$; and
- $\mathcal{P} \equiv_{\mathcal{Q}}^c \text{DMS}(\mathcal{Q}, \mathcal{P})$.

Proof. We start by observing that Theorem 3.2.20, claiming that killed atoms constitute an unfounded set, also holds for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries. Indeed, no assumption on the finiteness of ground programs is made in proving Theorem 3.2.20. By the same argument, Lemma 3.2.21 and Theorem 3.2.22 hold for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries. Hence, from these claims it follows that the Dynamic Magic Set technique for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries is sound. For proving the completeness, we have to consider Lemma 3.2.25 and Theorem 3.2.26. In particular, Lemma 3.2.25 has been proved by induction on the structure of the magic variant, which may be transfinite for an infinite stable model. However, we observe that for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries the definition of the magic variant does not use induction. Hence, the proof of Lemma 3.2.25 can be easily adapted for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries. For proving Theorem 3.2.26, instead, no assumption on the finiteness of ground programs has been made, so it can be extended to $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries with minor effort. \square

4.3 Decidability Theorem

The decidability of brave and cautious reasoning for $\text{Datalog}_{\text{FR}}^{\vee, \neg_s}$ queries is proved by providing a map to finitely ground programs. More specifically, we show that the application of Dynamic Magic Sets to finitely recursive queries generates finitely ground programs, for which query answering is known to be decidable. We start by showing some properties of programs generated by the application of Dynamic Magic Sets to finitely recursive queries.

Lemma 4.3.1. *Let Q be a finitely recursive query over a $\text{Datalog}_{\text{FS}}^{\vee, \neg}$ program \mathcal{P} , and $\text{DMS}(Q, \mathcal{P})$ the program obtained by applying Dynamic Magic Sets on Q and \mathcal{P} . Then, each cycle of dependencies in $\text{DMS}(Q, \mathcal{P})$ involving some predicates of \mathcal{P} is also present in \mathcal{P} .*

Proof. Consider first a cycle of dependencies in $\text{DMS}(Q, \mathcal{P})$ containing some magic predicates. In this case only magic predicates are involved in the cycle because magic rules have the form

$$\text{magic_q}(\bar{s}) \text{ :- magic_p}(\bar{t}).$$

Thus, the other cycles in $\text{DMS}(Q, \mathcal{P})$ only comprise standard predicates. These cycles depend only on modified rules, which have been obtained from rules in \mathcal{P} . Only magic atoms have been added to these rules, so the cycle is also present in \mathcal{P} . \square

The lemma above can be used for proving that Dynamic Magic Sets generate stratified programs when applied on $\text{Datalog}_{\text{FR}}^{\vee, \neg_s}$ queries. This property of Dynamic Magic Sets is due to the particular restriction imposed on the choice of SIPS in this section. In fact, in general the magic set rewriting of a stratified program can produce unstratified negation even if disjunction and functions are disallowed [41].

Corollary 4.3.2. *Let Q be a finitely recursive query over a $\text{Datalog}_{\text{FS}}^{\vee, \neg_s}$ program \mathcal{P} , and $\text{DMS}(Q, \mathcal{P})$ the program obtained by applying Dynamic Magic Sets on Q and \mathcal{P} . Then, $\text{DMS}(Q, \mathcal{P})$ is a $\text{Datalog}_{\text{FS}}^{\vee, \neg_s}$ program, that is, stratification with respect to negation is preserved.*

Proof. By applying Lemma 4.3.1, each cycle of dependencies in $\text{DMS}(Q, \mathcal{P})$ which involves some predicates of \mathcal{P} is also present in the original program \mathcal{P} . Each other cycle involves only magic predicates and positive dependencies because magic rules contain no negative literals. \square

All magic rules generated by Dynamic Magic Sets are characterized by atomic heads and positive bodies. For finitely recursive queries, magic rules are also characterized by bodies consisting of exactly one magic atom. It can be proved that the program comprising these magic rules is unique and finite.

Lemma 4.3.3. *Let Q be a finitely recursive query over a $\text{Datalog}_{\text{FS}}^{\vee, \neg_s}$ program \mathcal{P} . Let $R_{Q, \mathcal{P}}^{\text{mgc}}$ be the program which comprises all magic rules produced while computing $\text{DMS}(Q, \mathcal{P})$. Then, $R_{Q, \mathcal{P}}^{\text{mgc}}$ has a unique and finite stable model that will be denoted by M^* .*

Proof. Since $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$ is positive and normal, it admits a unique stable model M^* . We shall show that each ground magic atom in M^* corresponds to a relevant atom for \mathcal{Q} with respect to \mathcal{P} , from which finiteness of M^* follows. Let $\text{magic_q}(\bar{s})\vartheta$ be an atom in M^* (ϑ a substitution) different from the magic seed. Hence, $\text{magic_q}(\bar{s})\vartheta$ must be supported by a magic rule

$$r^*\vartheta : \text{magic_q}(\bar{s})\vartheta :- \text{magic_p}(\bar{t})\vartheta.$$

in $\text{Ground}(\text{DMS}(\mathcal{Q}, \mathcal{P}))$ such that $\text{magic_p}(\bar{t})\vartheta$ belongs to M^* . We shall show that $\text{q}(\bar{s})\vartheta$ is relevant for $\text{p}(\bar{t})\vartheta$ in this case. In fact, the magic rule r^* in $\text{DMS}(\mathcal{Q}, \mathcal{P})$ has been generated from a rule $r \in \mathcal{P}$ such that:

- the atom $\text{p}(\bar{t})$ occurs in $H(r)$; and
- the atom $\text{q}(\bar{s})$ belongs to $\text{ATOMS}(r) \setminus \{\text{p}(\bar{t})\}$.

Recall that, by previous observations, all variables in r also appear in $\text{p}(\bar{t})$. Hence, $r\vartheta$ belongs to $\text{Ground}(\mathcal{P})$ and is such that:

- $\text{p}(\bar{t})\vartheta$ is an atom in its head;
- $\text{q}(\bar{s})\vartheta$ occurs in the rule.

Therefore, $\text{q}(\bar{s})\vartheta$ is relevant for $\text{p}(\bar{t})\vartheta$. This is enough for proving that each atom in M^* corresponds to an atom which is relevant for \mathcal{Q} with respect to \mathcal{P} because the query seed is the only fact in $R_{\mathcal{Q}, \mathcal{P}}^{mgc}$. \square

We can now link $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries and finitely ground programs by means of the following theorem.

Theorem 4.3.4. *Let \mathcal{Q} be a finitely recursive query over a $\text{Datalog}_{\text{FS}}^{\vee, \neg^s}$ program \mathcal{P} , and $\text{DMS}(\mathcal{Q}, \mathcal{P})$ the program obtained by applying Dynamic Magic Sets on \mathcal{Q} and \mathcal{P} . Then, $\text{DMS}(\mathcal{Q}, \mathcal{P})$ is a finitely ground program.*

Proof. Consider a component ordering

$$\gamma = \langle C_1, \dots, C_n \rangle$$

for $\text{DMS}(\mathcal{Q}, \mathcal{P})$. We have to show that the intelligent instantiation of $\text{DMS}(\mathcal{Q}, \mathcal{P})$ with respect to γ , $\text{DMS}(\mathcal{Q}, \mathcal{P})^\gamma$, is finite. We start by observing that the components with non-magic predicates are disjoint from the components with magic predicates because of Lemma 4.3.1. Therefore, for proving that $\text{DMS}(\mathcal{Q}, \mathcal{P})^\gamma$ is finite, we have to consider two kinds of components. For a component C_i containing magic predicates, $\text{DMS}(\mathcal{Q}, \mathcal{P})_i^\gamma$ is a subset of M^* , which is finite by Lemma 4.3.3. For a component C_i containing standard predicates, we consider a modified rule $r' \in P(C_i)$ having the following form:

$$r' : \text{h}_1(\bar{u}_1) \vee \dots \vee \text{h}_m(\bar{u}_m) :- \text{magic_h}_1(\bar{u}_1), \dots, \text{magic_h}_m(\bar{u}_m), \\ \text{b}_1(\bar{v}_1), \dots, \text{b}_k(\bar{v}_k), \text{not } \text{b}_{k+1}(\bar{v}_{k+1}), \dots, \text{not } \text{b}_n(\bar{v}_n).$$

We observe that in any γ , C_i follows each component which contains a predicate magic_h_j , for $j \in \{1, \dots, m\}$. Moreover, since \mathcal{Q} is finitely recursive on \mathcal{P} , each variable appearing in r' also appears in some magic atom in $B(r')$. Therefore, $\text{DMS}(\mathcal{Q}, \mathcal{P})_i^\gamma$ is finite also in this case, because M^* is finite. \square

We are now ready for proving the decidability of brave and cautious reasoning over finitely recursive queries.

Theorem 4.3.5. *Let \mathcal{Q} be a finitely recursive query over a $\text{Datalog}_{\text{FS}}^{\vee, \neg^s}$ program \mathcal{P} . Then, deciding whether \mathcal{P} cautiously (or bravely) entails \mathcal{Q} is computable.*

Proof. Let $\text{DMS}(\mathcal{Q}, \mathcal{P})$ be the program obtained by applying Dynamic Magic Sets on \mathcal{Q} and \mathcal{P} . Then, the following equivalences are established by Theorem 4.2.4:

- $\mathcal{P} \equiv_{\mathcal{Q}}^b \text{DMS}(\mathcal{Q}, \mathcal{P})$; and
- $\mathcal{P} \equiv_{\mathcal{Q}}^c \text{DMS}(\mathcal{Q}, \mathcal{P})$.

Since $\text{DMS}(\mathcal{Q}, \mathcal{P})$ is finitely ground by Theorem 4.3.4, the decidability of brave and cautious reasoning follows from Theorem 4.1.6. \square

4.4 Expressive Power of Finitely Recursive Programs

The restrictions that guarantee the decidability of reasoning for $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ queries do not limit the expressiveness of the class. Indeed, all computable functions can be expressed by $\text{Datalog}_{\text{FR}}^{\vee, \neg^s}$ programs (even without using disjunction and negation). In this section we formalize this result by showing how to encode a deterministic Turing machine as a positive program with functions. Input strings are encoded by queries which are finitely recursive if the Turing machine halts. Essentially, we adapt a well-known result in classic first-order logic to our notation and terminology: All computable functions can be represented by Horn clauses [64].

A Turing machine \mathcal{M} with semi-infinite tape is a 5-tuple

$$\mathcal{M} = \langle \Sigma, \mathcal{S}, \mathbf{s}_i, \mathbf{s}_f, \delta \rangle,$$

where:

- Σ is an alphabet (i.e., a set of symbols);
- \mathcal{S} is a set of states;
- $\mathbf{s}_i, \mathbf{s}_f \in \mathcal{S}$ are two distinct states, which are respectively used for representing the initial and final state of \mathcal{M} ;
- δ is a transition function, that is,

$$\delta : \mathcal{S} \times \Sigma \longrightarrow \mathcal{S} \times \Sigma \times \{\leftarrow, \rightarrow\}.$$

Given an input string

$$x = \mathbf{x}_1 \cdots \mathbf{x}_n,$$

the initial configuration of \mathcal{M} is such that:

- the current state is \mathbf{s}_i ;
- the tape contains x followed by an infinite sequence of blank symbols (represented by \sqcup), a special tape symbol in Σ ;³

³We assume x does not contain any blank symbol.

- the head is over the first symbol of the tape.

The other configurations assumed by \mathcal{M} with input x are then obtained by means of the transition function δ . In particular, if \mathbf{s} and \mathbf{v} are the current state and symbol, respectively, and

$$\delta(\mathbf{s}, \mathbf{v}) = (\mathbf{s}', \mathbf{v}', \mathbf{m}),$$

the successive configuration is obtained by means of the following operation:

1. \mathcal{M} overwrites \mathbf{v} with \mathbf{v}' ;
2. \mathcal{M} moves its head according to $\mathbf{m} \in \{\leftarrow, \rightarrow\}$;
3. \mathcal{M} changes its state to \mathbf{s}' .

We assume that \mathcal{M} never moves its head to the left of the first symbol of the tape. The machine \mathcal{M} *accepts* x if the final state \mathbf{s}_f is reached at some point of the computation.

