Project Number:	IST-2001-33570
Project Acronym:	INFOMIX
Project Full Name:	Infomix: Boosting the Information Integration
Deliverable Number:	D4.3
Title:	**Optimization techniques for the internal integration formalism.**
Authors:	Nicola Leone, Pasquale Rullo, Giovambattista Ianni
Workpackage:	WP4
Document Type:	Deliverable
Distribution:	
Status:	Final
Document file:	WP4_T1_D4.3_0102.ps
Version:	1.2
Number of pages:	28
Due Date:	September 30, 2003
Delivery Date:	October 6, 2003
Partners contributed:	UNICAL, TUWIEN, UNIROME1, RODAN
Partners owning:	UNICAL, TUWIEN, UNIROME1, RODAN
Man Months:	11.5

Short Description:

The report presents the optimization methods and some smart algorithms in order to efficiently support the highly expressive computational-logic formalism, namely DLP$^{\mathcal{A}}$, which is needed in INFOMIX as the internal integration formalism for modelling the whole process of query answering in data integration.
Specifically, we try to overcome the deficiencies of scalability with current implementations of stable model engines, by designing a strategy for query answering which avoid useless computations, by modifying the original program with certain rules, acting as filters for the relevant information needed for the processing.
The technique is the most general appeared so far in the literature, since it applies to any disjunctive datalog program, even enriched with constraints, general unstratified negation and aggregate functions, thus, fully supporting all the features of DLP$^{\mathcal{A}}$.

Document Change Record		
Version	**Date**	**Reason for Change**
v.1.0	September 20, 2003	First draft
v.1.1	September 30, 2003	Review from the project parterns
v.1.2	October 3, 2003	Final version

Contents

1 Introduction **4**
1.1 Contributions and Organization 5

2 Preliminaries on DLP Programs **6**
2.1 Syntax 6
2.2 Semantics and Query Equivalent DLP Programs 6

3 Efficient Evaluation of Logic Programs **8**
3.1 Magic-Set Method for Datalog Queries 8
3.2 Related Works on Extensions of the Magic-Set Method 10

4 Magic-Set Method for Positive DLP Programs **11**
4.1 Adornment Step: SIPS for Disjunctive Programs 13
4.2 Generation and Modification Steps 16
4.3 Postprocessing 16

5 Magic-Set Method for Positive DLP Programs with Aggregates **17**
5.1 Syntax and Semantics of Aggregates in a Nutshell 17
5.2 Modification of the SIPS 18

6 Extension to Arbitrary DLP$^{\mathcal{A}}$ Programs **19**
6.1 Program with Constraints 20
6.2 General Unstratified Programs: Dangerous Rules 22
6.3 Query Equivalence Results 24

7 Conclusions **26**

1 Introduction

The definition of the Internal Integration Formalism (IIF) adopted in the INFOMIX system has been already specified in the report D4.1, where we evidenced that both the project proposal in itself and the specifications given in all the previous reports (especially the specifications in D2.2 for the run-time system architecture) made it clear that this formalism would have been based on a suitable extension of logic programming.

Even though disjunctive logic programming (DLP) fits all the requirements for the project, we realized that the expressiveness of standard DLP is not sufficient to express queries requiring to "aggregate data", which often occur in real application settings in data integration. Thus, the IIF has been based on a suitable extension of standard DLP by aggregate functions ($\text{DLP}^{\mathcal{A}}$) in order to support some forms of aggregate queries in the user language. Under a computational point of view, the resulting formalism turned out to be equivalent to standard DLP, since 'brave reasoning' for ground programs is Σ_2^P-complete whereas 'cautious reasoning' for ground programs is Π_2^P-complete.

While thinking at the imminent work to be done for the implementation of the INFOMIX system, one could be disappointed by these complexities results, for they showing the reasoning on $\text{DLP}^{\mathcal{A}}$ programs to be computationally expensive. Nonetheless these results are not surprising at all, since many recent studies (see, e.g., [1, 12, 3, 7, 6]) had already evidenced that the intrinsic complexity of query answering in data integration systems (even though in different settings) is often complete for the class co-NP, or even for the class Π_2^P.

Consequently, as the integration problem we are dealing with has been proved to be "intrinsically difficult", our efforts must be spend in designing specialized algorithms for speeding up the computation of any $\text{DLP}^{\mathcal{A}}$ program in the INFOMIX integration engine and for ensuring the scalability of our approach. We point out that these optimizations are crucial for an effective and efficient implementation of the INFOMIX system, since the major drawback with current implementations of stable model engines, such as DLV or Smodels, is that the evaluation of queries over large data sets quickly becomes infeasible because of lacking scalability [6].

In this report we face this drawback and we observe that a promising line of research towards the design of a scalable stable model engine which efficiently supports the highly expressive computational-logic formalism adopted in INFOMIX is the extension of deductive database techniques, like the binding propagation techniques [20], to the languages DLP and $\text{DLP}^{\mathcal{A}}$. Intuitively, the goal of the application of binding propagation techniques to disjunctive Datalog queries is to exploit the constants appearing in the query to reduce the size of the instantiation by eliminating "a priori" a number of ground instances of the rules which cannot contribute to the derivation of the query goal.

In [13], the author proposes an optimization technique for the evaluation of bound disjunctive datalog queries, which consists of a suitable adaptation of the Magic-Set method [20, 2, 4] originally defined for (nondisjunctive) datalog queries only. The Magic-Set method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of certain rules, acting as filters for the relevant information needed for the query processing. It is the most well-known technique for the optimization of positive recursive programs due to its efficiency and its generality, even though other

focused methods such as the supplementary magic set and other special techniques for linear and chain queries have been proposed as well (see, e.g., [15, 20, 19]). Moreover, [16, 17] have also shown that the Magic-Set method performs well in the optimization of non-recursive programs, too. The research on variants of the Magic-Set method is still going on: In [14] an elaborated technique for disjunctive programs based on [13] is described, and in [5] a technique for the more restricted class of stratified programs is given.

1.1 Contributions and Organization

In this report we preliminarily provide fresh and refined ideas (w.r.t. [14]) for extending the Magic-Set method to disjunctive Datalog queries; in particular, the rewriting in [13, 14] bloat the programs with additional predicates and rules, which are avoided to great extent by the techniques presented in this paper. Specifically, we present an extension of the Magic-Set method to disjunctive Datalog queries, referred to as Disjunctive Magic-Set (DMS) which is similar in spirit to the Auxiliary Predicates Method (APM) of [13, 14], but offers a novel integrated algorithm (instead of a phased one) which avoids the generation of many auxiliary rules needed in APM.

Subsequently, we show how the Magic-Set method can be suitably extended for dealing with general unstratified programs and with aggregate functions as well. This further extension is referred to as Extended Magic-Set (EMS), and to the best of our knowledge represents the most general approach for the optimization of disjunctive programs presented so far in the literature.

Moreover, while considering general unstratified programs we also present a more focused strategy for the optimization of programs enriched with constraints, i.e., a disciplined and expressive form of unstratified negation. The motivation beyond such further optimization lies in the fact that constraints represent a natural way to extend database semantics: They define properties which are supposed to be satisfied by all instances over a given database schema explicitly, and, hence, they are very often used for declaratively expressing hard problems, such as NP problems and problems in the second level of the polynomial hierarchy (Σ_p^2 and Π_p^2 problems), by exploiting the *guess-and-check* technique, where the *guess* part is expressed by means of disjunctive rules and the *check* part is expressed by means of constraints.

Interestingly, we observed that any technique implementing a focused top-down strategy for query answering in the presence of unstratified negation produces a query that, generally, is not equivalent to the original one. The reason is that constraints express conditions that must hold for every ground instance of the program also including the atoms which are not relevant for the query. Moreover, despite the case of disjunction only, we observed that different properties hold by swapping from brave to cautious reasoning.

We conclude by mentioning that, after this report, our attention within the INFOMIX project will be devoted to the implementation of all the algorithms proposed so far, and in performing extensive experiments using some benchmarks reported in the literature in order to asses the usefulness of the technique and in order to evaluate the impact of the proposed extensions of the Magic-Set method w.r.t. the traditional DLV engine.

The rest of the present document is structured as follows. In the next section, we

briefly recall the syntax and the semantics of DLP. Then, in Section 3 we overview the original proposal of Magic-Set method for positive Datalog programs and we analyze the extensions proposed for dealing with disjunctive rules. In Section 4 we present the method DMS, by formally providing all the algorithms needed for its actual implementation, whereas in Section 5 we further extend the range of applicability of the technique by considering DLP$^{\mathcal{A}}$ programs. Section 6 deals with the design of the algorithms for propagating the binding into disjunctive Datalog programs with unstratified negation, and finally, in Section 7 we draw our conclusions.

2 Preliminaries on DLP Programs

In this report we are interested in studying some strategies for the efficient evaluation of a query, which modify the original program by means of certain rules, acting as filters for the relevant information needed for the query processing. We stress that this modification must preserve the semantics of the original program, i.e., the whole process of query rewriting must be transparent to the final user of the integration system.

Thus, before describing the most well know strategy for the optimization of positive datalog queries, i.e., the Magic-Set method, we briefly recall the syntax and the semantics of DLP programs and provide the formal notion of query equivalent programs.

2.1 Syntax

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n \ \mathtt{:-} \ b_1, \cdots, b_k, \ \text{not } b_{k+1}, \cdots, \ \text{not } b_m. \tag{1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \ \text{not } b_{k+1}, ..., \ \text{not } b_m$ is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a fact, and we usually omit the " :- " sign.

The following notation will be useful for further discussion. If r is a rule of form (1), then $H(r) = \{a_1, \ldots, a_n\}$ is the set of the literals in the head and $B(r) = B^+(r) \cup B^-(r)$ is the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$.

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

2.2 Semantics and Query Equivalent DLP Programs

For any program $\mathcal{P}$, let $U_{\mathcal{P}}$ (the Herbrand Universe) be the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant ψ is added to $U_{\mathcal{P}}$. For any

[1] In positive programs negation as failure (not) does not occur, while strong negation ($\neg$) may be present.

program $\mathcal{P}$, let $B_\mathcal{P}$ be the set of all ground (classical) literals that can be constructed from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$ (note that, for each atom p, $B_\mathcal{P}$ contains also the strongly negated literal $\neg p$). For any rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$.

