Available online at www.sciencedirect.com

ScienceDirect

Journal of Computer and System Sciences 73 (2007) 584–609

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.elsevier.com/locate/jcss

Magic Sets and their application to data integration [☆]

Wolfgang Faber [*,1], Gianluigi Greco, Nicola Leone

Department of Mathematics, University of Calabria, 87030 Rende, Italy

Received 13 January 2006; received in revised form 23 May 2006

Available online 30 November 2006

Abstract

Recently, effective methods model query-answering in data integration systems and inconsistent databases in terms of cautious reasoning over Datalog$^\neg$ programs under the stable model semantics. Since this task is computationally expensive (co-NP-complete), there is a clear need of suitable techniques for query optimization, in order to make such methods feasible for data-intensive applications.

We propose a generalization of the well-known Magic Sets technique to Datalog$^\neg$ programs with (possibly unstratified) negation under the stable model semantics. Our technique produces a new program whose evaluation is more efficient (due to a smaller instantiation) in general, while preserving full query-equivalence for both brave and cautious reasoning, provided that the original program is consistent. Soundness under cautious reasoning is always guaranteed, even if the original program is inconsistent.

In order to formally prove the correctness of our Magic Sets transformation, we introduce a novel notion of modularity for Datalog$^\neg$ under the stable model semantics, which is more suitable for query answering than previous module definitions. We prove that a query on such a module can be evaluated independently from the rest of the program, while preserving soundness under cautious reasoning. Importantly, for consistent programs, both soundness and completeness are guaranteed for brave reasoning and cautious reasoning.

Our Magic Sets optimization constitutes an effective method for enhancing the performance of data integration systems in which query-answering is carried out by means of cautious reasoning over Datalog$^\neg$ programs. In fact, results of experiments in the EU project INFOMIX, show that Magic Sets are fundamental for the scalability of the system.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Logic programming; Stable models; Data integration; Magic Sets; Answer sets; Modularity

1. Introduction

Datalog$^\neg$ programs are function-free logic programs where negation may occur in the bodies of rules [1]. Datalog$^\neg$ with stable model semantics[2] [2,3] is a very expressive query language in a precise mathematical sense: Under brave

☆ This work was partially supported by MIUR under projects “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione”, “ONTO-DLV: Un ambiente basato sulla Programmazione Logica Disgiuntiva per il trattamento di Ontologie”, and “tocai.it: Tecnologie Orientate alla Conoscenza per Aggregazioni di Imprese in Internet”.

* Corresponding author.
E-mail addresses: wf@wfaber.com (W. Faber), ggreco@mat.unical.it (G. Greco), leone@mat.unical.it (N. Leone).

[1] Funded by an APART grant of the Austrian Academy of Sciences.

[2] Unless explicitly specified, Datalog$^\neg$ will always denote Datalog with negation under stable model semantics in this paper.

0022-0000/$ – see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2006.10.012

(cautious) reasoning[3] Datalog$^\neg$ allows to express every query that is decidable in the complexity class NP (co-NP) [4]. In the 90s, Datalog$^\neg$ was not considered very much in the database community, mainly because of the high complexity of its evaluation. However, recently there has been renewed the interest in this language because of two main factors: The emergence of database applications strictly requiring the co-NP expressiveness of Datalog$^\neg$ (see below and Section 6), along with the availability of a couple of effective Datalog$^\neg$ systems, like DLV [5] and Smodels [6].

Our motivation to study optimization techniques for Datalog$^\neg$ comes from our work within the EU project "INFOMIX: Boosting Information Integration." INFOMIX is a powerful data integration system, which is able to deal with both inconsistent and incomplete information. Following many recent proposals (see, e.g., [7–12]), query answering in the INFOMIX data integration system is reduced to cautious reasoning on Datalog$^\neg$ programs under stable model semantics. This reduction is possible since query answering in data integration systems is co-NP-complete (in our setting and also in many other data integration frameworks [10,12]), and since cautious reasoning on (unstratified) Datalog$^\neg$ programs under the stable model semantics is able to express all problems in co-NP [13].

Dealing with a co-NP-complete problem may appear infeasible in a database setting where input could be very large. However, our present results show that suitable optimization techniques can "localize" the computation and limit the inefficient (co-NP) computation to a fragment of the input, which will be small in many cases, obtaining fast query-answering, even in a powerful data integration framework. The optimization strategy relies on an extension of the well-known Magic Set method [1,14–16] for the class of Datalog$^\neg$ programs. This method exploits the fact that while answering a user query, often only a certain part of the stable models needs to be considered, so there is no need to compute these models in their entirety. Thus, its aim is to focus the instantiation of the program to those ground rules that are really needed to answer the query, by propagating binding information from the query goal into the program rules.

The main contributions of the paper are as follow.

▷ We define the new notions of *independent set* and *module* for Datalog$^\neg$, allowing us to identify program fragments which can be evaluated "independently," disregarding the rest of the program. The new notion of module is crucial for proving the correctness of our Magic Set method. It is related to the splitting sets of [17], and to the modules of [18]; but our notion has stronger semantic properties, which are useful for the computation.

▷ We design an extension of the Magic Set algorithm for general Datalog$^\neg$ programs ($\mathtt{MS}^\neg$ algorithm for short). We show that unstratified negation requires bindings to be propagated also body-to-head in certain cases. Doing that is necessary for rules which may inhibit some query answers. We identify such *dangerous rules* and perform a body-to-head propagation on them in our $\mathtt{MS}^\neg$ method. We prove that the rewriting generated by $\mathtt{MS}^\neg$ is query equivalent to the input program $\mathcal{P}$ (under both brave and cautious semantics), provided that $\mathcal{P}$ is consistent. If the program is inconsistent, soundness under cautious semantics and completeness under brave semantics are still guaranteed by $\mathtt{MS}^\neg$.

▷ We show that our method can be profitably exploited for query optimization in data integration systems, which have to deal with incompleteness and inconsistency. Several approaches based on database repair [7–12] involve a transformation to Datalog$^\neg$. These approaches usually guarantee that the resulting Datalog$^\neg$ program is consistent, and hence $\mathtt{MS}^\neg$ guarantees query equivalence. Note that no previous Magic Set technique is applicable, since these programs are unstratified and are to be evaluated under stable models semantics. We have applied $\mathtt{MS}^\neg$ in the data integration system INFOMIX, and have performed preliminary experiments on a realistic application scenario. The results confirmed the viability and effectiveness of our approach.

▷ We analyze also the complexity of the main computational issues arising in this framework. Here, the key notion is that of dangerous predicate of a program $\mathcal{P}$, cf. Definition 3.1. Checking if a predicate is dangerous is interesting from the viewpoint of the computational complexity: We show that, while checking the first condition of Definition 3.1 is NP-complete, deciding if a predicate is dangerous is tractable, and NL-complete in particular.

The rest of the paper is organized as follows. Section 2 overviews some basic notions on Datalog$^\neg$ programs and relates them to the problem of repairing data integration systems. In Section 3, we formally define a new notion of modularity for Datalog$^\neg$, which overcomes the deficiencies of the existing modularity concepts, and is well-suited for

[3] Note that brave and cautious consequences are also called possible and certain answers, respectively.

query answering in Datalog$^\neg$. This section provides the basic technical machinery needed for proving the correctness of our Magic Set technique for Datalog$^\neg$. In Section 4 we define our Magic Set method for Datalog programs and report on its properties. Relevant complexity issues arising in this framework are addressed in Section 5. An application of our results in a data integration setting is presented in Section 6, while in Section 7 we discuss related work in the literature. In Section 8 we draw our conclusions.

2. Logic-based approaches for querying data integration systems

In this section, we report some preliminaries on Datalog$^\neg$ queries under the stable model semantics, and we give an overview of the role they play in supporting query answering in data integration systems.

2.1. Preliminaries on Datalog$^\neg$ queries

We start with basic definitions for Datalog$^\neg$ queries. We assume the existence of three countable set containing *predicate symbols*, *constant symbols*, and *variable symbols*. The set of variable symbols is assumed to be disjoint from the sets of constant symbols. In this paper, predicate and constant symbols are alphanumeric strings starting with a lowercase character or a digit, while variable symbols are alphanumeric strings starting with an uppercase character. Each predicate symbol is associated with a non-negative integer, referred to as its *arity*. The basic semantic elements are atoms, which are formed of a predicate, constants, and variables. Literals are possibly negated atoms.

Definition 2.1. An *atom* $p(t_1, \dots, t_k)$ is composed of a predicate symbol p of *arity* k and terms $t_1, \dots, t_k$, each of which is either a constant or a variable. A *literal* is either an atom a or its negation not a.

Rules are formed of literals, and may be read as (possibly partial) definitions for the meaning of predicates.

Definition 2.2. A (Datalog$^\neg$) *rule* r is of the form

$$h \ :\!-\ b_1, \dots, b_m, \text{not}\ b_{m+1}, \dots, \text{not}\ b_n. \tag{1}$$

where $h, b_1, \dots, b_n$ are atoms and $0 \leqslant m \leqslant n$.

Given a rule r of the form (1), the atom h is called the *head* of r and is denoted by $H(r)$. Furthermore, the conjunction of literals $b_1, \dots, b_m, \text{not}\ b_{m+1}, \dots, \text{not}\ b_n$ is called the *body* of r, and we denote by $B^+(r)$ the set $\{b_1, \dots, b_m\}$, which is called the *positive body* of r, while $B^-(r)$ denotes the set $\{b_{m+1}, \dots, b_n\}$, called the *negative body* of r. The set of all atoms appearing in the body of r is denoted by $B(r)$, that is, $B(r) = B^+(r) \cup B^-(r)$.

A rule r is *positive* if $B^-(r) = \emptyset$. Given a set of atoms A, let $\text{not}.A = \{\text{not}\ a \mid a \in A\}$. Since the order of body literals does not matter in the semantics considered in this paper, we sometimes denote a rule r as $H(r)\ :\!-\ B^+(r), \text{not}.B^-(r)$. Finally, let $Atoms(r) = \{H(r)\} \cup B(r)$ denote the set of atoms in r, and $Atoms(\mathcal{P}) = \{Atoms(r) \mid r \in \mathcal{P}\}$ for a program $\mathcal{P}$.

A *defining rule* for a predicate p is a rule r whose head predicate is p (i.e., $H(r) = p(t_1, \dots, t_k)$). If, for a rule r, $B(r) = \emptyset$ holds, then the rule is a *fact*, usually denoted as $H(r)$. (the $:\!-$ symbol is omitted). A rule r is *positive* if $B^-(r) = \emptyset$. Throughout the paper, we assume that rules are *safe*, i.e., each variable occurring in the head or negative body of a rule r occurs also in $B^+(r)$ [1].

Programs are simply sets of rules, and may be read as definitions for predicates.

Definition 2.3. A *datalog program with negation* (Datalog$^\neg$ program for short) $\mathcal{P}$ is a finite set of rules. A *datalog program* (Datalog program) $\mathcal{P}$ is a finite set of positive rules.

If all defining rules of a predicate p are facts, then p is an *EDB predicate*; otherwise p is an *IDB predicate*. A set of facts for EDB predicates of a program $\mathcal{P}$ is called an *EDB* (for $\mathcal{P}$).[4] Let $\mathcal{P}$ be a Datalog$^\neg$ program and let $\mathcal{F}$ be an

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

EDB. Then, we denote by $\mathcal{P}_{\mathcal{F}}$ the program $\mathcal{P}_{\mathcal{F}} = \mathcal{P} \cup \mathcal{F}$. Note that this definition allows that a program $\mathcal{P}$ already contains EDB facts, and indeed in the remainder of this section we assume that $\mathcal{P}$ refers to a program containing suitable EDB facts.

Finally, we define the query itself, which may refer to predicates defined by an accompanying program.

Definition 2.4. A *query* $\mathcal{Q}$ is an IDB atom.[5]

Given a query $\mathcal{Q}$ and a program $\mathcal{P}$, we say that a subset $\mathcal{P}_1$ of $\mathcal{P}$ *covers* $\mathcal{Q}$ if all rules of $\mathcal{P}$ defining the predicate of $\mathcal{Q}$ are contained in $\mathcal{P}_1$.

We call an atom, rule, program, or query *ground*, if it does not contain any variables. The *universe* $U_{\mathcal{P}}$ of a program $\mathcal{P}$ is the set of all constants in $\mathcal{P}$, and the *base* $B_{\mathcal{P}}$ of $\mathcal{P}$ be the set of ground atoms constructible from predicates in $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Given a program $\mathcal{P}$, we denote by $Ground(\mathcal{P})$ the set of all the rules obtained by applying to each rule $r \in \mathcal{P}$ all possible substitutions from the variables in r to the set of all the constants occurring in $\mathcal{P}$ (sometimes referred to as the active domain of $\mathcal{P}$).

An *interpretation* for $\mathcal{P}$ is a subset I of $B_{\mathcal{P}}$. Given an interpretation I and a set T of rules, the *restriction of I to T* is $I/_T = I \cap B_T$. In a similar way, the restriction of a set S of interpretations to T is defined as $S/_T = \{I/_T \mid I \in S\}$. A ground atom a is true w.r.t. an interpretation I if $a \in I$; otherwise, it is false. The body of a ground rule r is true w.r.t. I if all atoms in $B^+(r)$ are true w.r.t. I and all atoms in $B^-(r)$ are false w.r.t. I. An interpretation I *satisfies* a ground rule $r \in Ground(\mathcal{P})$ if the head of r is true w.r.t. I whenever the body of r is true w.r.t. I. An interpretation I is a *model* of a Datalog$^\neg$ program $\mathcal{P}$ if I satisfies all rules in $Ground(\mathcal{P})$.

Each (positive) Datalog program $\mathcal{P}$ has a least (under subset inclusion) model, which is denoted by LM($\mathcal{P}$) and is the unique *stable model* of $\mathcal{P}$.

For programs with negation, the definition of stable model relies on the notion of Gelfond–Lifschitz transformation [19].

Definition 2.5. Given a Datalog$^\neg$ program $\mathcal{P}$ and an interpretation I, the *Gelfond–Lifschitz transform* $\mathcal{P}^I$ is defined as $\{H(r) \text{ :- } B^+(r) \mid r \in Ground(\mathcal{P}) : I \cap B^-(r) = \emptyset\}$.

Based on this transformation, we define the intended meaning of a program $\mathcal{P}$, its stable models.

Definition 2.6. The set of *stable models*[6] of a Datalog$^\neg$ program $\mathcal{P}$, denoted by SM($\mathcal{P}$), is the set of interpretations I, such that $I = \text{LM}(\mathcal{P}^I)$.

A program $\mathcal{P}$ is *consistent* if SM($\mathcal{P}$) $\neq \emptyset$, otherwise it is *inconsistent*. A program $\mathcal{P}$ is *data consistent* if $\mathcal{P}_{\mathcal{F}}$ is consistent for each EDB $\mathcal{F}$.

Definition 2.7. Given a ground atom a and a Datalog$^\neg$ program $\mathcal{P}$, a is a *cautious* (*or certain*) *consequence* of $\mathcal{P}$, denoted by $\mathcal{P} \models_c a$, if $\forall M \in \text{SM}(\mathcal{P})$: $a \in M$; a is a *brave* (*or possible*) *consequence* of $\mathcal{P}$, denoted by $\mathcal{P} \models_b a$, if $\exists M \in \text{SM}(\mathcal{P})$: $a \in M$.

Given a query $\mathcal{Q}$, $Ans_c(\mathcal{Q}, \mathcal{P})$ denotes the set of substitutions ϑ, such that $\mathcal{P} \models_c \mathcal{Q}\vartheta$; $Ans_b(\mathcal{Q}, \mathcal{P})$ denotes the set of substitutions ϑ, such that $\mathcal{P} \models_b \mathcal{Q}\vartheta$.

Example 2.8. Consider the following program $\mathcal{P}^a$:

```
a(X, Y) :- f(X, Y), not b(X, Y).     b(X, Y) :- f(X, Y), not a(X, Y).
c(X, Y) :- a(X, Y).                  c(X, Y) :- b(X, Y).
```

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

[6] Stable models of Datalog$^\neg$ programs are also called *answer sets* in the literature.

and EDB $\mathcal{F}_a = \{\texttt{f(k, l)}\}$. Then, $Ground(\mathcal{P}^a_{\mathcal{F}_a})$ is

$$\begin{array}{lll} \texttt{a(k, l) :- f(k, l), not b(k, l).} & \texttt{b(k, l) :- f(k, l), not a(k, l).} & \\ \texttt{c(k, l) :- a(k, l).} & \texttt{c(k, l) :- b(k, l).} & \texttt{f(k, l).} \end{array}$$

and the stable models of $\mathcal{P}^a_{\mathcal{F}_a}$ are $\{\texttt{f(k, l), a(k, l), c(k, l)}\}$ and $\{\texttt{f(k, l), b(k, l), c(k, l)}\}$.