A Turing machine \mathcal{M} can be simulated by means of a Datalog $_{\text{FS}}^{\vee, \neg, s}$ program. In particular, configurations of \mathcal{M} are encoded by instances of

$$\text{conf}(\mathbf{S}, \mathbf{L}, \mathbf{V}, \mathbf{R}),$$

where:

- \mathbf{S} is the current state;
- \mathbf{L} is the list of symbols on the left of the head in reverse order;
- \mathbf{V} is the symbol under the head;
- \mathbf{R} is a finite list of symbols on the right of the head containing at least all the non-blank symbols.

Hence, the initial configuration of \mathcal{M} with input x is encoded by the query $\mathcal{Q}_{\mathcal{M}(x)}$ below:

$$\begin{aligned} \text{conf}(\mathbf{s}_i, [], \mathbf{x}_1, [\mathbf{x}_2, \dots, \mathbf{x}_n])? & \quad \text{if } n > 0; \\ \text{conf}(\mathbf{s}_i, [], \sqcup, [])? & \quad \text{otherwise.} \end{aligned}$$

The machine \mathcal{M} is implemented by a program $\mathcal{P}_{\mathcal{M}}$ containing a rule

$$\text{conf}(\mathbf{s}_f, \mathbf{L}, \mathbf{V}, \mathbf{R}).$$

for representing the final state \mathbf{s}_f , and a set of rules implementing the transition function δ . More specifically, for each state $\mathbf{s} \in \mathcal{S} \setminus \{\mathbf{s}_f\}$ and for each symbol $\mathbf{v} \in \Sigma$, the program $\mathcal{P}_{\mathcal{M}}$ contains the following rules:

$$\begin{aligned} \text{conf}(\mathbf{s}, [\mathbf{V}|\mathbf{L}], \mathbf{v}, \mathbf{R}) & \text{ :- } \text{conf}(\mathbf{s}', \mathbf{L}, \mathbf{V}, [\mathbf{v}'|\mathbf{R}]). & \quad \text{if } \delta(\mathbf{s}, \mathbf{v}) = (\mathbf{s}', \mathbf{v}', \leftarrow); \\ \text{conf}(\mathbf{s}, \mathbf{L}, \mathbf{v}, [\mathbf{V}|\mathbf{R}]) & \text{ :- } \text{conf}(\mathbf{s}', [\mathbf{v}'|\mathbf{L}], \mathbf{V}, \mathbf{R}). & \quad \text{if } \delta(\mathbf{s}, \mathbf{v}) = (\mathbf{s}', \mathbf{v}', \rightarrow); \\ \text{conf}(\mathbf{s}, \mathbf{L}, \mathbf{v}, []) & \text{ :- } \text{conf}(\mathbf{s}', [\mathbf{v}'|\mathbf{L}], \sqcup, []). & \quad \text{if } \delta(\mathbf{s}, \mathbf{v}) = (\mathbf{s}', \mathbf{v}', \rightarrow). \end{aligned}$$

Note that we do not explicitly represent the infinite sequence of blanks on the right of the tape; the last rule above effectively produces a blank whenever the head moves right of all explicitly represented symbols. Therefore, the atoms produced by a top-down evaluation of the query and program above represent only the effectively reached tape positions. We also note that the program above is not safe due to the rule encoding the final state. A safe program can be achieved by introducing a fact $\mathbf{sigma}(v)$ for each $v \in \Sigma$ and by replacing the unsafe rule with the following rules:

$$\begin{aligned} \mathbf{conf}(s_f, L, V, R) &:- \mathbf{sigma}(V), \mathbf{list}(L), \mathbf{list}(R). \\ \mathbf{list}([\]). \\ \mathbf{list}([V|L]) &:- \mathbf{sigma}(V), \mathbf{list}(L). \end{aligned}$$

However, it can be checked that the two programs are equivalent with respect to every query representing an initial configuration for \mathcal{M} . We can then prove that $\mathcal{Q}_{\mathcal{M}(x)}$ and $\mathcal{P}_{\mathcal{M}}$ actually implement the machine \mathcal{M} with input x .

Theorem 4.4.1. *Consider a Turing machine \mathcal{M} and an input string x . Let $\mathcal{Q}_{\mathcal{M}(x)}$ and $\mathcal{P}_{\mathcal{M}}$ be the query and the program encoding \mathcal{M} with input x as described above. Then, $\mathcal{P}_{\mathcal{M}}$ bravely and cautiously entails $\mathcal{Q}_{\mathcal{M}(x)}$ if and only if \mathcal{M} accepts x .*

Proof Sketch. We observe that the program $\mathcal{P}_{\mathcal{M}}$ bravely and cautiously entails the query $\mathcal{Q}_{\mathcal{M}(x)}$ if and only if the unique stable model of $\mathcal{P}_{\mathcal{M}}$ contains a sequence of ground atoms

$$\mathbf{conf}(\bar{t}_1), \dots, \mathbf{conf}(\bar{t}_m)$$

such that:

- $\mathbf{conf}(\bar{t}_1)$ is the query atom;
- $\mathbf{conf}(\bar{t}_m)$ is an instance of the atom $\mathbf{conf}(s_f, L, V, R)$, that is, the first argument of $\mathbf{conf}(\bar{t}_m)$ is s_f ;
- for each $i \in \{1, \dots, m-1\}$, the rule

$$\mathbf{conf}(\bar{t}_i) :- \mathbf{conf}(\bar{t}_{i+1}).$$

belongs to $\mathit{Ground}(\mathcal{P}_{\mathcal{M}})$; intuitively, the rule above is an instance of a rule implementing the transition function of \mathcal{M} .

Hence, from these observations and since instances of $\mathbf{conf}(\bar{t})$ represent configurations of \mathcal{M} , the claim follows. \square

We can now link computable sets and finitely recursive queries.

Theorem 4.4.2. *Let L be a computable set. Then, there is an Datalog $_{\text{FR}}^{\vee, \neg s}$ program \mathcal{P} deciding L ; that is, for each string x there is a query \mathcal{Q} such that \mathcal{Q} is finitely recursive on \mathcal{P} . Moreover, \mathcal{Q} is such that \mathcal{P} cautiously (or bravely) entails \mathcal{Q} if and only if $x \in L$.*

Proof. Since L is computable, there is a Turing machine \mathcal{M} computing it. Let $\mathcal{P} = \mathcal{P}_{\mathcal{M}}$ be the program encoding \mathcal{M} as described above. Moreover, for a string x , let $\mathcal{Q} = \mathcal{Q}_{\mathcal{M}(x)}$ be the query encoding the initial configuration of \mathcal{M} with input x as described above. Hence, by Theorem 4.4.1, $\mathcal{P}_{\mathcal{M}}$ cautiously and bravely entails $\mathcal{Q}_{\mathcal{M}(x)}$ if and only if $x \in L$. Therefore, it remains to show that $\mathcal{P}_{\mathcal{M}}$ is a Datalog $_{\text{FR}}^{\vee, \neg s}$ program. Clearly, $\mathcal{P}_{\mathcal{M}}$ is Datalog $_{\text{FS}}^{\vee, \neg s}$ (actually, it is even negation-free), so we have to prove that $\mathcal{Q}_{\mathcal{M}(x)}$ is finitely recursive on $\mathcal{P}_{\mathcal{M}}$. By construction of $\mathcal{P}_{\mathcal{M}}$, for each ground atom $\text{conf}(\bar{\tau})$ in $\mathcal{B}_{\mathcal{P}_{\mathcal{M}}}$ there is exactly one rule in $\text{Ground}(\mathcal{P}_{\mathcal{M}})$ having $\text{conf}(\bar{\tau})$ in head. This rule has at most one atom $\text{conf}(\bar{\tau}')$ in its body and implements either the transition function or the final state of \mathcal{M} . Thus, the atoms relevant for $\mathcal{Q}_{\mathcal{M}(x)}$ are exactly the atoms representing the configurations assumed by \mathcal{M} with input x . The claim then follows because \mathcal{M} halts in a finite number of steps by assumption. \square

We note that, when Dynamic Magic Sets are applied on the program and query used to encode a Turing machine, the magic predicates effectively encode all reachable configurations and a bottom-up evaluation of the magic program corresponds to a simulation of the Turing machine. Hence, only encodings of Turing machine invocations that visit all (infinitely many) tape cells are not finitely recursive. We also note that recognizing whether a Datalog $_{\text{FR}}^{\vee, \neg s}$ query is finitely recursive is RE-complete, that is, complete for the class of recursively enumerable decision problems.

Chapter 5

Implementation and Experiments

Dynamic Magic Sets have been implemented in DLV [47]. The achieved prototype system allows for efficient query answering over Disjunctive Datalog programs. In this chapter, the architecture of the prototype system is presented. Moreover, experimental results assessing the effectiveness of Dynamic Magic Sets are reported. Further experimental results on an application scenario using real-world data are reported in Chapter 6.

5.1 System Architecture

Disjunctive Magic Sets have been implemented and integrated into the core of the DLV system. The architecture of the prototype is presented in Figure 5.1. DLV supports both brave and cautious reasoning. Brave reasoning is performed if the command-line option `-FB` is specified, while `-FC` is used for cautious reasoning. The DMS algorithm is applied by default only for (partially) bound queries, and can be disabled by specifying the option `-ODMS-`. For completely free queries, instead, the application of DMS can be explicitly requested by specifying the option `-ODMS`. Finally, for completely bound queries the magic variant of the stable model witnessing the truth or the falsity of the query, depending on whether brave or cautious reasoning is adopted, can be printed by specifying the option `--print-model`.¹

In our prototype, the input query and program are processed by the *Magic Set Rewriter* module, which implements DMS. Then, the rewritten and optimized program is processed by the *Intelligent Grounding* module, which implements program instantiation. After that, the ground program is processed by the *Model Generator* module, which implements stable model search. Therefore, apart from the new *Magic Set Rewriter* module, the only modification to the original DLV is for the output: For ground queries the witnessing stable model is no longer printed by default, but only if `--print-model` is specified. An executable of the DLV system supporting Dynamic Magic Sets is available at <http://www.dlvsystem.com/magic/>.

¹Magic predicates are not printed.

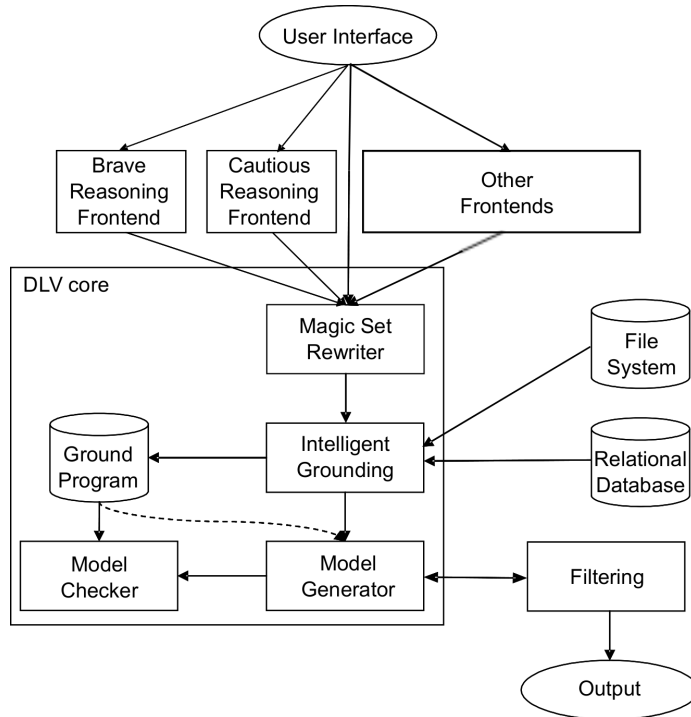


Figure 5.1: DLV prototype system architecture

5.2 Compared Methods, Benchmark Problems and Data

Effectiveness of Dynamic Magic Sets has been assessed empirically. The experiments comprise the following benchmarks:

- Strategic Companies;
- Simple Path;
- Related;
- Conformant Plan Checking;
- Super-Coherent Encodings for Related and Conformant Plan Checking.

In particular, the first three benchmarks have already been used to assess SMS in [38]. In the experiments, each benchmark is associated with a Disjunctive Datalog query. For each benchmark, the average execution time of DLV has been measured for:

- original queries (No Magic);
- queries optimized by Static Magic Sets (SMS);

- queries optimized by Dynamic Magic Sets (DMS).

A detailed description of each benchmark is provided below.