An interpretation I is a set of ground classical literals, i.e. $I \subseteq B_\mathcal{P}$ w.r.t. a program $\mathcal{P}$. A consistent interpretation $X \subseteq B_\mathcal{P}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive datalog program), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation $X \subseteq B_\mathcal{P}$ is an *answer set* for a positive disjunctive datalog program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.

The value of a ground atom l w.r.t. an interpretation I, $value_I(l)$, is $true$ if $l \in I$ and $false$ otherwise. The value of a ground negated literal not l is not $value_I(l)$. The truth value of a conjunction of ground literals $C = l_1, \ldots, l_n$ is the minimum over the values of the l_i, i.e., $value_I(C) = min(\{value_I(l_i) \mid 1 \leq i \leq n\})$, while the value $value_I(D)$ of a disjunction $D = l_1 \texttt{ v } \cdots \texttt{ v } l_n$ is their maximum, i.e., $value_I(D) = max(\{value_I(l_i) \mid 1 \leq i \leq n\})$; if $n = 0$, then $value_I(C) = true$ and $value_I(D) = false$. A ground rule r is *satisfied* by I if $value_I(H(r)) \geq value_I(B(r))$. Thus, a rule r with empty body is satisfied by I if $value_I(H(r)) = true$. In the following we also assume the existence of rules with empty head which define constraints, i.e. rules which are satisfied only if the body is false ($value_I(B(r)) = false$). An interpretation M for $\mathcal{P}$ is a model of $\mathcal{P}$ if M satisfies all rules in $Ground(\mathcal{P})$.

The model-theoretic semantics for positive $\mathcal{P}$ assigns to $\mathcal{P}$ the set $\mathcal{MM}(\mathcal{P})$ of its *minimal models*, where a model M for $\mathcal{P}$ is minimal, if no proper subset of M is a model for $\mathcal{P}$. The more general *disjunctive stable model semantics* generalizes stable model semantics, previously defined for normal programs [10] and also applies to programs with (unstratified) negation [11, 18].

The *reduct* or *Gelfond-Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_\mathcal{P}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by

- deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds;
- deleting the negative body from the remaining rules.

A stable model of a program $\mathcal{P}$ is a set $X \subseteq B_\mathcal{P}$ such that X is a minimal model of $Ground(\mathcal{P})^X$. The set of all stable models for a program $\mathcal{P}$ is denoted by $\mathcal{SM}(\mathcal{P})$. For positive $\mathcal{P}$, stable model and minimal model semantics coincide, i.e. $\mathcal{SM}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$.

Let $\mathcal{F}$ be a set of facts. Then, a *query* $\mathcal{Q}$ for $\mathcal{P}$ over the set of facts $\mathcal{F}$ is a conjunction of atoms, that defines a mapping from $\mathcal{P} \cup \mathcal{F}$ to a finite (possibly empty) set of tuples of atoms for $\mathcal{Q}$. Given a query $\mathcal{Q}$ and an interpretation M of $\mathcal{P}$, $\vartheta(\mathcal{Q}, M)$ denotes the set of substitutions for the variables in $\mathcal{Q}$ such that $\mathcal{Q}$ is true in M. The answer to a query $\mathcal{Q}$ over the facts in $\mathcal{F}$, under the *brave* semantics, denoted by $Ans_b(\mathcal{Q}, \mathcal{F}, \mathcal{P})$, is the set $\cup_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{SM}(\mathcal{P} \cup \mathcal{F})$. The answer to a query $\mathcal{Q}$ over the facts in $\mathcal{F}$, under the *cautious* semantics, denoted by $Ans_c(\mathcal{Q}, \mathcal{F}, \mathcal{P})$, is the set $\cap_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{SM}(\mathcal{P} \cup \mathcal{F})$.

$\mathcal{P}'$ is *bravely* (resp. *cautiously*) *sound* for $\mathcal{P}$ w.r.t. $\mathcal{Q}$, if for any set $\mathcal{F}$ of facts $Ans_b(\mathcal{Q},\mathcal{F},\mathcal{P}') \subseteq Ans_b(\mathcal{Q},\mathcal{F},\mathcal{P})$ (resp. $Ans_c(\mathcal{Q},\mathcal{F},\mathcal{P}') \subseteq Ans_c(\mathcal{Q},\mathcal{F},\mathcal{P})$).

$\mathcal{P}'$ is *bravely* (resp. *cautiously*) *complete* for $\mathcal{P}$ w.r.t. $\mathcal{Q}$, if for any set $\mathcal{F}$ of facts $Ans_b(\mathcal{Q},\mathcal{F},\mathcal{P}') \supseteq Ans_b(\mathcal{Q},\mathcal{F},\mathcal{P})$ (resp. $Ans_c(\mathcal{Q},\mathcal{F},\mathcal{P}') \supseteq Ans_c(\mathcal{Q},\mathcal{F},\mathcal{P})$).

Finally, $\mathcal{P}$ and $\mathcal{P}'$ are *bravely* (resp. *cautiously*) *equivalent* w.r.t. $\mathcal{Q}$, denoted by $\mathcal{P} \equiv_{\mathcal{Q},b} \mathcal{P}'$ (resp. $\mathcal{P} \equiv_{\mathcal{Q},c} \mathcal{P}'$), if for any set $\mathcal{F}$ of facts $Ans_b(\mathcal{Q},\mathcal{F},\mathcal{P}) = Ans_b(\mathcal{Q},\mathcal{F},\mathcal{P}')$ (resp. $Ans_c(\mathcal{Q},\mathcal{F},\mathcal{P}) = Ans_c(\mathcal{Q},\mathcal{F},\mathcal{P}')$).

3 Efficient Evaluation of Logic Programs

In this section we overview the original proposal of Magic-Set method for positive Datalog programs and we analyze the extensions proposed for dealing with disjunctive rules.

3.1 Magic-Set Method for Datalog Queries

The Magic-Set method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of additional rules, which narrow the search to what is relevant for answering the query. We will illustrate the technique by considering the rule $\mathtt{path(X,Y)}$:- $\mathtt{path(X,Z)}$, $\mathtt{path(Z,Y)}$. together with query $\mathtt{path(1,5)}$?.

The key idea is to analyze the binding propagation which would be done during a top-down computation. To this end, for each occurrence of an atom we record for each argument whether it would be bound (b) or free (f). This information is denoted as a string of the letters b and f.

The construction starts from the query, in which the adorned predicates can be obtained by marking each constant with b, and all the variables with f. For instance, the adorned version of the query is $\mathtt{path}^{\mathtt{bb}}(\mathtt{1},\mathtt{5})$. These predicates constitute the basis of the method, which consists of the following steps:

1. Adornment Step: Each bound argument of an adorned predicate in the query can be used for propagating its information into the body of the rules defining it. In particular, given an adorned predicate $\mathtt{p}^{\alpha}$, each rule r having the atom $\mathtt{p}(\mathtt{t_1},\ldots,\mathtt{t_n})$ in the head is adorned by exploiting a particular Sideways Information Passing Strategy (*SIPS*). A SIPS simulates the data flow occurring in the top-down evaluation of the query, by iteratively processing all the predicates in r. Let $\mathtt{q}$ be an atom that has not yet been processed, then its adorned version is created by assuming constants and variables occurring in already considered atoms to be bound, which is denoted by $\mathtt{v} \rightarrow_{\mathtt{X}} \mathtt{q}$, where $\mathtt{X}$ is the set of the variables assumed to be bound which propagate their values into $\mathtt{q}$, and $\mathtt{v}$ is the set of the predicates in which these variables occur.

Definition 3.1 Let r be a rule having $\mathtt{p}$ in the head, and let $\mathtt{p}^{\alpha}$ be an adornment. A *SIPS* for r is a labelled bipartite graph $\langle V_1 \cup V_2, E\rangle$, where V_1 is the set of subset of $B(r) \cup \{\mathtt{p}^{\alpha}\}$, $V_2 \in B(r)$, and E is a set of arcs satisfying the following conditions:

1. each arc is of the form $v \rightarrow_X s$, where $v \in V_1$ and $s \in V_2$, where X is a non-empty set of variables such that (i) each variable in X appears in s and in either a bound

argument position of $\mathtt{p}^\alpha$ or a positive body literal of v, and (ii) for each literal in v there exists a sequence of literals $v = l_0, l_1, ..., l_m = s$ with l_i and l_{i+1} sharing at least a common argument.

2. there exists a total order of $B(r) \cup \{\mathtt{p}^\alpha\}$ in which
   (a) $\mathtt{p}^\alpha$ precedes all members of $B(r)$,
   (b) any literal which does not appear in the graph follows every literal that appears in the graph, and
   (c) for each arc $v \rightarrow_X s$, if $u \in v$ the u precedes s. □

It is well know that if we are able to construct a SIPS for a given rule r and a predicate $\mathtt{p}^\alpha$, then we can use its edges for simulating the data flow from the head to the body of a rule, and, hence, for deriving the adornment of the rule.

We point out that, for each rule, it is possible to derive different SIPS, associated to all the possible permutations of the atoms appearing in the body; the choosing of a strategy does not matter in the case of positive programs, while in the presence of possibly negated atoms much attention is needed; in fact, we have implemented a method that enables the preservation of the safety of the rules.

For instance, $\mathtt{path}^{\mathtt{bb}}(\mathtt{1},\mathtt{5})$ passes its binding information to $\mathtt{path}(\mathtt{X},\mathtt{Z})$ as $\mathtt{path}^{\mathtt{bb}}(\mathtt{1},\mathtt{5}) \rightarrow_{\{\mathtt{X},\mathtt{Y}\}} \mathtt{path}(\mathtt{X},\mathtt{Z})$, which causes the generation of the adorned predicate $\mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z})$. Then we apply $\{\mathtt{path}^{\mathtt{bb}}(\mathtt{1},\mathtt{5}),\ \mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z})\} \rightarrow_{\{\mathtt{Y},\mathtt{Z}\}} \mathtt{path}(\mathtt{Z},\mathtt{Y})$, generating $\mathtt{path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$. The resulting adorned rule is $\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z}),\ \mathtt{path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$.

