Magic-Sets for *Datalog* with Existential Quantifiers

Mario Alviano, Nicola Leone, Marco Manna*,
Giorgio Terracina, and Pierfrancesco Veltri

Department of Mathematics, University of Calabria, Italy
{alviano,leone,manna,terracina,veltri}@mat.unical.it

Abstract. *Datalog*$^\exists$ is the extension of *Datalog* allowing existentially quantified variables in rule heads. This language is highly expressive and enables easy and powerful knowledge-modelling, but the presence of existentially quantified variables makes reasoning over *Datalog*$^\exists$ undecidable in the general case. Restricted classes of *Datalog*$^\exists$, such as *Shy*, have been proposed in the literature with the aim of enabling powerful, yet decidable query answering on top of *Datalog*$^\exists$ programs. However, in order to make such languages attractive it is necessary to guarantee good performance for query answering tasks. This paper works in this direction: improving the performance of query answering on *Datalog*$^\exists$. To this end, we design a rewriting method extending the well-known Magic-Sets technique to any *Datalog*$^\exists$ program. We demonstrate that our rewriting method preserves query equivalence on *Datalog*$^\exists$, and can be safely applied to *Shy* programs. We therefore incorporate the Magic-Sets method in DLV$^\exists$, a system supporting *Shy*. Finally, we carry out an experiment assessing the positive impact of Magic-Sets on DLV$^\exists$, and the effectiveness of the enhanced DLV$^\exists$ system compared to a number of state-of-the-art systems for ontology-based query answering.

1 Introduction

Datalog is one of the best-known rule-based languages, and extensions of it are used in a wide context of applications. *Datalog* is especially useful in various Artificial Intelligence applications as it allows for effective encodings of incomplete knowledge. However, in recent years an important shortcoming of *Datalog*-based languages became evident, especially in the context of Semantic Web applications: The language does not permit the generation and the reasoning about unnamed individuals in an obvious way. In particular, it is weak in supporting many cases of existential quantification needed in the field of ontology-based query answering (QA), which is becoming more and more a challenging task [10,12,8,16] attracting also interest of commercial companies.

As an example, big companies such as Oracle are adding ontological reasoning modules on top of their existing software. In this context, queries are not merely evaluated on an extensional relational database D, but over a logical theory combining D with an *ontological theory* Σ. More specifically, Σ describes rules and constraints for inferring intensional knowledge from the extensional data stored in D. Thus, for a conjunctive query (CQ) q, it is not only checked whether D entails q, but rather whether $D \cup \Sigma$ does.

* Marco Manna's work was supported by the European Commission through the European Social Fund and by Calabria Region.

A key issue in ontology-based QA is the design of the language that is provided for specifying the ontological theory Σ. In this regard, Datalog$^{\pm}$ [8], the novel family of *Datalog*-based languages proposed for tractable QA over ontologies, is arousing increasing interest. This family, generalizing well-known ontology specification languages, is mainly based on *Datalog*$^{\exists}$, the natural extension of *Datalog* [1] that allows $\exists$-quantified variables in rule heads. Unfortunately, a major challenge for *Datalog*$^{\exists}$ is decidability. In fact, without any restriction, QA over *Datalog*$^{\exists}$ is not decidable; thus, the identification of subclasses for which QA is decidable is desirable.

Different *Datalog*$^{\exists}$ fragments have been proposed in the literature, which essentially rely on four main syntactic paradigms called *guardedness* [7], *weak-acyclicity* [14], *stickiness* [9] and *shyness* [17]. The complexity of QA on these fragments, which offer different levels of expressivity, ranges from $\mathbf{AC}_0$ to **EXP**. Hence, optimization techniques are crucial to make QA effectively usable in real world scenarios, especially for those fragments providing high degrees of expressiveness.

The contribution of this paper goes exactly in this direction. We first focus on optimization strategies for improving QA tasks over decidable *Datalog*$^{\exists}$ fragments, and in particular on the well-known Magic-Sets optimization. We then focus on *Shy*, a *Datalog*$^{\exists}$ class based on shyness, enabling tractable QA, offering a good balance between expressivity and complexity, and suitable for an efficient implementation.

The original Magic-Sets technique was introduced for *Datalog* [4]. Many authors have addressed the issue of extending Magic-Sets to broader languages, including non-monotonic negation [13], disjunctive heads [15,2], and uninterpreted function symbols [11,3]. In order to bring Magic-Sets to the more general framework of *Datalog*$^{\exists}$, two main difficulties must be faced: the first is, obviously, the presence of existentially quantified variables; the second regards the correctness proof of a Magic-Sets rewriting. In fact, while a *Datalog* program can be associated with a universal model that comprises finitely many atoms, the universal model of a *Datalog*$^{\exists}$ program comprises in general infinitely many atoms. These difficulties are faced and solved in this paper, whose main contributions are as follows:

- We design a Magic-Sets rewriting algorithm handling existential quantifiers, and thus suitable for *Datalog*$^{\exists}$ programs in general.
- We demonstrate that our Magic-Sets algorithm preserves query equivalence for any *Datalog*$^{\exists}$ program.
- We show how Magic-Sets can be safely applied to *Shy* programs.
- We implement the Magic-Sets strategy in DLV$^{\exists}$, a bottom-up evaluator of CQs over *Shy* programs.
- We experiment on QA over a well-known benchmark ontology, named LUBM. The results evidence the optimization potential provided by Magic-Sets and confirm the effectiveness of DLV$^{\exists}$, which outperforms all compared systems in the benchmark.

2 *Datalog*$^{\exists}$

In this section we introduce *Datalog*$^{\exists}$ programs and CQs, and we equip such structures with a formal semantics.

2.1 Preliminaries

The following notation will be used throughout the paper. We always denote by Δ_C, Δ_N, $\Delta_\forall$ and $\Delta_\exists$, countably-infinite pairwise-disjoint domains of *terms* called *constants*, *nulls*, *universal variables* and *existential variables*, respectively; by Δ, the union of these four domains; by t, a generic *term*; by $\mathtt{c}$, $\mathtt{d}$ and $\mathtt{e}$, constants; by φ, a null; by $\mathtt{X}$ and $\mathtt{Y}$, variables; by $\mathbf{X}$ and $\mathbf{Y}$, sets of variables; by Π an alphabet of *predicate symbols* each of which, say $\mathtt{p}$, has a fixed nonnegative arity, denoted by $\mathsf{arity}(\mathtt{p})$; by $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, *atoms* being expressions of the form $\mathtt{p}(t_1, \ldots, t_k)$, where $\mathtt{p}$ is a predicate symbol and $t_1, \ldots, t_k$ is a *tuple* of terms (also denoted by $\bar{t}$). Moreover, if the tuple of an atom consists of only constants and nulls, then this atom is called *ground*; if $T \subseteq \Delta_C \cup \Delta_N$, then $\mathsf{base}(T)$ denotes the set of all ground atoms that can be formed with predicate symbols in Π and terms from T; if $\mathbf{a}$ is an atom, then $\mathsf{pred}(\mathbf{a})$ denotes the predicate symbol of $\mathbf{a}$; if ς is any formal structure containing atoms, then $\mathsf{dom}(\varsigma)$ denotes all the terms from $\Delta_C \cup \Delta_N$ occurring in the atoms of ς.