Strategic Companies

In the Strategic Companies problem, a collection

$$C = c_1, \dots, c_m$$

of companies is given, for some $m \geq 1$. Each company produces some goods in a set G , and each company $c_i \in C$ is possibly controlled by a set of owner companies $O_i \subseteq C$. In this context, a set $C' \subseteq C$ of companies is a *strategic set* if it is minimal among all sets satisfying the following conditions:

- Companies in C' produce all goods in G ;
- $O_i \subseteq C'$ implies $c_i \in C'$, for each $i = 1, \dots, m$.

In our benchmark, two distinct companies $c_i, c_j \in C$ are also provided in input, and the existence of a strategic set C' such that c_i and c_j occur in C' has to be checked. We used the instances submitted to the Second Answer Set Competition,² which are subjected to two additional restrictions:

- each product is produced by at most four companies; and
- each company is controlled by at most four companies.

Under these restrictions the problem is still Σ_2^P -complete [17], i.e., complete for the second level of the polynomial hierarchy. Instances of Strategic Companies are encoded by means of the predicates `producedBy` and `controlledBy`. In particular, there is a fact

$$\text{producedBy}(p, c_a, c_b, c_c, c_d)$$

when a product p is produced by companies c_a, c_b, c_c and c_d , and a fact

$$\text{controlledBy}(c, c_a, c_b, c_c, c_d),$$

when a company c is controlled by companies c_a, c_b, c_c and c_d . If a product p is produced by less than four companies (but at least one), the atom `producedBy(p, c_a, c_b, c_c, c_d)` contains repetitions of companies. For instance, `producedBy(p, c_a, c_b, c_c, c_c)` is used for representing that p is produced solely by c_a, c_b and c_c . Analogously, if a company c is controlled by less than four companies. The problem is encoded by the query

$$\text{st}(c_i), \text{st}(c_j)?$$

over the following program:

$$\begin{aligned} \text{st}(C_1) \vee \text{st}(C_2) \vee \text{st}(C_3) \vee \text{st}(C_4) &:- \text{producedBy}(P, C_1, C_2, C_3, C_4). \\ \text{st}(C) &:- \text{controlledBy}(C, C_1, C_2, C_3, C_4), \text{st}(C_1), \text{st}(C_2), \text{st}(C_3), \text{st}(C_4). \end{aligned}$$

²<http://www.cs.kuleuven.be/~dtai/events/ASP-competition/index.shtml>

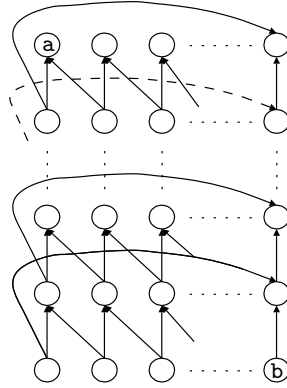


Figure 5.2: Structure of Simple Path and Related instances

We point out that conjunctive queries are allowed in DLV: Essentially, new rules and symbols are introduced as described in Chapter 2. In this case DLV replaces the original query by

$$q(c_i, c_j)?$$

and adds the following rule:

$$q(c_i, c_j) :- st(c_i), st(c_j).$$

The companies c_i and c_j belong to a strategic set C' if and only if the query is bravely true.

Simple Path

The Simple Path problem can be formulated as follows:

Given a directed graph G and two nodes a and b , does there exist a unique path connecting a to b in G ?

Instances of Simple Path are encoded by means of the predicate **edge**. In particular, there is a fact

$$\mathbf{edge}(u, v)$$

when an arc connects node u to node v in graph G . The structure of G is the same used in [38] and consists of a square matrix of nodes connected as shown in Figure 5.2. Instances of this benchmark have been generated by varying the number of nodes. The problem is encoded by the query

$$\mathbf{sp}(a, b)?$$

over the following program:

```

sp(X, X) v not_sp(X, X) :- edge(X, Y).
sp(X, Y) v not_sp(X, Y) :- sp(X, Z), edge(Z, Y).
path(X, Y) :- sp(X, Y).
path(X, Y) :- not_sp(X, Y).
not_sp(X, Z) :- path(X, Y1), path(X, Y2), Y1 ≠ Y2,
                edge(Y1, Z), edge(Y2, Z).

```

The last rule above derives all pairs of nodes which are connected by two different paths in G . The first two rules, instead, guess whether there is a unique path between two nodes. It can be shown that there is a unique path in G connecting a to b if and only if the query is bravely true. Note that a simpler encoding could be obtained by using stratified negation, but such an encoding would have prevented the comparison with SMS.

Related

The Related problem can be formulated as follows:

Given a genealogy graph storing information of relationship (father/brother) among people, and given two distinct people a and b , is b a possible ancestor of a ?

Instances of Related are encoded by means of the predicate `related`. In particular, there is a fact

$$\text{related}(x, y)$$

when a person x is known to be related to a person y , that is, when x is the father or a brother of y . The structure of the “genealogy” graph, reported in Figure 5.2, is the same used in [38] and coincides with the one used for testing Simple Path. Also in this case, instances have been generated by varying the number of nodes in the graph (thus, the number of people in the genealogy). The problem is encoded by the query

$$\text{ancestorOf}(b, a)?$$

over the following program:

```

fatherOf(X, Y) v brotherOf(X, Y) :- related(X, Y).
ancestorOf(X, Y) :- fatherOf(X, Y).
ancestorOf(X, Y) :- fatherOf(X, Z), ancestorOf(Z, Y).

```

The query is bravely true if and only if b is a possible ancestor of a .

Conformant Plan Checking

The Conformant Plan Checking problem has been designed for highlighting the dynamic optimization provided by DMS to the stable model search phase. The problem is inspired by a setting in planning, in particular, testing whether a given plan is conformant with respect to a state transition diagram [35]. A

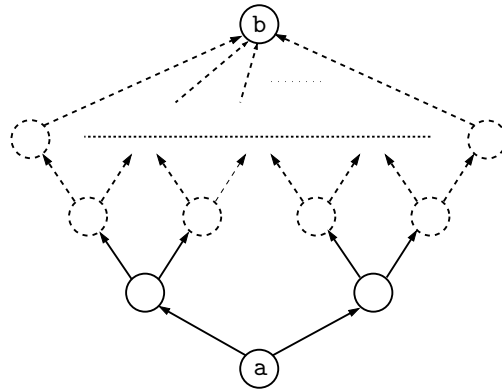


Figure 5.3: Structure of Conformant Plan Checking instances

state transition diagram is essentially a directed graph formed of nodes, representing states, and arcs labeled by actions, meaning that executing the action in the source state will lead to the target state. In the considered setting non-determinism is allowed, that is, executing an action in one state might lead nondeterministically to one of several successor states. A plan is a sequence of actions, and it is conformant with respect to a given initial state a and a goal state b if each possible execution of the action sequence leads from a to b .

In the benchmark, we assume that the action selection process has already been performed. The state transition diagram is then assumed to be already reduced to the transitions that may occur when executing the given plan. Furthermore, we assume that there are at most two possible successor states for each state. More specifically, the transition graphs in our experiment have the shape of binary trees rooted in the initial state a , augmented by one additional node and arcs such that all leaves are connected to the final state b , as depicted in Figure 5.3. This can also be viewed as whether all outgoing paths of a node in a directed graph reach a particular confluence node. In the benchmark, instances are encoded by means of the predicate `ptrans`. In particular, there is a fact

$$\text{ptrans}(s, s_i, s_j)$$

when s_i and s_j are the possible successor states of a state s . If s is a leaf of the tree in the transition graph, and only in this case, $s_i = s_j = b$. The problem is encoded using the query

$$\text{reach}(a, b)?$$

over the following program:

```
trans(X,Y) v trans(X,Z) :- ptrans(X,Y,Z).
reach(X,Y) :- trans(X,Y).
reach(X,Y) :- reach(X,Z), trans(Z,Y).
```

The plan is conformant if and only if the query is cautiously true.

Super-Coherent Encodings for Related and Conformant Plan Checking

The experiment is completed by two Datalog_{SC}^{V,¬} encodings for Related and Conformant Plan Checking. In particular, Related is encoded by the following program:

```

fatherOf(X,Y) :- related(X,Y), not brotherOf(X,Y).
brotherOf(X,Y) :- related(X,Y), not fatherOf(X,Y).
ancestorOf(X,Y) :- fatherOf(X,Y).
ancestorOf(X,Y) :- fatherOf(X,Z), ancestorOf(Z,Y).

```

For each fact `related(x,y)`, either `fatherOf(x,y)` or `brotherOf(x,y)` is guessed by the first and second rules above. Then, the transitive closure of the relation `fatherOf` is computed as usual by the last two rules. Conformant Plan Checking, instead, is encoded by the following program:

```

trans(X,Y) :- ptrans(X,Y,Z), Y ≠ Z, not trans(X,Z).
trans(X,Z) :- ptrans(X,Y,Z), Y ≠ Z, not trans(X,Y).
trans(X,Y) :- ptrans(X,Y,Y).
reach(X,Y) :- trans(X,Y).
reach(X,Y) :- reach(X,Z), trans(Z,Y).

```

For each state in the input transition graph, if two different successor states are given, a nondeterministic choice is made by the first and second rules above. Otherwise, if there is just one successor state, a deterministic choice is made by the third rule. Note that the condition $Y \neq Z$ is important in the first two rules, since dropping it would render the program incoherent (and thus incorrect) for all instances containing states with only one successor state (all of the instances considered here).

5.3 Results and Discussion

The experiment has been performed on a 3GHz Intel[®] Xeon[®] processor system with 4GB RAM under the Debian 4.0 operating system with a GNU/Linux 2.6.23 kernel. The DLV prototype used has been compiled using GCC 4.3.3. For each instance of the benchmark, we have allowed a maximum running time of 600 seconds (10 minutes) and a maximum memory usage of 3GB.

On all considered problems DMS performs better or equal to SMS. Both DMS and SMS improve DLV without Magic Sets. We point out that the time needed for rewriting is included only for DMS. In fact, we have been unable to obtain an implementation of Static Magic Sets, and thus performed SMS rewritings manually. The results are analyzed in detail below.

Strategic Companies. The results for Strategic Companies are reported in Figure 5.4 and in Table 5.1. Instances do not have a uniform structure, but are ordered by size in the graph. Without magic sets only the smallest two instances are solved in the allotted time. Substantial performance gains are provided by

SMS and DMS, which essentially perform equally in this benchmark. In fact, we have verified that SMS and DMS produce exactly the same rewritten program for the query and program encoding Strategic Companies.

Simple Path. The results for Simple Path are shown in Figure 5.5 and in Table 5.2. Without magic sets only the smallest instances are solved in the allotted time, with a very steep increase in execution time. Even if SMS provides a sensible performance gain, it scales worse than DMS. In this case the gap between SMS and DMS is mostly due to the grounding of the additional predicates introduced by SMS.

Related. The results for Related are reported in Figure 5.6 and in Table 5.3. An even steeper increase in runtime with respect to Simple Path is exhibited by the execution without magic sets, while the benefit provided by DMS are much more evident in this case. In particular, DMS appears to have an exponential speedup over SMS. In this case DMS outperforms SMS thanks to the optimization potential provided to stable model search by the magic atoms.

Conformant Plan Checking. The results for Conformant Plan Checking are shown in Figure 5.7 and in Table 5.4. Without magic sets only the smallest instances are solved. Also SMS does not scale well at all, while DMS appears to be exponentially faster than SMS. In this case DMS takes advantage of the conditional relevance implemented by its magic atoms. In fact, the magic atoms of DMS provide further optimization potential to stable model search: Parts of the ground program are disabled according to previously made assumptions. On the contrary, magic atoms of SMS have deterministic definitions and cannot provide any optimization potential to stable model search.

Super-Coherent Encodings for Related and Conformant Plan Checking. The results for these benchmarks are reported in Figures 5.8–5.9 and in Tables 5.5–5.6. SMS has been not considered because its correctness has not been proved for programs with recursive negation. We observe that the results obtained with super-coherent encodings are almost the same as those obtained by the positive encodings discussed previously. We point out that DLV performs exponentially better with DMS also in this case.