As in the above example, adorning a rule may generate new adorned predicates ($\mathtt{path}^{\mathtt{bf}}$). Thus, a natural way for carrying out the adornment consists in repeating this step until all adorned predicates have been processed. The resulting program is called *adorned program.*

2. Generation Step: The adorned program is used to generate *magic rules*, which simulate the top-down evaluation scheme. Preliminarily, we define a transformation over adorned atoms. For each adorned atom $\mathtt{p}^\alpha$, its *magic version* $\boldsymbol{magic}(\mathtt{p}^\alpha)$ consists of the atom $\mathtt{magic_p}^\alpha$ in which all arguments labelled f in α are eliminated.

Then, for each adorned atom $\mathtt{p}^\alpha$ in the body of any adorned rule r_a we generate a magic rule r_m such that (i) the head of r_m consists of $\boldsymbol{magic}(\mathtt{p}^\alpha)$, and (ii) the body of r_m consists of the magic version of the head atom of r_a, followed by all of the predicates of r_a which can propagate the binding on $\mathtt{p}$. For instance in our example we generate $\mathtt{magic_path}^{\mathtt{bf}}(\mathtt{X})$:- $\mathtt{magic_path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$. and $\mathtt{magic_path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$:- $\mathtt{magic_path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}),\ \mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z})$.

3. Modification Step: The adorned rules are modified by including magic atoms generated in Step 2 in the rule bodies. The resultant rules are called *modified rules.* For each adorned rule whose head is $\mathtt{h}$, we extend the rule body by inserting $\boldsymbol{magic}(\mathtt{h})$. In our example $\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{magic_path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}),\ \mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z}),\ \mathtt{path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$. is generated.

4. Processing of the Query: In the steps described so far, the query is only used for deriving the original set of adorned predicates, but also *query rules* are needed: For each adorned atom $\mathtt{g}^{\alpha}$ of the query, (1) the fact $\boldsymbol{magic}(\mathtt{g}^{\alpha})$ is asserted, and (2) a rule $\mathtt{g} \texttt{ :- } \mathtt{g}^{\alpha}$ is produced. Note that (1) provides a starting point for the simulated top-down evaluation, while (2) is necessary to provide answers to the original query. In our example we generate $\mathtt{magic_path}^{\mathtt{bb}}(\mathtt{1},\mathtt{5})$. and $\mathtt{path}(\mathtt{X},\mathtt{Y}) \texttt{ :- } \mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$.

The complete rewritten program consists of the magic, modified, and query rules. Given a program $\mathcal{P}$, a query $\mathcal{Q}$, and the rewritten program $\mathcal{P}'$, it has been shown that $\mathcal{P} \equiv_{\mathcal{Q},b} \mathcal{P}'$ and $\mathcal{P} \equiv_{\mathcal{Q},c} \mathcal{P}'$ hold (see e.g. [20]).

3.2 Related Works on Extensions of the Magic-Set Method

In this section we review the technique for the propagation of bindings into positive disjunctive queries proposed in [14], which since now on will be denoted as APM method.

Given a positive disjunctive Datalog program $\mathcal{P}$, APM preliminary constructs a suitable normal program, called the *extended standard* version of $\mathcal{P}$, and denoted by $esv(\mathcal{P})$. This program is derived by replacing each disjunctive rule $a_1 \vee ... \vee a_m \leftarrow B$ with

1. m rules of the form $a_i \leftarrow B \quad$ for $1 \leq i \leq m$, and
2. $m \times (m-1)$ rules of the form $a_i \leftarrow a_j, B \quad$ for $1 \leq i, j \leq m$ and $i \neq j$.

As the programs $\mathcal{P}$ and $esv(\mathcal{P})$ may be not equivalent, the second step of APM is to derive a program that must be equivalent to the original one, with the following procedure:

- Each derived predicate symbol g in $esv(\mathcal{P})$ is replaced with a new predicate symbol G obtaining the program $ESV(\mathcal{P})$.
- The program $RV(\mathcal{P})$ is constructed from $\mathcal{P}$ by inserting into the body of each disjunctive rule of the form $a_1 \vee ... \vee a_m \leftarrow B$, all the head predicates with the symbol replaced as in $ESV(\mathcal{P})$, i.e., deriving a rule of the form $a_1 \vee ... \vee a_m \leftarrow A_1, ..., A_m, B$.
- The rewritten program $Rew(\mathcal{P})$ is defined to be the union of $RV(\mathcal{P})$ and $ESV(\mathcal{P})$.

In [14], it has been proved that the programs $\mathcal{P}$ and $Rew(\mathcal{P})$ have actually the same semantics (w.r.t. the predicates in $\mathcal{P}$). In fact, adding the atoms defined in $ESV(\mathcal{P})$ inside the body of the rules of $\mathcal{P}$, does not make any effective restriction, as for every ground atom $a(t)$ appearing in the head of a disjunctive ground rule r there is a ground atom $A(t)$ which is derived from $ESV(\mathcal{P})$.

Thus, the idea is to not use directly the program $ESV(\mathcal{P})$ to restrict the rules in $\mathcal{P}$, but rather to consider a different program which makes an effective and sound restriction, i.e., the program obtained by applying the Magic-Set method to the standard positive datalog program $ESV(\mathcal{P})$.

It is obvious that even though this approach is correct, it bloats the programs with additional predicates and rules, namely the ones in $ESV(\mathcal{P})$ which are called collecting rules by the author, by reducing the overall efficiency. Our approach offers instead a novel integrated algorithm (instead of a phased one) which avoids the generation of many

Input: A Datalog$^\vee$ program $\mathcal{P}$, and a query $\mathtt{g_1}(\mathtt{t_1}), \ldots, \mathtt{g_n}(\mathtt{t_n})$.
Output: The optimized program $\mathcal{P}'$.
var S: **stack** of adorned predicates; *modifiedRules*,*magicRules*: **set** of rules;
begin
1. **if** $\mathtt{g_1}(\mathtt{t_1}), \ldots, \mathtt{g_n}(\mathtt{t_n})$ has some IDB predicate **then**
2. $\langle adornedQuery, S\rangle$:=***AdornQuery***$(\mathtt{g_1}(\mathtt{t}), \ldots, \mathtt{g_n}(\mathtt{t}), \mathcal{P})$;
3. $\langle modifiedRules, magicRules\rangle$:=***BuildQueryRules***$(adornedQuery)$;
4. **while** $S \neq \emptyset$ **do**
5. p^α:=S.**pop()**;
6. **for each** rule $r \in \mathcal{P}$: $\mathtt{p(t)} \vee \mathtt{p_1(t_1)} \vee \ldots \vee \mathtt{p_n(t_n)} \text{ :- } \mathtt{q_1(s_1)}, \ldots, \mathtt{q_m(s_m)}$ **do**
7. **let** r_s be the rule $\mathtt{p(t)} \text{ :- not } \mathtt{p_1(t_1)}, \ldots, \text{not } \mathtt{p_n(t_n)}, \mathtt{q_1(s_1)}, \ldots, \mathtt{q_m(s_m)}$;
8. r_a:=***Adorn***(r_s, p^α, S);
9. $magicRules := magicRules \bigcup$ ***Generate***(r_a);
10. r_m:=***Modify***(r_a);
11. $modifiedRules := modifiedRules \bigcup \{r_m\}$;
12. **end for**
13. **end while**
14. P':=$magicRules \cup modifiedRules$;
15. P':=***StripOffAdornment***$(\mathcal{P}')$;
16. **return** $\mathcal{P}'$;
17. **end if**
end.

Figure 1: Disjunctive Magic-Set Method

auxiliary rules needed in APM. Moreover, our technique is much more general as it can be applied also in the presence of constraints and general unstratified programs as well.

To the best of our knowledge this approach is the most general presented so far in the literature.

4 Magic-Set Method for Positive DLP Programs

In this section we present a technique for the optimization of disjunctive datalog programs (DLP programs), which has been implemented and integrated into the DLV system [8, 9]. Basically, the technique performs the adornment on a suitable non-disjunctive version of the rules, and uses the generated magic predicates for modifying the original disjunctive rules.

Differing from traditional implementations of the magic-set technique, our proposal is not conceptually divided into *adornment*, *generation*, and *modification* phases. Instead, it exploits a stack S of predicates for storing all the adorned predicates to be used for propagating the binding of the query: at each step, an element is removed from S, and it is used for processing each rule r of a given program $\mathcal{P}$, by contextually generating the associated modified and magic rules. Moreover, we point out that the algorithm can be used for non-disjunctive rules as a special case.

In more detail, the algorithm (see Figure 1) takes in input a disjunctive datalog program $\mathcal{P}$ and a query $\mathtt{g}_1(\mathtt{t}_1), \ldots, \mathtt{g}_n(\mathtt{t}_n)$?. If the query contains some non-free IDB predicates, it outputs a program $\mathcal{P}'$ consisting of a set of *modified*, *magic*, and *query* rules. The adorned query and the query rules are built in step *2* and *3* by means of the functions ***AdornQuery*** and ***BuildQueryRules***, the former being used for adorning the query and the latter for implementing the step *Processing of the Query* of the previous section.

The function ***AdornQuery*** is shown in Figure 2. It takes in input a query of the form $\mathtt{g}_1(\mathtt{t}), \ldots, \mathtt{g}_n(\mathtt{t})$ and outputs a tuple $\langle adornedQuery, S\rangle$, where the first argument is the adorned query, i.e., the query whose predicate are adorned as described above, while the second argument is the set of the adorned predicates obtained during the computation.

Input: A query $\mathtt{g}_1(\mathtt{t}), \ldots, \mathtt{g}_n(\mathtt{t})$.
Output: The adorned query, and a set of adorned predicates.
var S: **stack** of adorned predicates; *adornedQuery*: **set** of literals;
$\alpha_1, ...\alpha_n$: **string** over $\{b, f\}$;
begin
for each α_i **do** α_i:=";
for each IDB not propositional predicate $\mathtt{g}_i(\mathtt{t}_1, ..., \mathtt{t}_m)$ **do**
for each $\mathtt{t}_j$ argument of $\mathtt{g}_i$ **do**
if t_j is a constant **then** $\alpha_i[j]$:=b; **else** $\alpha_i[j]$:=f; **end if**
end for
if $\mathtt{g}_i^{\alpha_i}(\mathtt{t}_1, ..., \mathtt{t}_m) \notin \mathtt{S}$ **then** S.**push**($\mathtt{g}_i^{\alpha_i}$);
end for
return $\langle \mathtt{g}_1^{\alpha_1}(\mathtt{t}), \ldots, \mathtt{g}_n^{\alpha_n}(\mathtt{t}), S\rangle$.
end.