A *mapping* is a function $\mu : \Delta \to \Delta$ s.t. $c \in \Delta_C$ implies $\mu(c) = c$, and $\varphi \in \Delta_N$ implies $\mu(\varphi) \in \Delta_C \cup \Delta_N$. Let T be a subset of Δ. The application of μ to T, denoted by $\mu(T)$, is the set $\{\mu(t) \mid t \in T\}$. The restriction of μ to T, denoted by $\mu|_T$, is the mapping μ' s.t. $\mu'(t) = \mu(t)$ for each $t \in T$, and $\mu'(t) = t$ for each $t \notin T$. In this case, we also say that μ is an *extension* of μ', denoted by $\mu \supseteq \mu'$. For an atom $\mathbf{a} = \mathtt{p}(t_1, \ldots, t_k)$, we denote by $\mu(\mathbf{a})$ the atom $\mathtt{p}(\mu(t_1), \ldots, \mu(t_k))$. For a formal structure ς containing atoms, we denote by $\mu(\varsigma)$ the structure obtained by replacing each atom $\mathbf{a}$ of ς with $\mu(\mathbf{a})$. The *composition* of a mapping μ_1 with a mapping μ_2, denoted by $\mu_2 \circ \mu_1$, is the mapping associating each $t \in \Delta$ to $\mu_2(\mu_1(t))$. Let ς_1 and ς_2 be two formal structures containing atoms. A *homomorphism* from ς_1 to ς_2 is a mapping h s.t. $h(\varsigma_1)$ is a substructure of ς_2 (for example, if ς_1 and ς_2 are sets of atoms, $h(\varsigma_1) \subseteq \varsigma_2$). A *substitution* is a mapping σ s.t. $t \in \Delta_N$ implies $\sigma(t) = t$, and $t \in \Delta_V$ implies $\sigma(t) \in \Delta_C \cup \Delta_N \cup \{t\}$.

2.2 Programs and Queries

A *Datalog*$^\exists$ *rule* r is a finite expression of the following form:

$$\forall \mathbf{X} \exists \mathbf{Y} \ \ \mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]} \leftarrow \mathbf{conj}_{[\mathbf{X}]} \tag{1}$$

where (i) $\mathbf{X} \subseteq \Delta_\forall$ and $\mathbf{Y} \subseteq \Delta_\exists$ (next called $\forall$-variables and $\exists$-variables, respectively); (ii) $\mathbf{X}' \subseteq \mathbf{X}$; (iii) $\mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]}$ stands for an atom containing only and all the variables in $\mathbf{X}' \cup \mathbf{Y}$; and (iv) $\mathbf{conj}_{[\mathbf{X}]}$ stands for a *conjunction* of zero or more atoms containing only and all the variables in $\mathbf{X}$. Constants are also allowed in r. In the following, $\mathsf{head}(r)$ denotes $\mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]}$, and $\mathsf{body}(r)$ the set of atoms in $\mathbf{conj}_{[\mathbf{X}]}$. Universal quantifiers are usually omitted to lighten the syntax, while existential quantifiers are omitted only if $\mathbf{Y}$ is empty. In the second case, r coincides with a standard *Datalog* rule. If $\mathsf{body}(r) = \emptyset$, then r is usually referred to as a *fact*. In particular, r is called *existential* or *ground* fact according to whether r contains some $\exists$-variable or not, respectively. A *Datalog*$^\exists$ program P is a finite set of *Datalog*$^\exists$ rules. We denote by $\mathsf{preds}(P) \subseteq \Pi$ the predicate symbols occurring in P, by $\mathsf{data}(P)$ all the atoms constituting the ground facts of P, and by $\mathsf{rules}(P)$ all the rules of P being not ground facts. A predicate $\mathtt{p} \in \Pi$ is called *intentional* if there is a rule $r \in \mathsf{rules}(P)$ s.t. $\mathtt{p} = \mathsf{pred}(\mathsf{head}(r))$; otherwise, $\mathtt{p}$ is called *extensional*. We denote by $\mathsf{idb}(P)$ and $\mathsf{edb}(P)$ the sets of the intentional and extensional predicates occurring in P, respectively.

Example 1. The next rules belong to a *Datalog*$^\exists$ program hereafter called *P-Jungle*:

r_1 : ∃Z pursues(Z,X) ← escapes(X)
r_2 : hungry(Y) ← pursues(Y,X), fast(X)
r_3 : pursues(X,Y) ← pursues(X,W), prey(Y)
r_4 : afraid(X) ← pursues(Y,X), hungry(Y), strongerThan(Y,X).

This program describes a funny scenario where an escaping, yet fast animal X may induce many other animals to be afraid. Data for *P-Jungle* could be escapes(gazelle), fast(gazelle), prey(antelope), strongerThan(lion,antelope), and possibly pursues(lion,gazelle). We will use *P-Jungle* as a running example. □

Given a *Datalog*$^\exists$ program P, a *conjunctive query* (CQ) q over P is a first-order expression of the form $\exists \mathbf{Y}\ \mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$, where $\mathbf{X} \subseteq \Delta_\forall$ are its free variables, $\mathbf{Y} \subseteq \Delta_\exists$, and $\mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$ is a conjunction containing only and all the variables in $\mathbf{X} \cup \mathbf{Y}$ and possibly some constants. To highlight the free variables, we write $q(\mathbf{X})$ instead of q. Query q is called *Boolean CQ* (BCQ) if $\mathbf{X} = \emptyset$. Moreover, q is called *atomic* if $\mathbf{conj}$ is an atom. Finally, $\mathsf{atoms}(q)$ denotes the set of atoms in $\mathbf{conj}$.

Example 2. Animals pursed by a *lion* and stronger than some other animal can be retrieved by means of a CQ ∃Y pursues(lion,X), strongerThan(X,Y). □

2.3 Semantics and Query Answering

Given a set S of atoms and an atom $\mathbf{a}$, we say S entails $\mathbf{a}$ ($S \models \mathbf{a}$ for short) if there is a substitution σ s.t. $\sigma(\mathbf{a}) \in S$. Let $P \in$ *Datalog*$^\exists$. A set $M \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$ is a *model* for P ($M \models P$) if $M \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$ for each $r \in P$ of the form (1) and substitution σ s.t. $\sigma(\mathsf{body}(r)) \subseteq M$. Let $\mathsf{mods}(P)$ denote the set of models of P. Let $M \in \mathsf{mods}(P)$. A BCQ q is *true* w.r.t. M ($M \models q$) if there is a substitution σ s.t. $\sigma(\mathsf{atoms}(q)) \subseteq M$. Analogously, the answer of a CQ $q(\mathbf{X})$ w.r.t. M is the set $\mathsf{ans}(q, M) = \{\sigma|_{\mathbf{X}} : \sigma \text{ is a substitution } \wedge\ M \models \sigma|_{\mathbf{X}}(q)\}$. The answer of a CQ $q(\mathbf{X})$ w.r.t. a program P is the set $\mathsf{ans}_P(q) = \{\sigma : \sigma \in \mathsf{ans}(q, M)\ \forall M \in \mathsf{mods}(P)\}$. Note that for a BCQ q either $\mathsf{ans}_P(q) = \{\sigma|_\emptyset\}$ or $\mathsf{ans}_P(q) = \emptyset$; in the first case we say that q is *cautiously true* w.r.t. P, denoted by $P \models q$.