We obtain that $\mathcal{P}^a_{\mathcal{F}_a} \models_c \texttt{c(k, l)}$, but $\mathcal{P}^a_{\mathcal{F}_a} \not\models_c \texttt{a(k, l)}$ and $\mathcal{P}^a_{\mathcal{F}_a} \not\models_c \texttt{b(k, l)}$. On the other hand, $\mathcal{P}^a_{\mathcal{F}_a} \models_b \texttt{c(k, l)}$, $\mathcal{P}^a_{\mathcal{F}_a} \models_b \texttt{a(k, l)}$ and $\mathcal{P}^a_{\mathcal{F}_a} \models_b \texttt{b(k, l)}$ all hold. Hence, $Ans_c(\texttt{a(W, X)}, \mathcal{P}^a_{\mathcal{F}_a}) = \emptyset$, $Ans_b(\texttt{a(W, X)}, \mathcal{P}^a_{\mathcal{F}_a}) = \{\{\texttt{W/k, X/l}\}\}$, $Ans_c(\texttt{c(W, X)}, \mathcal{P}^a_{\mathcal{F}_a}) = \{\{\texttt{W/k, X/l}\}\}$, $Ans_b(\texttt{c(W, X)}, \mathcal{P}^a_{\mathcal{F}_a}) = \{\{\texttt{W/k, X/l}\}\}$.

Definition 2.9. Let $\mathcal{P}$ and $\mathcal{P}'$ be Datalog$^\neg$ programs and $\mathcal{Q}$ be a query. Then, $\mathcal{P}$ is *brave-sound* w.r.t. $\mathcal{P}'$ and $\mathcal{Q}$, denoted $\mathcal{P} \subseteq^b_{\mathcal{Q}} \mathcal{P}'$, if $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) \subseteq Ans_b(\mathcal{Q}, \mathcal{P}'_{\mathcal{F}})$ is guaranteed for each EDB $\mathcal{F}$; $\mathcal{P}$ is *cautious-sound* w.r.t. $\mathcal{P}'$ and $\mathcal{Q}$, denoted $\mathcal{P} \subseteq^c_{\mathcal{Q}} \mathcal{P}'$, if $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) \subseteq Ans_c(\mathcal{Q}, \mathcal{P}'_{\mathcal{F}})$ for all $\mathcal{F}$.

$\mathcal{P}$ is *brave-complete* (respectively, *cautious-complete*) w.r.t. $\mathcal{P}'$ and $\mathcal{Q}$, denoted $\mathcal{P} \supseteq^b_{\mathcal{Q}} \mathcal{P}'$ (respectively, $\mathcal{P} \supseteq^c_{\mathcal{Q}} \mathcal{P}'$) if $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) \supseteq Ans_b(\mathcal{Q}, \mathcal{P}'_{\mathcal{F}})$ (respectively, $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) \supseteq Ans_c(\mathcal{Q}, \mathcal{P}'_{\mathcal{F}})$).

Finally, $\mathcal{P}$ and P' are *brave-equivalent* (respectively, *cautious-equivalent*) w.r.t. $\mathcal{Q}$, denoted by $\mathcal{P} \equiv^b_{\mathcal{Q}} \mathcal{P}'$ (respectively, $\mathcal{P} \equiv^c_{\mathcal{Q}} \mathcal{P}'$), if $\mathcal{P} \subseteq^b_{\mathcal{Q}} \mathcal{P}'$ and $\mathcal{P} \supseteq^b_{\mathcal{Q}} \mathcal{P}'$ (respectively, $\mathcal{P} \subseteq^c_{\mathcal{Q}} \mathcal{P}'$ and $\mathcal{P} \supseteq^c_{\mathcal{Q}} \mathcal{P}'$).

With every program $\mathcal{P}$, we associate a marked directed graph $DG_{\mathcal{P}} = (N, E)$, called the *predicate dependency graph* of $\mathcal{P}$, where (i) each predicate of $\mathcal{P}$ is a node in N, and (ii) there is an arc (a, b) in E directed from node a to node b if there is a rule $r \in \mathcal{P}$ such that two predicates a and b of literals appear in $B(r)$ and $H(r)$, respectively. Such an arc is marked if a appears in $B^-(r)$. A *cycle* of $DG_{\mathcal{P}}$ is a sequence of nodes $C = n_1, \ldots, n_k$, such that each n_i $(1 < i < k)$ occurs exactly once in C, $n_1 = n_k$, and each (n_i, n_{i+1}) $(1 \leqslant i < k)$ is an arc in $DG_{\mathcal{P}}$. An *odd cycle* in $DG_{\mathcal{P}}$ is a cycle $C = n_1, \ldots, n_k$ such that an odd number of the arcs (n_i, n_{i+1}) $(1 \leqslant i < k)$ is marked. In analogy, one can also define the *atom dependency graph* $DG^A_{\mathcal{P}}$ of a ground program $\mathcal{P}$, by considering atoms rather than predicates.

2.2. Data integration and logic programming

Data integration systems offer transparent access to different sources by providing users with the so-called *global schema*, which users can query in order to extract data relevant for their aims. Then, the systems are in charge of accessing each source separately, and combining local results into the global answer, by using a set of *mapping assertions*, specifying the relationship between the sources and the global schema (cf. [20]). Usually, the global schema $\mathcal{G}$ is relational enriched with integrity constraints; mapping assertions are specified over the (relational) source schema in a language which often amounts to a fragment of Datalog; and, user queries are assumed to be *conjunctive queries*, i.e., select-project-join SQL queries, over the vocabulary of $\mathcal{G}$.

The main semantical issue in data integration systems is that, since integrated sources are originally autonomous, their data, filtered through the mapping, are likely not to satisfy the constraints on the global schema.

Example 2.10. Consider the global schema $\mathcal{G}_p$ consisting of the relation `person(SSN, Name)`, for which the social security number (`SSN`) is a *key*. Assume that `person(SSN, Name)` is populated by integrating data coming from sources $\texttt{s}_1$ and $\texttt{s}_2$, so that the following tuples are associated with $\mathcal{G}_P$: {`person(‘24832’, ‘John’)`, `person(‘15673’, ‘Mark’)`, `person(‘15673’, ‘Nick’)`}. Specifically, the first two tuples are retrieved from $\texttt{s}_1$, while the latter is retrieved from $\texttt{s}_2$. It can be noticed that the resulting database is “inconsistent,” since two different persons are now associated with the number `‘15673’`.

An approach to remedy to this problem that has lately received a lot of interest in the literature (see, e.g., [10,11, 21–28]) is based on the notion of *repair* for an inconsistent database as introduced in [29]. Roughly speaking, a repair of a database is a new database that satisfies the constraints in the schema, and minimally differs from the original one.

Example 2.11. There are two alternative ways of removing the inconsistency in the database of Example 2.10, leading to two repairs: R_1 consisting of the facts {person('24832', 'John'), person('15673', 'Mark')} and R_2 consisting of the facts {person('24832', 'John'), person('15673', 'Nick')}.

Since multiple repairs might be singled out for an inconsistent database, the standard approach in answering user queries is to compute those answers that are true in every possible repair, called *consistent answers* in the literature.

As an example, 'John' is the only consistent answer to the query Q_1 over $\mathcal{G}_p$ asking for the names of all the persons registered in the database; on the other hand, both '15673' and '24832' are answers to the query Q_2 asking for all the social security numbers. From this, we can identify two crucial aspects of query answering in data integration systems:

- First, the repair approach will retrieve as much information as possible from an inconsistent database. Indeed, even though '15673' is involved in a key conflict, it can still be returned as an answer to Q_2 since this social security number occurs in both R_1 and R_2.
- Second, the semantics of consistent query answering does not boil down to techniques where all the conflicts are deleted from the database (in which case, in fact, the answer to Q_2 would only be '24832').

In the light of the observations above, data integration systems have to be designed to compute *all* of the possible repairs. Therefore, it is not surprising that computing consistent query answers for conjunctive queries is harder (formally, co-NP-complete) than computing answers to conjunctive queries in absence of conflicts (polynomial time). Because of this intrinsic complexity, the database community has recently renewed the interest in Datalog$^\neg$ programs under the stable model semantics, and several techniques have been proposed which formalize the repair semantics in this logic framework [7–12]. Indeed, it is well known that Datalog$^\neg$ programs under stable model semantics are able to express all problems in co-NP [13]. Specifically, the idea common to these works is to encode the constraints of the global schema $\mathcal{G}$ into a Datalog$^\neg$ program Π, such that the stable models of this program yield the repairs of the database retrieved from the sources.

Example 2.12. To have some intuition on how logic programs may encode the repair semantics, consider again Example 2.10 and the following program:

$$\texttt{person}(X,Y) \text{ :- } \texttt{person}(X,Z),\ \text{not } \overline{\texttt{person}}(X,Z).$$
$$\overline{\texttt{person}}(X,Y) \text{ :- } \texttt{s}_1(X,Y),\ \texttt{person}(X,Y1),\ Y \neq Y1.$$
$$\overline{\texttt{person}}(X,Y) \text{ :- } \texttt{s}_2(X,Y),\ \texttt{person}(X,Y1),\ Y \neq Y1.$$

When evaluated over {s_1('24832', 'John'), s_1('15673', 'Mark'), s_2('15673', 'Nick')}, the program has two stable models that are in one-to-one correspondence with the repairs R_1 and R_2.

We would like to stress again that Datalog$^\neg$ programs are *not* used as a query language for data integration systems. Indeed, the systems are assumed to be queried by means of conjunctive queries (i.e., the select-project-join fragment of standard SQL). However, Datalog$^\neg$ programs can be used for *encoding* repair computation, as in the example above. In this way, consistent answering of a conjunctive query over the data integration system coincides with cautious reasoning over a corresponding Datalog$^\neg$ query. We will describe, in more details, such a transformation in Section 6.2. Transformations of this kind are attractive as they provide a specification of repair semantics, which is executable on top of stable model engines, such as DLV [5] or Smodels [6].

3. Dangerous rules and modules

For ensuring query equivalence of subprograms in the negation-free or stratified settings, it is sufficient to single out the rules which are relevant for the query by following the head-to-body direction. Negation under the stable semantics also gives rise to (partial) inconsistency, which may be triggered by activating an inconsistent part of the program in the body-to-head direction. A suitable notion of module, which is aimed at ensuring query equivalence also in presence of unstratified negation, should take into account the possible activation of inconsistent parts of the program in the body-to-head direction. To this end we will first present a way to identify the parts of the program which might give rise to inconsistencies.

Definition 3.1. Let $\mathcal{P}$ be a program (respectively, ground program), and d be a predicate (respectively, atom) of $\mathcal{P}$. Then, we say that d is *dangerous* if either

(1) d occurs in an odd cycle of $DG_{\mathcal{P}}$ (respectively, $DG^A_{\mathcal{P}}$), or
(2) d occurs in the body of a rule with a dangerous head predicate (respectively, atom).

A rule r is *dangerous*, if it contains a dangerous predicate (respectively, atom) in the head.

In principle, one can differentiate between conditional (that depend on the truth values of atoms) and unconditional (independent of interpretations) sources of inconsistencies. In the approach we present here, we are concerned with the first type—and, we will focus on ground programs. In particular, "isolated" (i.e., unconditional) inconsistencies are not covered, though one could easily come up with a modified definition to account also for these. Intuitively, an *independent atom set* of a ground program $\mathcal{P}$ is a set S of atoms whose semantics is not affected (apart from unconditional inconsistencies) by the remaining atoms of $\mathcal{P}$, and can therefore be evaluated by disregarding the other atoms. Independent atom sets induce a corresponding module of $\mathcal{P}$.

Definition 3.2. An *independent atom set* of a ground program $\mathcal{P}$ is a set $S \subseteq B_{\mathcal{P}}$ such that for each atom $a \in S$ the following holds:

(1) if $a = H(r)$ for a rule $r \in \mathcal{P}$ then $Atoms(r) \subseteq S$, and
(2) if a appears in the body of a dangerous rule $r \in \mathcal{P}$ then $Atoms(r) \subseteq S$.

A subset T of a program $\mathcal{P}$ is a *module* if $T = \{r \mid H(r) \in S\}$ for some independent atom set S.

Note that, intuitively, the first condition selects relevant atoms in the usual body-to-head direction; while the second condition follows a body-to-head direction.

Example 3.3. Consider the following program $\mathcal{P}^1$:

$$z \text{ :- } y, \text{not } z. \quad y \text{ :- } q. \quad p \text{ :- not } q. \quad q \text{ :- not } p. \quad a \text{ :- } p, \text{not } b. \quad b \text{ :- } p, \text{not } a.$$

Independent sets for $\mathcal{P}^1$ are $\{p, q, y, z\}$, $\emptyset$ and $\{p, q, y, z, a, b\}$, of which the first is the only non-trivial one. The corresponding module T of $\mathcal{P}^1$ is

$$z \text{ :- } y, \text{not } z. \quad y \text{ :- } q. \quad p \text{ :- not } q. \quad q \text{ :- not } p.$$

Note that all of these rules are dangerous.

We next state the relationships between stable models of a program and its modules.

Theorem 3.4. *Let T be a module of a ground program $\mathcal{P}$, then given an arbitrary EDB $\mathcal{F}$, the following holds*:

(1) $\mathrm{SM}(\mathcal{P}_{\mathcal{F}})/_{T_{\mathcal{F}}} \subseteq \mathrm{SM}(T_{\mathcal{F}})$, *and*
(2) $\mathrm{SM}(T_{\mathcal{F}}) = \mathrm{SM}(\mathcal{P}_{\mathcal{F}})/_{T_{\mathcal{F}}}$, *if $\mathcal{P}_{\mathcal{F}}$ is consistent.*

Proof. Item (1) follows directly from Lemma 5.1 of [18] (because each module T of a program $\mathcal{P}$ is also a potential use module of [18], denoted by $\mathcal{P} \setminus T \rhd T$).

(2) Since (1) holds also for consistent programs, what remains to show is $\mathrm{SM}(T_{\mathcal{F}}) \subseteq \mathrm{SM}(\mathcal{P}_{\mathcal{F}})/_{T_{\mathcal{F}}}$ for consistent $\mathcal{P}_{\mathcal{F}}$.

By virtue of Lemma 5.1 of [18] (stating that $\mathrm{SM}(\mathcal{P}_{\mathcal{F}}) = \bigcup_{M \in \mathrm{SM}(\mathcal{P}_{1\mathcal{F}})} \mathrm{SM}(\{a. \mid a \in M\} \cup \mathcal{P}_2))$, for any EDB $\mathcal{F}$, we have $\mathrm{SM}(\mathcal{P}_{\mathcal{F}}) = \bigcup_{M \in \mathrm{SM}(T_{\mathcal{F}})} \mathrm{SM}(\{a. \mid a \in M\} \cup (\mathcal{P} \setminus T))$.

Note now that each odd cycle in $DG^A_{\mathcal{P}\setminus T}$ is independent from T in $DG^A_{\mathcal{P}}$, i.e. no node corresponding to an atom in T is reachable from the odd cycle in $DG^A_{\mathcal{P}\setminus T}$, and the odd cycle in $DG^A_{\mathcal{P}\setminus T}$ cannot be reached from any node in T. Indeed, if this was not the case, then if a node corresponding to an atom in T was reachable from the odd cycle in $DG^A_{\mathcal{P}\setminus T}$, then by Definition 3.2 also the rules forming the odd cycle would be in the underlying independent atom set; if the odd cycle in $DG^A_{\mathcal{P}\setminus T}$ could be reached from a node in T, then this node of T would be dangerous by Definition 3.1, and via Definition 3.2 all "linking" atoms and the odd cycle itself would have to be included because they are also dangerous.

Let C denote the *component* containing all odd cycles in $DG^A_{\mathcal{P}\setminus T}$, i.e. C contains all rules the head of which is in an odd cycle, reachable from an odd cycle or reaches an odd cycle. Note that $Atoms(C) \cap Atoms(\mathcal{P} \setminus C) = \emptyset$, so adding facts made from atoms of $\mathcal{P} \setminus C$ cannot change (other than including the additional atoms) or invalidate $\mathrm{SM}(C_{\mathcal{F}})$. So, for any EDB $\mathcal{F}$, we can decompose any stable model of $\mathcal{P}_{\mathcal{F}}$ into its "C-part" and its "non-C-part": $\mathrm{SM}(\mathcal{P}_{\mathcal{F}}) = \{S \cup S_c \mid S \in \mathrm{SM}((\mathcal{P} \setminus C) \cup \mathcal{F}),\ S_c \in \mathrm{SM}(C \cup \mathcal{F})\}$ (C is independent from S w.r.t. stable models). Note that this decomposition implies that if any of $(\mathcal{P} \setminus C) \cup \mathcal{F}$ or $C \cup \mathcal{F}$ admit no stable model, also $\mathcal{P}_{\mathcal{F}}$ does not. Since by hypothesis $\mathrm{SM}(\mathcal{P}_{\mathcal{F}}) \neq \emptyset$, it follows thus that $\mathrm{SM}(C \cup \mathcal{F}) \neq \emptyset$.