Figure 2: Algorithm ***AdornQuery***

The function ***BuildQueryRules*** is instead shown in Figure 3.

Input: An adorned query $\mathtt{g}_1^{\alpha_1}(\mathtt{t}), \ldots, \mathtt{g}_n^{\alpha_n}(\mathtt{t})$.
Output: The set *modifiedRules* and *magicRules* derived from the query.

begin
for each $\mathtt{g}_i(\mathtt{t}_1, ..., \mathtt{t}_m)$ **do**
magicRules:=*magicRules* $\cup$ {***magic***($\mathtt{g}_i^{\alpha_i}$)};
modifiedRules:=*modifiedRules* $\cup$ {$\mathtt{g}_i(\mathtt{t})$:- $\mathtt{g}_i^{\alpha_i}(\mathtt{t})$};
end for
return $\langle$ *modifiedRules*,*magicRules*$\rangle$.
end.

Figure 3: Algorithm ***BuildQueryRules***

Example 4.1 Consider the disjunctive datalog rule p(X) v p(Y) :- a(X, Y), r(X). and the query p(1)?. The function ***AdornQuery*** outputs the query $\mathtt{p}^{\mathtt{b}}(1)$?, and pushes the

adorned predicate $\mathtt{p}^{\mathtt{b}}$ on the stack S. The function ***BuildQueryRules*** outputs the rule $\mathtt{p(X)}$:- $\mathtt{p}^{\mathtt{b}}\mathtt{(X)}$ and the fact $\mathtt{magic_p}^{\mathtt{b}}\mathtt{(1)}$. □

The core of the technique (steps *5-12*) is repeated until the stack S is empty, i.e., until there is no further adorned predicate to be propagated. Next, we briefly explain the main ideas that are used for propagating the binding into disjunctive rules.
Let $\mathtt{p}^{\alpha}$ be an adorned predicate that has been removed from the stack S in step *4*, and consider a disjunctive rule r in $\mathcal{P}$ of the form

$$r: \ \mathtt{p(t)\ v\ p_1(t_1)\ v\ \ldots\ v\ p_n(t_n)} \text{ :- } \mathtt{q_1(s_1),\ \ldots,\ q_m(s_m)}.$$

in which the binding of $\mathtt{p}^{\alpha}$ should be propagated from the occurrence of $\mathtt{p(t)}$ in its head. The difference w.r.t. the case of non-disjunctive programs is that the propagation also affects the atoms $\mathtt{p_1(t_1)}, \ldots, \mathtt{p_n(t_n)}$ in the head. A simple idea is to replace r with a rule r_s without disjunction of the form

$$r_s: \ \mathtt{p(t)} \text{ :- not } \mathtt{p_1(t_1)}, \ldots, \text{not } \mathtt{p_n(t_n),\ q_1(s_1),\ \ldots,\ q_m(s_m)}.$$

in which the head atoms are pushed into the negative body since they cannot yield any binding. r_s can then be adorned in a more or less standard way. The only peculiarity is the presence of negated atoms, which require a slightly refined SIPS, in which negated atoms are always processed last.

4.1 Adornment Step: SIPS for Disjunctive Programs

Rather then processing sequentially each predicate in the body of the rule to be adorned (as shown in the previous section), we focus on *admissible* predicates, which is needed for specifying the literals that can be used for propagating the binding.

Definition 4.2 A literal l is *admissible*, if (1) l is a built-in comparison literal ($<=$, $<$, $>$, $>=$, $!=$) and both its argument are bound, or (2) l is a built-in equality literal ($==$) and at least one of its arguments is bound, or (3) l is a built-in arithmetic literal ($+$, $*$) and at least one of its arguments is bound, or (4) l is a negative literal, and all its arguments are bound. Any other positive non built-in literal is *admissible*. □

Moreover, in the case of many admissible literals, we need a notion of preference for deciding the literal to be selected. The idea beyond the following definition is to prefer the literals which are the most useful for propagating the binding, i.e., the literals having the maximum number of bound arguments.

Definition 4.3 Let l_1 and l_2 be two admissible literals. Then, l_1 is *preferable* to l_2 if (1) l_1 is a propositional EDB literal, or (2) l_2 is a propositional IDB literal, or (3) l_1 has some bound arguments, while l_2 as not, or (4) l_1 is an EDB literal with all its arguments bound, or (5) both l_1 and l_2 have some bound arguments, but l_1 is EDB or built-in, or (6) both l_1 and l_2 are IDB, they have both some bound arguments but l_1 has more bound arguments than l_2, or (7) both l_1 and l_2 have no bound arguments, where l_1 is EDB or built-in or l_2 is IDB. In all other cases l_2 is preferable to l_1.

Finally, given a set of literals L, we say that $l \in L$ is *preferred* if there exists no $l' \in L$ preferable to l. □

The SIPS we adopt selects at each step the *preferred* predicate among the ones that have to be adorned. Literal which are not *admissible* are eventually processed sequentially.

Example 4.4 Consider the rule $\mathtt{q(X,Y,C)}$:- $\mathtt{b(X,Y,Z,W)}$, not $\mathtt{c(D,C)}$, $\mathtt{q(Z,W,D)}$. where $\mathtt{b}$ and $\mathtt{c}$ are *EDB* predicates, and the query $\mathtt{p(1,C)}$? When adorning with the adorned query predicate $\mathtt{q}^{\mathtt{bbf}}(\mathtt{1,2,C})$, we first select the predicate $\mathtt{b(X,Y,Z,W)}$, which is preferred, as it satisfies Condition 5 in Definition 4.3 when compared with $\mathtt{q(Z,W,D)}$, and since $\mathtt{c(D,C)}$ is not admissible for it not satisfying Condition 4 in Definition 4.2. Hence, we have $\{\mathtt{q}^{\mathtt{bbf}}(\mathtt{X,Y,C})\} \rightarrow_{\{\mathtt{X,Y}\}} \mathtt{b(X,Y,Z,W)}$.

Then, the set of bound variables becomes $\{\mathtt{X,Y,Z,W}\}$; since $\mathtt{c(D,C)}$ is still not admissible, we chose the predicate $\mathtt{q(Z,W,D)}$, obtaining $\{\mathtt{q}^{\mathtt{bbf}}(\mathtt{X,Y,C}), \mathtt{b(X,Y,Z,W)}\} \rightarrow_{\{\mathtt{X,Y,Z,W}\}} \mathtt{q(Z,W,D)}$.

Finally, we select $\{\mathtt{q}^{\mathtt{bbf}}(\mathtt{X,Y,C}), \mathtt{b(X,Y,Z,W)}, \mathtt{q(Z,W,D)}\} \rightarrow_{\{\mathtt{X,Y,Z,W,D}\}} \mathtt{c(D,C)}$. and we generate the adorned rule $\mathtt{q}^{\mathtt{bbf}}(\mathtt{X,Y,C})$:- $\mathtt{b(X,Y,Z,W)}$, $\mathtt{q}^{\mathtt{bbf}}(\mathtt{Z,W,D})$, not $\mathtt{c}^{\mathtt{bf}}(\mathtt{D,C})$. □

The rule r_s is then adorned by means of the function ***Adorn*** which is shown in Figure 4. It takes in input a rule r to be adorned w.r.t. $\mathtt{p}^{\alpha}$ and the stack S of adorned predicates.

The algorithm stores in the variable *remainingLiterals* all the literals in the body of the rule r, that have still to be adorned; first, it create a set of *boundVariables* by inserting all the variables of $\mathtt{p(t)}$ whose adornment is b. By taking a look at Definition 3.1, the careful reader may notice that such a set corresponds to the labels of each edge of the form $v \rightarrow_X s$.

Then, at each step, in the case there exists some *admissible* literal in $B(r)$, it is chosen the preferred literal in this set, for being adorned. Such a literal is the right term of the arc $v \rightarrow_X s$, while the left term consists of all the predicates selected so far plus the original adorning predicate. The iterations continue till all the literal have been considered or there exists no other *admissible* literal. Finally, literals that are not *admissible* and have not yet considered are processed.

Note that the algorithm makes use of the subroutine ***AdornLiteral*** that takes the literal chosen with the above criteria, adorns it and updates the variables *adornedPredicates*, *remainingLiterals*, S, and *boundVariables*. Moreover, if the adorned literal generated by this way is not in *adornedPredicates* than it is pushed on the stack S, and, hence, can be used for adorning some other rule. informally described in the previous section. It produces an adorned rule from an adorned predicate and a suitable unadorned rule by employing the refined SIPS and pushes all newly adorned predicates on the stack S.

Hence, in step *8* let us assume that the rule r_a obtained by means of this adornment is of the form

$$r_a : \ \mathtt{p}^{\alpha}(\mathtt{t}) \text{ :- not } \mathtt{p}_1^{\alpha_1}(\mathtt{t}_1), \ldots, \text{not } \mathtt{p}_n^{\alpha_n}(\mathtt{t}_n), \mathtt{q}_1^{\beta_1}(\mathtt{s}_1), \ldots, \mathtt{q}_m^{\beta_m}(\mathtt{s}_m).$$

Example 4.5 Consider again the program and the query of Example 4.1. When $\mathtt{p}^{\mathtt{b}}$ is considered, we first select the head predicate $\mathtt{p(X)}$. Then, we construct the rule $\mathtt{p(X)}$:- not $\mathtt{p(Y)}, \mathtt{a(X,Y)}, \mathtt{r(X)}$. whose adorned version is $\mathtt{p}^{\mathtt{b}}(\mathtt{X})$:- not $\mathtt{p}^{\mathtt{b}}(\mathtt{Y}), \mathtt{a(X,Y)}, \mathtt{r}^{\mathtt{b}}(\mathtt{X})$.