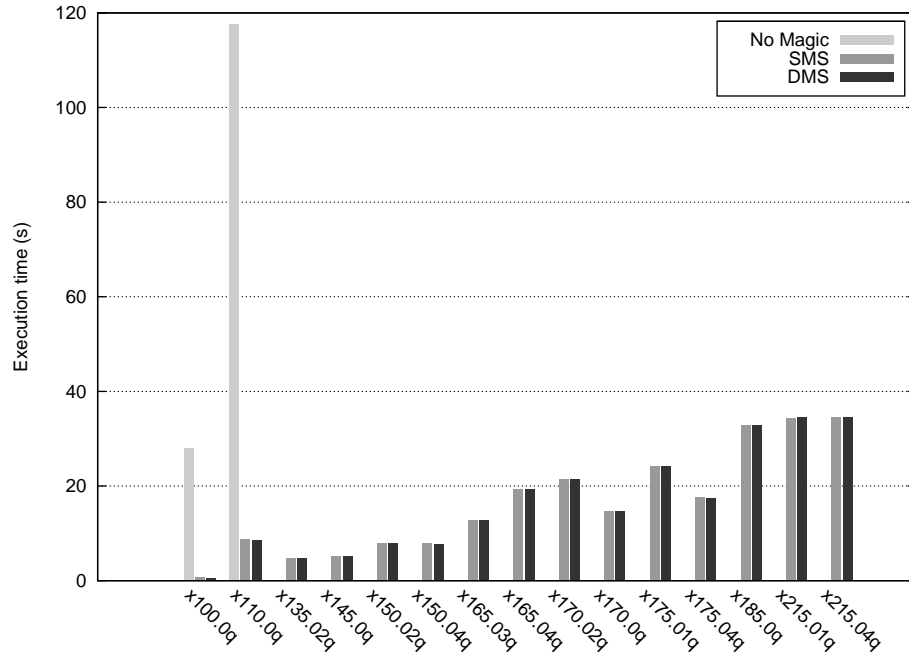


Figure 5.4: Average execution time for Strategic Companies

Table 5.1: Average execution time for Strategic Companies

Instance Name	No Magic (seconds)	SMS (seconds)	DMS (seconds)
x100.0q	28.07	0.61	0.60
x110.0q	117.67	8.66	8.58
x135.02q	—	4.68	4.64
x145.0q	—	5.21	5.18
x150.02q	—	7.84	7.80
x150.04q	—	7.81	7.78
x165.03q	—	12.82	12.80
x165.04q	—	19.32	19.33
x170.02q	—	21.45	21.37
x170.0q	—	14.75	14.74
x175.01q	—	24.27	24.15
x175.04q	—	17.53	17.48
x185.0q	—	32.86	32.86
x215.01q	—	34.39	34.53
x215.04q	—	34.46	34.54

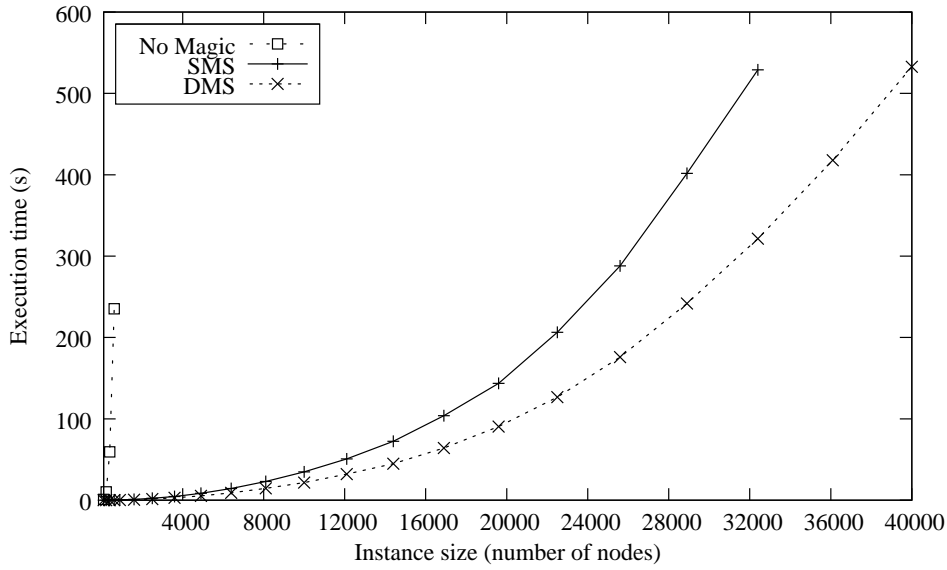


Figure 5.5: Average execution time for Simple Path

Table 5.2: Average execution time for Simple Path

Instance Size (number of nodes)	No Magic (seconds)	SMS (seconds)	DMS (seconds)
100	1.08	0.01	0.00
225	10.30	0.02	0.02
400	59.44	0.08	0.04
625	235.33	0.16	0.09
900	—	0.33	0.20
1 600	—	0.95	0.60
2 500	—	2.27	1.42
3 600	—	4.62	2.89
4 900	—	8.47	5.29
6 400	—	14.42	8.97
8 100	—	22.94	14.51
10 000	—	34.76	21.72
12 100	—	50.70	32.09
14 400	—	72.47	44.79
16 900	—	103.85	64.10
19 600	—	143.52	90.43
22 500	—	206.30	126.77
25 600	—	288.07	175.81
28 900	—	401.70	241.75
32 400	—	528.91	321.45
36 100	—	—	417.90
40 000	—	—	532.79

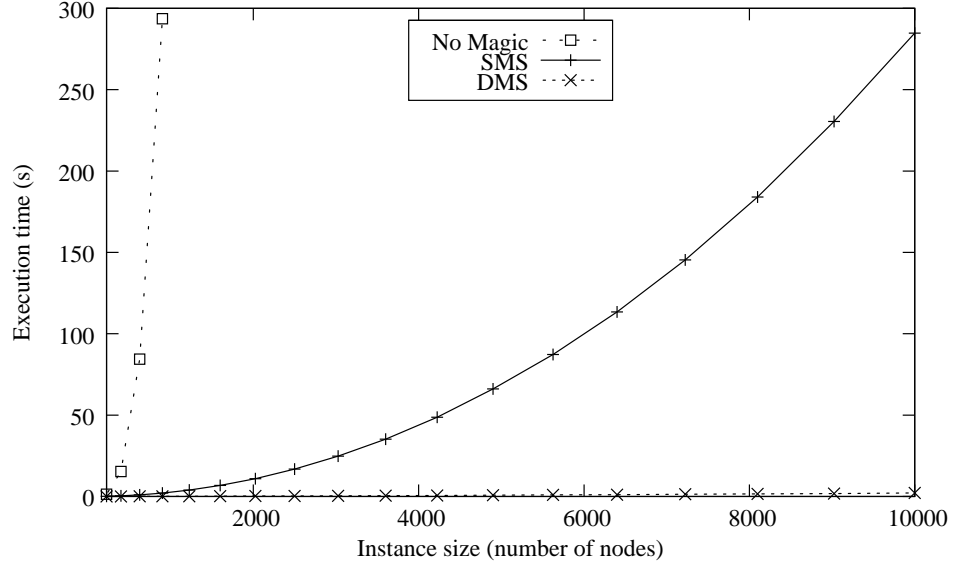


Figure 5.6: Average execution time for Related

Table 5.3: Average execution time for Related

Instance Size (number of nodes)	No Magic (seconds)	SMS (seconds)	DMS (seconds)
225	1.42	0.12	0.03
400	15.31	0.40	0.04
625	84.40	1.00	0.06
900	293.58	2.10	0.09
1 225	—	3.96	0.13
1 600	—	6.77	0.18
2 025	—	10.93	0.25
2 500	—	16.83	0.33
3 025	—	24.76	0.40
3 600	—	35.20	0.50
4 225	—	48.72	0.62
4 900	—	66.06	0.77
5 625	—	87.26	0.94
6 400	—	113.42	1.11
7 225	—	145.33	1.36
8 100	—	184.06	1.56
9 025	—	230.44	1.80
10 000	—	284.74	2.15

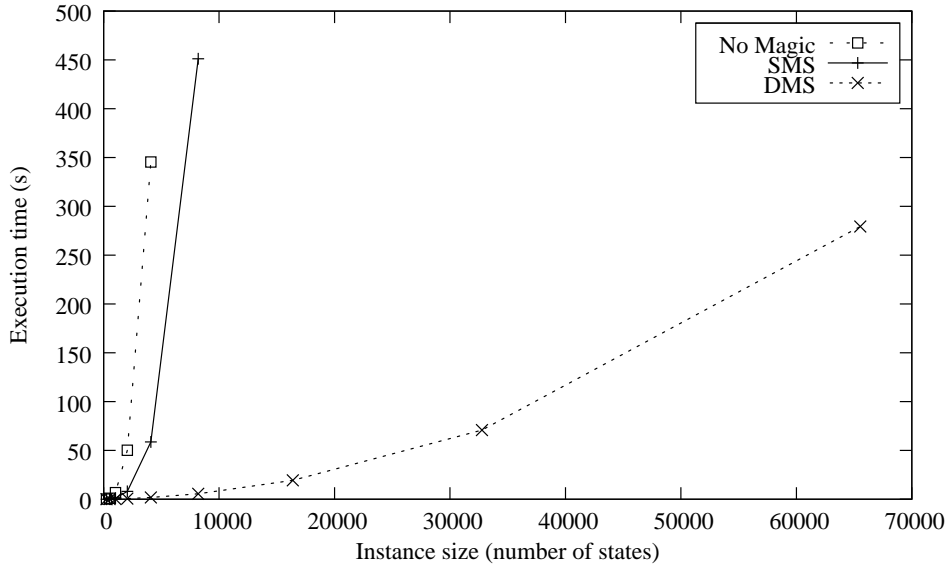


Figure 5.7: Average execution time for Conformant Plan Checking

Table 5.4: Average execution time for Conformant Plan Checking

Instance Size (number of states)	No Magic (seconds)	SMS (seconds)	DMS (seconds)
256	0.18	0.05	0.04
512	1.04	0.21	0.09
1 024	6.74	1.21	0.21
2 048	50.23	8.03	0.59
4 096	345.47	58.72	1.74
8 192	—	451.14	5.62
16 384	—	—	19.35
32 768	—	—	70.83
65 536	—	—	279.41

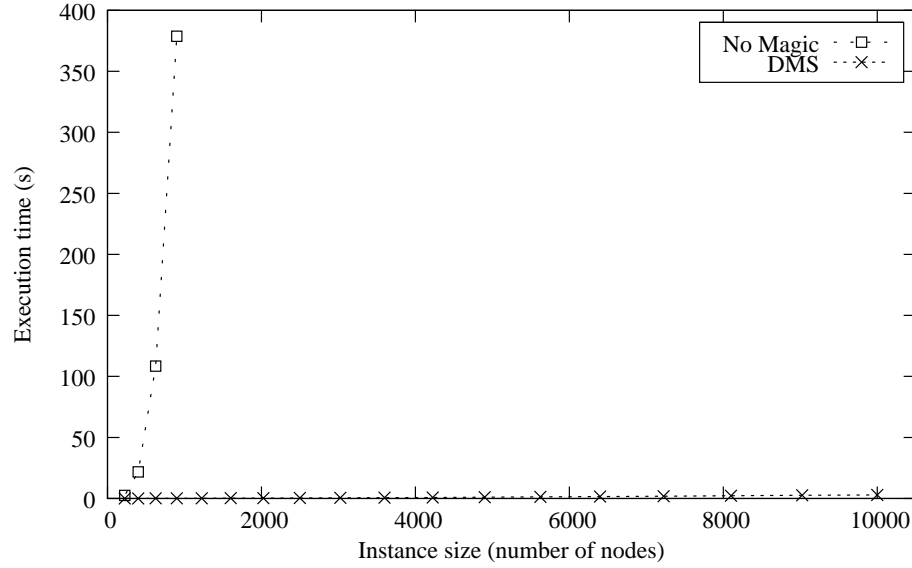


Figure 5.8: Average execution time for Related — Super-Coherent Encoding

Table 5.5: Average execution time for Related — Super-Coherent Encoding

Instance Size (number of nodes)	No Magic (seconds)	DMS (seconds)
225	2.48	0.03
400	21.72	0.05
625	108.50	0.08
900	378.81	0.11
1 225	—	0.17
1 600	—	0.27
2 025	—	0.31
2 500	—	0.42
3 025	—	0.54
3 600	—	0.68
4 225	—	0.85
4 900	—	1.04
5 625	—	1.28
6 400	—	1.51
7 225	—	1.81
8 100	—	2.18
9 025	—	2.57
10 000	—	2.95