Query answering (QA) is the problem of computing $\mathsf{ans}_P(q)$, where P is a *Datalog*$^\exists$ program and q a CQ. It is well-known that QA can be carried out by using a *universal model* of P [14], that is, a model U of P s.t. for each $M \in \mathsf{mods}(P)$ there is a homomorphism h satisfying $h(U) \subseteq M$. In this regard, given a universal model U of P, for each CQ $q(\mathbf{X})$ and for each substitution σ s.t. $\sigma(\mathbf{X}) \subseteq \Delta_C$, it has been shown that $\sigma \in \mathsf{ans}_P(q)$ iff $\sigma \in \mathsf{ans}(q, U)$ [14]. However, although each *Datalog*$^\exists$ program admits a universal model, deciding whether a substitution belongs to $\mathsf{ans}_P(q)$ is undecidable in the general case [14]. Finally, we mention the CHASE as a well-known procedure for constructing a universal model for a *Datalog*$^\exists$ program. (See Appendix A for details.)

3 Magic-Sets for *Datalog*$^\exists$

The original Magic-Sets technique was introduced for *Datalog* [4]. In order to bring it to the more general framework of *Datalog*$^\exists$, we have to face two main difficulties.

The first is that originally the technique was defined to handle $\forall$-variables only. How does the technique have to be extended to programs containing $\exists$-variables? The second difficulty, which is eventually due to the first one, concerns how to establish the correctness of an extension of Magic-Sets to *Datalog*$^{\exists}$. In fact, any *Datalog* program is characterized by a unique universal model of finite size. In this case, the correctness of Magic-Sets can be established by proving that the universal model of the rewritten program (modulo auxiliary predicates) is a subset of the universal model of the original program and contains all the answers for the input query. On the other hand, a *Datalog*$^{\exists}$ program may have in general many universal models of infinite size. Due to this difference, it is more difficult to prove the correctness of a Magic-Sets technique.

The difficulty associated with the presence of $\exists$-variables is circumvented by means of the following observation: A hypothetical top-down evaluation of a query over a *Datalog*$^{\exists}$ program would only consider the rules whose head atoms unify with the (sub)queries. Therefore, the Magic-Sets algorithm has to skip those rules whose head atoms have some $\exists$-variables in arguments that are bound from the (sub)queries. Concerning the second difficulty, we prove the correctness of the new Magic-Sets technique by considering all models of original and rewritten programs, showing that the same set of substitution answers is determined for the input query.

3.1 Magic-Sets Algorithm

Magic-Sets stem from SLD-resolution, which roughly acts as follows: Each rule r s.t. $\sigma(\mathsf{head}(r)) = \sigma'(q)$, where σ and σ' are two substitutions, is considered in a first step. Then, the atoms in $\sigma(\mathsf{body}(r))$ are taken as subqueries, and the procedure is iterated. During this process, if a (sub)query has some arguments bound to constant values, this information is used to limit the range of the corresponding variables in the processed rules, thus obtaining more targeted subqueries when processing rule bodies. Moreover, bodies are processed in a certain sequence, and processing a body atom may bind some of its arguments for subsequently considered body atoms. The specific propagation strategy adopted in a top-down evaluation scheme is called *sideways information passing strategy* (SIPS). Roughly, a SIPS is a strict partial order over the atoms of each rule which also specifies how the bindings originate and propagate [5].

In order to properly formalize our Magic-Sets algorithm, we first introduce adornments, a convenient way for representing binding information for intentional predicates.

Definition 1 (Adornments). *Let p be a predicate of arity k. An adornment for p is a string $\alpha = \alpha_1 \cdots \alpha_k$ defined over the alphabet $\{b, f\}$. The i-th argument of p is considered* bound *if $\alpha_i = b$, or* free *if $\alpha_i = f$ ($i \in [1..k]$).*

Binding information can be propagated in rule bodies according to a SIPS.

Definition 2 (SIPS). *Let r be a Datalog*$^{\exists}$ *rule and α an adornment for $\mathsf{pred}(\mathsf{head}(r))$. A SIPS for r w.r.t. α is a pair $(\prec_r^\alpha, f_r^\alpha)$, where: $\prec_r^\alpha$ is a strict partial order over $\mathsf{atoms}(r)$ s.t. $\mathbf{a} \in \mathsf{body}(r)$ implies $\mathsf{head}(r) \prec_r^\alpha \mathbf{a}$; f_r^α is a function assigning to each atom $\mathbf{a} \in \mathsf{atoms}(r)$ the subset of the variables in $\mathbf{a}$ that are made bound after processing $\mathbf{a}$; f_r^α must guarantee that $f_r^\alpha(\mathsf{head}(r))$ contains only and all the variables of $\mathsf{head}(r)$ corresponding to bound arguments according to α.*

Algorithm 1: MS(q,P)

Input : an atomic query $q = \mathtt{g}(u_1, \ldots, u_k)$ and a *Datalog*$^\exists$ program P
Output: an optimized *Datalog*$^\exists$ program

1 **begin**
2 $\quad \alpha := \alpha_1 \cdots \alpha_k$, where $\alpha_i = b$ if $u_i \in \Delta_C$, and $\alpha_i = f$ otherwise ($i \in [1..k]$);
3 $\quad S := \{\langle \mathtt{g}, \alpha \rangle\}$; $\quad D := \emptyset$; $\quad R^{mgc} := \{\mathsf{mgc}(q, \alpha) \leftarrow\}$; $\quad R^{mod} := \emptyset$;
4 $\quad$ **while** $S \neq \emptyset$ **do**
5 $\quad\quad \langle \mathtt{p}, \alpha \rangle :=$ any element in S; $\quad S := S \setminus \{\langle \mathtt{p}, \alpha \rangle\}$; $\quad D := D \cup \{\langle \mathtt{p}, \alpha \rangle\}$;
6 $\quad\quad$ **foreach** $r \in \mathsf{rules}(P)$ *s.t.* $\mathsf{head}(r) = p(t_1, \ldots, t_n)$ **and**
$\quad\quad\quad\quad t_i \in \Delta_\exists$ *implies* $\alpha_i = f$ *(*$i \in [1..k]$*)* **do**
$\quad\quad\quad$ // $\mathbf{a} := \mathtt{p}(t_1, \ldots, t_n)$
7 $\quad\quad\quad R^{mod} := R^{mod} \cup \{\mathsf{head}(r) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge \mathsf{body}(r)\}$;
8 $\quad\quad\quad$ **foreach** $q(s_1, \ldots, s_m) \in \mathsf{body}(r)$ *s.t.* $q \in \mathsf{idb}(P)$ **do**
$\quad\quad\quad\quad$ // $\mathbf{b} := \mathtt{q}(s_1, \ldots, s_m)$
9 $\quad\quad\quad\quad B := \{\mathbf{c} \in \mathsf{body}(r) \mid \mathbf{c} \prec_r^\alpha \mathbf{b}\}$;
10 $\quad\quad\quad\quad \beta := \beta_1 \cdots \beta_m$, where $\beta_i = b$ if $s_i \in \Delta_C \cup f_r^\alpha(B)$, and
$\quad\quad\quad\quad\quad\quad \beta_i = f$ otherwise ($i \in [1..k]$);
11 $\quad\quad\quad\quad R^{mgc} := R^{mgc} \cup \{\mathsf{mgc}(\mathbf{b}, \beta) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge B\}$;
12 $\quad\quad\quad\quad$ **if** $\langle q, \beta \rangle \notin D$ **then** $S := S \cup \{\langle \mathtt{q}, \beta \rangle\}$;
13 $\quad$ **return** $R^{mgc} \cup R^{mod} \cup \{\mathbf{a} \leftarrow \mid \mathbf{a} \in \mathsf{data}(P)\}$;

The auxiliary atoms introduced by the algorithm are obtained as described below.