Given any $M \in \mathrm{SM}(T_{\mathcal{F}})$, as a special case of the reasoning above, we obtain also $\mathrm{SM}(\{a. \mid a \in M\} \cup (\mathcal{P} \setminus T)) = \{S \cup S_c \mid S \in \mathrm{SM}(\{a. \mid a \in M\} \cup ((\mathcal{P} \setminus T) \setminus C) \cup \mathcal{F}), S_c \in \mathrm{SM}(C \cup \mathcal{F})\}$ (note that $\mathcal{F} \subseteq \{a. \mid a \in M\}$). The program $\{a. \mid a \in M\} \cup ((\mathcal{P} \setminus T) \setminus C) \cup \mathcal{F}$ does not contain any odd cycle, and by results of [30] is consistent, i.e. it admits at least one stable model.

As a consequence, for any $M \in \mathrm{SM}(T_{\mathcal{F}})$ a stable model $S \in \mathrm{SM}(\{a. \mid a \in M\} \cup (\mathcal{P} \setminus T) \cup \mathcal{F})$ exists such that $S/_{T_{\mathcal{F}}} = M$, from which $\mathrm{SM}(T_{\mathcal{F}}) = \mathrm{SM}(\mathcal{P}_{\mathcal{F}})/_{T_{\mathcal{F}}}$ follows. □

Thus, each stable model of $\mathcal{P}_{\mathcal{F}}$, restricted to the alphabet of module T, is a stable model of $T_{\mathcal{F}}$. Moreover, if $\mathcal{P}_{\mathcal{F}}$ is consistent, then the stable models of $T_{\mathcal{F}}$ are precisely the stable models of $\mathcal{P}_{\mathcal{F}}$ restricted to T.

Corollary 3.5. *Let T be a module of a ground program $\mathcal{P}$ and $\mathcal{F}$ be an EDB. If $\mathcal{P}_{\mathcal{F}}$ is consistent, each stable model of $\mathcal{P}_{\mathcal{F}}$ can be obtained by enlarging a stable model of $T_{\mathcal{F}}$.*

Proof. From Theorem 3.4 we know that $\mathrm{SM}(T_{\mathcal{F}}) = \mathrm{SM}(\mathcal{P}_{\mathcal{F}})/_{T_{\mathcal{F}}}$ holds. So for each $S \in \mathrm{SM}(\mathcal{P}_{\mathcal{F}})$ it holds that $S/_{T_{\mathcal{F}}} = S \cap Atoms(T_{\mathcal{F}}) \in \mathrm{SM}(T_{\mathcal{F}})$ and therefore $S = S/_{T_{\mathcal{F}}} \cup Y$ for some Y. □

Both brave and cautious queries can be answered using a module covering it, while preserving soundness and completeness if the program is consistent.

Theorem 3.6. *Given query $\mathcal{Q}$, which is covered by module T of a ground program $\mathcal{P}$, and an EDB $\mathcal{F}$, such that $\mathcal{P}_{\mathcal{F}}$ is consistent, then it holds that*:

(1) $Ans_b(\mathcal{Q}, T_{\mathcal{F}}) = Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, *and*
(2) $Ans_c(\mathcal{Q}, T_{\mathcal{F}}) = Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$.

Proof. Since $\mathrm{SM}(\mathcal{P}_{\mathcal{F}}) \neq \emptyset$, the set of brave consequences of $\mathcal{P}_{\mathcal{F}}$ is $\bigcup_{s \in \mathrm{SM}(\mathcal{P}_{\mathcal{F}})} s$ and the set of cautious consequences of $\mathcal{P}_{\mathcal{F}}$ is $\bigcap_{s \in \mathrm{SM}(\mathcal{P}_{\mathcal{F}})} s$. Because of Theorem 3.4 we know that $\mathrm{SM}(T_{\mathcal{F}}) = \mathrm{SM}(\mathcal{P}_{\mathcal{F}})/_{T_{\mathcal{F}}}$. Therefore also $\bigcup_{s \in \mathrm{SM}(T_{\mathcal{F}})} s = (\bigcup_{s \in \mathrm{SM}(\mathcal{P}_{\mathcal{F}})} s)/_{T_{\mathcal{F}}}$ and $\bigcap_{s \in \mathrm{SM}(T_{\mathcal{F}})} s = (\bigcap_{s \in \mathrm{SM}(\mathcal{P}_{\mathcal{F}})} s)/_{T_{\mathcal{F}}}$. This means that brave and cautious consequences coincide for $T_{\mathcal{F}}$ and $\mathcal{P}_{\mathcal{F}}$ on the atoms defined by T, and since all atoms with the predicate of $\mathcal{Q}$ belong to the module T, we obtain $Ans_b(\mathcal{Q}, T_{\mathcal{F}}) = Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$ and $Ans_c(\mathcal{Q}, T_{\mathcal{F}}) = Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$. □

Answering a query using a covering module is always brave-complete and cautious-sound. For data consistent programs, full query equivalence is guaranteed.

Theorem 3.7. *Given query $\mathcal{Q}$, which is covered by module T of a ground program $\mathcal{P}$, it holds that*:

(1) $T \supseteq^b_{\mathcal{Q}} \mathcal{P}$ *and* $T \subseteq^c_{\mathcal{Q}} \mathcal{P}$;
(2) $T \equiv^b_{\mathcal{Q}} \mathcal{P}$ *and* $T \equiv^c_{\mathcal{Q}} \mathcal{P}$, *if $\mathcal{P}$ is data consistent.*

Proof. Let F be an arbitrary EDB. For (1), we have to show that $\mathcal{P}_\mathcal{F} \models_b \mathcal{Q}\theta$ implies $T_\mathcal{F} \models_b \mathcal{Q}\theta$ and $T_\mathcal{F} \models_c \mathcal{Q}\theta$ implies $\mathcal{P}_\mathcal{F} \models_b \mathcal{Q}\theta$ for any substitution θ. If $\mathrm{SM}(\mathcal{P}_\mathcal{F}) \neq \emptyset$, these implications follow from item (1) and item (2) of Theorem 3.6. If $\mathrm{SM}(\mathcal{P}_\mathcal{F}) = \emptyset$ then $\mathcal{P}_\mathcal{F} \models_c \mathcal{Q}\theta$ for any $\mathcal{Q}\theta \in \mathit{Atoms}(T)$, while $\mathcal{P}_\mathcal{F} \models_b \mathcal{Q}\theta$ for no $\mathcal{Q}\theta \in \mathit{Atoms}(T)$. Therefore in this case, the implications are trivially satisfied.

For (2) observe that $T \equiv^b_\mathcal{Q} \mathcal{P}$ holds if $\mathit{Ans}_b(\mathcal{Q}, T_{\mathcal{F}'}) = \mathit{Ans}_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}'})$ is true for each EDB $\mathcal{F}'$, whereas $T \equiv^c_\mathcal{Q} \mathcal{P}$ holds if $\mathit{Ans}_c(\mathcal{Q}, T_{\mathcal{F}'}) = \mathit{Ans}_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}'})$ is true for each EDB $\mathcal{F}'$. By hypothesis, for any such $\mathcal{F}'$, $\mathrm{SM}(\mathcal{P}_{\mathcal{F}'}) \neq \emptyset$ holds, and therefore the results follow from item (1) and item (2) of Theorem 3.6. □

We remark that the benefits of our notion of module rely on the property that programs without odd cycles are guaranteed to be consistent (cf. the proof of Theorem 3.4). One can generalize our notion of module to use any property Ψ, which guarantees consistency of a program. In particular, one would update the notion of dangerous predicates (or atoms) and rules, such that it refers to (the predicates defined by) minimal subprograms for which Ψ does not hold, augmented with all rules it depends on.

4. Magic Set method for Datalog$^\neg$ programs

In this section we present the Magic Set algorithm for general non-ground Datalog$^\neg$ programs ($\mathtt{MS}^\neg$ algorithm for short). After briefly recalling the Magic Set algorithm for positive Datalog queries, we discuss the key issues arising when dealing with Datalog$^\neg$ programs with unstratified negation. We then present the resulting $\mathtt{MS}^\neg$ method, and we show some query equivalence results.

4.1. *Datalog programs*

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 computation to what is relevant for answering the query. We next provide a brief and informal description of the Magic Set rewriting technique. The reader is referred to [31] for a detailed presentation. The method is structured in four main phases, which are summarized in Fig. 1, and which are informally illustrated below by example, considering the query `path(1, 5)` on the following program:

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

Adornment (steps 1–6): The key idea is to materialize, by suitable *adornments*, binding information for IDB predicates which would be propagated during a top-down computation. Adornments are strings of the letters `b` and `f`, denoting “bound” and “free,” respectively, for each argument of an IDB predicate. First, adornments are created for

```
Input: A Datalog program $\mathcal{P}$, and a query $\mathcal{Q} = \mathtt{g}(\bar{\mathtt{t}})$.
Output: The optimized program $\mathtt{MS}(\mathcal{Q}, \mathcal{P})$.
var $S$: stack of adorned predicates; adornedRules, modifiedRules, magicRules: set of rules;
begin
 1.  modifiedRules := $\emptyset$; magicRules := BuildQuerySeeds($\mathcal{Q}$, $S$);
 2.  while $S \neq \emptyset$ do
 3.     $p^\alpha := S$.pop();
 4.     for each rule $r \in \mathcal{P}$ with $H(r) = p(\bar{t}_p)$ do
 5.        adornedRules := adornedRules $\cup$ Adorn($r$, $p^\alpha$, $S$);
 6.  end while
 7.  for each rule $r_a \in$ adornedRules do
 8.     magicRules := magicRules $\cup$ Generate($r_a$);
 9.  for each rule $r_a \in$ adornedRules do
 10.    modifiedRules := modifiedRules $\cup$ {Modify($r_a$)};
 11. $\mathtt{MS}(\mathcal{Q}, \mathcal{P})$ := magicRules $\cup$ modifiedRules;
 12. return $\mathtt{MS}(\mathcal{Q}, \mathcal{P})$;
end.
```

Fig. 1. Magic Set algorithm for Datalog programs.

query predicates so that an argument occurring in the query is adorned with the letter b if it is a constant, or with the letter f if it is a variable. This is carried out by the function ***BuildQuerySeeds***$(\mathcal{Q}, S)$, which pushes on a stack S the newly adorned predicates. The function also produces some *magicRules* whose need will be explained later. Then, each adorned predicate is used to propagate its information into the body of the rules defining it, thereby simulating a top-down evaluation. This is carried out by the function ***Adorn***(r, p^α, S). We note that adorning a rule may generate new adorned predicates, which are pushed on the stack S by ***Adorn***(r, p^α, S). Thus, the adornment step is repeated until all adorned predicates have been processed, yielding the *adorned program*.

For simplicity of the presentation, we next adopt the "basic" Magic Set method as defined in [14], in which binding information within a rule comes only from the adornment of the head predicate, from EDB predicates in the (positive) rule body, and from constants. In other words, an adornment of type b is induced by a constant, or by a variable occurring either as an argument in a position of type b in the head predicate or in an EDB predicate. It is worthwhile noting that this way of passing bindings is not crucial for our method, and it can be adapted to follow the so-called "generalized" Magic Set method [15]. In this enhanced setting, bindings may also be generated by IDB predicates in rule bodies. In particular, an appropriate strategy for Sideways Information Passing (*SIP*) has to be specified for each rule, fixing the body ordering and the way in which bindings are generated. In this respect, the "basic" method we rely on in this work uses a particular, predetermined SIP for all rules.

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

$\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- edge(X, Y).
$\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- edge(X, Z), $\mathtt{path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$.

Generation (steps 7–8): The adorned program is used to generate *magic rules*, which simulate the top-down evaluation scheme and single out the atoms which are relevant for deriving the input query. Let the *magic version* ***magic***$(\mathtt{p}^\alpha(\bar{v}))$ for an adorned atom $\mathtt{p}^\alpha(\bar{v})$ be defined as the atom $\mathtt{magic\_p}^\alpha(\bar{v}')$, where $\bar{v}'$ is obtained from $\bar{v}$ by eliminating all arguments corresponding to an f label in α, and where $\mathtt{magic\_p}^\alpha$ is a new predicate symbol obtained by attaching the prefix "magic_" to the predicate symbol $\mathtt{p}^\alpha$. Then, for each adorned atom B in the body of an adorned rule r_a, a magic rule r_m is generated such that (i) the head of r_m consists of ***magic***(B), and (ii) the body of r_m consists of the magic version of the head atom of r_a, followed by all the (EDB) atoms of r_a which can propagate the binding on B. For each adorned rule r_a, the generation of the magic rules is carried out by the function ***Generate***(r_a).

Modification (steps 9–10): The adorned rules are subsequently modified by including magic atoms generated in the rule bodies, which limit the range of the head variables avoiding the inference of facts which cannot contribute to deriving the query. The resulting rules are called *modified rules*. Each adorned rule r_a is modified by ***Modify***(r_a) as follows. Let H be the head atom of r_a. Then, atom ***magic***(H) is inserted in the body of the rule, and the adornments of all other predicates are stripped off. We would like to point out that stripping off the adornments serves mainly for facilitating the equivalence proofs; one may also leave the adornments (also in the query) intact, as it was done in the original definition of Magic Sets.

Processing of the Query (step 1): For each adorned atom $\mathtt{g}^\alpha$ of the query, the *magic seed* or query rule ***magic***$(\mathtt{g}^\alpha)$. (a fact) is produced by the function ***BuildQuerySeeds***$(\mathcal{Q}, S)$. For instance, in our example we generate $\mathtt{magic\_path}^{\mathtt{bb}}(1,5)$. This step can, in fact, be executed contextually with the adornment of the query.

The complete rewritten program consists of the magic, modified, and query rules. Given a Datalog program $\mathcal{P}$, a query $\mathcal{Q}$, and the rewritten program $\mathcal{P}'$, it is well known (see e.g. [1]) that $\mathcal{P}$ and $\mathcal{P}'$ are equivalent w.r.t. $\mathcal{Q}$, i.e., $\mathcal{P} \equiv^b_{\mathcal{Q}} \mathcal{P}'$ and $\mathcal{P} \equiv^c_{\mathcal{Q}} \mathcal{P}'$ hold (since brave and cautious semantics coincide for Datalog programs).

Example 4.2. The complete rewriting of our running example is:

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

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

In this rewriting, $\texttt{magic\_path}^{bb}(\texttt{X},\texttt{Y})$ represents the start- and end-nodes of all potential sub-paths of paths from 1 to 5. Therefore, when answering the query, only these sub-paths will be actually considered in bottom-up computations. The careful reader may check that this rewriting is in fact equivalent to the original program w.r.t. the query path(1, 5).

4.2. Binding propagation in Datalog$^\neg$ programs: some key issues

The first issue to be faced when generalizing the Magic Set technique to Datalog$^\neg$ programs is to extend the strategy for propagating binding information for rules with negative literals. Negative literals may "receive" bindings (i.e. some of their arguments can be bound via the head atom), but they cannot provide new bindings for variables. In this light, they should be treated like IDB predicates (even if they are EDB predicates) in the "basic" Magic Set method.

Example 4.3. Consider the rule

q(X, Y, D) :- b(X, Y, Z, W), q(Z, W, D), not c(D, Y).

where b and c are *EDB* predicates, while q is an *IDB* predicate, and the query q(1, 2, V).

When adorning the rule with the adorned query predicate $\texttt{q}^{bbf}$, first predicate b is processed, followed by q and c. Since b is an EDB predicate, it binds variables Z and W (X and Y are already bound due to the adornment of the head atom). While c is also an EDB predicate, it cannot bind D, as it occurs negated. In this way, q(Z, W, D) becomes $\texttt{q}^{bbf}(\texttt{Z},\texttt{W},\texttt{D})$, while, as usual, the EDB predicates are not adorned. Thus, the adorned rule is:

$\texttt{q}^{bbf}(\texttt{X},\texttt{Y},\texttt{D})$:- b(X, Y, Z, W), $\texttt{q}^{bbf}(\texttt{Z},\texttt{W},\texttt{D})$, not c(D, Y).

However, the most critical issue to be faced when extending Magic Sets to Datalog$^\neg$ programs is that the way of determining the set of relevant rules must be extended. For positive programs, binding propagations from head to body are sufficient to guarantee query equivalence. As we will demonstrate in the following example, this is not sufficient for Datalog$^\neg$, even if the program is guaranteed to be consistent for every EDB. The source of this phenomenon is essentially the same as for modular query answering as discussed in Section 3. In particular, we can extend the Magic Set method by using the notion of dangerous rules and predicates, in analogy to Section 3. This will give rise to propagations from body to head.