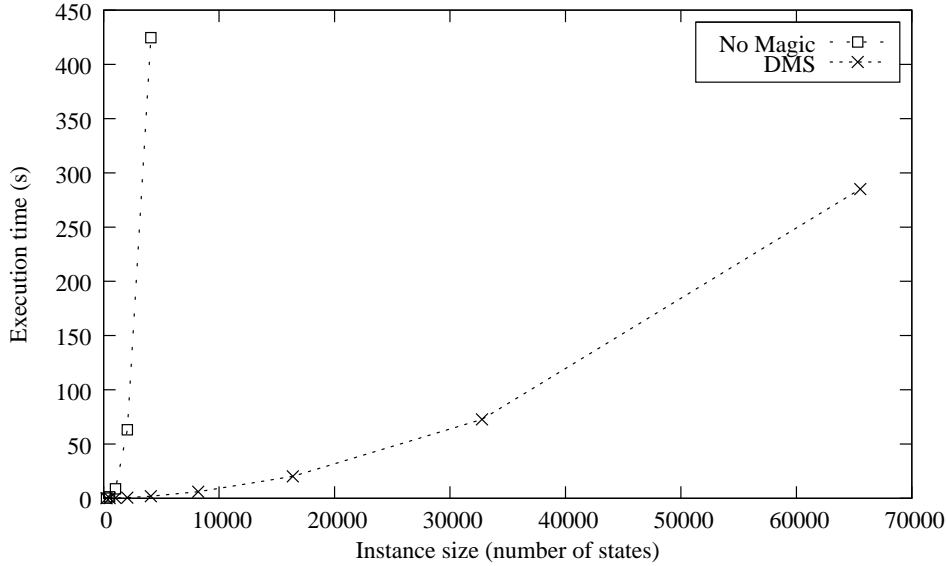


Figure 5.9: Average execution time for Conformant Plan Checking — Super-Coherent Encoding

Table 5.6: Average execution time for Conformant Plan Checking — Super-Coherent Encoding

Instance Size (number of nodes)	No Magic (seconds)	DMS (seconds)
256	0.21	0.04
512	1.31	0.10
1 024	8.74	0.25
2 048	63.15	0.66
4 096	424.62	1.91
8 192	—	6.04
16 384	—	20.17
32 768	—	72.77
65 536	—	284.98

Chapter 6

Application to Data Integration

In this chapter we give a brief account of a case study that evidences the impact of Dynamic Magic Sets when used on programs that realize data integration systems. The chapter is structured as follows. We first give an overview of data integration systems in Section 6.1. Then, in Section 6.2 we show how data integration systems can be implemented using Disjunctive Datalog programs. Finally, in Section 6.3 we discuss about the results of our experiments, aimed at assessing the impact of Dynamic Magic Sets on a data integration system involving real-world data.

6.1 Data Integration Systems

The main goal of data integration systems is to offer transparent access to heterogeneous sources by providing users with a *global schema*: The global schema can be queried without having to know from what sources the data come from. In fact, given a query over the global schema, the task of a data integration system is to identify and access the data sources which are relevant for answering the query, and to combine the obtained data. Data integration systems use a set of *mapping assertions* which specify the relationship between the data sources and the global schema. Following [45], we formalize a data integration system \mathcal{I} as a triple

$$\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle,$$

where:

1. \mathcal{G} is the *global (relational) schema*, that is, a pair

$$\langle \Psi, \Sigma \rangle,$$

where Ψ is a finite set of relation symbols, each one with an associated positive arity, and Σ is a finite set of *integrity constraints* (ICs) expressed on the symbols in Ψ . ICs are first-order assertions that are intended to be satisfied by database instances.

2. \mathcal{S} is the *source schema*, constituted by the schemas of the various sources that are part of the data integration system. In particular, we assume that \mathcal{S} is a relational schema of the form

$$\mathcal{S} = \langle \Psi', \emptyset \rangle,$$

which means that there are no integrity constraints on the sources. This assumption implies that data stored at the sources are locally consistent, which is common in data integration. Indeed, sources are in general external to the integration system, which is not in charge of analyzing or restoring their consistency.

3. \mathcal{M} is the *mapping* establishing the relationship between \mathcal{G} and \mathcal{S} . In our framework the mapping follows the *global-as-view* (GAV) approach, that is, each global relation is associated with a *view*—a conjunctive query over the sources. We assume that these associations are represented as Datalog rules.

Since integrated sources are originally autonomous, the main semantical issue in data integration systems is that data obtained by applying the mapping assertions may not satisfy the constraints of the global schema. This problem has lately received a lot of interest in the literature; see for instance [19, 16, 13, 12, 23, 22, 31, 32, 4, 21]. An approach for tackling this problem is based on the notion of *repair* for an inconsistent database, as introduced in [5]. Roughly speaking, a repair of an inconsistent database is a new database that satisfies the constraints in the schema, and minimally differs from the original one. Since an inconsistent database might possess multiple repairs, the standard approach in answering user queries is to return the answers that are true in every possible repair. These are called *consistent answers* in the literature.

6.2 Consistent Query Answering

There is an intuitive relationship between consistent query answering over data integration systems and queries over Datalog^{V,∇} programs. Indeed, for each data integration system

$$\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle,$$

it is possible to associate a Datalog^{V,∇} program $\Pi(\mathcal{I})$ such that each stable model of $\Pi(\mathcal{I})$ corresponds to a possible repair of \mathcal{I} . Hence, a query \mathcal{Q} over \mathcal{G} can be answered by performing cautious reasoning for \mathcal{Q} over $\Pi(\mathcal{I})$. In fact, various authors considered the idea of encoding the constraints of the global schema \mathcal{G} into various kinds of logic programs; see for instance [6, 36, 8, 19, 16, 21]. Some of these approaches use logic programs with unstratified negation [19], whereas other approaches take advantage of Disjunctive Datalog programs with stratified negation [14, 52].

In this context, Magic Set techniques can provide crucial optimization potential. Indeed, the benefits of Magic Sets for optimizing logic programs obtained by translating data integration systems have already been discussed in the literature. In particular, a translation of data integration systems to Datalog[∇] programs has been presented in [52], where a Magic Set technique (defined in [30]) has been profitably used for optimizing the query answering.

We now report an alternative transformation which produces Datalog^{V,¬s} programs, that is, no unstratified occurrences of negation are present in the programs obtained by this transformation. This rewriting has been devised and used within the EU project INFOMIX on data integration [46].

Consider a data integration system

$$\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle,$$

where

$$\mathcal{G} = \langle \Psi, \Sigma \rangle$$

and \mathcal{M} is specified as a Datalog program. Moreover, let \mathcal{D} be a database for \mathcal{G} , represented as a set of facts over the relational predicates in \mathcal{G} . We assume that each constraint over the global schema is either a *key* or an *exclusion dependency*. A set of attributes \bar{x} is a key for a relation \mathbf{r} if

$$(\mathbf{r}(\bar{x}, \bar{y}) \wedge \mathbf{r}(\bar{x}, \bar{z})) \rightarrow \bar{y} = \bar{z}, \quad \forall \{\mathbf{r}(\bar{x}, \bar{y}), \mathbf{r}(\bar{x}, \bar{z})\} \subseteq \mathcal{D}.$$

An exclusion dependency holds between a set of attributes \bar{x} of a relation \mathbf{r} and a set of attributes \bar{w} of a relation \mathbf{s} if

$$(\mathbf{r}(\bar{x}, \bar{y}) \wedge \mathbf{s}(\bar{w}, \bar{z})) \rightarrow \bar{x} \neq \bar{w}, \quad \forall \{\mathbf{r}(\bar{x}, \bar{y}), \mathbf{s}(\bar{w}, \bar{z})\} \subseteq \mathcal{D}.$$

The disjunctive rewriting of \mathcal{I} is the Datalog^{V,¬s} program

$$\Pi(\mathcal{I}) = \Pi_{\mathcal{M}} \cup \Pi_{\text{KD}} \cup \Pi_{\text{ED}} \cup \Pi_{\text{COLL}},$$

where:

1. For each Datalog rule in \mathcal{M} of the form

$$\mathbf{r}(\bar{u}) \text{ :- } \mathbf{s}_1(\bar{v}_1), \dots, \mathbf{s}_m(\bar{v}_m).$$

where \mathbf{r} is a relation in \mathcal{G} and $\mathbf{s}_1, \dots, \mathbf{s}_m$ are relations in \mathcal{S} , the program $\Pi_{\mathcal{M}}$ contains the following rule:

$$\mathbf{r}_{\mathcal{D}}(\bar{u}) \text{ :- } \mathbf{s}_1(\bar{v}_1), \dots, \mathbf{s}_m(\bar{v}_m).$$

2. For each relation \mathbf{r} in \mathcal{G} , and for each key defined over a set of its attributes \bar{x} , the program Π_{KD} contains the following rules:

$$\begin{aligned} \mathbf{r}_{\text{out}}(\bar{x}, \bar{y}) \vee \mathbf{r}_{\text{out}}(\bar{x}, \bar{z}) \text{ :- } & \mathbf{r}_{\mathcal{D}}(\bar{x}, \bar{y}), \mathbf{r}_{\mathcal{D}}(\bar{x}, \bar{z}), Y_1 \neq Z_1. \\ & \vdots \\ \mathbf{r}_{\text{out}}(\bar{x}, \bar{y}) \vee \mathbf{r}_{\text{out}}(\bar{x}, \bar{z}) \text{ :- } & \mathbf{r}_{\mathcal{D}}(\bar{x}, \bar{y}), \mathbf{r}_{\mathcal{D}}(\bar{x}, \bar{z}), Y_m \neq Z_m. \end{aligned}$$

where $\bar{y} = Y_1, \dots, Y_m$, and $\bar{z} = Z_1, \dots, Z_m$.

3. For each exclusion dependency between a set of attributes $\bar{x} = X_1, \dots, X_m$ of a relation \mathbf{r} and a set of attributes $\bar{w} = W_1, \dots, W_m$ of a relation \mathbf{s} , the

program Π_{ED} contains the following rule:

$$\mathbf{r}_{out}(\bar{x}, \bar{y}) \vee \mathbf{s}_{out}(\bar{w}, \bar{z}) :- \mathbf{r}_{\mathcal{D}}(\bar{x}, \bar{y}), \mathbf{s}_{\mathcal{D}}(\bar{w}, \bar{z}), X_1 = W_1, \dots, X_m = W_m.^1$$

4. For each relation \mathbf{r} in \mathcal{G} , the program Π_{COLL} contains the following rule:

$$\mathbf{r}(\bar{x}) :- \mathbf{r}_{\mathcal{D}}(\bar{x}), \text{not } \mathbf{r}_{out}(\bar{x}).$$

Given a data integration system \mathcal{I} and the associated rewriting $\Pi(\mathcal{I})$, it can be shown that:

For each query \mathcal{Q} over \mathcal{G} , and for each source database \mathcal{F} over \mathcal{S} , the consistent query answers to \mathcal{Q} precisely coincide with the set $Ans_c(\mathcal{Q}, \Pi(\mathcal{I}) \cup \mathcal{F})$ of cautious answers for the query \mathcal{Q} over the program $\Pi(\mathcal{I}) \cup \mathcal{F}$ (see Section 2.2.2 for a formal definition of Ans_c).

Below is an example of data integration system.

Example 6.2.1. Let `red_wine`(NAME, MAKER) and `white_wine`(NAME, MAKER) be two global relations obtained from a source schema containing the relations `italian_wine`(NAME, TYPE, MAKER) and `french_wine`(NAME, TYPE, MAKER). In particular, the mapping is given by the following Datalog program:

$$\begin{aligned} \text{red_wine}(N, M) & :- \text{italian_wine}(N, \text{"red"}, WM). \\ \text{red_wine}(N, M) & :- \text{french_wine}(N, \text{"red"}, WM). \\ \text{white_wine}(N, M) & :- \text{italian_wine}(N, \text{"white"}, WM). \\ \text{white_wine}(N, M) & :- \text{french_wine}(N, \text{"white"}, WM). \end{aligned}$$

In the global schema we assume the following integrity constraints:

- NAME is a key of `red_wine`;
- NAME is a key of `white_wine`;
- There is an exclusion dependency between the argument NAME of `red_wine` and the argument NAME of `white_wine` (red wines are not white wines and vice versa).