Definition 3 (Magic Atoms). *Let* $\mathbf{a} = p(t_1, \ldots, t_k)$ *be an atom and* α *be an adornment for* $\mathtt{p}$*. We denote by* $\mathsf{mgc}(\mathbf{a}, \alpha)$ *the magic atom* $\mathtt{mgc\_p}^\alpha(\bar{t})$*, where:* $\bar{t}$ *contains all terms in* $t_1, \ldots, t_k$ *corresponding to bound arguments according to* α*; and* $\mathtt{mgc\_p}^\alpha$ *is a new predicate symbol (we assume that no standard predicate in* P *has the prefix "*$\mathtt{mgc\_}$*").*

We are now ready to describe the MS algorithm (Algorithm 1), associating each atomic query q over a *Datalog*$^\exists$ program P with a rewritten and optimized program MS(q,P). (More complex queries can be encoded by means of auxiliary rules.) The algorithm uses two sets, S and D, to store pairs of predicates and adornments to be propagated and already processed, respectively. Magic and modified rules are stored in the sets R^{mgc} and R^{mod}, respectively. The algorithm starts by producing the adornment associated with the query (line 1), which is paired with the query predicate and put into S (line 2). Moreover, the algorithm stores a ground fact named *query seed* into R^{mgc} (line 2). Sets D and R^{mod} are initially empty (line 2).

After that, the main loop of the algorithm is repeated until S is empty (lines 3–11). More specifically, a pair $\langle \mathtt{p}, \alpha \rangle$ is moved from S to D (line 4), and each rule r s.t. $\mathsf{head}(r) = \mathbf{a}$ and $\mathsf{pred}(\mathbf{a}) = \mathtt{p}$ is considered (lines 5–11). Considered rules are constrained to comply with the binding information from α, that is, no existential variables have to receive a binding during this process (line 5). The algorithm adds to R^{mod} a rule named *modified rule* which is obtained from r by adding $\mathsf{mgc}(\mathbf{a}, \alpha)$ to its body.

Binding information from α are then passed to body atoms according to a specific SIPS (lines 7–11). Specifically, for each body atom $\mathbf{b} = \mathtt{q}(\bar{s})$, the algorithm determines

the set B of predecessor atoms in the SIPS (line 8), from which an adornment string β for $\mathtt{q}$ is built (line 9). B and β are then used to generate a *magic rule* whose head atom is $\mathsf{mgc}(\mathbf{b}, \beta)$, and whose body comprises $\mathsf{mgc}(\mathbf{a}, \alpha)$ and atoms in B (line 10). Moreover, the pair $\langle \mathtt{q}, \beta \rangle$ is added to S unless it was already processed in a previous iteration (that is, unless $\langle \mathtt{q}, \beta \rangle \in D$; line 11). Finally, the algorithm terminates returning the program obtained by the union of R^{mgc}, R^{mod} and $\{\mathbf{a} \leftarrow \mid \mathbf{a} \in \mathsf{data}(P)\}$ (line 12).

Example 3. Resuming program *P-Jungle* of Example 1, we now give an example of the application of Algorithm 1. In particular, we consider SIPS s.t. atoms are totally ordered from left-to-right and binding information is propagated whenever possible. In this setting, Algorithm 1 run on query `afraid(antelope)` and *P-Jungle* yields the following rewritten program:

```
mgc_afraid^b(antelope)  ←
mgc_pursues^fb(X)  ←  mgc_afraid^b(X)
mgc_pursues^ff  ←  mgc_pursues^fb(Y)
mgc_pursues^bf(Y)  ←  mgc_hungry^b(Y)
mgc_hungry^b(Y)  ←  mgc_afraid^b(X), pursues(Y,X)

∃Z pursues(Z,X)  ←  mgc_pursues^fb(X), escapes(X)
∃Z pursues(Z,X)  ←  mgc_pursues^ff, escapes(X)
hungry(Y)  ←  mgc_hungry^b(Y), pursues(Y,X), fast(X)
pursues(X,Y)  ←  mgc_pursues^fb(Y), pursues(X,W), prey(Y)
pursues(X,Y)  ←  mgc_pursues^ff, pursues(X,W), prey(Y)
pursues(X,Y)  ←  mgc_pursues^bf(X), pursues(X,W), prey(Y)
afraid(X)  ←  mgc_afraid^b(X), pursues(Y,X), hungry(Y),
              strongerThan(Y,X)
```

A detailed description is reported in Appendix B.1. □

3.2 Query Equivalence Result

We start by establishing a relationship between the model of P and those of MS(q,P). The relationship is given by means of the next definition.

Definition 4 (Magic Variant). *Let* $I \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$*, and* $\{\mathsf{var}_i(I)\}_{i \in \mathbb{N}}$ *be the following sequence:* $\mathsf{var}_0(I) = I$*; for each* $i \geq 0$*,* $\mathsf{var}_{i+1}(I) = \mathsf{var}_i(I) \cup \{\mathbf{a} \in I \mid \exists \alpha$ *s.t.* $\mathsf{mgc}(\mathbf{a}, \alpha) \in \mathsf{var}_i(I)\} \cup \{\mathsf{mgc}(\mathbf{a}, \alpha) \mid \exists r, \sigma$ *s.t.* $r \in R^{mgc} \wedge \sigma(\mathsf{head}(r)) = \mathsf{mgc}(\mathbf{a}, \alpha) \wedge \sigma(\mathsf{body}(r)) \subseteq \mathsf{var}_i(I)\}$*. The fixpoint of this sequence is denoted by* $\mathsf{var}(I)$*.*

We point out that the magic variant of a set of atoms I comprises magic atoms and a subset of I. Intuitively, these atoms are enough to achieve a model of MS(q,P) if I is a model of P. This intuition is formalized below and proven in Appendix C.

Lemma 1. *If* $M \models P$*, then* $\mathsf{var}(M) \models \mathrm{MS}(q, P)$.

The soundness of Algorithm 1 w.r.t. QA can be now established.

Theorem 1 (Soundness). *If* $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$*, then* $\sigma \in \mathsf{ans}_P(q)$.

Proof. Assume $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$. Let $M \models P$. By Lemma 1, $\mathsf{var}(M) \models \mathrm{MS}(q, P)$. Since $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$ by assumption, $\sigma(q) \in \mathsf{var}(M)$. Thus, $\sigma(q) \in M$ because $\mathsf{var}(M)$ comprises magic atoms and a subset of M by construction. □

To prove the completeness of Algorithm 1 w.r.t. QA we identify a set of atoms that are not entailed by the rewritten program but not due to the presence of magic atoms.

Definition 5 (Killed Atoms). *Let* $M \models \mathrm{MS}(q, P)$. *The set* $\mathsf{killed}(M)$ *is defined as follows:* $\{\mathbf{a} \in \mathsf{base}(\Delta) \setminus M \mid$ *either* $\mathsf{pred}(\mathbf{a}) \in \mathsf{edb}(P)$, *or* $\exists \alpha$ *s.t.* $\mathsf{mgc}(\mathbf{a}, \alpha) \in M\}$.

Since the falsity of killed atoms is not due to the Magic-Sets rewriting, one expects that their falsity can also be assumed in the original program. This intuition is formalized below and proven in Appendix C.