$\mathtt{p}^{\mathtt{b}}$ is also applicable on the head predicate $\mathtt{p(Y)}$, producing the adorned rule $\mathtt{p}^{\mathtt{b}}(\mathtt{Y})$:- not $\mathtt{p}^{\mathtt{b}}(\mathtt{X}), \mathtt{a(X,Y)}, \mathtt{r}^{\mathtt{b}}(\mathtt{X})$. The predicate $\mathtt{r}^{\mathtt{b}}$ is pushed onto the stack S during the processing, and can be eventually used for adorning other rules having the predicate $\mathtt{r}$ in their head. □

Input: A rule r with $\mathtt{p}(\mathtt{t})$ in the head, an adorned predicate $\mathtt{p}^{\alpha}$, the stack of adorned predicates S, and the set of adorned predicates *adornedPredicates*.
Output: The adorned rule of r w.r.t. $\mathtt{p}^{\alpha}$.
var r_a: adorned rule; *remainingLiterals*: **set** of literals; *existsAdmissibleLiteral*: **boolean**; l,l': literal; *boundVariables*: **set** of variables;
Procedure ***AdornLiteral***(l: literal);
begin
  α_1:=";
  if l is not an EDB predicate **then**
    for each $\mathtt{t_j}$ argument of $l(\mathtt{t_1},...,\mathtt{t_m})$ **do**
      if t_j is a constant or a variable in *boundVariables* **then** $\alpha_1[\mathtt{j}]$:=b;
      else $\alpha_1[\mathtt{j}]$:=f; **end if**
    end for
    if $l^{\alpha_1}(\mathtt{t_1},...,\mathtt{t_m}) \notin$ *adornedPredicates* **then** S.**push**($\mathtt{g_i}^{\alpha_i}$);
  end if
  append $l^{\alpha_1}(\mathtt{t_1},...,\mathtt{t_m})$ into the body of r_a;
  adornedPredicates:= *adornedPredicates*$\cup\{l^{\alpha_1}\}$;
  remainingLiterals:=*remainingLiterals* -$\{l(\mathtt{t_1},...,\mathtt{t_m})\}$;
  for each $\mathtt{t_j}$ argument of $l(\mathtt{t_1},...,\mathtt{t_m})$ **do**
    if t_j is a variable **then** *boundVariables*:=*boundVariables* $\cup\{t_j\}$;
  end for
end;
begin
  remainingLiterals:=$B(r)$;
  existsAdmissibleLiteral:=**true**;
  boundVariables;
  for each argument $\mathtt{t_j}$ in $\mathtt{p}(\mathtt{t_1},...,\mathtt{t_m})$ **do**
    if t_j is a variable and $\alpha[\mathtt{j}] =' \mathtt{b}'$ **then** *boundVariables*:=*boundVariables* $\cup\{t_j\}$;
  let r_a be the rule $\mathtt{p}^{\alpha}(\mathtt{t}) :-$;
  while (*remainingLiterals*$!=\emptyset \wedge$ *existsAdmissibleLiteral*) **do**
    if $\exists l' \in$ *remainingLiterals* such that l' is *admissible* **then**
      let l be the *preferred* literal among those *admissible*;
      AdornLiteral(l);
    else
      existsAdmissibleLiteral:=**false**;
    end if
  end while
  for each $l \in$ *remainingLiterals* **do** ***adornLiteral***(l);
  return r_a;
end.

Figure 4: Algorithm ***Adorn***

4.2 Generation and Modification Steps

The algorithm Disjunctive Magic Set (reported in Figure 1), after having adorned a rule r by generating r_a, uses r_a for generating the magic rules in step *8*. Since r_a is not a disjunctive rule, the technique is very similar to the case of Datalog queries, and is carried out by means of the function ***Generate***, implementing the *Generation Step* sketched in the previous section. For instance, from Example 4.5, we derive the following magic rules:

$\mathtt{magic_r^b(X)} \text{ :- } \mathtt{magic_p^b(X), p^b(Y), a(X,Y).}$ $\quad$ $\mathtt{magic_p^b(Y)} \text{ :- } \mathtt{magic_p^b(X), r^b(X), a(X,Y).}$
$\mathtt{magic_r^b(X)} \text{ :- } \mathtt{magic_p^b(Y), p^b(X), a(X,Y).}$ $\quad$ $\mathtt{magic_p^b(X)} \text{ :- } \mathtt{magic_p^b(Y), r^b(X), a(X,Y).}$

Step *9* generates the modified rules. Essentially, a disjunctive rule is reconstructed by reversing the push performed in steps *5-6*. Hence, the function ***Modify*** constructs a rule of the following form

$$\mathtt{p}^{\alpha}(\mathtt{t}) \,\mathtt{v}\, \mathtt{p}_1^{\alpha_1}(\mathtt{t}_1) \,\mathtt{v}\, \ldots \,\mathtt{v}\, \mathtt{p}_n^{\alpha_n}(\mathtt{t}_n) \text{ :- } \textbf{\textit{magic}}(\mathtt{p}^{\alpha}(\mathtt{t})), \textbf{\textit{magic}}(\mathtt{p}_1^{\alpha_1}(\mathtt{t}_1)), \ldots, \textbf{\textit{magic}}(\mathtt{p}_n^{\alpha_n}(\mathtt{t}_n)), \mathtt{q}_1(\mathtt{s}_1)^{\beta_1}, \ldots, \mathtt{q}_m^{\beta_m}(\mathtt{s}_m).$$

Example 4.6 In our running example, we derive the following set of modified rules:

$$\mathtt{p^b(X)} \,\mathtt{v}\, \mathtt{p^b(Y)} \text{ :- } \mathtt{magic_p^b(X), magic_p^b(Y), a(X,Y), r^b(X).}$$
$$\mathtt{p^b(Y)} \,\mathtt{v}\, \mathtt{p^b(X)} \text{ :- } \mathtt{magic_p^b(Y), magic_p^b(X), a(X,Y), r^b(X).}$$

□

4.3 Postprocessing

Since the adornment of the head of disjunctive rules may give rise to multiple rules, the semantics of the program may be affected: Consider, for instance, the rule $\mathtt{p(X,Y)} \;\mathtt{v}\; \mathtt{q(Y,X)} \text{ :- } \mathtt{a(X,Y).}$ in which p and q are mutually exclusive (due to minimality). Assuming the adornments $\mathtt{p^{bf}}$ and $\mathtt{q^{bf}}$ to be propagated, we obtain:

$$\mathtt{p^{bf}(X,Y)} \;\mathtt{v}\; \mathtt{q^{fb}(Y,X)} \text{ :- } \mathtt{magic_p^{bf}(X),\; magic_q^{fb}(X),\; a(X,Y).}$$
$$\mathtt{p^{fb}(X,Y)} \;\mathtt{v}\; \mathtt{q^{bf}(Y,X)} \text{ :- } \mathtt{magic_p^{fb}(Y),\; magic_q^{bf}(Y),\; a(X,Y).}$$

These two rules could support two atoms $\mathtt{p^{bf}(m,n)}$ and $\mathtt{q^{bf}(n,m)}$, while in the original program $\mathtt{p(m,n)}$ and $\mathtt{p(n,m)}$ could not hold simultaneously. The method in [14] circumvents this problem by defining auxiliary predicates for collecting all the different adornments. For instance, in the above example, by considering the predicate p two rules of the form $\mathtt{collect_\ p(X,Y)} \text{ :- } \mathtt{p^{fb}(X,Y).}$ and $\mathtt{collect_\ p(X,Y)} \text{ :- } \mathtt{p^{bf}(X,Y).}$ are added.

In our implementation we do not use collecting predicates, rather we remove the adornments directly from the modified rules, by means of the function ***StripOffAdornments*** in step *14*. Obviously, the function does not affect magic predicates. For instance, the rules from Example 4.6 result in the following rules:

$$\mathtt{p(X)} \,\mathtt{v}\, \mathtt{p(Y)} \text{ :- } \mathtt{magic_p^b(X), magic_p^b(Y), a(X,Y), r(X).}$$
$$\mathtt{p(Y)} \,\mathtt{v}\, \mathtt{p(X)} \text{ :- } \mathtt{magic_p^b(Y), magic_p^b(X), a(X,Y), r(X).}$$

The rewritten program $\mathcal{P}'$ consists of the above stripped rules, and the magic rules.

5 Magic-Set Method for Positive DLP Programs with Aggregates

In this section, we provide an extension of the Magic-Set Method for programs with aggregate functions. Before describing the technique, we preliminary overview the syntax and the semantics of DLP$^{\mathcal{A}}$ (for a more formal treatment we refer to the deliverable D4.1).

5.1 Syntax and Semantics of Aggregates in a Nutshell

A (DLP$^{\mathcal{A}}$) *set* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard literals.[2] A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among #`count`, #`min`, #`max`, #`sum`. Let I be an interpretation. The informal semantics of the aggregate functions are as follows:

- #**count**$\{Vars : Conj\}$. **#count** returns the number of ground instances of $Vars$ which satisfy the conjunction $Conj$ w.r.t. the interpretation I.
- #**sum**$\{Vars : Conj\}$. **#sum** computes the sum of all the values of the first variable in *Vars* in the ground instances satisfying the conjunction $Conj$ w.r.t. the interpretation I.
- #**min**$\{Vars : Conj\}$. Let X be the first variable of *Vars*. Then, the function **#min** returns the minimum value of the variable X among those taken in the ground instances of *Vars* making the conjunction $Conj$ true w.r.t. the interpretation I.
- #**max**$\{Vars : Conj\}$. Let X be the first variable of *Vars*. Then, the function **#max** returns the maximum value of the variable X among those taken in the ground instances of *Vars* making the conjunction $Conj$ true w.r.t. the interpretation I.

An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted. Given an interpretation I, the aggregate literal is true if and only if the value taken by the aggregate function $f(\mathcal{S})$ belongs to the range specified by the guard values Lg and Rg.

An *atom* is either a standard (DLP) atom or an aggregate atom. A *literal* L is either an atom A or an atom A preceded by the default negation symbol *not*; if A is an aggregate atom, then L is an *aggregate literal*.

[2] Intuitively, a symbolic set $\{X : a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X : \exists Y s.t.\ a(X,Y), p(Y)\ is\ true\}$. Note that also negative literals may occur in the conjunction $Conj$ of a symbolic set.