Hence, the disjunctive rewriting of this data integration system is the following Datalog ^{\vee, \neg_s} program:

$$\begin{aligned} \text{red_wine}_{\mathcal{D}}(N, M) & :- \text{italian_wine}(N, \text{"red"}, WM). \\ \text{red_wine}_{\mathcal{D}}(N, M) & :- \text{french_wine}(N, \text{"red"}, WM). \\ \text{white_wine}_{\mathcal{D}}(N, M) & :- \text{italian_wine}(N, \text{"white"}, WM). \\ \text{white_wine}_{\mathcal{D}}(N, M) & :- \text{french_wine}(N, \text{"white"}, WM). \\ \\ \text{red_wine}_{out}(N, M) \vee \text{red_wine}_{out}(N, M') & :- \text{red_wine}_{\mathcal{D}}(N, M), \\ & \quad \text{red_wine}_{\mathcal{D}}(N, M'), M \neq M'. \\ \text{white_wine}_{out}(N, M) \vee \text{white_wine}_{out}(N, M') & :- \text{white_wine}_{\mathcal{D}}(N, M), \\ & \quad \text{white_wine}_{\mathcal{D}}(N, M'), M \neq M'. \end{aligned}$$

¹Actually, the following equivalent rule is used: $\mathbf{r}_{out}(\bar{x}, \bar{y}) \vee \mathbf{s}_{out}(\bar{x}, \bar{z}) :- \mathbf{r}_{\mathcal{D}}(\bar{x}, \bar{y}), \mathbf{s}_{\mathcal{D}}(\bar{x}, \bar{z})$.

$$\begin{aligned} \text{red_wine}_{\text{out}}(N, M) \vee \text{white_wine}_{\text{out}}(N, M') & :- \text{red_wine}_{\mathcal{D}}(N, M), \\ & \quad \text{white_wine}_{\mathcal{D}}(N, M'). \\ \\ \text{red_wine}(N, M) & :- \text{red_wine}_{\mathcal{D}}(N, M), \text{ not } \text{red_wine}_{\text{out}}(N, M). \\ \text{white_wine}(N, M) & :- \text{white_wine}_{\mathcal{D}}(N, M), \text{ not } \text{white_wine}_{\text{out}}(N, M). \end{aligned}$$

□

6.3 Experimental Results

In the INFOMIX project, a case study has been conducted, developing a data integration system which models aspects of the information system of the University “La Sapienza” in Rome. In particular, the global schema consists of 14 global relations with 29 constraints, while the data sources include 29 relations of 3 legacy databases and 12 wrappers generating relational data from web pages. This amounts to more than 24MB of data regarding students, professors and exams in several faculties of the university. Personal data has been scrambled for privacy requirements (see, for instance, Query 3 in Figure 6.5).

On the INFOMIX schema we have tested five typical queries, each one with particular characteristics, which model different use cases. The full encodings of the tested queries are reported in Figures 6.1, 6.3, 6.5, 6.7, and 6.9, where we underlined source relations. Note that the INFOMIX system performs a preliminary selection of the source relations involved in an input query, so that only those table which are required for answering the query are actually processed by DLV. In the experiment, we considered datasets of increasing size obtained by adding records produced by putting a prefix to all the arguments of the original tuples. For each query and for each dataset considered, we measured the average execution time of the INFOMIX system with and without Dynamic Magic Sets.

The experiment has been performed by running the INFOMIX prototype system on a 3GHz Intel[®] Xeon[®] processor system with 4GB RAM under the Debian 4.0 operating system with a GNU/Linux 2.6.23 kernel. A DLV prototype supporting Dynamic Magic Sets and compiled using GCC 4.3.3 has been used as the computational core of the INFOMIX system. For each instance, we allowed a maximum running time of 600 seconds (10 minutes) and a maximum memory usage of 3GB.

The results of the experiment are represented in Figures 6.2, 6.4, 6.6, 6.8, and 6.10, and reported in Tables 6.1–6.5. In particular, the achieved results confirm that Dynamic Magic Sets considerably improve performances of Datalog systems. More specifically, on Queries 1–4 the response time of DLV scales much better with Dynamic Magic Sets. Indeed, the average execution time of DLV with DMS appears essentially linear, while without magic sets the performance of DLV is quite obviously non-linear. We also observe that there is basically no improvement on Query 5. Indeed, for this query all data seems to be relevant, which means that magic sets cannot have any positive effect. However, it is important to note that the application of DMS does not incur any significant overhead in this case.

```

courseD(X1, X2) :- esame(-, X1, X2, -).

courseD(X1, X2) :- esame_diploma(X1, X2).

exam_recordD(X1, X2, Z, W, X4, X5, Y) :- affidamenti_ing_informatica(X2, X3, Y),
    dati_esami(X1, -, X2, X5, X4, -, Y), dati_professori(X3, Z, W).

exam_recordout(X1, X2, X3, X4, Y5, Y6, Y7) v exam_recordout(X1, X2, X3, X4, Z5, Z6, Z7) :-
    exam_recordD(X1, X2, X3, X4, Y5, Y6, Y7), exam_recordD(X1, X2, X3, X4, Z5, Z6, Z7),
    Y5 ≠ Z5.

exam_recordout(X1, X2, X3, X4, Y5, Y6, Y7) v exam_recordout(X1, X2, X3, X4, Z5, Z6, Z7) :-
    exam_recordD(X1, X2, X3, X4, Y5, Y6, Y7), exam_recordD(X1, X2, X3, X4, Z5, Z6, Z7),
    Y6 ≠ Z6.

exam_recordout(X1, X2, X3, X4, Y5, Y6, Y7) v exam_recordout(X1, X2, X3, X4, Z5, Z6, Z7) :-
    exam_recordD(X1, X2, X3, X4, Y5, Y6, Y7), exam_recordD(X1, X2, X3, X4, Z5, Z6, Z7),
    Y7 ≠ Z7.

course(X1, X2) :- courseD(X1, X2), not courseout(X1, X2).

exam_record(X1, X2, X3, X4, X5, X6, X7) :- exam_recordD(X1, X2, X3, X4, X5, X6, X7),
    not exam_recordout(X1, X2, X3, X4, X5, X6, X7).

query1(CD) :- course(C, CD), exam_record("09089903", C, -, -, -, -).

query1(CD)?

```

Figure 6.1: Encoding of Query 1 — INFOMIX benchmark

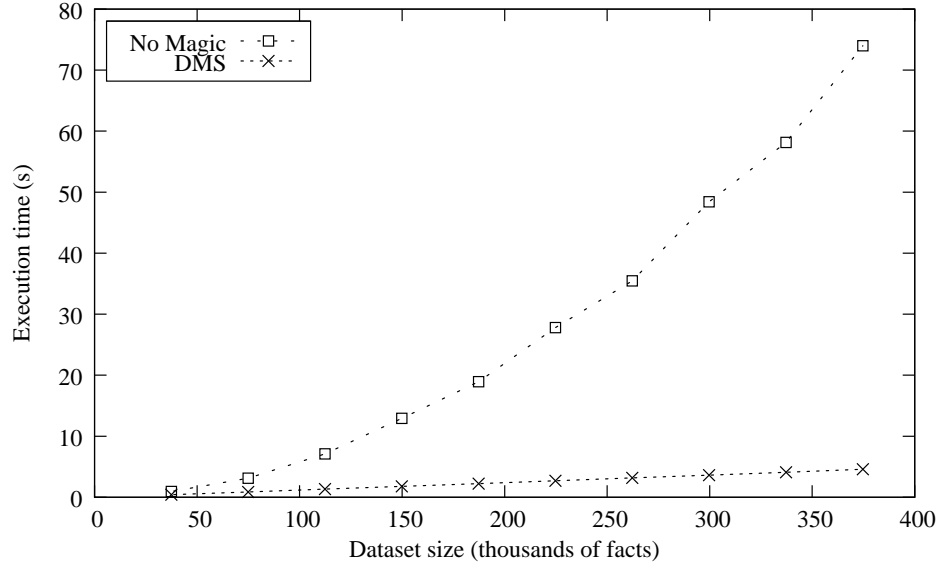


Figure 6.2: Average execution time for Query 1 — INFOMIX benchmark

Table 6.1: Average execution time for Query 1 — INFOMIX benchmark

Instance Size (number of facts)	No Magic (seconds)	DMS (seconds)
37 470	0.95	0.42
74 940	3.13	0.87
112 400	7.12	1.33
149 870	12.92	1.78
187 340	18.93	2.24
224 810	27.81	2.70
262 280	35.46	3.18
299 740	48.42	3.62
337 210	58.16	4.09
374 680	73.99	4.58

```

studentD(X1, X2, X3, X4, X5, X6, X7) :- diploma_maturita(Y, X7),
      studente(X1, X3, X2, -, -, -, -, -, -, X6, X5, -, -, X4, -, -, -, Y, -).

student(X1, X2, X3, X4, X5, X6, X7) :- studentD(X1, X2, X3, X4, X5, X6, X7),
      not studentout(X1, X2, X3, X4, X5, X6, X7).

query2(SFN, SLN, COR, ADD, TEL, HSS) :-
      student("09089903", SFN, SLN, COR, ADD, TEL, HSS).

query2(SFN, SLN, COR, ADD, TEL, HSS)?

```

Figure 6.3: Encoding of Query 2 — INFOMIX benchmark

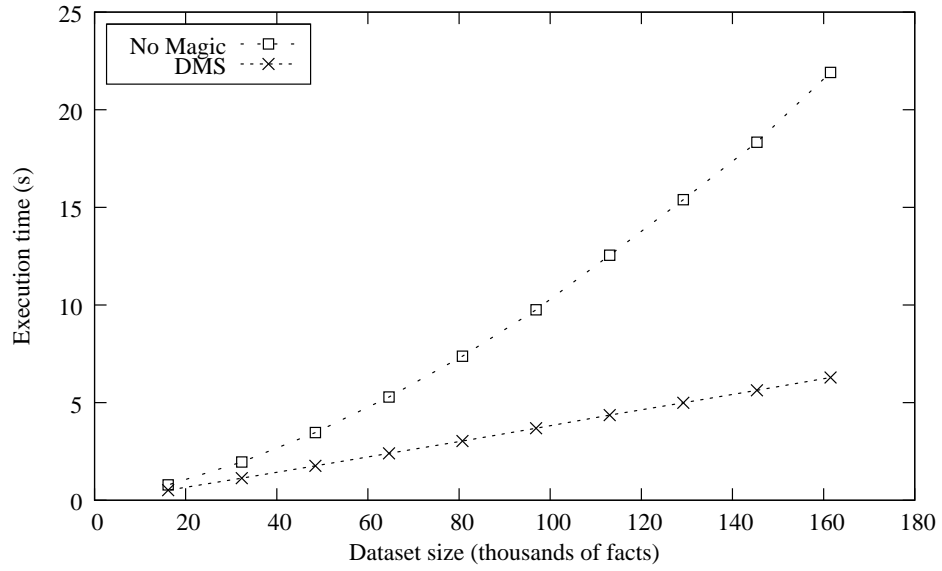


Figure 6.4: Average execution time for Query 2 — INFOMIX benchmark

Table 6.2: Average execution time for Query 2 — INFOMIX benchmark

Instance Size (number of facts)	No Magic (seconds)	DMS (seconds)
16 150	0.79	0.52
32 300	1.96	1.13
48 460	3.47	1.76
64 610	5.29	2.4
80 760	7.38	3.03
96 910	9.75	3.69
113 060	12.55	4.36
129 220	15.39	4.99
145 370	18.34	5.63
161 520	21.91	6.29

```

studentD(X1, X2, X3, X4, X5, X6, X7) :- diploma_maturita(Y, X7),
      studente(X1, X3, X2, -, -, -, -, -, -, -, X6, X5, -, -, X4, -, -, -, Y, -).

student_course_planD(X1, X2, X3, X4, X5) :- orientamento(Y1, X3),
      piano_studi(X1, X2, Y1, X4, Y2, -, -, -, -), stato(Y2, X5).

student(X1, X2, X3, X4, X5, X6, X7) :- studentD(X1, X2, X3, X4, X5, X6, X7),
      not studentout(X1, X2, X3, X4, X5, X6, X7).

student_course_plan(X1, X2, X3, X4, X5) :- student_course_planD(X1, X2, X3, X4, X5),
      not student_course_planout(X1, X2, X3, X4, X5).

query3(SID, SLN, R) :- student(SID, "ZNEPB", SLN, -, -, -, -),
      student_course_plan(-, SID, -, R, "APPROVATO SENZA MODIFICHE").

query3(SID, SLN, R)?