Example 4.4. Consider the program $\mathcal{P}_2$ consisting of the following rules:

z(X) :- y(X), not z(X).
y(X) :- q(X, Y).
p(X, Y) :- d(X, Y), not q(X, Y).
q(X, Y) :- d(X, Y), not p(X, Y).
a(X) :- p(X, Y), not b(X).
b(X) :- p(X, Y), not a(X).

together with the query $\mathcal{Q}_2$ = p(a, X), and the set of facts $\mathcal{F}_2$ = {d(a, b)}. We assume that d is an EDB predicate, while all other predicates are IDB. The stable models of $\mathcal{P}_2 \cup \mathcal{F}_2$ are {p(a, b), a(a), d(a, b)} and {p(a, b), b(a), d(a, b)}, so we get $Ans_c(\mathcal{Q}_2, \mathcal{P}_2 \cup \mathcal{F}_2) = Ans_b(\mathcal{Q}_2, \mathcal{P}_2 \cup \mathcal{F}_2)$ = {{X/b}}. Note that q(a, b) cannot occur in any stable model.

When applying the Magic Set technique, we obtain the following adorned program:

$\texttt{p}^{bf}(\texttt{X},\texttt{Y})$:- d(X, Y), not $\texttt{q}^{bb}(\texttt{X},\texttt{Y})$.
$\texttt{q}^{bb}(\texttt{X},\texttt{Y})$:- d(X, Y), not $\texttt{p}^{bb}(\texttt{X},\texttt{Y})$.
$\texttt{p}^{bb}(\texttt{X},\texttt{Y})$:- d(X, Y), not $\texttt{q}^{bb}(\texttt{X},\texttt{Y})$.

Then, the generation step produces the following magic program $Magic(\mathcal{Q}_2, \mathcal{P}_2)$:

$\texttt{magic\_p}^{\texttt{bf}}(\texttt{a}).$
$\texttt{magic\_q}^{\texttt{bb}}(\texttt{X},\texttt{Y}) \coloneq \texttt{magic\_p}^{\texttt{bf}}(\texttt{X}), \texttt{d}(\texttt{X},\texttt{Y}).$
$\texttt{magic\_p}^{\texttt{bb}}(\texttt{X},\texttt{Y}) \coloneq \texttt{magic\_q}^{\texttt{bb}}(\texttt{X},\texttt{Y}).$
$\texttt{magic\_q}^{\texttt{bb}}(\texttt{X},\texttt{Y}) \coloneq \texttt{magic\_p}^{\texttt{bb}}(\texttt{X},\texttt{Y}).$

Finally, the original rules are modified as follows to yield $Modified(\mathcal{Q}_2, \mathcal{P}_2)$:

$\texttt{p}(\texttt{X},\texttt{Y}) \coloneq \texttt{magic\_p}^{\texttt{bf}}(\texttt{X}), \texttt{d}(\texttt{X},\texttt{Y}), \text{not } \texttt{q}(\texttt{X},\texttt{Y}).$
$\texttt{q}(\texttt{X},\texttt{Y}) \coloneq \texttt{magic\_q}^{\texttt{bb}}(\texttt{X},\texttt{Y}), \texttt{d}(\texttt{X},\texttt{Y}), \text{not } \texttt{p}(\texttt{X},\texttt{Y}).$
$\texttt{p}(\texttt{X},\texttt{Y}) \coloneq \texttt{magic\_p}^{\texttt{bb}}(\texttt{X},\texttt{Y}), \texttt{d}(\texttt{X},\texttt{Y}), \text{not } \texttt{q}(\texttt{X},\texttt{Y}).$

Let us now evaluate the rewritten program over the set of facts $\mathcal{F}_2 = \{\texttt{d}(\texttt{a},\texttt{b})\}$. It is easy to see that the evaluation of the magic rules leads to the derivation of the atoms $\texttt{magic\_p}^{\texttt{bf}}(\texttt{a})$, $\texttt{magic\_q}^{\texttt{bb}}(\texttt{a},\texttt{b})$, and $\texttt{magic\_p}^{\texttt{bb}}(\texttt{a},\texttt{b})$. Therefore, by substituting these values in $Modified(\mathcal{Q}_2, \mathcal{P}_2)$, we get the following relevant rules of the ground program (all other rules contain a false magic atom in the body and are therefore trivially satisfied):

$\texttt{p}(\texttt{a},\texttt{b}) \coloneq \texttt{magic\_p}^{\texttt{bf}}(\texttt{a}), \texttt{d}(\texttt{a},\texttt{b}), \text{not } \texttt{q}(\texttt{a},\texttt{b}).$
$\texttt{q}(\texttt{a},\texttt{b}) \coloneq \texttt{magic\_q}^{\texttt{bb}}(\texttt{a},\texttt{b}), \texttt{d}(\texttt{a},\texttt{b}), \text{not } \texttt{p}(\texttt{a},\texttt{b}).$
$\texttt{p}(\texttt{a},\texttt{b}) \coloneq \texttt{magic\_p}^{\texttt{bb}}(\texttt{a},\texttt{b}), \texttt{d}(\texttt{a},\texttt{b}), \text{not } \texttt{q}(\texttt{a},\texttt{b}).$

The above program (evaluated over $\mathcal{F}_2$) has two stable models, say M_1 and M_2, such that $M_1/_{\mathcal{P}_{2\mathcal{F}_2}} = \{\texttt{p}(\texttt{a},\texttt{b}), \texttt{d}(\texttt{a},\texttt{b})\}$ and $M_2/_{\mathcal{P}_{2\mathcal{F}_2}} = \{\texttt{q}(\texttt{a},\texttt{b}), \texttt{d}(\texttt{a},\texttt{b})\}$. Thus, letting $\texttt{MS}(\mathcal{P}_2) = Magic(\mathcal{Q}_2, \mathcal{P}_2) \cup Modified(\mathcal{Q}_2, \mathcal{P}_2)$, we have that: $Ans_c(\mathcal{Q}_2, \texttt{MS}(\mathcal{P}_2) \cup \mathcal{F}_2) = \emptyset$, and $Ans_b(\mathcal{Q}_2, \texttt{MS}(\mathcal{P}_2) \cup \mathcal{F}_2) = \{\{\texttt{X}/\texttt{b}\}\}$. By comparing these answers with those we got for the original program, we conclude that the magic rewriting $\texttt{MS}(\mathcal{P}_2)$ is not cautious-complete w.r.t. $\mathcal{P}_2$.

In general the application of the traditional Magic Set method on unstratified programs would guarantee cautious-soundness and brave-completeness, but it would not ensure cautious-completeness and brave-soundness. Looking at the example above, we notice that the reason for the cautious-incompleteness lies in the fact that the first rule of $\mathcal{P}_2$ acts as a constraint imposing any atom of the form $\texttt{y}(\texttt{X})$ to be not contained in any model. Then, from the second rule we also conclude that we cannot derive any fact of the form $\texttt{q}(\texttt{X},\texttt{Y})$, as this would entail $\texttt{y}(\texttt{X})$. It follows that the constraint "indirectly" influences the query on predicate $\texttt{p}$, since the model M_2 such that $M_2/_{\mathcal{P}_{2\mathcal{F}_2}} = \{\texttt{q}(\texttt{a},\texttt{b}), \texttt{d}(\texttt{a},\texttt{b})\}$ cannot be extended to be a model for program $\mathcal{P}_{2\mathcal{F}_2}$. Note that the first and second rules are *dangerous* rules.

In order to overcome this problem, we next present a Magic Set rewriting which does, as the new module notion developed in Section 3, take dangerous rules into account. For the program of Example 4.4, our method will recognize and use the fact that the first and second rules of $\mathcal{P}_2$ are *dangerous* and will propagate the binding coming from $\texttt{q}$ to $\texttt{y}$ and on to $\texttt{z}$, thereby adding the first and second rules to the module identified by the rewritten program.

4.3. $\texttt{MS}^{\neg}$ *algorithm*

In this section, we describe the peculiarities of our rewriting technique. Our general architecture is slightly different from the definitions in text books such as [1] or [32]: Instead of staging the computation into adornment, generation and rewriting passes, where in each stage the whole program is processed, we perform all of these passes in a single step for each rule to be adorned. That is, we employ an adornment triggered strategy, employing a stack (or, equivalently, a queue) of adorned predicates, which have to be processed. Whenever a predicate is adorned (initially from the query), we check whether it has been processed already, and if not, we push it onto the aforementioned stack. An iterative process pops an unprocessed adorned predicate from the stack, and processes all rules in which the predicate occurs in the head. During the processing, the binding is propagated, potentially creating new adorned predicates, and in the same processing step, magic and modified rules are generated.

An advantage of this method is that the adorned program does not have to be saved. Moreover, as we shall see, the handling of dangerous rules becomes easier in this setting. In the following, we will refer to the function ***BuildQuerySeeds***$(\mathcal{Q}, S)$, ***Adorn***(r, p^{α}, S), ***Generate***(r_a), and ***Modify***(r_a) informally described in Section 4.1 for the standard Magic Set method for (positive) Datalog.

```
Input: A Datalog¬ program $\mathcal{P}$, and a query $\mathcal{Q} = \mathtt{g}(\bar{\mathtt{t}})$.
Output: The optimized program $\mathtt{MS}^{\neg}(\mathcal{Q}, \mathcal{P})$.
var $S$: stack of adorned predicates; modifiedRules, magicRules: set of rules;
begin
 1. modifiedRules := $\emptyset$; magicRules := BuildQuerySeeds($\mathcal{Q}$, $S$);
 2. while $S \neq \emptyset$ do
 3.    $p^{\alpha}$ := $S$.pop();
 4.    for each rule $r \in \mathcal{P}$ with $H(r) = p(\bar{t}_p)$ do
 5.       $r_a$ := Adorn($r$, $p^{\alpha}$, $S$);
 6.       magicRules := magicRules $\cup$ Generate($r_a$);
 7.       modifiedRules := modifiedRules $\cup$ {Modify($r_a$)};
 8.    end for
 9.    for each dangerous rule $d \in \mathcal{P}$ of the form $\mathtt{h}(\bar{\mathtt{t}}_{\mathtt{h}}) :- \mathtt{Q}_1(\bar{\mathtt{t}}_1), \ldots, \mathtt{Q}_{\mathtt{m}}(\bar{\mathtt{t}}_{\mathtt{m}})$.
              where $Q_i = p$ or $Q_i$ = not $p$ do
10.       let $d_s$ be the rule $\mathtt{p}(\bar{\mathtt{t}}_{\mathtt{i}}) :- \mathtt{h}(\bar{\mathtt{t}}_{\mathtt{h}}), \mathtt{Q}_1(\bar{\mathtt{t}}_1), \ldots, \mathtt{Q}_{\mathtt{i-1}}(\bar{\mathtt{t}}_1), \mathtt{Q}_{\mathtt{i+1}}(\bar{\mathtt{t}}_1), \ldots, \mathtt{Q}_{\mathtt{m}}(\bar{\mathtt{t}}_{\mathtt{m}})$.;
11.       let $d_a$ := Adorn($d_s$, $\mathtt{p}^{\alpha}$, $S$);
12.       magicRules := magicRules $\cup$ Generate($d_a$);
13.    end for
14. end while
15. $\mathtt{MS}^{\neg}(\mathcal{Q}, \mathcal{P})$ := magicRules $\cup$ modifiedRules;
16. return $\mathtt{MS}^{\neg}(\mathcal{Q}, \mathcal{P})$;
end.
```

Fig. 2. Magic Set algorithm for Datalog$^{\neg}$ programs.

The algorithm $\mathtt{MS}^{\neg}$, reported in Fig. 2, implements our Magic Set method for Datalog$^{\neg}$ programs. Its input is a Datalog$^{\neg}$ program $\mathcal{P}$ and a query $\mathcal{Q}$. (Note that the algorithm can be used for positive programs as a special case.) The algorithm $\mathtt{MS}^{\neg}$ outputs a program $\mathtt{MS}^{\neg}(\mathcal{Q}, \mathcal{P})$ consisting of a set of *modified* and *magic* rules (denoted by *modifiedRules* and *magicRules*, respectively). The algorithm generates modified and magic rules on a rule-by-rule basis. To this end, it exploits a stack S of predicates for storing all the adorned predicates that are still to be used for propagating the query binding (the ***Adorn*** function pushes on S each adorned predicates it generates, which has not been previously rewritten). At each step, an element p^{α} is removed from S, and the rules defining p are processed one-at-a-time. Thus, adorned rules do not have to be stored.

The main steps of the algorithm $\mathtt{MS}^{\neg}$ are illustrated by means of the program $\mathcal{P}_2$ and the query $\mathcal{Q}_2$ of Example 4.4.

The computation starts in step 1 by initializing *modifiedRules* to the empty set. Then, the function ***BuildQuerySeeds*** is used for storing in *magicRules* the magic seeds, and pushing on the stack S the adorned predicates of $\mathcal{Q}$. Note that we do not generate any query rules, because the transformed program will not contain adornments.

Example 4.5. Given the query $\mathcal{Q}_2$ and the program $\mathcal{P}_2$ of Example 4.4, ***BuildQuerySeeds*** creates $\mathtt{magic\_p}^{\mathtt{bf}}(\mathtt{a})$ and pushes $\mathtt{p}^{\mathtt{bf}}$ onto the stack S.

The core of the technique (steps 3–13) is repeated until the stack S is empty, i.e., until there is no further adorned predicate to be propagated. Specifically, an adorned predicate $\mathtt{p}^{\alpha}$ is removed from the stack S in step 3, and its binding is propagated.

In steps 4–8, the binding of $\mathtt{p}^{\alpha}$ is propagated head-to-body in each rule r of $\mathcal{P}$ having an atom $\mathtt{p}(\bar{\mathtt{t}})$ in the head. This propagation is as in the standard Magic Set method for stratified Datalog$^{\neg}$ programs (using an appropriate SIPS as discussed in Section 4.2).

Example 4.6. We continue from Example 4.5. Taking the predicate $\mathtt{p}^{\mathtt{bf}}$ from the stack entails the adornment of the rule $\mathtt{p}(\mathtt{X}, \mathtt{Y}) :- \mathtt{d}(\mathtt{X}, \mathtt{Y})$, not $\mathtt{q}(\mathtt{X}, \mathtt{Y})$. This yields the rule $\mathtt{p}^{\mathtt{bf}}(\mathtt{X}, \mathtt{Y}) :- \mathtt{d}(\mathtt{X}, \mathtt{Y})$, not $\mathtt{q}^{\mathtt{bb}}(\mathtt{X}, \mathtt{Y})$., and the predicate $\mathtt{q}^{\mathtt{bb}}$ is eventually pushed on the stack. Then, we proceed with the generation of one magic ($\mathtt{magic\_q}^{\mathtt{bb}}(\mathtt{X}, \mathtt{Y}) :- \mathtt{magic\_p}^{\mathtt{bf}}(\mathtt{X}), \mathtt{d}(\mathtt{X}, \mathtt{Y})$.) and one modified rule ($\mathtt{p}(\mathtt{X}, \mathtt{Y}) :- \mathtt{magic\_p}^{\mathtt{bf}}, \mathtt{d}(\mathtt{X}, \mathtt{Y})$, not $\mathtt{q}(\mathtt{X}, \mathtt{Y})$.).

Steps 9–13 performs the propagation of the binding through each *dangerous* rule d in $\mathcal{P}$ of the form $\mathtt{h}(\bar{\mathtt{t}}_\mathtt{h})$:- $\mathtt{Q}_1(\bar{\mathtt{t}}_1), \ldots, \mathtt{Q}_\mathtt{m}(\bar{\mathtt{t}}_\mathtt{m})$., in which the predicate p occurs in $\mathtt{Q}_\mathtt{i}(\bar{\mathtt{t}}_\mathtt{i})$ inside the positive or negative body. These steps are, in fact, required to guarantee cautious-completeness and brave-soundness for consistent programs, as suggested in Section 4.2. In this case, in order to simulate body-to-head propagations and to minimize the effort of doing so, we swap head and the matching body literal and apply the standard method as if it was a head-to-body propagation. So, the rule d is first replaced by an "inverted" rule d_s of the form $\mathtt{p}(\bar{\mathtt{t}}_\mathtt{i})$:- $\mathtt{h}(\bar{\mathtt{t}}_\mathtt{h}), \mathtt{Q}_1(\bar{\mathtt{t}}_1), \ldots, \mathtt{Q}_{\mathtt{i}-1}(\bar{\mathtt{t}}_{\mathtt{i}-1}), \mathtt{Q}_{\mathtt{i}+1}(\bar{\mathtt{t}}_{\mathtt{i}+1}), \ldots, \mathtt{Q}_\mathtt{m}(\bar{\mathtt{t}}_\mathtt{m})$., which has been obtained by swapping the head atom with the body atom (possibly occurring negated) propagating the binding. Then, the adornment can be carried out as usual by means of the function ***Adorn***. Since this "inverted" rule was not part of the original program and its only purpose is generating binding information, it will not give rise to a modified rule, but only to magic rules.