Lemma 2. *If* $M \models MS(q, P)$, $M' \models P$ *and* $M' \supseteq M$, *then* $M' \setminus \mathsf{killed}(M) \models P$.

We can finally prove the completeness of Algorithm 1 w.r.t. QA, which then establishes the correctness of Magic-Sets for queries over *Datalog*$^\exists$ programs.

Theorem 2 (Completeness). *If* $\sigma \in \mathsf{ans}_P(q)$, *then* $\sigma \in \mathsf{ans}(q, MS(q, P))$.

Proof. Assume $\sigma \in \mathsf{ans}_P(q)$. Let $M \models \mathrm{MS}(q, P)$. Let $M' \models P$ and be s.t. $M' \supseteq M$. By Lemma 2, $M' \setminus \mathsf{killed}(M) \models P$. Since $\sigma \in \mathsf{ans}_P(q)$ by assumption, $\sigma(q) \in M' \setminus \mathsf{killed}(M)$. Note that all instances of the query which are not in M are contained in $\mathsf{killed}(M)$ because the query seed belongs to M. Thus, $\sigma(q) \in M$ holds. □

4 Magic-Sets for *Shy* Programs

Among various *Datalog*$^\exists$ subclasses making QA computable, we are going to focus on *Shy* [17], an attractive *Datalog*$^\exists$ fragment which guarantees both easy recognizability and efficient answering even to CQs. After recalling basic definitions and computational results about *Shy*, we show how to guarantee shyness in the rewritten of a *Shy* program.

4.1 *Shy* Programs

Intuitively, the key idea behind *Shy* programs relays on the following *shyness* property: *During a chase execution on a Shy program P, nulls (propagated body-to-head in ground rules) do not meet each other to join.*

We now introduce the notion of *null-set* of a position in an atom. More precisely, $\varphi^r_{\mathtt{X}}$ denotes the "representative" null that can be introduced by the $\exists$-variable $\mathtt{X}$ occurring in rule r. (If $\langle r, \mathtt{X}\rangle \neq \langle r', \mathtt{X}'\rangle$, then $\varphi^r_{\mathtt{X}} \neq \varphi^{r'}_{\mathtt{X}'}$.) Let P be a *Datalog*$^\exists$ program, $\mathbf{a}$ be an atom, and $\mathtt{X}$ a variable occurring in $\mathbf{a}$ at position i. The *null-set* of position i in $\mathbf{a}$ w.r.t. P, denoted by $\mathsf{nullset}(i, \mathbf{a})$, is inductively defined as follows: In case $\mathbf{a}$ is the head of some rule $r \in P$, $\mathsf{nullset}(i, \mathbf{a})$ is the singleton $\{\varphi^r_{\mathtt{X}}\}$ if $\mathtt{X} \in \Delta_\exists$; otherwise ($\mathtt{X} \in \Delta_\forall$), $\mathsf{nullset}(i, \mathbf{a})$ is the intersection of every $\mathsf{nullset}(j, \mathbf{b})$ s.t. $\mathbf{b} \in \mathsf{body}(r)$ and $\mathtt{X}$ occurs at position j in $\mathbf{b}$. In case $\mathbf{a}$ is not a head atom, $\mathsf{nullset}(i, \mathbf{a})$ is the union of $\mathsf{nullset}(i, \mathsf{head}(r))$ for each $r \in P$ s.t. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathbf{a})$.

A representative null φ *invades* a variable $\mathtt{X}$ that occurs at position i in an atom $\mathbf{a}$ if φ is contained in $\mathsf{nullset}(i, \mathbf{a})$. A variable $\mathtt{X}$ occurring in a conjunction $\mathbf{conj}$ is *attacked* in $\mathbf{conj}$ by a null φ if each occurrence of $\mathtt{X}$ in $\mathbf{conj}$ is invaded by φ. A variable $\mathtt{X}$ is *protected* in $\mathbf{conj}$ if it is attacked by no null.

Definition 6. *Let Shy be the class of all Datalog$^\exists$ programs containing only shy rules, where a rule r is called* shy *w.r.t. a program P if the following conditions are satisfied:*

- *If a variable $\mathtt{X}$ occurs in more than one body atom, then $\mathtt{X}$ is protected in* $\mathsf{body}(r)$.
- *If two distinct $\forall$-variables are not protected in* $\mathsf{body}(r)$ *but occur both in* $\mathsf{head}(r)$ *and in two different body atoms, then they are not attacked by the same null.* □

According to Definition 6, program *P-Jungle* of Example 1 is *Shy*. Let $\mathbf{a}_1, \ldots, \mathbf{a}_{12}$ be the atoms of rules r_1–r_4 in left-to-right/top-to-bottom order, and $\mathsf{nullset}(1, \mathbf{a}_1)$ be $\{\varphi^{r_1}_{\mathtt{Z}}\}$. To show the shyness of *P-Jungle*, we first propagate $\varphi^{r_1}_{\mathtt{Z}}$ (head-to-body) to $\mathsf{nullset}(1, \mathbf{a}_4)$, $\mathsf{nullset}(1, \mathbf{a}_7)$, and $\mathsf{nullset}(1, \mathbf{a}_{10})$. Next, this singleton is propagated (body-to-head) from $\mathbf{a}_4$, $\mathbf{a}_7$ and $\mathbf{a}_3$ to $\mathsf{nullset}(1, \mathbf{a}_3)$, $\mathsf{nullset}(1, \mathbf{a}_6)$ and $\mathsf{nullset}(1, \mathbf{a}_{11})$, respectively. Finally, we observe that rules r_1–r_3 are trivially shy, and that r_4 also is because variable $\mathtt{Y}$ is not invaded in $\mathbf{a}_{12}$ even if $\varphi^{r_1}_{\mathtt{Z}}$ invades $\mathtt{Y}$ both in $\mathbf{a}_{10}$ and $\mathbf{a}_{11}$.

Shy enjoys the following notable computational properties:

- Checking whether a program is *Shy* is doable in polynomial-time.
- Query answering over *Shy* is polynomial-time computable in data complexity.[1]

4.2 Preserving Shyness in the Magic-Sets Rewriting

In Section 3, the correctness of MS has been established for *Datalog$^\exists$* programs in general. Our goal now is to preserve the desirable shyness property in the rewritten of a *Shy* program. In fact, shyness is not preserved by MS per sé. Resuming Example 3, MS run on query `afraid(antelope)` and program *P-Jungle* may produce from r_4 a rule $\mathtt{mgc\_hungry}^b\mathtt{(Y)} \leftarrow \mathtt{mgc\_afraid}^b\mathtt{(X)}, \mathtt{pursues(Y,X)}$, which assumes $\mathtt{hungry}(\varphi)$ relevant whenever some $\mathtt{pursues}(\varphi,\mathtt{X})$ is derived, for any $\varphi \in \Delta_N$. However, shyness guarantees that any extension of this substitution for r_4 is actually annihilated by $\mathtt{strongerThan(Y,X)}$, which thus enforces protection on $\mathtt{Y}$. Unfortunately, SIPS cannot represent this kind of information in general, and thus MS may yield a non-shy program. Actually, the rewritten program in Example 3 is not shy because it contains rule $\mathtt{hungry(Y)} \leftarrow \mathtt{mgc\_hungry}^b\mathtt{(Y), pursues(Y,X), fast(X)}$.