A *(*DLP$^{\mathcal{A}}$*) rule* r is a construct

$$a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n \ \mathtt{:-} \ b_1, \cdots, b_k, \ \text{not} \ b_{k+1}, \cdots, \ \text{not} \ b_m.$$

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are literals, and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \ \text{not} \ b_{k+1}, ..., \text{not} \ b_m$ is the *body* of r. A *(*DLP$^{\mathcal{A}}$*) program* is a set of DLP$^{\mathcal{A}}$ rules.

5.2 Modification of the SIPS

Let us consider an aggregate literal of the form $L_g \leq f\{Vars : Conj\} \leq U_g$, where $Vars$ are variables local w.r.t. the function f, while $Conj$ is a conjunction of literals. As in $Conjs$ there may appear some variables that are used into other predicates of a given rule, the idea is to exploit such variables for propagating the binding into the aggregate literals, too.

Thus, in the adornment step, literals in $Conj$ can be treated as they were part of the rule; nonetheless some further attention is needed for ensuring the correctness of the SIPS implemented. In fact, literals in $Conj$ have not to be used for propagating bindings to other literals, and, hence, they should be considered at the end of the adornment process. In order to implement in an efficient way this behavior, we simple extend the concepts of *admissible* and *preferable* literal, exposed in Definition 4.2 and 4.3, respectively.

Definition 5.1 An aggregate literal $L_g \leq f\{Vars : Conj\} \leq U_g$ is *admissible* if all the global variables of $Conjs$ are bound. Let l_a be an admissible aggregate literal and l be an admissible literals. Then l_a is *preferable* to l if l is an aggregate literal, and l_a has a bigger number of bound variables. In all the other cases, l is *preferable* to l_a. □

We point out that, by using the above notions, for the adornment we can exploit the same strategy presented in the function ***Adorn***.

Example 5.2 Let us consider the following rule containing an aggregate literal

$$\mathtt{q(X)} :- \mathtt{a(X)}, \ 2 \leq \mathit{count}(\mathtt{Y} : \mathtt{r(X,Y,Z)}) \leq 5, \ \mathtt{p(X,Z)}.$$

and the query $\mathtt{q(5)}$?. Then, when propagating the binding $\mathtt{q^b}$ we have that the set of the bound variables is $\{\mathtt{X}\}$; in the SIPS, the predicate $\mathtt{p(X,Z)}$ is preferable to $\mathtt{r(X,Y,Z)}$, and, hence we have the following edge in the SIPS we enforce the following edge:

$$\{\mathtt{q^b(X)}\} \rightarrow_{\{\mathtt{X}\}} \mathtt{p(X,Z)}.$$

Thus, the set of bound variables will include $\mathtt{Z}$, and can be used for propagating the biding to $\mathtt{r}$:

$$\{\mathtt{q^b(X)}, \mathtt{p^{bf}(X,Z)} \rightarrow_{\{\mathtt{X,Z}\}} \mathtt{r(X,Y,Z)}.$$

Finally, the adorned rule is as follows

$$\mathtt{q^b(X)} :- \mathtt{a(X)}, \ 2 \leq \mathit{count}(\mathtt{Y} : \mathtt{r^{bfb}(X,Y,Z)}) \leq 5, \ \mathtt{p^{bf}(X,Z)}.$$

□

Input: A Datalog$^\vee$ program $\mathcal{P}$, and a query $\mathtt{g_1}(\mathtt{t_1}), \ldots, \mathtt{g_n}(\mathtt{t_n})$.
Output: The optimized program $\mathcal{P}'$.
var S: **stack** of adorned predicates; *modifiedRules*,*magicRules*: **set** of rules;
begin
if $\mathtt{g_1}(\mathtt{t_1}), \ldots, \mathtt{g_n}(\mathtt{t_n})$ has some IDB predicate **then**
$\langle adornedQuery, S\rangle$:=***AdornQuery***$(\mathtt{g_1}(\mathtt{t}), \ldots, \mathtt{g_n}(\mathtt{t}), \mathcal{P})$;
$\langle modifiedRules, magicRules\rangle$:=***BuildQueryRules***$(adornedQuery)$;
while $S \neq \emptyset$ **do**
p^α:=S.**pop()**;
`TestForDisjunctiveRule`$(\mathtt{p}^\alpha)$;
`TestForConstraint`$(\mathtt{p}^\alpha)$;
`TestForDangerous`$(\mathtt{p}^\alpha)$;
end while
P':=$magicRules \cup modifiedRules$;
P':=***StripOffAdornment***$(\mathcal{P}')$;
return $\mathcal{P}'$;
end if
end.

Figure 5: Extended Magic-Set Method

The generation of the magic rules has to consider the predicates in $Conj$ other than the aggregate literal in itself. For instance, in the above example, we generate the following set of magic rules:

$$\mathtt{magic_r}^{\mathtt{bfb}}(\mathtt{X},\mathtt{Z}) \text{ :- } \mathtt{magic_q}^{\mathtt{b}}(\mathtt{X}),\ \mathtt{a}(\mathtt{X}).$$
$$\mathtt{magic_p}^{\mathtt{bf}}(\mathtt{Z}) \text{ :- } \mathtt{magic_q}^{\mathtt{b}}(\mathtt{X}),\ \mathtt{a}(\mathtt{X}).$$

Note that the aggregate literal has been not taken into account; moreover, in the second rule the literal $\mathtt{r}^{\mathtt{bfb}}$ has been not inserted as it is useless for the propagation of the binding (as in the standard case).

Finally, the generation of the modified rule remains unchanged:

$$\mathtt{q}^{\mathtt{b}}(\mathtt{X}) :- \mathtt{magic}^{\mathtt{b}}_{\mathtt{q}}(\mathtt{X}),\ \mathtt{a}(\mathtt{X}),\ 2 \leq \mathit{count}(\mathtt{Y} : \mathtt{r}^{\mathtt{bfb}}(\mathtt{X},\mathtt{Y},\mathtt{Z})) \leq 5,\ \mathtt{p}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z}).$$

6 Extension to Arbitrary DLP$^\mathcal{A}$ Programs

In this section we present an extension of the technique for propagating the binding into DLP$^\mathcal{A}$ program with unstratified negation. By exploiting the modularity of our approach in processing each rule at time, this extension do not need to reformulate the algorithms presented so far.

For the sake of clarity we reported in Figure 5 the structure of the whole algorithm, called Extended Magic-Set method (EMS), in which each adorned predicate is used for processing sequentially the disjunctive rules, the constraints and the dangerous rules (whose meaning will be soon clarified).

The procedure `TestForDisjunctiveRule` has been described in the Section 4 and corresponds to the steps 6-12 of the algorithm reported in Figure 1. Thus, we start by

Procedure `TestForConstraint`($\mathtt{p}^\alpha$**: adorned predicate**);
begin
1. **for each** constraint $c \in \mathcal{C}$ of the form $:-\mathtt{p}(\mathtt{t}), \mathtt{q}_1(\mathtt{t}), \ldots, \mathtt{q}_\mathtt{m}(\mathtt{t})$ **do**
2. **let** c_s be the rule $\mathtt{p}(\mathtt{t}) :- \mathtt{p}(\mathtt{t}),\ \mathtt{q}_1(\mathtt{t}), \ldots, \mathtt{q}_\mathtt{m}(\mathtt{t})$;
3. **let** c_a:=***Adorn***(c_s,$\mathtt{p}^\alpha$,S);
4. *magicRules* := *magicRules* $\bigcup$ ***Generate***(r_a);
5. **let** c_m be the constraint with the body of c_a;
6. **for each** adorned predicate $\neg\mathtt{q}_\mathtt{i}^\beta(\mathtt{t})$ in c_a **do**
7. **insert** ***magic***($\mathtt{q}_\mathtt{i}^\beta(\mathtt{t})$) into c_m;
8. *modifiedRules*:=*modifiedRules* $\bigcup \{c_m\}$;
9. **end for**

end;

Figure 6: Modification in the Disjunctive Magic-Set Method

considering the propagation into constraints, i.e., the function `TestForConstraint`.

6.1 Program with Constraints

Despite standard positive DLP$^\mathcal{A}$ programs without constraints, in which we propagate bindings from the head of rules into the body, the problem with constraint rules is that we need the bindings to be propagated also through the constraints. For instance, if we are interested in knowing whether `p(1)` is true in a program where there is the constraint `:- p(X), q(X).`, then we also need to evaluate the truth value of `q(1)`.

Thus, the truth value of each ground atom in a constraint depends on the truth value of the other atoms appearing in the same ground constraint, and, hence, in a more abstract perspective, constraint rules propagate the binding from one predicate in the body to all the other predicates.

The procedure `TestForConstraint`, which is reported in Figure 6 addresses the problem of propagating the binding of a given adorned predicate $\mathtt{p}^\alpha$ into a constraint. We shall next provide more details on this procedure.

Adornment: Given an adorned predicate $\mathtt{p}^\alpha$, and a constraint of $\mathcal{P}$

$$c: \ \ :- \ \mathtt{p}(\mathtt{t}), \mathtt{p}_1(\mathtt{t}), \mathtt{q}_1(\mathtt{t}), \ldots, \mathtt{q}_\mathtt{m}(\mathtt{t}).$$

the idea consist of replacing c, with a rule c_s of the form

$$c_s: \ \mathtt{p}(\mathtt{t}) \ :- \ \mathtt{p}(\mathtt{t}),\ \mathtt{q}_1(\mathtt{t}), \ldots, \mathtt{q}_\mathtt{m}(\mathtt{t}).$$

Note that the above rule can be used for simulating the information passing strategy from the predicate $\mathtt{p}(\mathtt{t})$ to all the other predicates of the constraints, i.e., $\mathtt{q}_1(\mathtt{t}), ..., \mathtt{q}_\mathtt{n}(\mathtt{t})$. Moreover, we point out that the predicate $\mathtt{p}(\mathtt{t})$ is maintained in the body of the rule c_s and is forced to be considered as the first predicate in the adornment, in order to preserve the safety of the rule (a variable may appear only into $\mathtt{p}(\mathtt{t})$), and in order to guarantee a correct application of the SIPS (since no binding is omitted).

The rule c_s can be adorned in the usual manner, by making use of the function ***Adorn***, implementing the adorning step as for described in the previous section, thus, obtaining a rule c_a of the form $\mathtt{p}^{\alpha}(\mathtt{t}) :- \mathtt{p}^{\beta}(\mathtt{t}), \mathtt{q}_1^{\beta_1}(\mathtt{t}), \ldots, \mathtt{q}_m^{\beta_m}(\mathtt{t})$.