Example 4.7. Continuing from Example 4.6, when $\mathtt{q}^{\mathtt{bb}}$ is removed from the stack, it can be used for adorning the body of the dangerous rule y(X) :- q(X, Y). Hence, we obtain first the "inverted" rule q(X, Y) :- y(X) and adorn it, obtaining $\mathtt{q}^{\mathtt{bb}}$(X, Y) :- $\mathtt{y}^{\mathtt{b}}$(X). which gives rise to one magic rule: magic_$\mathtt{y}^{\mathtt{b}}$(X) :- magic_$\mathtt{q}^{\mathtt{bb}}$(X, Y). Moreover, $\mathtt{y}^{\mathtt{b}}$ is pushed onto the stack, which subsequently causes the adornment and rewriting (hence inclusion into the rewritten program) of the first and second rule of $\mathcal{P}_2$.

Finally, after all the adorned predicates have been processed the algorithm outputs the program $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$.

Example 4.8. The complete rewriting of program $\mathcal{P}_2$ w.r.t. query $\mathcal{Q}_2$ ($\mathtt{MS}^\neg(\mathcal{Q}_2, \mathcal{P}_2)$) consists of the magic rules:

magic_$\mathtt{p}^{\mathtt{bf}}$(a).
magic_$\mathtt{q}^{\mathtt{bb}}$(X, Y) :- magic_$\mathtt{p}^{\mathtt{bf}}$(X), d(X, Y).
magic_$\mathtt{p}^{\mathtt{bb}}$(X, Y) :- magic_$\mathtt{q}^{\mathtt{bb}}$(X, Y).
magic_$\mathtt{y}^{\mathtt{b}}$(X) :- magic_$\mathtt{q}^{\mathtt{bb}}$(X, Y).
magic_$\mathtt{q}^{\mathtt{bf}}$(X) :- magic_$\mathtt{y}^{\mathtt{b}}$(X).
magic_$\mathtt{z}^{\mathtt{b}}$(X) :- magic_$\mathtt{y}^{\mathtt{b}}$(X).
magic_$\mathtt{z}^{\mathtt{b}}$(X) :- magic_$\mathtt{z}^{\mathtt{b}}$(X).
magic_$\mathtt{p}^{\mathtt{bb}}$(X, Y) :- magic_$\mathtt{q}^{\mathtt{bf}}$(X), d(X, Y).
magic_$\mathtt{q}^{\mathtt{bb}}$(X, Y) :- magic_$\mathtt{p}^{\mathtt{bb}}$(X, Y).

plus the rewritten rules:

p(X, Y) :- magic_$\mathtt{p}^{\mathtt{bf}}$(X), d(X, Y), not q(X, Y).
q(X, Y) :- magic_$\mathtt{q}^{\mathtt{bb}}$(X, Y), d(X, Y), not p(X, Y).
y(X) :- magic_$\mathtt{y}^{\mathtt{b}}$(X), q(X, Y).
z(X) :- magic_$\mathtt{z}^{\mathtt{b}}$(X), y(X), not z(X).
q(X, Y) :- magic_$\mathtt{q}^{\mathtt{bf}}$(X), d(X, Y), not p(X, Y).
p(X, Y) :- magic_$\mathtt{p}^{\mathtt{bb}}$(X, Y), d(X, Y), not q(X, Y).

It is worth noting that the rewritten program does not contain rules for predicates a and b, and indeed they are not relevant for answering $\mathcal{Q}_2$. Notice that $\mathtt{MS}^\neg(\mathcal{Q}_2, \mathcal{P}_2) \cup \mathcal{F}_2$ admits only one stable model M, such that $M/_{\mathcal{P}_2\mathcal{F}_2} = \{\mathtt{p(a,b)}, \mathtt{d(a,b)}\}$. Hence, $Ans_c(\mathcal{Q}_2, \mathtt{MS}^\neg(\mathcal{Q}_2, \mathcal{P}_2) \cup \mathcal{F}_2) = Ans_b(\mathcal{Q}_2, \mathtt{MS}^\neg(\mathcal{Q}_2, \mathcal{P}_2) \cup \mathcal{F}_2) = \{\{\mathtt{X/b}\}\}$, and so all answers are preserved w.r.t. $\mathcal{P}_2$ and $\mathcal{Q}_2$.

4.4. Query equivalence results

We conclude the presentation of the Magic Set algorithm for Datalog$^\neg$ programs by formally proving its soundness and completeness under certain conditions. The result is shown by establishing a correspondence between a program $\mathcal{P}$ and its transformed program $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$ with respect to some query $\mathcal{Q}$.

To show this result, we will employ the notion of *simplification*: Given a ground program $\mathcal{P}$, an EDB $\mathcal{F}$, and a subprogram $U \subseteq \mathcal{P}$, for which no predicate defined by U is defined by $\mathcal{P} \setminus U$, and where $U_{\mathcal{F}}$ admits exactly one stable model S. Then simplify$(\mathcal{P}, U)$ denotes the program $\{H(r) \text{ :- } B^+(r)/_{\mathcal{P}\setminus U}, \text{not } B^-(r)/_{\mathcal{P}\setminus U} \mid r \in (\mathcal{P} \setminus U), B^+(r)/_U \subseteq S, B^-(r)/_U \cap S = \emptyset\}$, which can be thought of as the partial evaluation w.r.t. S. In other words, all predicates defined by U will be eliminated from $\mathcal{P} \setminus U$, based on the single answer set S: Atoms which are true in S and occur in a positive rule body are eliminated from a rule, just as atoms which are false in S and occur in a negative body. Rules, which contain an atom, which is false in S in the positive body, are eliminated, as are rules which contain a true atom in the negative body. This simplification is needed to get rid of the magic predicates, which are not present in the original program.

Lemma 4.9. *Let $\mathcal{P}$ be a Datalog$^\neg$ program $\mathcal{P}$, $\mathcal{Q}$ a query, and $\mathcal{F}$ be an EDB. Furthermore, we denote by* magic$(\mathcal{Q}, \mathcal{P})$ *the set of magic rules in* $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$. *Then it holds that* $\mathcal{P}'' = \text{simplify}(\mathit{Ground}(\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F}), \mathit{Ground}(\text{magic}(\mathcal{Q}, \mathcal{P}) \cup \mathcal{F}))$ *is a module of* $\mathcal{P}' = \text{simplify}(\mathit{Ground}(\mathcal{P} \cup \mathcal{F}), \mathcal{F})$.

Proof. Observe that $\mathcal{P}'' \subseteq \mathcal{P}'$ holds. Assume that $\mathcal{P}''$ is not a module of $\mathcal{P}'$. Then at least one of the following condition holds: (1) $\exists r' \in \mathcal{P}' \setminus \mathcal{P}'', r'' \in \mathcal{P}''$: $H(r') = H(r'')$; (2) $\exists r'' \in \mathcal{P}''$: $\exists b \in B(r'')$: $\exists r' \in \mathcal{P}' \setminus \mathcal{P}''$: $b = H(r')$; (3) $\exists r'' \in \mathcal{P}''$: $\exists r' \in \mathcal{P}' \setminus \mathcal{P}''$: $H(r'') \in B(r')$ and r' is dangerous. One can show that all of (1), (2), and (3) lead to contradictions, and hence the result follows.

(1) For all rules in $\mathcal{P}$ with head predicate h, in $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$ there exists a copy for each adornment that was generated for h on line 5 of Fig. 2. So for any simplified ground instance r of such a rule with head atom $h(c_1, \ldots, c_n)$, either $magic\_h^a(c_1, \ldots, c_m)$ holds for at least one adornment a of h, or it does not hold for any adornment of h. In the former case, for each rule in $\mathcal{P}$ with h in its head, a corresponding rule with $magic\_h^a$ in its body exists in $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$ because of line 7 of Fig. 2. If $magic\_h^a(c_1, \ldots, c_m)$ holds for no adornment a, no simplified ground version of r is in $\mathcal{P}''$. In total, for each ground atom $h(c_1, \ldots, c_n)$ in $\mathcal{P}'$, either all or none of its defining rules are in $\mathcal{P}''$.

(2) Assume that r'' (the head of which is $h(c_1, \ldots, c_h)$) stems from a rule r''_o, which was adorned by a, such that $magic\_h^a(c_1, \ldots, c_{h_1})$ follows from the magic rules. Each IDB body atom of r''_o has received some adornment based on a, in which bound arguments either directly share bound variables (w.r.t. a) with h, or via some EDB atoms. For any body predicate b, this gives rise to a magic rule r_m: $magic\_b^{a_1}(\bar{t}_{b_1}) \text{ :- } magic\_h^a(\bar{t}_a), B.$ by means of line 6 of Fig. 2, where B contains only those EDB atoms relevant for bound arguments of b. Concerning r' it contains some $b(d_1, \ldots, d_b)$ of the body of r'' in its head, and its originating rule r'_o is adorned by a_1. So in $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$ a rule r'_m: $b(\bar{t}_{b_2}) \text{ :- } magic\_b^{a_1}(\bar{t}_{b_3}), B'.$ occurs, generated by line 7 of Fig. 2. Note that for all bound arguments $d_1, \ldots, d_k$ of $b(d_1, \ldots, d_b)$ w.r.t. a_1, $magic\_b^{a_1}(d_1, \ldots, d_k)$ follows from the magic rules because of r_m. So whenever a simplified ground instance of r'_o with $b(d_1, \ldots, d_b)$ in the head exists, so does one of r'_m, which is hence in $\mathcal{P}''$.

(3) Observe first that the set of instantiations of dangerous rules in $\mathcal{P}$ is a superset of the set of dangerous rules in $\mathit{Ground}(\mathcal{P}_{\mathcal{F}})$, which is in turn a superset the set of dangerous rules in any simplification of $\mathit{Ground}(\mathcal{P}_{\mathcal{F}})$. So any dangerous rule in $\mathcal{P}'$ is also dangerous in $\mathcal{P}$. Therefore, the originating rule $r'_o \in \mathcal{P}$ of r' must have been adorned and "inverted" by line 10 of Fig. 2, adorning in the following also the head of r'_o. So the dangerous rule r'_o eventually also gives rise to a modified rule in $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$. Then, by the same argument as in (2), a magic rule obtained from the "inverted" rule must exist in $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$, such that one of its instantiations matches the bound arguments of $H(r')$. So r' is in $\mathcal{P}''$ iff it is in $\mathcal{P}'$. □

Armed with this lemma we can prove the following.

Theorem 4.10. *Let $\mathcal{P}$ be a Datalog$^\neg$ program, let $\mathcal{Q}$ be a query, and $\mathcal{F}$ be an EDB. Then, the following holds*:

(1) $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P}) \subseteq^c_{\mathcal{Q}} \mathcal{P}$ *and* $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P}) \supseteq^b_{\mathcal{Q}} \mathcal{P}$;
(2) $\mathit{Ans}_b(\mathcal{Q}, \mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})_{\mathcal{F}}) = \mathit{Ans}_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, *if $\mathcal{P}_{\mathcal{F}}$ is consistent*;
(3) $\mathit{Ans}_c(\mathcal{Q}, \mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})_{\mathcal{F}}) = \mathit{Ans}_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, *if $\mathcal{P}_{\mathcal{F}}$ is consistent*;
(4) $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P}) \equiv^b_{\mathcal{Q}} \mathcal{P}$ *and* $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P}) \equiv^c_{\mathcal{Q}} \mathcal{P}$, *if $\mathcal{P}$ is data consistent.*

Proof. First, observe that both grounding and simplification w.r.t. a part U of a program $\mathcal{P}$, such that head predicates of U do not occur in $\mathcal{P} \setminus U$, preserve query equivalence, as long as the query does not contain predicates of U. Note that by definition queries do not contain EDB predicates, and they also cannot contain magic predicates. Then, since by virtue of Lemma 4.9, simplify($\mathit{Ground}(\mathtt{MS}^{\neg}(\mathcal{Q},\mathcal{P}) \cup \mathcal{F})$, $\mathit{Ground}(\text{magic}(\mathcal{Q},\mathcal{P}) \cup \mathcal{F})$) is a module of simplify($\mathit{Ground}(\mathcal{P} \cup \mathcal{F}), \mathcal{F}$), we can apply Theorems 3.6 and 3.7 to yield the result for the grounded and simplified program, which by the consideration above also proves the result as stated. □

5. Complexity issues

In this section, we discuss some relevant complexity issues related to the Magic Set technique for Datalog$^{\neg}$ programs. We start by investigating the intrinsic complexity of the notion of dangerous predicate, which is crucial for generating $\mathtt{MS}^{\neg}(\mathcal{Q},\mathcal{P})$.

Recall that a predicate is dangerous if either it occurs in an odd cycle of the predicate dependency graph or it occurs in the body of a rule with a dangerous head predicate. The following theorem shows that checking whether the first condition is satisfied is in general intractable.

Theorem 5.1. *Let $\mathcal{P}$ be a program and d be a predicate. Then, deciding whether d occurs in an odd cycle of $DG_{\mathcal{P}}$ is* NP*-complete.*

Proof. Membership in NP is trivial since we can guess a sequence of arcs of $DG_{\mathcal{P}}$ and verify in polynomial time that they form a cycle which contains an odd number of marked arcs.

As for the hardness, we can exploit a reduction from the problem of deciding, given a directed graph $G = (N, E)$, whether a node $n \in N$ occurs in a cycle made of an odd number of edges, which has been proved to be NP-complete in [33]. To this aim, given G we build a program $\mathcal{P}(G)$ such that predicates in $\mathcal{P}(G)$ are in one-to-one correspondence with nodes in N; there is exactly one rule of the form `a :- not b.`, for each edge from b to a in E, and no other rule is in $\mathcal{P}(G)$. Thus, each arc in $DG_{\mathcal{P}(G)}$ is marked and n occurs in a cycle with an odd number of edges in G if and only if the predicate corresponding to n occurs in a cycle of $DG_{\mathcal{P}(G)}$ with an odd number of marked edges. □

Note that, regarding complexity, there is a big difference between the problem of deciding whether a program contains an odd cycle and the problem of deciding whether a particular predicate occurs in an odd cycle: The former problem is polynomially solvable, while the latter is NP-complete, as shown in Theorem 5.1. Intuitively, this difference is due to the fact that the existence of a closed walk (i.e. a closed path where intermediate nodes may occur more than once) with an odd number of edges implies the existence of an odd cycle, and deciding whether a graph contains a closed walk is feasible in polynomial time. However, the occurrence of a node in a closed walk with an odd number of edges does not imply the occurrence of this node in an odd cycle.

In the light of Theorem 5.1, one might expect that the problem of deciding whether a predicate is dangerous is intractable as well. However, we next evidence that this is not the case. Indeed, the set of dangerous predicates is a superset of the set of predicates occurring in odd cycles, and relaxing the constraints on the set to be computed indeed leads to a tractable notion.

Theorem 5.2. *Let $\mathcal{P}$ be a program and d be a predicate. Then, deciding whether d is dangerous is* NL*-complete.*

Proof. NL-*Hardness.* Recall that, given a directed graph $G = (N, E)$ and two nodes s and t in G, deciding whether there exists a path from s to t is the canonical problem complete for non-deterministic logspace (see, e.g., [34]). We reduce this problem to checking if a predicate is dangerous.

To this end, given graph $G = (N, E)$, we build a program $\mathcal{P}(G)'$ such that: predicates in $\mathcal{P}(G)'$ are in one-to-one correspondence with nodes in N. $\mathcal{P}(G)'$ contains the rule `s :- not t`. Additionally, it contains the rule `a :- b` for each edge from b to a in E. No other rule is in $\mathcal{P}(G)'$. The program $\mathcal{P}(G)'$ is clearly constructible in logarithmic space.

By construction of $\mathcal{P}(G)'$, there is a path from s to t in G if and only if `s` occurs in an odd cycle of $DG_{\mathcal{P}(G)'}$.[7] Therefore, the existence of a path from s to t in G implies that s is dangerous. On the other hand, if there is no path from s to t in G, then $DG_{\mathcal{P}(G)'}$ has no odd cycle, and program $\mathcal{P}(G)'$ has no dangerous predicates at all. Hence, there

[7] Recall that an odd cycle of $DG_{\mathcal{P}(G)'}$ is a cycle of $DG_{\mathcal{P}(G)'}$ containing an odd number of marked edges.

exists a path from s to t in G if and only if s is a dangerous predicate for $\mathcal{P}(G)'$, and the NL-hardness of deciding whether a predicate is dangerous follows.

NL-*Membership.* We construct a non-deterministic Turing machine *TM'* which decides whether a given predicate is dangerous, using logarithmic space. We first construct another non-deterministic Turing machine *TM*, which also works using logarithmic space, which decides whether a predicate d is on a closed walk in the dependency graph, where the closed walk contains an odd number of marked (negative) edges. Note that d is dangerous if such a closed walk exists, as it is either directly on an odd cycle or reached from an odd cycle. *TM'* is then just an extension for recognizing also predicates reached from a closed walk identified by *TM*.