The problem described above originates by the inability to represent in SIPS that no join on nulls is required to evaluate *Shy* programs. We thus explicitly encode this information in rules by means of the following transformation strategy: Let r be a rule of the form (1) in a program P, and #dom be an auxiliary predicate not occurring in P. We denote by $r^\star$ the rule obtained from r by adding a body atom $\mathsf{\#dom}(\mathtt{X})$ for each protected variable $\mathtt{X}$ in $\mathsf{body}(r)$. Moreover, we denote by $P^\star$ the program comprising each rule $r^\star$ s.t. $r \in P$, and each fact $\mathsf{\#dom}(\mathtt{c}) \leftarrow$ s.t. $\mathtt{c} \in \mathsf{dom}(P)$. (Note that the introduction of these facts is not really required because #dom can be treated as a built-in predicate, thus introducing no computational overhead.)

[1] In this setting, $\mathsf{data}(P)$ are the only input while q and $\mathsf{rules}(P)$ are considered fixed.

Proposition 1. *If P is Shy, then $P^\star$ is shy as well and* $\mathsf{mods}(P) = \mathsf{mods}(P^\star)$.

Now, for an atomic query q over a *Shy* program P, in order to preserve shyness, we apply Algorithm 1 to $P^\star$ and force SIPS to comply with the following restriction: Let $r \in P^\star$ and α be an adornment. For each $\mathbf{a}, \mathbf{b} \in \mathsf{body}(r)$ s.t. $\mathbf{a} \prec_r^\alpha \mathbf{b}$, and for each variable $\mathtt{X}$ occurring in both $\mathbf{a}$ and $\mathbf{b}$, SIPS $(\prec_r^\alpha, f_r^\alpha)$ is s.t. $\mathbf{a} \prec_r^\alpha \#\mathsf{dom}(\mathtt{X}) \prec_r^\alpha \mathbf{b}$. (An example is reported in Appendix B.2.)

Theorem 3. *Let q be an atomic query. If P is Shy, then* $\mathrm{MS}(q, P^\star)$ *is Shy.*

Proof. All arguments of magic predicates have empty null-sets. Indeed, each variable in the head of a magic rule r either occurs in the unique magic atom of $\mathsf{body}(r)$, or appears as the argument of a #dom atom. Consequently, all rules in R^{mgc} are shy. Moreover, each rule in R^{mod} is obtained from a rule of $P^\star$ by adding a magic atom to its body. No attack can be introduced in this way because arguments of magic atoms have empty null-sets. Thus, since the original rule is shy, the modified rule is also shy. □

In order to handle CQs of the form $\exists\mathbf{Y}\ \mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$, we first introduce a rule r_q of the form $q(\mathbf{X}) \leftarrow \mathbf{conj}$. We then compute $P' = \mathrm{MS}(q(\mathbf{X}), (P \cup \{r_q\})^\star)$ further restricting the SIPS for r_q to not propagate bindings via attacked variables, that is, to be s.t. $\mathtt{Z} \in f_{r_q}^\alpha(\mathbf{conj})$ implies that $\mathtt{Z}$ is protected in $\mathbf{conj}$ (where α is the adornment for q). After that, we remove from P' the rule associated with the query, thus obtaining a *Shy* program P''. Finally, we evaluate the original query $\exists\mathbf{Y}\ \mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$ on program P''.

5 Experimental Results and Discussion

We incorporated Magic-Sets in DLV$^\exists$ [17], a system supporting QA over *Shy*. Empirical evidence of the effectiveness of the implemented system is provided by means of an experiment on the well-known benchmark suite LUBM (see `http://swat.cse.`

Table 1. Query evaluation time (seconds) of DLV$^\exists$ and improvements (IMP) of Magic-Sets

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}
lubm-10														
DLV$^\exists$	3.40	3.21	0.93	1.37	5.73	2.29	5.12	3.97	4.83	3.53	0.33	0.86	5.26	1.88
DLV$^\exists$+MS	1.83	1.95	0.63	0.39	1.20	0.48	2.95	1.08	3.45	2.54	0.08	0.85	0.76	1.88
IMP	46%	39%	32%	72%	79%	79%	42%	73%	29%	28%	76%	1%	86%	0%
lubm-30														
DLV$^\exists$	11.90	11.49	2.09	4.40	18.42	8.07	18.02	13.53	15.87	12.42	1.13	2.93	18.95	6.41
DLV$^\exists$+MS	6.20	6.28	1.44	1.28	3.91	1.67	9.85	3.11	11.82	7.95	0.24	2.85	2.42	6.23
IMP	48%	45%	31%	71%	79%	79%	45%	77%	26%	36%	79%	3%	87%	3%
lubm-50														
DLV$^\exists$	21.15	19.05	3.72	7.71	31.80	14.46	31.47	23.63	28.96	21.80	1.99	5.48	32.50	11.52
DLV$^\exists$+MS	10.86	11.39	2.42	2.23	6.36	3.03	16.32	5.23	20.30	14.10	0.39	5.32	4.13	11.49
IMP	49%	40%	35%	71%	80%	79%	48%	78%	30%	35%	80%	3%	87%	0%

lehigh.edu/projects/lubm/). It refers to a university domain and includes a synthetic data generator, which we used to generate three increasing data sets, namely lubm-10, lubm-30 and lubm-50. LUBM incorporates a set of 14 queries referred to as q_1–q_{14}, where q_2, q_6, q_9 and q_{14} contain no constants. Tests have been carried out on an Intel Xeon X3430, 2.4 GHz, with 4 Gb Ram, running Linux Operating System. For each query, we allowed 7200 seconds (two hours) or running time and 2 Gb of memory.

We first evaluated the impact of Magic-Sets on DLV$^\exists$. Specifically, we measured the time taken by DLV$^\exists$ to answer the 14 LUBM queries with and without the application of Magic-Sets. Results are reported in Table 1, where times do not include data parsing and loading as they are not affected by Magic-Sets. On the considered queries, Magic-Sets reduce running time of 50% in average, with a peak of 87% on q_{13}. If only queries with no constants are considered, the average improvement of Magic-Sets is 37%, while the average improvement rises up to 55% for queries with at least one constant. We also point out that the average improvement provided by Magic-Sets is always greater than 25% if q_{12} and q_{14} are not considered. Regarding these two queries, Magic-Sets do not provide any improvement because the whole data sets are relevant for their evaluation.

Next, we compared DLV$^\exists$ enhanced by Magic-Sets with three state-of-the-art reasoners, namely Pellet [18], OWLIM-SE [6] and OWLIM-Lite [6]. Results are reported in Table 2, where times include the *total time* required for query answering. We measured the total time, including data parsing and loading, because ontology reasoning is usually performed in contexts where data and knowledge rapidly vary, even within hours. DLV$^\exists$ significantly outperforms all other systems in all tested queries and data sets. Comparing the other systems, OWLIM-Lite is in general faster than Pellet and OWLIM-SE. Pellet is faster than OWLIM-SE on lubm-10, but it answered no tested queries in the allotted time on lubm-30 and lubm-50.