Example 6.1 Consider the following constraint

$$\mathtt{:-\ a(Y,V),\ \neg q(Z),\ p(V,Z).}$$

where a is an EDB predicate, while q and p are IDB predicates. Moreover, assume that in the algorithm of Figure 1 we are propagating the binding of the predicate $\mathtt{p}^{\mathtt{bf}}$. The algorithm generates a temporary rule of the form

$$\mathtt{p(V,Z)\ :-\ p(V,Z),\ a(Y,V), \neg q(Y,Z).}$$

that can be adorned as usual, obtaining the rule

$$\mathtt{p}^{\mathtt{bf}}\mathtt{(V,Z)\ :-\ p}^{\mathtt{bf}}\mathtt{(V,Z),\ a(Y,V),\ \neg q}^{\mathtt{bb}}\mathtt{(Y,Z).}$$

that generates in its turn the adorned predicate $\mathtt{q}^{\mathtt{bb}}$. Notice that if $\mathtt{p(V,Z)}$ were not processed as the first predicate, then the adornment would generate the predicate $\mathtt{p}^{\mathtt{bb}}$ after receiving the binding for Z from q. □

Generation: The magic rules can be derived from c_a by means of the function ***Generate***, described in the previous section. Note that in this step the procedure will generate each time a useless rule in which the binding is propagated from $\mathtt{p(t)}$ to itself. For instance, the adorned rule generated in the above example

$$\mathtt{magic_p}^{\mathtt{bf}}\mathtt{(V)\ :-magic_p}^{\mathtt{bf}}\mathtt{(V).}$$
$$\mathtt{magic_q}^{\mathtt{bb}}\mathtt{(Y,Z)\ :-magic_p}^{\mathtt{bf}}\mathtt{(V),\ a(Y,V),\ p}^{\mathtt{bf}}\mathtt{(V,Z).}$$

where the first rule is obviously redundant.

We point out that the creation of this rule is a side effect of the 'trick' of inserting the predicate which is propagating the binding also in the body of the rule, and it is automatically avoided in our implementation.

Modification: In the step *5*, the modified constraint c_m is generated by considering the body of the adorned rule c_a. Nonetheless, in steps *6-7* the algorithm introduces some additional computations, which is needed for solving some semantic problem that could arise in certain situations, as shown in the following example.

Example 6.2 Consider the program $\mathcal{P}$

$$\mathtt{p(X,Y)\ :-\ a(X,Y).}$$
$$\mathtt{:-\ b(X,Y), \neg p(X,Y).}$$

and the query $\mathtt{p(1,2)}$? over the set facts $\mathcal{F} = \{\mathtt{a(1,2),\ a(3,4),\ b(1,2),\ b(3,4)}\}$. Since the only stable model of $\mathcal{P} \cup \mathcal{F}$ is $M = \mathcal{F} \cup \{\mathtt{p(1,2),\ p(3,4)}\}$, the query is true under both the brave and cautious semantics. After the application of the algorithm `TestForConstraint`, we obtain the following rules

$$\mathtt{p}^{\mathtt{bb}}\mathtt{(X,Y)\ :-\ magic_p}^{\mathtt{bb}}\mathtt{(X,Y), a(X,Y.)}$$
$$\mathtt{:-\ b(X,Y), \neg p}^{\mathtt{bb}}\mathtt{(X,Y).}$$

plus the set of rules

$$\texttt{p(X, Y)} \ \texttt{:-}\ \texttt{p}^{\texttt{bb}}\texttt{(X, Y)}.$$
$$\texttt{magic_p}^{\texttt{bb}}\texttt{(1, 2)}.$$

obtained from the rewriting of the query by means of the function ***BuildQueryRules***, described in the previous section. Now we point out that such a program has no stable model; in fact, when considering the predicate $\texttt{b(3, 4)}$ there is no way for satisfying the constraints, since $\texttt{p}^{\texttt{bb}}\texttt{(1, 2)}$ is the only instantiation of $\texttt{p}^{\texttt{bb}}$ that is derived. □

A solution to this problem is that, in the modification step, for each adorned predicate $\neg\texttt{p}^{\alpha}$ in the rule c_a, the corresponding magic predicate is generated by means of the function ***magic*** and appended into the modified constraint c_m.

For instance, in the above example, the rewritten constraint has the form

$$\texttt{:- b(X, Y)}, \neg\texttt{p}^{\texttt{bb}}\texttt{(X, Y)}, \ \texttt{magic_p}^{\texttt{bb}}\texttt{(X, Y)}.$$

and, consequently, the program maintains the original semantics.

6.2 General Unstratified Programs: Dangerous Rules

The propagation of the binding to constraints only is not sufficient for preserving the semantics of the original program in all the cases, as shown in the following example.

Example 6.3 Consider the program $\mathcal{P}$

$$\texttt{p(X)} \ \texttt{:-}\ \texttt{a(X).}$$
$$\texttt{q(X)} \ \texttt{:-}\ \texttt{p(X).}$$
$$\texttt{:-}\ \texttt{q(1).}$$

and the query $\texttt{p(1)}$? over the set of facts $\mathcal{F} = \{\texttt{a(1)}\}$. We point out that $\texttt{p}$ is not involved in any constraint, and that the query is false under brave reasoning since $\texttt{:-q(1)}$ is not satisfied.

Now, let us consider the rewritten program obtained by applying the Magic-Set method. It consists of the rule

$$\texttt{p}^{\texttt{b}}\texttt{(X)} \ \texttt{:-magic_p}^{\texttt{b}}\texttt{(X), a(X).}$$

and the fact $\texttt{magic_p}^{\texttt{b}}\texttt{(1)}$. Notice that the constraint is not present in the rewritten version since there is no way for passing the binding of the query to the predicate $\texttt{q}$. It follows that the original semantics is completely changed. □

The problem with the above example is that, even though $\texttt{p(1)}$ does not occur in any constraint, it directly enables an other constraint by means of the derivation of $\texttt{q(1)}$. Thus, the rule $\texttt{q(X)} \ \texttt{:-}\ \texttt{p(X).}$ acts someway as a constraint, for it directly enabling $\texttt{:-q(1).}$, and, hence, it must be considered for biding propagation into its body.

Roughly, we next propose a technique for propagating the binding not only in the head of the rules, but also in the body of those rules that act as constraints. Throughout this section we assume the dependency graph of $\mathcal{P}$ is given[3].

[3] DLV computes this graph in the grounding phase.

Procedure TestForDangerous(p^α: **adorned predicate**);

begin
1. **for each** dangerous rule d of the form $r_1(t)$ v ... v $r_n(t)$:- $p(t), q_1(t), \ldots, q_m(t)$ **do**
2. **let** d_s be the rule $p(t)$ derives $p(t), q_1(t), ..., q_m(t), r_1(t), ..., r_n(t)$.;
3. **let** d_a:=***Adorn***(d_a,p^α,S);
4. *magicRules* := *magicRules* $\bigcup$ ***Generate***(d_a);
5. **end for**
end;

Figure 7: Modification in the Disjunctive Magic-Set method

Definition 6.4 Let $\mathcal{P}$ be a datalog program, and p be a predicate of $\mathcal{P}$. Then, we say that p is *dangerous* if at least one of the following condition hold:

- p is in the body of a constraint;
- p is in the body of a rule, having an head predicate involved in the dependency graph in a cycle with an odd number of edges;
- p is in the body of a rule, having a *dangerous* head predicate.

Moreover, a rule r is said *dangerous*, if it contains a *dangerous* predicate. □

Note that given a dependency graph, deciding whether a rule is dangerous can be done in polynomial time, by means of a visit in the dependency graph.

Example 6.5 Consider the program

$$\begin{array}{ll} r_1: & p(X) \text{ :- } a(X),\ r(Y,Z),\ b(X,Y). \\ r_2: & r(X,Y) \text{ :- } c(X),\ b(Y). \\ r_3: & q(X) \text{ :- } a(X),\ r(X,Y). \\ c_1: & \text{ :- } q(X). \end{array}$$

and the query $p(2)$? over the facts $\{a(2),\ a(4),\ b(2,4),\ b(4,2),\ c(2),\ c(4)\}$. Then q is a dangerous predicate as it appears into a constraint; r_3 defining q is dangerous, and, hence, r is dangerous in its turn. Thus, the constraint c_1 will be considered for the adornment step, even though there is no way for propagating directly the binding from $p(2)$ to it. □

The function TestForDangerous which manages the presence of dangerous rule in the Magic-Set method has been reported in Figure 7. Given the predicate p^α to be used for the adornment, each dangerous rule d having $p(t)$ in its body is processed as follows. Other then designing a complete new adorning mechanism, we simple reuse the procedures for the case of normal programs; indeed, if the dangerous rule has the form

$$r_1(t) \text{ v } ... \text{ v } r_n(t) \text{ :- } p(t), q_1(t), \ldots, q_m(t).$$

then we construct the rule d_s:

$$p(t) \text{ :- } p(t), q_1(t), ..., q_m(t), r_1(t), ..., r_n(t).$$

Notice that d_s can be adorned in the standard way, and also the magic rules can be generated as usual. An important aspect is that d_s have not to be used for generating a modified rule, as it is only a 'temporary' rule generated for propagating the adornment.

Example 6.6 In the program of Example 6.5, the adornment with $\mathtt{p}^{\mathtt{b}}$ produces the following adorned rules, pushing $\mathtt{r}^{\mathtt{bb}}$ on the stack.

$$\mathtt{p}^{\mathtt{b}}(\mathtt{X}) \ \mathtt{:-}\ \mathtt{a}(\mathtt{X}),\ \mathtt{b}^{\mathtt{bf}}(\mathtt{X},\mathtt{Y}),\ \mathtt{r}^{\mathtt{bb}}(\mathtt{Y},\mathtt{Z}).$$
$$\mathtt{r}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}) \ \mathtt{:-}\ \mathtt{c}(\mathtt{X}),\ \mathtt{b}(\mathtt{Y}).$$

As $\mathtt{r}$ is a dangerous predicate, the rule r_3 must be rewritten as follows:

$$\mathtt{r}(\mathtt{X},\mathtt{Y}) \ \mathtt{:-}\ \mathtt{r}(\mathtt{X},\mathtt{Y}),\ \mathtt{a}(\mathtt{X}),\ \mathtt{q}(\mathtt{X}).$$

and will be adorned as follows:

$$\mathtt{r}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}) \ \mathtt{:-}\ \mathtt{r}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}),\ \mathtt{a}(\mathtt{X}),\ \mathtt{q}^{\mathtt{b}}(\mathtt{X}).$$

causing in its turn the adornment of the constraint with the procedure `TestForConstraint`, generating $\mathtt{:-}\ \mathtt{q}^{\mathtt{b}}(\mathtt{X})$. □

6.3 Query Equivalence Results

The above technique requires a finer analysis for proving its correctness, since in the general case it produces a query that is not equivalent to the original one. The reason is that constraints express conditions that must hold for every ground instance of the program also including the atoms which are not relevant for the query. This observation will be clarified by the following example.