Given a program $\mathcal{P}$ and a predicate a_1, we first show how to build a non-deterministic logspace Turing machine *TM* which checks whether there is a sequence of predicates of the form $a_1, \ldots, a_n, a_{n+1}$ where $a_{n+1} = a_1$ such that (i) there is an arc, say e_i, from a_i to a_{i+1} in $DG_{\mathcal{P}}$, for each $1 \leqslant i \leqslant n$, and (ii) the number of marked arcs in $\{e_1, \ldots, e_n\}$ is odd (this sequence is a closed walk of $DG_{\mathcal{P}}$ having an odd number of marked arcs). The machine *TM* computes the sequence $a_1, \ldots, a_{n+1}$ by non-deterministically selecting, at each step i, the predicate a_{i+1} in the set of the predicates occurring in the head of some rule, say r, having a_i in the body. The machine also uses a *flip* bit which is initialized to 0, and which is flipped if a_i occurs negatively in r. Then, each time a new predicate is selected a counter is incremented so that the machine may halt by returning *no* if the counter exceeded the number of nodes in N, say m. Moreover, in the case where a_1 is reached again, the machine returns *yes* if and only if the value of *flip* is 1.

We can easily check that *TM* is correct. Note that (1) predicates and rules of $\mathcal{P}$ can be indexed in logspace, (2) looking for a rule whose head contains a given predicate just requires an additional logspace index, and (3) the counter over the number of iterations needs a logarithmic number of bits since *TM* halts in at most m steps. So, *TM* needs only logarithmic space.

We can now face the problem of deciding whether d is dangerous by modifying *TM* into a machine *TM'* as follows. The predicate a_1 received in input by *TM* is, in fact, guessed by *TM'* at the very beginning of the computation. Then, *TM'* performs the same steps as *TM*. In the cases where *TM* outputs *no*, *TM'* outputs *no* as well. Otherwise, i.e., when a_1 is reached again (and *TM* returned *yes*), *TM'* starts a new computational branch, employing the standard NL reachability method, to assess whether there is a path from d to a_1 in the dependency graph $DG_{\mathcal{P}}$, and outputs *yes* iff such a path exists. *TM'* clearly is in non-deterministic logspace, and it only remains to show its correctness, which we do next.

- Assume d is dangerous. Then, d occurs in an odd cycle of $DG_{\mathcal{P}}$ or it occurs in the body of a rule having a dangerous head predicate. In both cases, the dependency graph $DG_{\mathcal{P}}$ has a closed walk W with an odd number of marked arcs, and there is a path pt in $DG_{\mathcal{P}}$ from d to an element $\bar{a}$ of W (note that W contains an odd cycle of $DG_{\mathcal{P}}$, and all elements in W are dangerous predicates; moreover pt will be a subset of W, if d occurs in W). In the initial phase (where *TM'* behaves like *TM*) *TM'* may initially guess $a_1 = \bar{a}$ and the other elements of the closed walk W in the right sequence, up to reaching $a_{n+1} = \bar{a}$ again. The reachability algorithm employed by *TM'* will then detect the existence of the path pt in $DG_{\mathcal{P}}$ from d to $\bar{a}$, and *TM'* will correctly output *yes*.
- Assume *TM'* outputs *yes*. Then, the first part of the computation of *TM'* (where it behaves like *TM*) has detected a sequence $W = [a_1, \ldots, a_{n+1}]$ of predicates witnessing the satisfaction of conditions (i) and (ii) above. *TM'* has then, in the second part of the computation, detected a path pt from d to a_1 (recall that $a_{n+1} = a_1$). The sequence W is, in fact, a closed walk in the dependency graph $DG_{\mathcal{P}}$ having an odd number of marked arcs. Consequently, there exists a subset of the nodes in W, say C, which forms a cycle of $DG_{\mathcal{P}}$ having an odd number of marked arcs. Therefore, each predicate a in W is dangerous: either a is in the odd cycle C, or there exists a path from a to a node in C (actually, to all nodes of C). In particular, this is true also for a_1 which is dangerous. Since *TM'* has detected a path pt from d to the dangerous predicate a_1, then d is dangerous as well, and the *yes* answer of *TM'* is correct. □

Thus, checking if a predicate is dangerous, is computationally easier than checking if it occurs in an odd cycle (of the dependency graph $DG_{\mathcal{P}}$), even if membership in an odd cycle is part of the definition of the notion of dangerous predicate. Intuitively, this is possible because for checking whether a predicate d is dangerous, we can circumvent the problem of checking membership in odd cycles. Indeed, instead of checking that d reaches (in $DG_{\mathcal{P}}$) a node a of an odd cycle, we can equivalently check that d reaches a node a of a closed walk having an odd number of marked arcs (which is feasible in *NL*).

Note that the membership part in the proof above suggests a practical algorithm for testing whether a predicate/rule is dangerous by means of a deterministic Turing machine. Indeed, one can use a slight modification of the algorithm for computing the transitive closure of a graph, where information about the oddness of the number of marked arcs connecting any pair of vertices is stored. For a program $\mathcal{P}$, let $\mathit{Preds}(\mathcal{P})$ be the set of IDB predicates of $\mathcal{P}$, and let $|\mathcal{P}|$ denote the number of the rules in $\mathcal{P}$.

Theorem 5.3. *Let $\mathcal{P}$ be a Datalog$^\neg$ program. Computing the set of all the dangerous rules is feasible in $O(|\mathit{Preds}(\mathcal{P})|^3 + |\mathcal{P}|)$.*

Proof. We give an algorithm which first computes the set of all dangerous predicates, which is then used to identify the dangerous rules.

Let $DG_{\mathcal{P}} = (N, E)$ be the dependency graph of $\mathcal{P}$, with $N = \{n_1, \ldots, n_m\}$, from which nodes corresponding to EDB predicates have been deleted (these nodes are sources in the graph, and w.l.o.g., we assume that no rule in $\mathcal{P}$ defines an EDB predicate). We use a dynamic programming algorithm consisting in $m = |\mathit{Preds}(\mathcal{P})|$ steps. At each step i, where $0 \leqslant i \leqslant m$ for each pair of predicates (nodes in the dependency graph $DG_{\mathcal{P}}$), say (n_α, n_β), we compute two bits, say $e^i_{(n_\alpha,n_\beta)}$ and $o^i_{(n_\alpha,n_\beta)}$, denoting whether there is a path from n_α to n_β in the dependency graph $DG_{\mathcal{P}}$ involving an even or an odd number of marked arcs, respectively, touching the nodes in $\{n_1, \ldots, n_i\}$ only (apart from n_α and n_β).

The process is initialized by setting $e^0_{(n_\alpha,n_\beta)}$ to 1 if $DG_{\mathcal{P}}$ contains an unmarked arc (n_α, n_β), and otherwise to 0. Symmetrically, $o^0_{(n_\alpha,n_\beta)}$ is set to 1 if $DG_{\mathcal{P}}$ contains a marked arc (n_α, n_β), and otherwise to 0.

Then, at each step i, where $0 < i \leqslant m$ we set $e^{i+1}_{(n_\alpha,n_\beta)} := 1$ if $e^i_{(n_\alpha,n_\beta)} = 1$ or if $e^i_{(n_\alpha,n_{i+1})} = e^i_{(n_{i+1},n_\beta)} = 1$, or if $o^i_{(n_\alpha,n_{i+1})} = o^i_{(n_{i+1},n_\beta)} = 1$; otherwise $e^{i+1}_{(n_\alpha,n_\beta)} := 0$. On the other hand, $o^{i+1}_{(n_\alpha,n_\beta)} := 1$ if $o^i_{(n_\alpha,n_\beta)} = 1$ or if $e^i_{(n_\alpha,n_{i+1})} \neq o^i_{(n_{i+1},n_\beta)}$ and $o^i_{(n_\alpha,n_{i+1})} \neq e^i_{(n_{i+1},n_\beta)}$; otherwise $o^{i+1}_{(n_\alpha,n_\beta)} := 0$.

By means of standard inductive arguments, we can show that $o^m_{(n_\alpha,n_\alpha)}$ is equal to 1 if and only if there is a walk closed in n_α having an odd number of marked arcs. This entails that some of the cycles in which the closed walk can be decomposed contains an odd number of marked arcs, and n_α is dangerous (see the arguments in proof of Theorem 5.2). Thus, $o^m_{(n_\alpha,n_\alpha)}$ is equal to 1 if and only if n_α is dangerous. Now, recall that a predicate n_β is dangerous if there is a path from n_β to n_α in the dependency graph, where n_α is dangerous. To check this condition it is sufficient to verify that $o^m_{(n_\beta,n_\alpha)} + e^m_{(n_\beta,n_\alpha)} > 0$ and $o^m_{(n_\alpha,n_\alpha)} = 1$. It follows that the set of dangerous predicates can be computed with a linear scan over the bits associated to pairs of nodes. Thus, the set of dangerous predicates can be computed by means of an algorithm which requires m steps, and such that each step requires in turn updating m^2 entries.

Finally, given the set of dangerous predicates we can scan the rules in $\mathcal{P}$ to select as dangerous only those whose head contains a dangerous predicate. This requires $|\mathcal{P}|$ steps at most. □

We are now in the position to analyze the complexity of the $\mathtt{MS}^\neg$ algorithm. We next assume that the maximum arity of the program predicates is bound by a constant, which is often the case in practical applications. For instance, in the case of the data integration application, described in Section 6, the maximum arity of predicates is 4. (Note that this assumption automatically holds when the complexity is measured according to data complexity [35].)

Theorem 5.4. *Let $\mathcal{P}$ be a Datalog$^\neg$ program, and let $\mathcal{Q}$ be a query. Then, $\mathtt{MS}^\neg(\mathcal{Q}, \mathcal{P})$ can be computed in time $O(\ell\ |\mathcal{P}|^3)$, where ℓ is the maximum number of literals in the body of rules in $\mathcal{P}$, and $|\mathcal{P}|$ is the number of rules in $\mathcal{P}$.*

Proof. The execution of the Magic Set algorithm (in Fig. 2) requires firstly the computation of all the dangerous rules (to be used in step *9*), which can be carried out in $O(|\mathit{Preds}(\mathcal{P})|^3 + |\mathcal{P}|)$, by Theorem 5.3. Then, we have to repeat $O(|\mathit{Preds}(\mathcal{P})|)$ times the main loop (cf. steps *2–14*), i.e., one repetition for each possible adornment of predicates in $\mathcal{P}$. At each step, we must scan all the rules in $\mathcal{P}$, and for each rule, in the worst case, we have to invoke the functions ***Adorn***, ***Generate***, and, possibly (while adorning a rule that is not dangerous) ***Modify***. In particular, the dominant operations are in the function ***Adorn***, which has to adorn the rule as well as to push on the stack S only those adorned predicates that are not yet used to propagate their binding information. To check this latter condition, we can simply maintain the list of all the processed adorned predicates and make a containment test (in $O(|\mathit{Preds}(\mathcal{P})|)$) for each newly generated adornment—notice that these adornments are $O(\ell)$. Moreover, propagating the binding in the rule requires just a scan of all the variables in it, which is again feasible in $O(\ell)$.

Therefore, the value $O(|Preds(\mathcal{P})|^3 + |\mathcal{P}| + \ell|Preds(\mathcal{P})|^2|\mathcal{P}|)$ is a bound on the running time of $\mathtt{MS}^{\neg}(\mathcal{Q}, \mathcal{P})$. Notice, in particular, that checking for dangerous predicates does not represent an overhead w.r.t. the basic Magic Set method. Finally, the result follows by noting that $|Preds(\mathcal{P})| = O(|\mathcal{P}|)$. □

Note that the maximum number ℓ of literals in the body of rules in $\mathcal{P}$ provides an upper bound on the number of iterations of the algorithm, under the assumption that the arity of the predicates is bounded by a fixed constant. In fact, letting h denote the maximum arity of predicates in $\mathcal{P}$, $O(2^h)$ adornments might be pushed on the stack in the worst case, for each literal in $\mathcal{P}$; and, hence, the running time of $\mathtt{MS}^{\neg}(\mathcal{Q}, \mathcal{P})$ exponentially scales w.r.t. h.

Before leaving the section, it is worthwhile noting that the running time of the Magic Set algorithm is, in fact, polynomial in the size of the program $\mathcal{P}$, but it does not depend on the size of the EDB over which the rewritten program has to be evaluated. This is a very important property making the approach suitable for scaling over large databases.

6. An application to data integration

In this section we show an application of our Magic Set method for optimizing query answering in data integration systems, and report on the experience we have made with it in the course of the EU project INFOMIX on data integration.

6.1. Formal framework

Following [20], a data integration system $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where:

(1) $\mathcal{G}$ is the *global* (*relational*) *schema*, i.e., a tuple $\langle \Psi, \Sigma \rangle$, where Ψ is a finite set of relation symbols, each with an associated positive arity, and Σ is a finite set of *integrity constraints* (ICs) expressed on the symbols in Ψ, i.e., assertions that are intended to be satisfied by database instances. Within the INFOMIX setting, three kinds of IC are considered: *key constraints*, *exclusion dependencies*, and *inclusion dependencies*. Throughout this section, we assume the schema to be *non-key-conflicting* [36] without mentioning this property explicitly.

(2) $\mathcal{S}$ is the *source schema*, constituted by the schemas of the various sources that are part of the data integration system. We assume that $\mathcal{S}$ is a relational schema of the form $\mathcal{S} = \langle \Psi', \emptyset \rangle$, i.e., there are no integrity constraints on the sources. This assumption implies that data stored at the sources are locally consistent; this is a common assumption in data integration, because 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* which establishes the relationship between $\mathcal{G}$ and $\mathcal{S}$. In our framework, the mapping follows the GAV approach, i.e., each global relation is associated with a *view*—a stratified Datalog$^{\neg}$ query, over the sources.

Example 6.1. Let us consider a data integration system $\mathcal{I}_0 = \langle \mathcal{G}_0, \mathcal{S}_0, \mathcal{M}_0 \rangle$ which is a simplification of the Demo Scenario of INFOMIX. For details on the full scenario we refer to [37].

The schema $\mathcal{G}_0$ consists of the relations `professor(IDP, Pname, Phomepage)`, `student(IDS, Sname, Saddress)`, and `exam_data(IDP, IDS, Exam, Mark)`. The associated constraints in Σ_0 state that: (i) (*key constraints*) the keys of `professor`, `student`, and `exam_data` are `IDP`, `IDS`, and (`IDP`, `IDS`, `Exam`), respectively, (ii) (*exclusion dependency*) a professor cannot be a student, and (iii) (*inclusion dependencies*) the identifiers of professors and students in the relation `exam_data` must be in the relations `professor` and `student`, respectively. The source schema $\mathcal{S}_0$ comprises the relations $\mathtt{s}_1$, $\mathtt{s}_2$, $\mathtt{s}_3$, and $\mathtt{s}_4$ (of arity 3, 3, 4, and 4, respectively), while the mapping $\mathcal{M}_0$ is defined by the datalog program formed by the rules:

```
professor(X, Y, Z) :- s1(X, Y, Z).        professor(X, Y, Z) :- s4(Z, Y, _ , X).
student(X, Y, Z) :- s2(Y, X, Z).          exam_data(X, Y, Z, W) :- s3(Y, X, Z, W).
```

Answering a query over the global schema $\mathcal{G}$ is done by first populating $\mathcal{G}$ with the data retrieved from a given database instance $\mathcal{D}$ for the sources according to the mapping $\mathcal{M}$, and evaluating the query on this "retrieved" database. However, while carrying out such an integration, it often happens that the retrieved (global) database, denoted by $ret(\mathcal{I}, \mathcal{D})$, is inconsistent w.r.t. Σ, since data stored in local and autonomous sources will in general not satisfy all constraints expressed on the global schema.

Example 6.2. Consider again the data integration system of Example 6.1, and a database instance $\mathcal{D}_0$ for the source $\mathcal{S}_0$ consisting of the following facts: $\mathtt{s}_1$(‘NI65’, ‘Nick’, ‘www.nick.edu’), $\mathtt{s}_4$(‘www.frank.edu’, ‘Frank’, ‘Rome’, ‘FR70’), $\mathtt{s}_4$(‘www.nick.mat.edu’, ‘Nick’, ‘Rome’, ‘NI65’), $\mathtt{s}_2$(‘John’, ‘JO75’, ‘Cosenza’), and $\mathtt{s}_3$(‘JO75’, ‘MA62’, ‘DiscreteMathematics’, ‘30’). Then, the retrieved global database $ret(\mathcal{I}_0, \mathcal{D}_0)$ consists of the facts:

```
professor(‘NI65’, ‘Nick’, ‘www.nick.edu’).
professor(‘NI65’, ‘Nick’, ‘www.nick.mat.edu’).
professor(‘FR70’, ‘Frank’, ‘www.frank.edu’).
student(‘JO75’, ‘John’, ‘Cosenza’).
exam_data(‘MA62’, ‘JO75, ‘DiscreteMathematics’, ‘30’).
```

Notice that $ret(\mathcal{I}_0, \mathcal{D}_0)$ is inconsistent; indeed, (i) there is a violation of the key for the relation professor (cf. two different web pages are associated with ‘NI65’), and (ii) there is a violation of the inclusion dependency stating that identifiers for the professors in exam_data must be in the relation professor as well (cf. ‘MA62’).