```

Figure 6.5: Encoding of Query 3 — INFOMIX benchmark

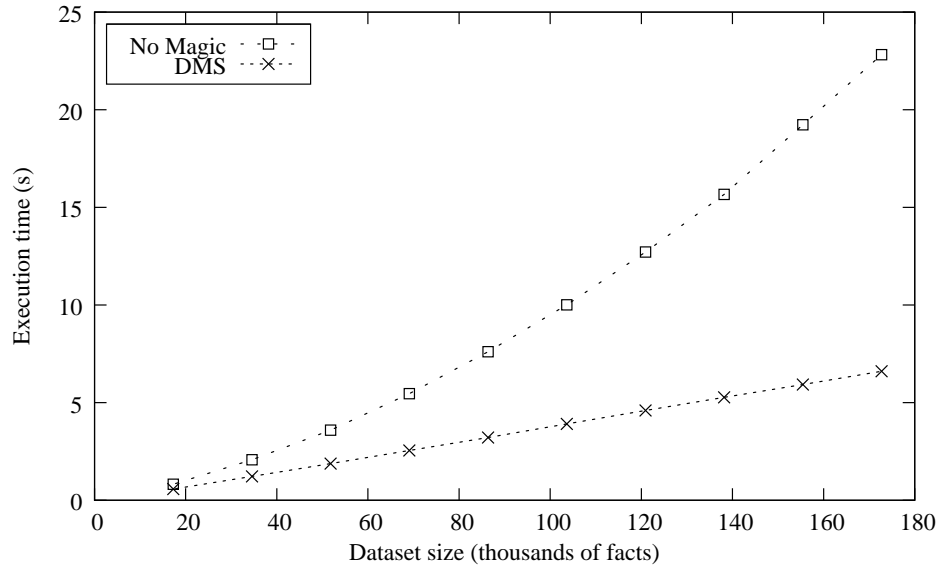


Figure 6.6: Average execution time for Query 3 — INFOMIX benchmark

Table 6.3: Average execution time for Query 3 — INFOMIX benchmark

Instance Size (number of facts)	No Magic (seconds)	DMS (seconds)
17 270	0.82	0.57
34 550	2.07	1.22
51 820	3.59	1.88
69 090	5.46	2.55
86 370	7.61	3.21
103 640	10.01	3.91
120 910	12.71	4.60
138 180	15.66	5.27
155 460	19.23	5.93
172 730	22.82	6.61

```

studentD(X1, X2, X3, X4, X5, X6, X7) :- diploma_maturita(Y, X7),
      studente(X1, X3, X2, -, -, -, -, -, -, X6, X5, -, -, X4, -, -, -, Y, -).

courseD(X1, X2) :- esame(-, X1, X2, -).

courseD(X1, X2) :- esame_diploma(X1, X2).

student_course_planD(X1, X2, X3, X4, X5) :- orientamento(Y1, X3),
      piano_studi(X1, X2, Y1, X4, Y2, -, -, -, -), stato(Y2, X5).

plan_dataD(X1, X2, X3) :- dati_piano_studi(X1, X2, -),
      esame_ingegneria(X2, Y3, Y2, -), tipo_esame(Y2, X3).

student(X1, X2, X3, X4, X5, X6, X7) :- studentD(X1, X2, X3, X4, X5, X6, X7),
      not studentout(X1, X2, X3, X4, X5, X6, X7).

student_course_plan(X1, X2, X3, X4, X5) :- student_course_planD(X1, X2, X3, X4, X5)
      not student_course_planout(X1, X2, X3, X4, X5).

plan_data(X1, X2, X3) :- plan_dataD(X1, X2, X3),
      not plan_dataout(X1, X2, X3).

course(X1, X2) :- courseD(X1, X2), not courseout(X1, X2).

query4(F, S) :- course(CID, "RETILOGICHE"), plan_data(SCID, CID, -),
      student(SID, F, S, "ROMA", -, -, -), student_course_plan(SCID, SID, -, -, -).

query4(F, S)?

```

Figure 6.7: Encoding of Query 4 — INFOMIX benchmark

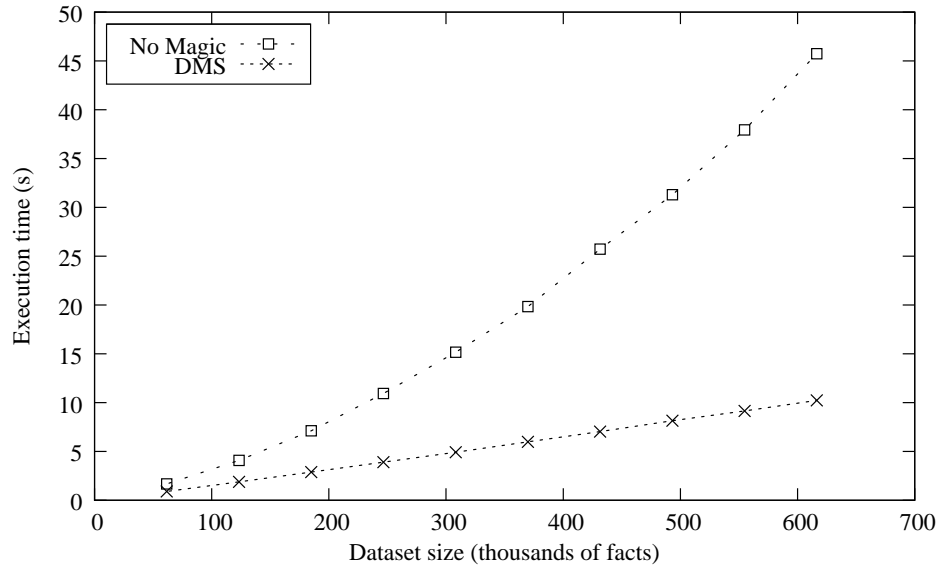


Figure 6.8: Average execution time for Query 4 — INFOMIX benchmark

Table 6.4: Average execution time for Query 4 — INFOMIX benchmark

Instance Size (number of facts)	No Magic (seconds)	DMS (seconds)
61 650	1.67	0.90
123 290	4.09	1.88
184 940	7.12	2.88
246 580	10.93	3.90
308 230	15.16	4.92
369 870	19.83	5.99
431 520	25.72	7.03
493 160	31.30	8.14
554 810	37.93	9.15
616 450	45.73	10.23

```

courseD(X1, X2) :- esame(-, X1, X2, -).

courseD(X1, X2) :- esame_diploma(X1, X2).

student_course_planD(X1, X2, X3, X4, X5) :- orientamento(Y1, X3),
    piano_studi(X1, X2, Y1, X4, Y2, -, -, -, -), stato(Y2, X5).

plan_dataD(X1, X2, X3) :- dati_piano_studi(X1, X2, -),
    esame_ingegneria(X2, Y3, Y2, -), tipo_esame(Y2, X3).

student_course_plan(X1, X2, X3, X4, X5) :- student_course_planD(X1, X2, X3, X4, X5),
    not student_course_planout(X1, X2, X3, X4, X5).

plan_data(X1, X2, X3) :- plan_dataD(X1, X2, X3),
    not plan_dataout(X1, X2, X3).

course(X1, X2) :- courseD(X1, X2), not courseout(X1, X2).

query5(D) :- course(E, D), plan_data(C, E, -),
    student_course_plan(C, "09089903", -, -, -).

query5(D)?

```

Figure 6.9: Encoding of Query 5 — INFOMIX benchmark

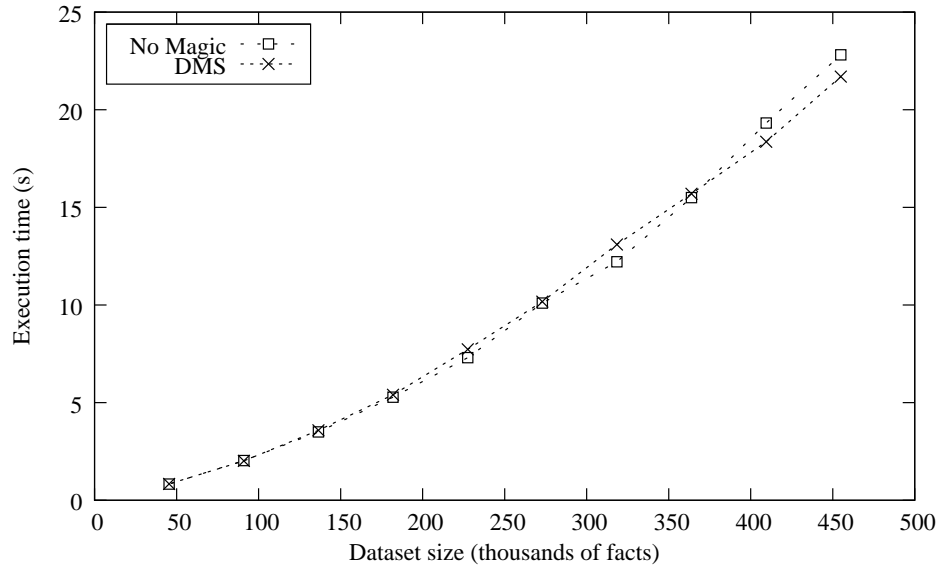


Figure 6.10: Average execution time for Query 5 — INFOMIX benchmark

Table 6.5: Average execution time for Query 5 — INFOMIX benchmark

Instance Size (number of facts)	No Magic (seconds)	DMS (seconds)
45 490	0.83	0.84
90 990	2.03	2.02
136 480	3.50	3.58
181 970	5.28	5.40
227 470	7.30	7.73
272 960	10.10	10.17
318 450	12.21	13.10
363 940	15.50	15.70
409 440	19.32	18.36
454 930	22.81	21.70

Chapter 7

Related Work

In this chapter we first discuss about the main body of work which is related to Dynamic Magic Sets, the technique for query optimization developed in this thesis. In particular, we discuss Magic Set techniques for Datalog and its extensions. The discussion is structured in sections grouping techniques which cover the same language. After that, we discuss some applications for which Dynamic Magic Sets have already been exploited. All of these applications refer to the preliminary work published in [25], in which the technique is referred to as Disjunctive Magic Sets.

Magic Sets for Datalog

In order to optimize query evaluation in bottom-up systems, like deductive database systems, many works have proposed the simulation of top-down strategies by means of suitable transformations. Among these transformations, Magic Sets for Datalog queries, here referred to as Classic Magic Sets, are one of the best known logical optimization techniques for database systems. The method, first developed in [7], has been analyzed and refined by many authors; see, for instance, [10, 65, 57, 63]. These works form the foundations of Dynamic Magic Sets.

Magic Sets for Datalog^{¬s}

Many authors have addressed the issue of extending Classic Magic Sets in order to deal with Datalog queries involving stratified negation. The main problem related to the extension of the technique to Datalog^{¬s} programs is how to assign a semantics to the rewritten programs. Indeed, while Datalog^{¬s} programs have a natural and accepted semantics, namely the perfect model semantics [3, 66], the application of Classic Magic Sets can introduce unstratified negation in the rewritten programs. Solutions have been presented in [41, 61, 42, 11]. In particular, in [41, 61] rewritten programs have been evaluated according to the well-founded semantics, a three-valued semantics for Datalog[¬] programs which is two-valued for stratified programs and for the associated rewritten programs. In [42, 11], instead, ad-hoc semantics have been defined. All of these methods are considerably different from Dynamic Magic Sets. Indeed, they take advantage

of the uniqueness of the intended model, which is not a property of Disjunctive Datalog programs.

Magic Sets for Datalog[¬]

Extending Classic Magic Sets to Datalog[¬] programs must face two major difficulties. First, for a Datalog[¬] program uniqueness of the intended model is no longer guaranteed, thus query answering in this setting involves a set of stable models in general. The second difficulty is that parts of a Datalog[¬] program may act as constraints, thus impeding a relevant interpretation to be a stable model. In [30] a Magic Set method for Datalog[¬] programs has been defined and proved to be correct for coherent programs. This method takes special precautions for relevant parts of the program that act as constraints, called *dangerous rules* in [30]. It can be observed that dangerous rules cannot occur in Datalog^{∨,¬s}, which allows for the simpler Dynamic Magic Sets to work correctly for this class of programs.

Magic Sets for Disjunctive Datalog

The first extension of Classic Magic Sets to Disjunctive Datalog is due to [37, 38], where Static Magic Sets have been presented and proved to be correct for Datalog[∨] programs. The main drawback of Static Magic Sets is the introduction of *collecting* predicates. Indeed, magic and collecting predicates of Static Magic Sets have deterministic definitions. As a consequence, their extension can be completely computed during program instantiation, which means that no further optimization potential can be provided to the subsequent stable model search. Moreover, while the correctness of Dynamic Magic Sets has been formally established for Datalog^{∨,¬s} programs in general (and also for Datalog^{∨,¬}_{SC} programs), the applicability of Static Magic Sets to Datalog^{∨,¬s} programs has not been considered in depth in [37, 38].