Table 2. Systems comparison: running time (sec.), solved queries ($\#_s$) and average time (G.Avg)

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}	$\#_s$	G.Avg
lubm-10																
DLV$^\exists$	5	4	2	4	6	1	6	4	8	5	<1	1	6	2	14	2.87
Pellet	82	84	84	82	80	88	81	89	95	82	82	89	82	84	14	84.48
OWLIM-Lite	33	–	33	33	33	33	4909	70	–	33	33	33	33	33	12	53.31
OWLIM-SE	105	105	105	105	105	105	105	106	106	105	105	105	105	105	14	105.14
lubm-30																
DLV$^\exists$	16	13	7	14	21	3	21	12	25	18	<1	5	23	8	14	9.70
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	107	–	107	106	107	106	–	528	–	107	106	106	107	106	11	123.18
OWLIM-SE	323	328	323	323	323	323	323	323	326	323	323	323	323	323	14	323.57
lubm-50																
DLV$^\exists$	27	23	12	23	35	6	34	22	42	31	<1	9	33	14	14	16.67
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	188	–	190	187	189	188	–	1272	–	189	187	187	189	187	11	223.79
OWLIM-SE	536	547	536	536	536	537	536	536	542	536	536	536	536	537	14	537.35

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1995.
2. Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. *Artificial Intelligence.* Elsevier. To appear.
3. Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming.* Cambridge University Press, 10(4–6):497–512, July 2010.
4. François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Int. Symposium on Principles of Database Systems*, pages 1–16, 1986.
5. Catriel Beeri and Raghu Ramakrishnan. On the power of magic. 10(1–4):255–259, 1991.
6. Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semant. Web*, 2:33–42, 2011.
7. Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In *Proc. of the 11th KR Int. Conf.*, pages 70–80, 2008. Revised version: `http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf`.
8. Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proc. of the 28th PODS Symp.*, pages 77–86, 2009.
9. Andrea Calì, Georg Gottlob, and Andreas Pieris. Advanced Processing for Ontological Queries. *PVLDB*, 3(1):554–565, 2010.
10. Andrea Calì, Georg Gottlob, and Andreas Pieris. New Expressive Languages for Ontological Query Answering. In *Proc. of the 25th AAAI Conf. on AI*, pages 1541–1546, 2011.
11. Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)*, volume 5753, pages 71–86. Springer Verlag, September 2009.
12. Diego Calvanese, Giuseppe Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *J. Autom. Reason.*, 39:385–429, 2007.
13. Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences*, 73(4):584–609, 2007.
14. Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *TCS*, 336(1):89–124, 2005.
15. Sergio Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. 15(2):368–385, March/April 2003.
16. Ilianna Kollia, Birte Glimm, and Ian Horrocks. SPARQL Query Answering over OWL Ontologies. In *Proc. of the 24th DL Int. Workshop*, volume 6643 of *LNCS*, pages 382–396. Springer, 2011.
17. Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Efficiently Computable Datalog$^{\exists}$ Programs. In *Proc. of the 13th KR Int. Conf.*, page Forthcoming, 2012.
18. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5(2):51–53, 2007.

A The chase for *Datalog*$^{\exists}$

Given a rule r of the form (1) and a set C of atoms, a *firing* substitution σ for r w.r.t. C is a substitution σ on $\mathbf{X}$ s.t. $\sigma(\mathsf{body}(r)) \subseteq C$. Next, given a firing substitution σ for r w.r.t. C, the *fire* of r on C due to σ infers $\hat{\sigma}(\mathsf{head}(r))$, where $\hat{\sigma}$ is an extension of σ on $\mathbf{Y} \cup \mathbf{X}$ associating each $\exists$-variable in $\mathbf{Y}$ to a different null.

Procedure 1 illustrates the overall *restricted chase procedure*. Importantly, we assume that different fires (on the same or different rules) always introduce different "fresh" nulls. The procedure consists of an exhaustive series of fires in a breadth-first (level-saturating) fashion, which leads as result to a (possibly infinite) $\mathsf{chase}(P)$.

Procedure 1: CHASE(P)

Input : *Datalog*$^{\exists}$ program P
Output: Universal Model $\mathsf{chase}(P)$

1 $C := \mathsf{data}(P)$;
2 *NewAtoms* $:= \emptyset$;
3 **foreach** $r \in P$ **do**
4 **foreach** *firing substitution* σ *for* r *w.r.t.* C **do**
5 **if** $(C \cup \text{NewAtoms}) \not\models \sigma(\mathsf{head}(r))$ **then**
6 $\mathsf{add}(\hat{\sigma}(\mathsf{head}(r)), \textit{NewAtoms})$;
7 **if** NewAtoms $\neq \emptyset$ **then**
8 $C := C \cup \textit{NewAtoms}$;
9 **go to** step 2;
10 **return** C;

B Magic-Sets Examples

In this section we first provide a detailed description of the application of Algorithm 1 to a query `afraid(antelope)` for program *P-Jungle* reported in Example 1. We assume that SIPS are s.t. atoms are totally ordered from left to right, and binding information are propagated whenever possible. Finally, we show how shyness can be preserved in the rewritten program.

B.1 The Magic-Sets Rewritten of a *Datalog*$^{\exists}$ Program

Algorithm 1 starts by producing the adornment b associated with the query. Sets S and R^{mgc} initially contain $\langle \texttt{afraid}, \alpha \rangle$ and $\texttt{mgc\_afraid}^b\texttt{(antelope)} \leftarrow$, respectively, while D and R^{mod} are empty. MS then enters its main loop. Pair $\langle \texttt{afraid}, \alpha \rangle$ is moved from S to D, and rule r_4 is considered. The following modified and magic rules are produced:

```
afraid(X)  ←  mgc_afraid^b(X), pursues(Y,X), hungry(Y),
               strongerThan(Y,X)
mgc_pursues^fb(X)  ←  mgc_afraid^b(X)
mgc_hungry^b(Y)  ←  mgc_afraid^b(X), pursues(Y,X)
```

and at the same time $\langle \texttt{pursues}, \mathit{fb} \rangle$ and $\langle \texttt{hungry}, b \rangle$ are added to S. After that, an element of S, say $\langle \texttt{pursues}, \mathit{fb} \rangle$, is moved to D, and rules r_1 and r_3 are considered. The following rules are produced:

```
∃Z pursues(Z,X)  ←  mgc_pursues^fb(X), escapes(X)
pursues(X,Y)  ←  mgc_pursues^fb(Y), pursues(X,W), prey(Y)
mgc_pursues^ff  ←  mgc_pursues^fb(Y)
```

and $\langle \texttt{pursues}, \mathit{ff} \rangle$ is added to S. Then, an element of S, say $\langle \texttt{hungry}, b \rangle$, is moved to D, and rule r_2 is considered. The following rules are produced:

```
hungry(Y)  ←  mgc_hungry^b(Y), pursues(Y,X), fast(X)
mgc_pursues^bf(Y)  ←  mgc_hungry^b(Y)
```

and $\langle \texttt{pursues}, \mathit{bf} \rangle$ is added to S. The algorithm then move an element of S, say $\langle \texttt{pursues}, \mathit{ff} \rangle$, to D, and considers rules r_1 and r_3, from which it proceduces the following rules:

```
∃Z pursues(Z,X)  ←  mgc_pursues^ff, escapes(X)
pursues(X,Y)  ←  mgc_pursues^ff, pursues(X,W), prey(Y)
mgc_pursues^ff  ←  mgc_pursues^ff.
```

In this case, no new pair is introduced in S. The last element of S, $\langle \texttt{pursues}, \mathit{bf} \rangle$, is then moved to D, and rule r_3 is considered. The following modified rules are produced:

```
pursues(X,Y)  ←  mgc_pursues^bf(X), pursues(X,W), prey(Y)
mgc_pursues^bf(X)  ←  mgc_pursues^bf(X)
```