To remedy this problem, several approaches (see, e.g., [7–12,38]) defined the semantics of a data integration system $\mathcal{I}$ in terms of the repairs $rep(\mathcal{I}, \mathcal{D})$ of the database $ret(\mathcal{I}, \mathcal{D})$. Intuitively, each repair $\mathcal{R} \in rep(\mathcal{I}, \mathcal{D})$ is obtained by properly adding and deleting facts from $ret(\mathcal{I}, \mathcal{D})$ in order to satisfy constraints in Σ. These repairs depend on the interpretation of the mappings in $\mathcal{M}$, which, in fact, impose restrictions or preferences on the possibility of adding or removing facts from $ret(\mathcal{I}, \mathcal{D})$ to repair constraint violations. In the INFOMIX project, we have adopted the *loosely-sound* semantics according to which mappings might retrieve only a subset of the tuples needed for answering the query.

Definition 6.3. Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a (GAV) data integration system where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, and let $\mathcal{D}$ be a database for $\mathcal{S}$. Then, a database $\mathcal{R}$ for $\mathcal{G}$ is a *repair* for $\mathcal{I}$ w.r.t. $\mathcal{S}$ under the loosely-sound semantics if:

- $\mathcal{R}$ satisfies all the constraints in Σ, and
- there is no database $\mathcal{R}'$ for $\mathcal{G}$ satisfying all the constraints in Σ such that $(ret(\mathcal{I}, \mathcal{D}) - \mathcal{R}') \subset (ret(\mathcal{I}, \mathcal{D}) - \mathcal{R})$.

Note that, according to this semantics, we can add an unbounded number of tuples to repair violations of inclusion dependencies; nonetheless, the semantics is loose in the sense that, in order to repair keys and exclusion dependencies, we are allowed to delete a minimal set of tuples.

Example 6.4. Consider again the data integration system of Example 6.1, and the retrieved global database in Example 6.2. According to the loosely-sound semantics, we can repair the violation of the key by deleting either the tuple professor(‘NI65’, ‘Nick’, ‘www.nick.edu’), or the tuple professor(‘NI65’, ‘Nick’, ‘www.nick.mat.edu’),—these two repairs are, in fact, the only minimal ones. Moreover, the inclusion dependency can be repaired by adding (at least) a tuple of the form professor(‘MA62’, X, Y), where X and Y are arbitrary values denoting the name and the home page address for the professor with code ‘MA62’.

Data integration systems are generally queried by means of *conjunctive queries*. Formally, a *conjunctive query* q over a database schema $\mathcal{G}$ is a rule of the form q: $q(\mathbf{u}) \leftarrow r_1(\mathbf{u_1}) \wedge \cdots \wedge r_n(\mathbf{u_n})$, where $n \geqslant 0$, $r_1, \ldots, r_n$ are relation names (not necessarily distinct) of $\mathcal{G}$; q is a relation name not in the schema of $\mathcal{G}$; and, $\mathbf{u}, \mathbf{u_1}, \ldots, \mathbf{u_n}$ are lists of terms (i.e., variables or constants) of appropriate length. Given a database $\mathcal{DB}$ for $\mathcal{G}$, the answer to q over $\mathcal{DB}$, denoted $q^{\mathcal{DB}}$, is the set of substitutions ϑ for the variables in $\mathbf{u}$, such that the formula $r_1(\theta(\mathbf{u_1})) \wedge \cdots \wedge r_n(\theta(\mathbf{u_n}))$ evaluates to true in $\mathcal{DB}$. Then, given a data integration system $\mathcal{I}$ and a source database $\mathcal{D}$, a query for $\mathcal{I}$ is simply a query for the global schema of $\mathcal{I}$; and, the answer to q is defined as the set $ans(q, \mathcal{I}, \mathcal{D}) = \bigcap_{\mathcal{R} \in rep(\mathcal{I}, \mathcal{D})} q^{\mathcal{R}}$.

Example 6.5. A conjunctive query for the data integration system $\mathcal{I}_0$ is $q_0(X) \leftarrow \texttt{professor}(X, Y, Z)$. This query asks for the identifiers of the professors. Given the form of the repairs described in Example 6.4, the answer to q_0 is {{X/'FR70'}, {X/'NI65'}, {X/'MA62'}}. Note that X/'NI65' and X/'MA62' belong to this answer, even though they are involved in constraint violations. On the other hand, for the query $q_1(Z) \leftarrow \texttt{professor}(X, Y, Z)$ asking for the home pages of the professors, the answer is {{Z/'www.frank.edu'}}, given that we cannot be cautiously sure that, for instance, 'www.nick.edu' is the web page for 'NI65'.

6.2. *Data integration using computational logic*

In order to design effective systems for query answering in data integration settings, in the INFOMIX project a transformation of query answering in the repair semantics to logic programs has been employed. Similar strategies have been used also in other approaches—see, e.g., [7–9,11].

Given a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, a database $\mathcal{D}$ for $\mathcal{S}$, and a query q over $\mathcal{G}$, the query q, the constraints Σ of $\mathcal{G}$ and the mapping assertions $\mathcal{M}$, are encoded into a logic program $\Pi(q, \mathcal{I})$ using unstratified negation, such that the stable models of $\Pi(q, \mathcal{I}) \cup \mathcal{D}$ yield the repairs of the global database.

Specifically, the program $\Pi(q, \mathcal{I})$ consists of the union of two disjoint programs: $\Pi_{\mathrm{ID}}(q, \mathcal{I})$ and $\Pi_{\mathrm{KEM}}(\mathcal{I})$. In particular, $\Pi_{\mathrm{ID}}(q, \mathcal{I})$ is the non-recursive Datalog program produced by the algorithm ID-rewrite specified in [10]—this is, in fact, a rewriting of the query where inclusion dependencies are intensionally compiled-in. $\Pi_{\mathrm{KEM}}(\mathcal{I})$ is instead a general Datalog$^{\neg}$ program whose aim is to retrieve data from $\mathcal{D}$ by means of the mappings in $\mathcal{I}$ and to resolve the conflicts for key and exclusion dependencies by a suitable use of unstratified negation. The way the rewriting is computed is informally described by means of the following example; formal definitions of the rewriting can be found in [39,40].

Example 6.6. Given the data integration system $\mathcal{I}_0$ and the query q_0 of Example 6.5, the algorithm ID-rewrite produces the following rules in $\Pi_{\mathrm{ID}}(\mathtt{q}_0, \mathcal{I}_0)$: $\mathtt{q_0(X)}$:- professor(X, Y, Z), $\mathtt{q_0(X)}$:- exam_data(X, Y, Z, W). Intuitively, the first rule is the straightforward encoding of the conjunctive query q_0, while the second rule has been added to take into account the inclusion dependency between relations professor and exam_data.

Moreover, the program $\Pi_{\mathrm{KEM}}(\mathcal{I}_0)$ can be partitioned as $\Pi_K(\mathcal{I}_0) \cup \Pi_E(\mathcal{I}_0) \cup \Pi_M(\mathcal{I}_0)$. Here, $\Pi_M(\mathcal{I}_0)$ is a Datalog program retrieving data from the sources by means of the mappings of $\mathcal{M}$ into suitable auxiliary predicates.

$$
\begin{array}{ll}
\texttt{professorD(X,Y,Z) :- s}_1\texttt{(X,Y,Z).} & \texttt{professorD(X,Y,Z) :- s}_4\texttt{(Z,Y,\_,X).} \\
\texttt{studentD(X,Y,Z) :- s}_2\texttt{(Y,X,Z).} & \texttt{exam\_dataD(X,Y,Z,W) :- s}_3\texttt{(Y,X,Z,W).}
\end{array}
$$

$\Pi_M(\mathcal{I}_0)$ is a Datalog$^{\neg}$ program resolving the key-constraints conflicts. It uses auxiliary overlined predicates; intuitively, if a fact $\overline{\mathtt{p}}\mathtt{(a)}$ is derived, then p(a) should not be in the (repair of the) retrieved database.

$$
\begin{array}{l}
\texttt{professor(X,Y,Z) :- professorD(X,Y,Z), not }\overline{\texttt{professor}}\texttt{(X,Y,Z).} \\
\overline{\texttt{professor}}\texttt{(X,Y,Z) :- professorD(X,Y,Z), professor(X,Y1,Z1), Y} \neq \texttt{Y1.} \\
\overline{\texttt{professor}}\texttt{(X,Y,Z) :- professorD(X,Y,Z), professor(X,Y1,Z1), Z} \neq \texttt{Z1.} \\
\\
\texttt{student(X,Y,Z) :- studentD(X,Y,Z), not }\overline{\texttt{student}}\texttt{(X,Y,Z).} \\
\overline{\texttt{student}}\texttt{(X,Y,Z) :- studentD(X,Y,Z), student(X,Y1,Z1), Y} \neq \texttt{Y1.} \\
\overline{\texttt{student}}\texttt{(X,Y,Z) :- studentD(X,Y,Z), student(X,Y1,Z1), Z} \neq \texttt{Z1.} \\
\\
\texttt{exam\_data(X,Y,Z,W) :- exam\_dataD(X,Y,Z,W), not }\overline{\texttt{exam\_data}}\texttt{(X,Y,Z,W).} \\
\overline{\texttt{exam\_data}}\texttt{(X,Y,Z,W) :- exam\_dataD(X,Y,Z,W), exam\_data(X,Y,Z,W1), W} \neq \texttt{W1.}
\end{array}
$$

To have an intuition on how these rules solve the conflicts in the data, consider the relation exam_data. Then, the last two rules are used to force that each tuple retrieved in exam_dataD(X, Y, Z, W) can be (virtually) materialized in a repair only if a tuple of the form exam_data(X, Y, Z, W1) is not materialized in its turn, where $\mathtt{W} \neq \mathtt{W1}$. By this way, repairs are guaranteed not to have conflicts on the key of exam_data (which is, in fact, formed by the first three attributes).

$\Pi_E(\mathcal{I}_0)$ is a Datalog$^\neg$ program resolving the exclusion-dependencies conflicts in a similar way as $\Pi_M(\mathcal{I}_0)$.

$\texttt{professor(X,Y,Z)} \texttt{:-} \texttt{professorD(X,Y,Z)},\ \text{not}\ \overline{\texttt{professor}}\texttt{(X,Y,Z)}.$

$\overline{\texttt{professor}}\texttt{(X,Y,Z)} \texttt{:-} \texttt{professorD(X,Y,Z)},\ \texttt{student(X,Y1,Z1)}.$

$\texttt{student(X,Y,Z)} \texttt{:-} \texttt{studentD(X,Y,Z)},\ \text{not}\ \overline{\texttt{student}}\texttt{(X,Y,Z)}.$

$\overline{\texttt{student}}\texttt{(X,Y,Z)} \texttt{:-} \texttt{student(X,Y,Z)},\ \texttt{professor(X,Y1,Z1)}.$

The correctness of the rewriting has been formally proved in [39,40].

Theorem 6.7. [39,40] *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, $\mathcal{D}$ be a database for $\mathcal{S}$, and q be a query over $\mathcal{G}$. Then, ans$(q, \mathcal{I}, \mathcal{D})$ coincides with $Ans_c(q, \Pi(q, \mathcal{I}) \cup \mathcal{D})$.*

Example 6.8. The program $\Pi(\texttt{q}_0, \mathcal{I}_0) \cup \mathcal{D}_0$ presented in Example 6.6 has two stable models M_1 and M_2. The part which is relevant for query answering is comprised of the predicates in $\Pi_{\text{ID}}(q_0, \mathcal{I}_0)$:

$M_1/_{\Pi_{\text{ID}}(q_0,\mathcal{I}_0)\cup\mathcal{D}_0} = \mathcal{D}_0 \cup \{\texttt{q}_0(\text{‘FR70’}), \texttt{q}_0(\text{‘MA62’}), \texttt{q}_0(\text{‘NI65’}),$
professor(‘NI65’, ‘Nick’, ‘www.nick.edu’),
professor(‘FR70’, ‘Frank’, ‘www.frank.edu’),
student(‘JO75’, ‘John’, ‘Cosenza’),
exam_data(‘MA62’, ‘JO75’, ‘DiscreteMathematics’, ‘30’)},

$M_2/_{\Pi_{\text{ID}}(q_0,\mathcal{I}_0)\cup\mathcal{D}_0} = \mathcal{D}_0 \cup \{\texttt{q}_0(\text{‘FR70’}), \texttt{q}_0(\text{‘MA62’}), \texttt{q}_0(\text{‘NI65’})$
professor(‘NI65’, ‘Nick’, ‘www.nick.mat.edu’),
professor(‘FR70’, ‘Frank’, ‘www.frank.edu’),
student(‘JO75’, ‘John’, ‘Cosenza’),
exam_data(‘MA62’, ‘JO75’, ‘DiscreteMathematics’, ‘30’)}.

Therefore, $Ans_c(\texttt{q}_0, \Pi(\texttt{q}_0, \mathcal{I}_0) \cup \mathcal{D}_0) = \{\{\text{X/‘FR70’}\}, \{\text{X/‘MA62’}\}, \{\text{X/‘NI65’}\}\}$.

Before leaving this section, we remark that while presented in the context of the loosely-sound semantics, our results on the application of the Magic Sets for the optimization of logic programs encoding the repair computation in data integration systems can be extended to other semantics (and, also, other kinds of rewritings) in the literature.

6.3. Magic Sets for data integration

The binding propagation techniques proposed in this paper can be profitably exploited to isolate the relevant part of a database. Importantly, our optimization fits perfectly the data integration framework. Indeed, the loosely-sound semantics for data integration always guarantees the existence of a database repair no matter of the types of constraints in Σ, provided that the schema is *non-key-conflicting* [36]. Thus, the resulting logic program (whose stable models are in a one-to-one correspondence with the repair) is consistent and our rewriting fully preserves the original semantics of the data-integration query, since query equivalence is ensured (Theorem 4.10).

Theorem 6.9. *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, $\mathcal{D}$ be a database for $\mathcal{S}$, and q be a query over $\mathcal{G}$. Then, ans$(q, \mathcal{I}, \mathcal{D})$ coincides with $Ans_c(q, \texttt{MS}^\neg(q, \Pi(q, \mathcal{I})) \cup \mathcal{D})$.*

Proof. In the loosely-sound semantics, $\Pi(q, \mathcal{I}) \cup \mathcal{D}$ is guaranteed to be consistent, because the existence of (at least) a repair is guaranteed by Definition 6.3. Thus, the result follows immediately by Theorem 6.7 and by the correctness of the application of the Magic Set technique (cf. Theorem 4.10). □

In order to test the effectiveness of the Magic Set technique for query optimization in data integration systems, we have carried out some experiments on the demonstration scenario of the INFOMIX project, which refers to the information system of the University "La Sapienza" in Rome. These experiments have been carried out in order to check whether our technique was working effectively and was bringing some benefits in a particular real world data-integration setting.

A detailed discussion of the results is available in [37]. The results confirmed that on various practical queries the performance is greatly improved by Magic Sets. For some queries, no binding propagations occurred, causing only a light overhead. Moreover, two main parameters influencing query answering time have been identified:

- the size of the retrieved global database, influencing mainly the instantiation time and the size of the ground logic program encoding repair computation; and,
- the number of conflicts (over key and exclusion dependencies), determining the number of repairs for the retrieved global database—in general, the number of repairs scales exponentially w.r.t. the number of conflicts.

Interestingly, the Magic Set technique provides benefits with respect to both parameters. Indeed, on the one hand, by limiting the computation on the fraction of the retrieved global database which is involved in the propagation of the query binding, it generally produces smaller ground programs than the basic ground encoding. On the other hand, by disregarding conflicts that are irrelevant for answering the query at hand (i.e., those conflicts that involve facts on which no binding propagation occurred), it has potential to lead to exponentially better performances than the evaluation on the whole retrieved global database.

These two factors can be seen as the main reasons for the gain with the Magic Set method that we have observed in our experiments. However, it is unclear whether these results can be generalized, and in particular whether they can entail predictions on the gain in any particular application. However, we conjecture that the performance mainly depends on the overall number of conflicts in the data and on the structure of the issued queries. A comprehensive quantitative evaluation of the benefits would require the definition of a benchmarking suite over very large real-world data, where queries are designed to reflect actual business needs. We are not aware of such a suite; compiling it and carrying out a systematic analysis of the performance of the Magic Set method requires major efforts and is left for future work.