Example 6.7 Consider the simple logic program consisting of the following two rules

$$\mathtt{q}(\mathtt{X}) \ \mathtt{:-}\ \mathtt{a}(\mathtt{X}).$$
$$\mathtt{:-}\ \mathtt{b}(\mathtt{X}).$$

and the query $\mathtt{q}(\mathtt{1})$?, over the set of facts $\{\mathtt{a}(\mathtt{1}),\ \mathtt{b}(\mathtt{2})\}$. Then, under the brave semantics the query is false, while it is true under the cautious semantics, since the program admits no model.

As for the rewritten program, we have

$$\mathtt{q}^{\mathtt{b}}(\mathtt{X}) \ \mathtt{:-}\ \mathtt{magic_q}^{\mathtt{b}}(\mathtt{X}),\ \mathtt{a}(\mathtt{X}).$$

and the magic fact $\mathtt{magic_q}^{\mathtt{b}}(\mathtt{1})$. Note that the constraint is not present since there exists no adorned predicate that may propagate the binding into it, even by considering the extension with the dangerous rules (this is because b is a EDB predicate). Obviously the answer is true under both the semantics. □

Given a query $\mathcal{Q}$ and a program $\mathcal{P}$, it should be now evident that under the brave semantics the evaluation of $\mathcal{Q}$ over the rewritten program $\mathcal{P}'$ may differ from the evaluation over $\mathcal{P}$. In particular it could be the case that a constraint is violated in $\mathcal{P}$ even though it does not appear any more in $\mathcal{P}'$. In order to formalize this intuition, we start by providing some definitions.

Let $\mathcal{P}$ be a program, $\mathcal{Q}$ be a query, and $\mathcal{P}'$ be the rewritten program obtained by the algorithm EMS. Then, it is easy to associate to each ground rule or ground constraint $r'_g \in \mathcal{P}'$ the corresponding ground rule $r_g \in \mathcal{P}$, from which r'_g is derived by means of the modification step. Indeed, we can define a function $\texttt{derived} : \mathit{ground}(\mathcal{P}') \mapsto \mathit{ground}(\mathcal{P})$, such that $r'_g = \texttt{derived}(r_g)$.

Now, the set of relevant rules of the program $\mathcal{P}$ w.r.t. the query $\mathcal{Q}$ is the set $\mathit{relevant}(\mathcal{P}, \mathcal{Q})$ of ground rules and ground constraint of $\mathcal{P}$, such that $\forall r_g \in \mathit{relevant}(\mathcal{P}, \mathcal{Q})$, there exists $r'_g \in \mathcal{P}'$ such that $r'_g = \texttt{derived}(r_g)$.

Definition 6.8 *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ be a query, and $\mathcal{P}'$ be the rewritten program w.r.t. $\mathcal{Q}$. Then, the set $\mathit{compl}(\mathcal{P}, \mathcal{Q}) = \mathit{ground}(\mathcal{P}) - \mathit{relevant}(\mathcal{P}, \mathcal{Q})$ is called the* complementary *program of $\mathcal{P}$ w.r.t. $\mathcal{Q}$.*

For instance, in the Example 6.7, the set $\mathit{compl}(\mathcal{P}, \mathcal{Q})$ consists of the rule $q(2) :- b(2)$ and of the constraint $:- b(2)$.

We notice that the above definition enable us to shed some lights in the comparison of the program $\mathcal{P}$ and $\mathcal{P}'$; indeed, the program $\mathit{compl}(\mathcal{P}, \mathcal{Q})$ has no "influence" on the program $\mathit{relevant}(\mathcal{P}, \mathcal{Q})$, and vice-versa. Now we are in the position to prove our main result.

Theorem 6.9 *Let $\mathcal{P}$ be a program, $\mathcal{Q}$ be a query, and $\mathcal{P}'$ be the rewritten program w.r.t. $\mathcal{Q}$. Then,*

1. *$\mathcal{P}'$ is* brave-complete *for $\mathcal{P}$ w.r.t. $\mathcal{Q}$,*
2. *$\mathcal{P}'$ is* cautious-sound *for $\mathcal{P}$ w.r.t. $\mathcal{Q}$.*

Proof. By definition, $\mathit{ground}(P) = \mathit{relevant}(\mathcal{P}, \mathcal{Q}) \cup \mathit{compl}(\mathcal{P}, \mathcal{Q})$. Moreover, from the construction of $\mathcal{P}_1 = \mathit{relevant}(\mathcal{P}, \mathcal{Q})$ and $\mathcal{P}_2 = \mathit{compl}(\mathcal{P}, \mathcal{Q})$ we have that there is no way for propagating an information from one to the other, when answering to the query $\mathcal{Q}$. Thus, we can evaluate $\mathcal{P}_1$ and $\mathcal{P}_2$ independently, and the set of stable models of $\mathcal{P}$ is such that $\mathcal{SM}(\mathcal{P}) = \{M_1 \cup M_2 |\ M_1 \in \mathcal{SM}(\mathcal{P}_1) \wedge M_2 \in \mathcal{SM}(\mathcal{P}_2)\}$.

Due to the soundness and completeness of the Magic-Set method for positive programs, we have that $\mathcal{SM}(P_1)$ coincides with $\mathcal{SM}(\mathcal{P}')$, i.e., the semantics of the part of $\mathcal{P}$ in which the binding has propagated is exactly the one of the rewritten program.

Now let us consider two possibilities. In the case $\mathcal{SM}(\mathcal{P}_2) \neq \emptyset$, then each stable model of $\mathcal{P}$ contains a stable model of $\mathcal{P}_1$ that can be used for answering the query. Thus, $\mathcal{Q}$ is evaluated in the same manner in $\mathcal{P}$ and in $\mathcal{P}'$ (no matter of the chosen semantics).

Conversely, if $\mathcal{SM}(\mathcal{P}_2) = \emptyset$, then $\mathcal{SM}(\mathcal{P})$ is empty too. Hence, the query $\mathcal{Q}$ is false in $\mathcal{P}$ under the brave semantics, but it is always true under the cautious. Conversely, $\mathcal{Q}$ could be evaluated both true or false in $\mathcal{P}'$.

Since $\mathcal{Q}$ is false in $\mathcal{P}$ under the brave semantics, we do not care of it, for concluding the *brave-complete* for $\mathcal{P}'$, i.e., each time the query $\mathcal{Q}$ is bravely true in $\mathcal{P}$, then it is bravely true in $\mathcal{P}'$ too.

Moreover, since $\mathcal{Q}$ is true under the cautious semantics, we do not care of it, for concluding the *cautious-soundness* for $\mathcal{P}'$, i.e., each time the query $\mathcal{Q}$ is cautiously true in $\mathcal{P}'$, then it is bravely true in $\mathcal{P}$ too. □

Notice that this theorem guarantees that in the INFOMIX setting, i.e., when calculating cautious answers, the magic set method is sound. Conversely, we do not care of completeness, as it is often assumed in data integration systems.

7 Conclusions

In this report we have presented the optimization methods and some smart algorithms in order to efficiently support the highly expressive computational-logic formalism which is needed in INFOMIX as the internal integration formalism for modelling the whole process of query answering in data integration.

Indeed, while an attractive feature of our approach is that logic programs serve as executable logical specifications of repair, and thus allow to state repair policies in a declarative manner rather than in a procedural way, an evident drawback with current implementations of stable model engines, such as DLV or Smodels, is that the evaluation of queries over large data sets quickly becomes infeasible because of lacking scalability.

In this report overcome these deficiencies, by designing a scalable strategy for query answering in which the original program is modified by means of certain rules, acting as filters for the relevant information needed for the query processing, and thus avoiding useless computations.

The technique is the most general appeared so far in the literature as it is applicable to any disjunctive datalog program, even enriched with constraints, general unstratified negation and aggregate functions, thus, fully supporting all the features of $\mathrm{DLP}^{\mathcal{A}}$.

References

[1] 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, 2000.

[2] 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.

[3] Pablo Barceló and Leopoldo Bertossi. Reapairing databases with annotated predicate logic. In *Proc. the 10th Int. Workshop on Non-Monotonic Reasoning (NMR 2002)*, pages 160–170, 2002.

[4] Catriel Beeri and Ragu Ramakrisnhan. On the power of magic. *Journal of Logic Programming*, 10(1–4):255–259, 1991.

[5] Andreas Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–110. ACM Press, 2003.

[6] 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)*, 2003. To appear.

[7] 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)*, 2003. To appear.

[8] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix and Ulrich Furbach and Anil Nerode, editor, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Berlin, 1997. Springer.

[9] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

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

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

[12] 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 Artificial Intelligence*, pages 348–364. Springer, 2001.

[13] 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.

[14] 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. Extended Abstract appeared as [13].

[15] Sergio Greco, Domenico Sacca, and Carlo Zaniolo. The PushDown Method to Optimize Chain Logic Programs (Extended Abstract). In *Proc. Int. Colloquim on Automata, Languages and Programming*, pages 523–534, 1995.

[16] Ashish Gupta and Inderpal Singh Mumick. Magic-sets Transformation in Nonrecursive Systems. In *Proceedings of the Thirteenth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-92)*, pages 354–367, 1992.

[17] 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.

[18] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[19] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Y. Vardi. Logical Query Optimization by Proof-Tree Transformation. *Journal of Computer and System Sciences*, 47(1):222–248, 1993.

[20] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, 1989.