No pair is added to S and thus the algorithm terminates. Note that for $\langle \texttt{pursues}, \mathit{bf} \rangle$ the algorithm does not consider rule r_1 as the first argument of `pursues` in $\mathsf{head}(r_1)$ is existentially quantified. To sum up, the complete rewritten program is the following:

```
mgc_afraid^b(antelope)  ←
mgc_pursues^fb(X)  ←  mgc_afraid^b(X)
mgc_pursues^ff  ←  mgc_pursues^fb(Y)
mgc_pursues^bf(Y)  ←  mgc_hungry^b(Y)
mgc_hungry^b(Y)  ←  mgc_afraid^b(X), pursues(Y,X)

∃Z pursues(Z,X)  ←  mgc_pursues^fb(X), escapes(X)
∃Z pursues(Z,X)  ←  mgc_pursues^ff, escapes(X)
hungry(Y)  ←  mgc_hungry^b(Y), pursues(Y,X), fast(X)
pursues(X,Y)  ←  mgc_pursues^fb(Y), pursues(X,W), prey(Y)
pursues(X,Y)  ←  mgc_pursues^ff, pursues(X,W), prey(Y)
pursues(X,Y)  ←  mgc_pursues^bf(X), pursues(X,W), prey(Y)
afraid(X)  ←  mgc_afraid^b(X), pursues(Y,X), hungry(Y),
              strongerThan(Y,X)
```

(Always satisfied rules have been omitted).

B.2 Preserving Shyness in the Magic-Sets Rewritten

Since program *P-Jungle* is *Shy*, we can first introduce program *P-Jungle**:

```
r1* : ∃Z pursues(Z,X)  ←  escapes(X), #dom(X)
r2* : hungry(Y)  ←  pursues(Y,X), #dom(X), fast(X)
r3* : pursues(X,Y)  ←  pursues(X,W), #dom(W), prey(Y), #dom(Y)
r4* : afraid(X)  ←  pursues(Y,X), #dom(Y), hungry(Y), #dom(X),
                    strongerThan(Y,X)
```

and then apply Algorithm 1, which thus yields the following *Shy* program:

```
mgc_afraid^b(antelope)  ←
mgc_pursues^fb(X)  ←  mgc_afraid^b(X)
mgc_pursues^ff  ←  mgc_pursues^fb(Y)
mgc_pursues^bf(Y)  ←  mgc_hungry^b(Y)
mgc_hungry^b(Y)  ←  mgc_afraid^b(X), pursues(Y,X), #dom(Y)

∃Z pursues(Z,X)  ←  mgc_pursues^fb(X), escapes(X), #dom(X)
∃Z pursues(Z,X)  ←  mgc_pursues^ff, escapes(X), #dom(X)
hungry(Y)  ←  mgc_hungry^b(Y), pursues(Y,X), #dom(X), fast(X)
pursues(X,Y)  ←  mgc_pursues^fb(Y), pursues(X,W), #dom(W),
                 prey(Y), #dom(Y)
pursues(X,Y)  ←  mgc_pursues^ff, pursues(X,W), #dom(W),
                 prey(Y), #dom(Y)
pursues(X,Y)  ←  mgc_pursues^bf(X), pursues(X,W), #dom(W),
                 prey(Y), #dom(Y)
afraid(X)  ←  mgc_afraid^b(X), pursues(Y,X), #dom(Y),
              hungry(Y), #dom(X), strongerThan(Y,X).
```

C Proofs

Proofs that have been omitted in Section 3.2 are reported below.

Proof (Proof of Lemma 1). Assume $M \models P$. Let $r \in \mathrm{MS}(q, P)$ be of the form (1) and σ be a substitution s.t. $\sigma(\mathsf{body}(r)) \subseteq \mathsf{var}(M)$. We have to show $\mathsf{var}(M) \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$. If $r \in R^{mgc}$, then $\sigma(\mathsf{head}(r)) \in \mathsf{var}(M)$ by construction of $\mathsf{var}(M)$, and so $\mathsf{var}(M) \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$ holds. Otherwise, $r \in R^{mod}$. Consider the rule $r' \in P$ from which r has been obtained (line 7 of Algorithm 1). Note that $r = \mathsf{head}(r') \leftarrow \mathsf{mgc}(\mathsf{head}(r), \alpha) \wedge \mathsf{body}(r')$, for some adornment α. Therefore, $\sigma(\mathsf{body}(r')) \subseteq M$, which combined with $M \models P$ gives $M \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$, i.e., there is a substitution σ' s.t. $\sigma' \circ \sigma|_{\mathbf{X}}(\mathsf{head}(r)) \in M$. Since $\mathsf{mgc}(\mathsf{head}(r), \alpha) \in \mathsf{var}(M)$ by assumption, $\sigma' \circ \sigma|_{\mathbf{X}}(\mathsf{head}(r)) \in \mathsf{var}(M)$ by Definition 4, and so $\mathsf{var}(M) \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$ holds also in this case.

Proof (Proof of Lemma 2). In order to show that $M' \setminus \mathsf{killed}(M) \models P$, we have to consider each rule $r \in P$ of the form (1) and each substitution σ s.t. $\sigma(\mathsf{body}(r)) \subseteq M' \setminus \mathsf{killed}(M)$. Our aim is thus to show $M' \setminus \mathsf{killed}(M) \models \mathbf{a}$, where $\mathbf{a} = \sigma|_{\mathbf{X}}(\mathsf{head}(r))$. Since $\sigma(\mathsf{body}(r)) \subseteq M' \setminus \mathsf{killed}(M) \subseteq M'$ and $M' \models P$, we have $M' \models \mathbf{a}$. Assume

by contradiction that for each σ' s.t. $\sigma'(\mathbf{a}) \in M'$, we have $\sigma'(\mathbf{a}) \in \mathsf{killed}(M)$. Thus, $\sigma'(\mathbf{a}) \notin M$ and $\mathsf{mgc}(\sigma'(\mathbf{a}), \alpha) \in M$ (for some adornment α). In this case we consider the rule $r' \in MS(q, P)$ s.t. $r' = \mathsf{head}(r) \leftarrow \mathsf{mgc}(\mathsf{head}(r), \alpha) \wedge \mathsf{body}(r)$. Since $M \models MS(q, P)$, $\mathsf{mgc}(\mathbf{a}, \alpha) \in M$ and $\sigma'(\mathbf{a}) \notin M$ (for each σ'), we have $\sigma(\mathsf{body}(r)) \not\subseteq M$. Let $\mathbf{b} \in \mathsf{body}(r)$ be s.t. $\sigma(\mathbf{b}) \notin M$ and $\sigma(B) \subseteq M$, where $B = \{\mathbf{c} \in \mathsf{body}(r) \mid \mathbf{c} \prec_r^{\alpha} \mathbf{b}\}$. Since MS($q$,$P$) contains a magic rule $\mathsf{mgc}(\mathbf{b}, \beta) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge B$ (line 11 of Algorithm 1), we can conclude that $\sigma(\mathbf{b})$ belongs to $\mathsf{killed}(M)$, which contradicts the original assumption $\sigma(\mathsf{body}(r)) \subseteq M' \setminus \mathsf{killed}(M)$. We can thus conclude that there is σ' s.t. $\sigma'(\mathbf{a}) \in M'$ and $\sigma'(\mathbf{a}) \notin \mathsf{killed}(M)$, from which we immediately obtain $M' \setminus \mathsf{killed}(M) \models \mathbf{a}$. □