7. Related work

7.1. Evaluation strategies for Datalog$^\neg$ programs

While answering a user query, often only a strict subset of all models and of each of these only a certain fragment needs to be considered. This intuition is exploited by top-down techniques which only consider atoms necessary to answer the actual query and often outperform bottom-up methods. In fact, top-down approaches exploit constants in the query to propagate bindings into rule bodies.

In order to optimize query evaluation in bottom-up systems (like deductive database systems), several works proposed the simulation of top-down strategies by means of suitable transformations introducing new predicates and rewriting clauses. Among them, the Magic Set method [1,14–16] is one of the best known techniques for the optimization of Datalog queries. Many extensions and refinements of Magic Sets have been proposed, addressing e.g. query constraints [41], modular stratification and well-founded semantics [42,43], integration into cost-based query optimization [44]. The research on enhancements to the Magic Set method is still going on. For instance, a Magic Set technique for the class of *soft-stratifiable* programs was recently presented [45], and in [46,47] Magic Sets techniques for disjunctive programs were proposed.

An extension of the Magic Set technique for *positive* Datalog programs with integrity constraints has been presented in [48]. The proposed method is shown to be *brave complete* and *cautious sound*. Comparing this method to our approach, we observe that: (1) Our method is more general than the method in [48], since the latter deals only with a strict subset of Datalog$^\neg$ (recall that an integrity constraint $:- C$ is just a shorthand for $p :- C, \text{not } p$); while our method supports full Datalog$^\neg$, allowing for unstratified negation. (2) Our method has much better semantic properties than [48]. Indeed, [48] does not guarantee query equivalence under any conditions; while we do guarantee full query equivalence, unless the input program is inconsistent (see Theorem 4.10). Query equivalence is in fact very relevant for data integration applications, and the key property which allowed us to employ our technique in the INFOMIX project.

We have introduced some strong modularity results which are strictly related to splitting sets, as defined in [17], or equivalently potential use modules as defined in [18]. The main difference is that our notion of modules and independent sets guarantee query equivalence for consistent programs, which does not hold for these previous notions. In particular, Theorem 3.6, and the second items of Theorem 3.4 and Theorem 3.7, respectively, do not hold for the modules as defined in [17] and [18].

Note that a different kind of query optimization for data integration has been described in [49]. This approach exploits a *localization strategy* to reduce complexity, in which the retrieved data is decomposed into two parts: facts which will possibly be touched by a repair and facts which for sure will not. Then, query answering is achieved by means of some *techniques for recombining* the decomposed parts which interleave logic programming and relational database engines. Therefore, [49] does not exploit constants that appear in the query to optimize query evaluation, but only inconsistent (w.r.t. constraints of the global schema) portions of the retrieved database. In fact, no systematic technique for query optimization in data integration systems exploiting binding propagations has been proposed in the literature so far.

7.2. Systems for handling inconsistency and incompleteness of data

Query answering in the presence of conflicting and incomplete data has been the subject of a large body of research in the last few years. In this paper, we have focused on data integration systems in a GAV setting where key constraints, exclusion dependencies, and inclusion dependencies can be issued over the global schema. A closely related framework is that of answering queries in *data exchange systems*.

Data exchange is the problem of translating an instance of a source schema into an instance of a target schema satisfying all the source-to-target dependencies, and all the integrity constraints issued over the target schema. In other words, we are interested in *materializing* the target instance that reflects the source as accurately as possible. A clear and comprehensive framework for data exchange in the relational context was recently developed by Fagin, Kolaitis, Miller, and Popa [50,51]. Within this framework, it emerged that several alternative ways of translating the data can be singled out (corresponding to the ways of enforcing the source-to-target dependencies and the constraints on the global schema). Hence, answering a user query Q over the target schema amounts (as in the case of data integration) at computing its *consistent answers*, i.e., the answer that are true in every possible translation.

However, because of the different aim of data exchange, the kinds of constraint investigated in this field differ from those typically issued over data integration systems. Indeed, many works (see, e.g., [50–53]) study the exchange problem when source-to-target constraints are *tuple generating dependencies* (TGDs) and when target constraints consist of *weakly-acyclic* TGDs and *equality generating dependencies* (EGDs). When dealing with these dependencies that can be "repaired" by just materializing some further tuples in the target instance or by recognizing that two tuples are equal, in [50] it has been shown that among the solutions of a solvable data exchange problem, there exists a most compact one (up to isomorphism), called the *core*. In fact, cores were recently proven to be computable in polynomial time [54].

It is worthwhile noting that in the data integration setting there is little hope to materialize (in polynomial time) a repair which is representative of all the repairs for the system. Indeed, when keys, functional dependencies, or exclusion dependencies are considered, repairs can also be obtained by means of tuple deletions (see, e.g., [28]), which lead to an intrinsically non-monotonic behavior and to the co-NP-hardness as informally discussed in Section 2.

Finally, other related approaches have been proposed in the context of query answering from uncertain, probabilistic, and incomplete databases (see, e.g., [55–58]). For instance, MYSTIQ [57] is a system for efficiently answering (uncertainty) queries on probabilistic databases, while *Trio* [58] is a system that supports accuracy and lineage of data as first class concepts, along with the data itself. In the context of data integration, uncertainty comes into play when several possible repairs can be singled out for an inconsistent/incomplete database. And, in fact, the classical *consistent* answering semantics can be simply viewed as the lower approximation to query results in an uncertain relational database, as recently argued in [59]. This correspondence suggests that other kinds of semantics (e.g., the *maybe* or probabilistic semantics) deserve further attention in both the field of data integration and exchange.

8. Conclusion

In this paper, we have presented a Magic Set technique for the whole class of Datalog$^\neg$ programs, by providing results that are relevant for both theory and practice. On the theory side, our modularity results provide a better understanding of the structural properties of Datalog$^\neg$, complementing and advancing on previous works on modularity properties of this language. Moreover, the $\mathtt{MS}^{\neg}$ algorithm generalizes Magic Sets, enlarging significantly their range of applicability to the full class of Datalog$^{\neg}$ programs under the stable model semantics. Importantly, our work can be profitably exploited for data integration systems. Results of experiments over the Demo Scenario of the INFOMIX project show that the application of our techniques allows us to solve very advanced data-integration tasks.

We conclude by observing that, our Magic Set technique can be profitably exploited in other approaches to data integration such as [7–9,11]. In fact, all these approaches reduce answering a user query to cautious reasoning over a logic program which is guaranteed to be consistent. Some of these approaches actually use disjunctive datalog programs, possibly with unstratified negation. In this respect, we point out that the algorithm of this paper can be coupled with the method in [47], which is defined on positive disjunctive programs, obtaining a Magic Set method for arbitrary disjunctive programs.

References

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

[2] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Logic Programming: Proceedings Fifth International Conference and Symposium, MIT Press, Cambridge, MA, 1988, pp. 1070–1080.

[3] N. Bidoit, C. Froidevaux, Negation by default and unstratifiable logic programs, Theoret. Comput. Sci. 78 (1991) 85–112.

[4] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surveys 33 (3) (2001) 374–425.

[5] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (3) (2006) 499–562.

[6] I. Niemelä, P. Simons, T. Syrjänen, Smodels: A system for answer set programming, in: C. Baral, M. Truszczyński (Eds.), Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, NMR 2000, Breckenridge, Colorado, USA, 2000.

[7] M. Arenas, L.E. Bertossi, J. 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, Springer, 2000, pp. 27–41.

[8] G. Greco, S. Greco, E. 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, in: Lecture Notes in Artificial Intelligence, vol. 2237, Springer, 2001, pp. 348–364.

[9] P. Barceló, L. Bertossi, Repairing databases with annotated predicate logic, in: Proc. the 10th Int. Workshop on Non-Monotonic Reasoning, NMR 2002, 2002, pp. 160–170.

[10] A. Calì, D. Lembo, R. 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, pp. 16–21.

[11] L. Bravo, L. Bertossi, Logic programming for consistently querying data integration systems, in: Proc. of the 18th Int. Joint Conf. on Artificial Intelligence, IJCAI 2003, 2003, pp. 10–15.

[12] J. Chomicki, J. Marcinkowski, Minimal-change integrity maintenance using tuple deletions, Inform. and Comput. 197 (1–2) (2005) 90–121.

[13] J. Schlipf, The expressive powers of logic programming semantics, J. Comput. System Sci. 51 (1) (1995) 64–86, abstract in: Proc. PODS '90, pp. 196–204.

[14] F. Bancilhon, D. Maier, Y. Sagiv, J.D. Ullman, Magic Sets and other strange ways to implement logic programs, in: Proc. Int. Symposium on Principles of Database Systems, 1986, pp. 1–16.

[15] C. Beeri, R. Ramakrishnan, On the power of magic, J. Logic Program. 10 (1–4) (1991) 255–259.

[16] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, R. Ramakrishnan, Magic is relevant, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 1990, pp. 247–258, URL: http://citeseer.nj.nec.com/article/mumick90magic.html.

[17] V. Lifschitz, H. Turner, Splitting a logic program, in: P. Van Hentenryck (Ed.), Proceedings of the 11th International Conference on Logic Programming, ICLP '94, MIT Press, Santa Margherita Ligure, 1994, pp. 23–37.

[18] T. Eiter, G. Gottlob, H. Mannila, Disjunctive datalog, ACM Trans. Database Systems 22 (3) (1997) 364–418.

[19] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Generation Computing 9 (1991) 365–385.

[20] M. Lenzerini, Data integration: A theoretical perspective, in: Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems, PODS 2002, 2002, pp. 233–246.

[21] L. Bertossi, J. Chomicki, A. Cortes, C. Gutierrez, Consistent answers from integrated data sources, in: Proc. of the 6th Int. Conf. on Flexible Query Answering Systems, FQAS 2002, 2002, pp. 71–85.

[22] L. Bertossi, J. Chomicki, Query answering in inconsistent databases, in: J. Chomicki, R. van der Meyden, G. Saake (Eds.), Logics for Emerging Applications of Databases, Springer, 2003, pp. 43–83 (Chapter 2).

[23] J. Chomicki, J. Marcinkowski, S. Staworko, Hippo: A system for computing consistent answers to a class of SQL queries, in: 9th International Conference on Extending Database Technology, EDBT 2004, Springer, 2004, pp. 841–844.

[24] J. Chomicki, J. Marcinkowski, S. Staworko, Computing consistent query answers using conflict hypergraphs, in: Proc. 13th ACM Conference on Information and Knowledge Management, CIKM 2004, ACM Press, 2004, pp. 417–426.

[25] A. Fuxman, E. Fazli, R.J. Miller, Conquer: Efficient management of inconsistent databases, in: SIGMOD Conference, 2005.

[26] A. Fuxman, R.J. Miller, First-order query rewriting for inconsistent databases, in: T. Eiter, L. Libkin (Eds.), Proceedings of the 10th International Conference on Database Theory, ICDT 2005, in: Lecture Notes in Comput. Sci., vol. 3363, Springer, 2005, pp. 337–351.

[27] M. Arenas, L. Bertossi, J. Chomicki, Scalar aggregation in FD-inconsistent databases, in: International Conference on Database Theory, ICDT 2001, Springer, 2001, pp. 39–53.

[28] J. Chomicki, J. Marcinkowski, Minimal-change integrity maintenance using tuple deletions, Inform. and Comput. 197 (2005) 90–121.
[29] M. Arenas, L.E. Bertossi, J. Chomicki, Consistent query answers in inconsistent databases, in: Proc. of the 18th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems, PODS '99, 1999, pp. 68–79.
[30] P.M. Dung, On the relations between stable and well-founded semantics of logic programs, Theoret. Comput. Sci. 105 (1) (1992) 7–25.
[31] J.D. Ullman, Principles of Database and Knowledge Base Systems, vol. 2, Computer Science Press, 1989.
[32] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison–Wesley, 1995.
[33] E.M. Arkin, C.H. Papadimitriou, M. Yannakakis, Modularity of cycles and paths in graphs, J. ACM 38 (2) (1991) 255–274.
[34] D.S. Johnson, A catalog of complexity classes, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. A, Elsevier Science, 1990 (Chapter 2).
[35] M.Y. Vardi, Complexity of relational query languages, in: Proceedings of the 14th Symposium on Theory of Computation, STOC, 1982, pp. 137–146.
[36] A. Calì, D. Lembo, R. Rosati, On the decidability and complexity of query answering over inconsistent and incomplete databases, in: Proc. of the 22st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems, PODS 2003, 2003, pp. 260–271.
[37] INFOMIX Project Team, Demo Scenario, Tech. Rep. INFOMIX S7-1, INFOMIX Project Consortium, available at http://sv.mat.unical.it/infomix, June 2004.
[38] A. Calì, D. Calvanese, G. De Giacomo, M. Lenzerini, Data integration under integrity constraints, Inform. Syst. 29 (2) (2004) 147–163.
[39] D. Lembo, M. Lenzerini, R. Rosati, Methods and techniques for query rewriting, Tech. Rep. D5.2, Infomix Consortium, October 2003.
[40] D. Lembo, Dealing with inconsistency and incompleteness in data integration, PhD thesis, Università di Roma "La Sapienza," 2004.
[41] P.J. Stuckey, S. Sudarshan, Compiling query constraints, in: Proceedings of the Thirteenth Symposium on Principles of Database Systems, PODS '94, ACM Press, 1994, pp. 56–67.
[42] K.A. Ross, Modular stratification and Magic Sets for datalog programs with negation, J. ACM 41 (6) (1994) 1216–1266.
[43] D.B. Kemp, D. Srivastava, P.J. Stuckey, Bottom-up evaluation and query optimization of well-founded models, Theoret. Comput. Sci. 146 (1995) 145–184.
[44] P. Seshadri, J.M. Hellerstein, H. Pirahesh, T.Y.C. Leung, R. Ramakrishnan, D. Srivastava, P.J. Stuckey, S. Sudarshan, Cost-based optimization for magic: Algebra and implementation, in: H.V. Jagadish, I.S. Mumick (Eds.), Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, ACM Press, 1996, pp. 435–446.
[45] A. Behrend, Soft stratification for Magic Set based query evaluation in deductive databases, in: Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press, 2003, pp. 102–110.
[46] S. Greco, Binding propagation techniques for the optimization of bound disjunctive queries, IEEE Trans. Knowledge and Data Engineering 15 (2) (2003) 368–385.
[47] C. Cumbo, W. Faber, G. Greco, Enhancing the magic-set method for disjunctive datalog programs, in: Proceedings of the 20th International Conference on Logic Programming—ICLP '04, in: Lecture Notes in Comput. Sci., vol. 3132, 2004, pp. 371–385.
[48] G. Greco, S. Greco, I. Trubitsyna, E. Zumpano, Optimization of bound disjunctive queries with constraints, theory and practice of logic programming.
[49] T. Eiter, M. Fink, G. Greco, D. Lembo, Efficient evaluation of logic programs for querying data integration systems, in: Proc. of the 19th Int. Conf. on Logic Programming, ICLP '03, 2003, pp. 163–177.
[50] R. Fagin, P. Kolaitis, R.J. Miller, L. Popa, Data exchange: Semantics and query answering, in: Special Issue for Selected Papers from the 2003 International Conference on Database Theory, Theoret. Comput. Sci. 336 (2005) 89–124.
[51] R. Fagin, P.G. Kolaitis, L. Popa, Data exchange: Getting to the core, ACM Trans. Database Systems 30 (1) (2005) 174–210.
[52] G. Gottlob, Computing cores for data exchange: New algorithms and practical solutions, in: PODS '05: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, New York, NY, 2005, pp. 148–159.
[53] P.G. Kolaitis, J. Panttaja, W. Tan, The complexity of data exchange, in: PODS '06: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, New York, NY, 2006, pp. 30–39.
[54] G. Gottlob, A. Nash, Data exchange: Computing cores in polynomial time, in: PODS '06: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, New York, NY, 2006, pp. 40–49.
[55] O. Benjelloun, A.D. Sarma, C. Hayworth, J. Widom, An introduction to ULDBs and the trio system, in: Special Issue on Probabilistic Databases, IEEE Data Engineering Bulletin 29 (2006) 5–16.
[56] N.N. Dalvi, G. Miklau, D. Suciu, Asymptotic conditional probabilities for conjunctive queries, in: T. Eiter, L. Libkin (Eds.), Proceedings of the 10th International Conference on Database Theory, ICDT '05, in: Lecture Notes in Comput. Sci., vol. 3363, Springer, 2005, pp. 289–305.
[57] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, D. Suciu, MYSTIQ: A system for finding more answers by using probabilities, in: SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, NY, 2005, pp. 891–893.
[58] J. Widom, Trio: A system for integrated management of data, accuracy, and lineage, in: Proceedings of the second Biennial Conference on Innovative Data Systems Research CIRD '05, 2005, pp. 262–276.
[59] L. Libkin, Data exchange and incomplete information, in: PODS '06: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, New York, NY, 2006, pp. 60–69.