Applications

Magic Sets have been applied in many contexts. In particular, [14, 52, 56, 39] have profitably used the optimization provided by Dynamic Magic Sets. In particular, in [14, 52] a data integration system has been presented. The system is based on Disjunctive Datalog and takes advantage of Dynamic Magic Sets for fast query answering. In [56, 39], instead, an algorithm for answering query over description logic knowledge bases has been presented. More specifically, the algorithm reduces a *SHIQ* knowledge base to a Disjunctive Datalog program, so that Dynamic Magic Sets can be used for query optimization.

Chapter 8

Conclusion

Magic Sets have already been assessed as an effective method for query optimization over positive recursive Datalog. Many works have analyzed the technique in this context [7, 10, 65, 57, 63], which has been referred to as Classic Magic Sets in this thesis. Other works have presented extensions of the technique to Datalog[∃] [3, 66, 41, 61, 42, 11] and Datalog[∃] [30]. All these works share the key concept of *relevance*: the relevant atoms for a given query are the atoms which can be reached during a top-down evaluation of the query, and relevant atoms are sufficient for answering the query.¹ Relevant atoms are identified by magic atoms based on suitable *sideways information passing strategies* (SIPS). Magic atoms are then used for limiting the instantiation of rule bodies during a bottom-up evaluation of the rewritten program.

The first extension of Classic Magic Sets to Disjunctive Datalog, referred to as Static Magic Sets in this thesis, has been presented in [37, 38]. The main contribution of Static Magic Sets is the extension of the notion of SIPS to Disjunctive Datalog rules. Indeed, the fact that binding information in this setting has to be passed also from head atoms to head atoms has been first observed in [37]. However, the main drawback of Static Magic Sets is that magic atoms have deterministic definitions. Essentially, the same concept of relevance adopted by Classic Magic Sets is used to identify the atoms which are required for answering a given query. It is a drawback because the extension of these magic atoms can be completely determined during program instantiation, which means that no further optimization can be achieved during the subsequent stable model search.

Dynamic Magic Sets, the technique developed in the work presented in this thesis, has been designed for overcoming the drawbacks of Static Magic Sets. The same notion of SIPS used by Static Magic Sets is adopted, but nondeterministic definitions of magic atoms are possibly introduced. Nondeterministic definitions take into account partial knowledge inferred during stable model search, which means that a new notion of *conditional relevance* is introduced by Dynamic Magic Sets: the relevant atoms for a given query are the atoms which can be reached during a top-down evaluation of the query according to all previously made assumptions. It is the notion of conditional relevance that allows Dynamic Magic Sets to effectively disable parts of the program during

¹The technique of [30] for Datalog[∃] programs must also take into account atoms occurring in odd cycles.

stable model search.

A strong relationship between magic sets and unfounded sets has been highlighted: Relevant atoms are either true or belong to some unfounded set. This relationship has been used for proving the correctness of Dynamic Magic Sets for the class of $\text{Datalog}^{\vee, \neg s}$ programs. Moreover, the correctness of Dynamic Magic Sets has been enlarged to the class of super-coherent programs ($\text{Datalog}_{SC}^{\vee, \neg}$), which include all odd-cycle-free programs.

Dynamic Magic Sets have also been used for proving decidability of query answering over stratified finitely recursive programs, a class of logic programs with uninterpreted function symbols denoted by $\text{Datalog}_{FR}^{\vee, \neg s}$. In particular, finitely recursive programs are mapped to finitely ground programs, for which decidability was already established in the literature. The mapping is given by Dynamic Magic Sets and opens the possibility of using DLV for experimenting with $\text{Datalog}_{FR}^{\vee, \neg s}$ programs.² The expressive power of $\text{Datalog}_{FR}^{\vee, \neg s}$ programs has also been analyzed, showing that all computable sets and functions can be expressed by programs in this class.

Dynamic Magic Sets have been empirically assessed by testing the implemented DLV prototype in many benchmarks. Among them are benchmarks from the literature which have already been used for assessing Static Magic Sets. The results show that Dynamic Magic Sets outperform Static Magic Sets in general, in some cases also by an exponential factor. Other benchmarks have been taken from a real application scenario, the INFOMIX project for consistent query answering over heterogeneous data sources. Also in these benchmarks the benefits of Dynamic Magic Sets are tangible.

To sum up, Dynamic Magic Sets have been proved correct and effective for $\text{Datalog}^{\vee, \neg s}$ and $\text{Datalog}_{SC}^{\vee, \neg}$ programs, and have been used for mapping $\text{Datalog}_{FR}^{\vee, \neg s}$ programs to finitely ground programs. Whether Dynamic Magic Sets may be applied to broader classes of programs, including syntactically extended languages like Disjunctive Datalog with recursive aggregates, constitutes an interesting direction for further research.

²Finitely ground programs are supported by DLV-complex [20], a prototype system extending DLV. They are also supported by DLV since version 2010-10-14.

Bibliography

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. Technical Report 09/2009, Dipartimento di Matematica, Università della Calabria, Italy, 2009. <http://www.wfaber.com/research/papers/TRMAT092009.pdf>.
- [3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Minker [55], pages 89–148.
- [4] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Scalar aggregation in fd-inconsistent databases. In *International Conference on Database Theory (ICDT-2001)*, pages 39–53. Springer Verlag, 2001.
- [5] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proc. of the 18th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'99)*, pages 68–79, 1999.
- [6] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Proc. of the 4th Int. Conf. on Flexible Query Answering Systems (FQAS 2000)*, pages 27–41. Springer-Verlag, 2000.
- [7] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Int. Symposium on Principles of Database Systems*, pages 1–16, 1986.
- [8] Pablo Barceló and Leopoldo Bertossi. Repairing databases with annotated predicate logic. In *Proc. the 10th Int. Workshop on Non-Monotonic Reasoning (NMR 2002)*, pages 160–170, 2002.
- [9] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On Finitely Recursive Programs. *Theory and Practice of Logic Programming*, 9(2):213–238, 2009.
- [10] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(1–4):255–259, 1991.
- [11] Andreas Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *PODS '03: Proceedings of the twenty-second*

- ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–110, New York, NY, USA, 2003. ACM.
- [12] Leopoldo Bertossi and Jan Chomicki. Query answering in inconsistent databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*, chapter 2, pages 43–83. Springer-Verlag, 2003.
- [13] Leopoldo Bertossi, Jan Chomicki, Alvaro Cortes, and Claudio Gutierrez. Consistent answers from integrated data sources. In *Proc. of the 6th Int. Conf. on Flexible Query Answering Systems (FQAS 2002)*, pages 71–85, 2002.
- [14] Leopoldo E. Bertossi and Loreto Bravo. Consistent query answers in virtual data integration systems. In *Inconsistency Tolerance*, volume 3300 of *LNCS*, pages 42–83. Springer, 2005.
- [15] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.
- [16] Loreto Bravo and Leopoldo Bertossi. Logic programming for consistently querying data integration systems. In *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, pages 10–15, 2003.
- [17] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.
- [18] Marco Cadoli and Marco Schaerf. A survey on complexity results for non-monotonic logics. *Journal of Logic Programming*, 17:127–160, 1993.
- [19] Andrea Cali, Domenico Lembo, and Riccardo Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, pages 16–21, 2003.
- [20] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable Functions in ASP: Theory and Implementation. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 407–424, Udine, Italy, December 2008. Springer.
- [21] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, 2005.
- [22] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Computing consistent query answers using conflict hypergraphs. In *Proc. 13th ACM Conference on Information and Knowledge Management (CIKM-2004)*, pages 417–426. ACM Press, 2004.
- [23] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Hippo: A system for computing consistent answers to a class of sql queries. In *9th International Conference on Extending Database Technology (EDBT-2004)*, pages 841–844. Springer Verlag, 2004.

- [24] Keith L. Clark. Negation as Failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
- [25] Chiara Cumbo, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Enhancing the magic-set method for disjunctive datalog programs. In *Proceedings of the 20th International Conference on Logic Programming – ICLP’04*, volume 3132 of *Lecture Notes in Computer Science*, pages 371–385, 2004.
- [26] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-Driven Disjunctive Answer Set Solving. In Gerhard Brewka and Jérôme Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 422–432, Sydney, Australia, 2008. AAAI Press.
- [27] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.
- [28] Thomas Eiter and Mantas Simkus. Bidirectional answer set programs with function symbols. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*. AAAI Press/IJCAI, 2009.
- [29] Wolfgang Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.
- [30] Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences*, 73(4):584–609, 2007.
- [31] Ariel Fuxman, Elham Fazli, and Renée J. Miller. Conquer: Efficient management of inconsistent databases. In *SIGMOD Conference*, 2005.
- [32] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. In Thomas Eiter and Leonid Libkin, editors, *Proceedings of the 10th International Conference on Database Theory (ICDT 2005)*, number 3363 in LNCS, pages 337–351. Springer, 2005.
- [33] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR’07*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271, Tempe, Arizona, May 2007. Springer Verlag.
- [34] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

- [35] Robert P. Goldman and Mark S. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings AIPS-96*, pages 110–117. AAAI Press, 1996.
- [36] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *Proc. of the 17th Int. Conf. on Logic Programming (ICLP'01)*, volume 2237 of *Lecture Notes in AI (LNAI)*, pages 348–364. Springer-Verlag, 2001.
- [37] Sergio Greco. Optimization of Disjunction Queries. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 441–455, Las Cruces, New Mexico, USA, November 1999. The MIT Press.
- [38] Sergio Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):368–385, March/April 2003.
- [39] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning in description logics by a reduction to disjunctive datalog. *Journal of Automated Reasoning*, 39(3):351–384, 2007.
- [40] Tomi Janhunnen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Partiality and Disjunctions in Stable Model Semantics. In Anthony G. Cohn, Fausto Giunchiglia, and Bart 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.
- [41] David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146:145–184, July 1995.
- [42] Jean-Marc Kerisit and Jean-Marc Pugin. Efficient query answering on stratified databases. In *FGCS*, pages 719–726, 1988.
- [43] Robert A. Kowalski. Predicate Logic as Programming Language. In *IFIP Congress*, pages 569–574, 1974.
- [44] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.
- [45] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002)*, pages 233–246, 2002.
- [46] Nicola Leone, Georg Gottlob, Riccardo Rosati, Thomas Eiter, Wolfgang Faber, Michael Fink, Gianluigi Greco, Giovambattista Ianni, Edyta Kalka, Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Marco Ruzzi, Witold Staniszkis, and Giorgio Terracina. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, June 2005. ACM Press.

- [47] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.
- [48] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.
- [49] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.
- [50] Yuliya Lierler and Vladimir Lifschitz. One More Decidable Class of Finitely Ground Programs. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *Lecture Notes in Computer Science*, pages 489–493, Pasadena, CA, USA, July 2009. Springer.
- [51] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.
- [52] Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Querying inconsistent databases using answer set programs. In *SUM 2007*, pages 74–88, 2007.
- [53] John McCarthy. Circumscription — a Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1–2):27–39, 1980.
- [54] Jack Minker. On Indefinite Data Bases and the Closed World Assumption. In Donald W. Loveland, editor, *Proceedings 6th Conference on Automated Deduction (CADE '82)*, volume 138 of *Lecture Notes in Computer Science*, pages 292–308, New York, 1982. Springer.
- [55] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.
- [56] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Fakultät für Wirtschaftswissenschaften, Universität Fridericiana zu Karlsruhe, 2006.
- [57] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 247–258, 1990.
- [58] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1995.
- [59] Raymond Reiter. On Closed World Data Bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

- [60] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.
- [61] Kenneth A. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of the ACM*, 41(6):1216–1266, 1994.
- [62] Mantas Simkus and Thomas Eiter. FDNC: Decidable Non-monotonic Disjunctive Logic Programs with Function Symbols. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR2007)*, volume 4790 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2007.
- [63] Peter J. Stuckey and S. Sudarshan. Compiling query constraints. In *Proceedings of the Thirteenth Symposium on Principles of Database Systems (PODS'94)*, pages 56–67. ACM Press, May 1994.
- [64] Sten-Åke Tärnlund. Horn clause computability. *BIT Numerical Mathematics*, 17(2):215–226, June 1977.
- [65] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.
- [66] Allen Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In Minker [55], pages 1149–1176.
- [67] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proceedings of the Seventh Symposium on Principles of Database Systems (PODS'88)*, pages 221–230, 1988.