A

Fast Query Answering over Existential Rules

NICOLA LEONE, MARCO MANNA, GIORGIO TERRACINA and PIERFRANCESCO VELTRI, University of Calabria, Italy

Enhancing Datalog with existential quantification gives rise to Datalog$^\exists$, a powerful knowledge representation language widely used in ontology-based query answering. In this setting, a conjunctive query is evaluated over a Datalog$^\exists$ program consisting of extensional data paired with so-called "existential" rules. Due to their high expressiveness, such rules make the evaluation of queries undecidable, even when the latter are atomic. Decidable generalizations of Datalog existential rules have been proposed in the literature (such as weakly-acyclic and weakly-guarded); but they pay the price of higher computational complexity, hindering the implementation of effective systems. Conversely, the results in this paper demonstrate that it is definitely possible to enable fast yet powerful query answering over existential rules, ensuring decidability without any complexity overhead.

On the theoretical side, we define the class of parsimonious programs which guarantees decidability of atomic queries. We then strengthen this class to strongly parsimonious programs ensuring decidability also for conjunctive queries. Since parsimony is an undecidable property, we single out Shy, an easily recognizable class of strongly parsimonious programs that generalizes Datalog while preserving its complexity even under conjunctive query evaluation. Shy generalizes also the class of linear existential programs, while it is uncomparable to the other main classes ensuring decidability.

On the practical side, we exploit our results to implement DLV$^\exists$, an effective system for query answering over parsimonious existential rules. To asses its efficiency, we carry out an experimental analysis, comparing DLV$^\exists$ against a number of state-of-the-art systems for ontology-based query answering. The results confirm the effectiveness of DLV$^\exists$, which outperforms all other systems.

1. INTRODUCTION

The computational problem of answering a Boolean query q against a logical theory consisting of an extensional database D paired with an ontology Σ is attracting the increasing attention of scientists in various fields of Computer Science, ranging from artificial intelligence [Baget et al. 2011; Calvanese et al. 2013; Gottlob et al. 2014] to database theory [Bienvenu et al. 2014; Gottlob et al. 2014; Bourhis et al. 2016] and logic [Prez-Urbina et al. 2010; Bárány et al. 2014; Gottlob et al. 2013]. This problem, best known as *ontology-based query answering* (OBQA) [Calì et al. 2009b], is usually stated as $D \cup \Sigma \models q$, and it is equivalent to checking whether q is satisfied by all models of $D \cup \Sigma$ according with the standard approach of first-order logics, yielding an open world semantics [Abiteboul et al. 1995].

1.1. Motivation

A key issue in OBQA is the design of the language that is provided for specifying the ontology Σ. This language should balance expressiveness and complexity. Ideally, the language should be: (1) intuitive and easy-to-understand; (2) decidable (i.e., OBQA should be decidable in this language); (3) efficiently computable; (4) powerful enough in terms of expressiveness; and (5) suitable for an efficient implementation.

In this regard, Datalog$^\pm$, the family of Datalog-based languages proposed by [Calì et al. 2009a] for tractable query answering over ontologies, is arousing increasing interest [Mugnier 2011]. This family has been introduced with the aim of "closing the gap between the Semantic Web and databases" [Calì et al. 2012] to provide the *Web of Data* with scalable formalisms that can benefit from exiting database technologies. In fact, Datalog$^\pm$ encompasses and generalizes well known ontology specification languages, such as two well-known families of Description Logics called $\mathcal{EL}$ [Brandt 2004; Baader et al. 2005; Rosati 2007] and *DL-Lite* [Calvanese et al. 2007; Artale et al. 2009], which collect the basic tractable languages for OBQA in the context of the Semantic Web and

databases. From a syntactic point of view, Datalog$^{\pm}$ is mainly based on Datalog$^{\exists}$, the natural extension of Datalog [Abiteboul et al. 1995] that allows existentially quantified variables in rule heads. For example, the following Datalog$^{\exists}$ (or "existential") rules

$$\begin{aligned} person(john) &\leftarrow \\ \exists Y\ hasFather(X,Y) &\leftarrow person(X) \\ person(Y) &\leftarrow hasFather(X,Y) \end{aligned}$$

state that John is a person, and that if X is a person, then X must have some father Y, who has to be a person as well. In general, Datalog$^{\pm}$ intends to collect all expressive extensions of Datalog which are based on *existential quantification*, *equality-generating dependencies*, *negative constraint*, *negation*, and *disjunction*. In particular, the "plus" symbol refers to any possible combination of these extensions, while the "minus" one imposes at least decidability, since Datalog$^{\exists}$ alone is already undecidable [Calì et al. 2013a].

The main decidable Datalog$^{\pm}$ fragments rely on the following four syntactic properties: *weak-acyclicity* [Fagin et al. 2005a], *guardedness* [Calì et al. 2013a], *linearity* [Calì et al. 2012], and *stickiness* [Calì et al. 2010]. And these properties have been exploited to define the basic classes of existential rules called weakly-acyclic, (weakly-)guarded, linear, and sticky(-join), respectively. Several variants and combinations of these classes have been defined and studied too [Baget et al. 2010a; Krötzsch and Rudolph 2011; Calì et al. 2012; Civili and Rosati 2012; Gottlob et al. 2013]. But there are also decidable "abstract" classes of Datalog$^{\exists}$ programs, called fes, bts and fus, depending on semantic properties [Mugnier 2011].

The proposed languages enjoy the simplicity of Datalog and are endowed with a number of desirable properties of ontology specification languages. Nevertheless, no proposed class satisfies conditions (1)–(5) stated above (see Section 8). In particular, some classes do not generalize Datalog; while those languages fully generalizing Datalog pay the price of higher computational complexity hindering the implementation of effective systems.

1.2. Summary of Contributions

In this work, we single out a new class of Datalog$^{\exists}$ programs, called shy, which enjoys a new semantic property called *parsimony* and results in a powerful and yet decidable ontology specification language that combines positive aspects of different Datalog$^{\pm}$ languages. With respect to properties (1)–(5) above, the class of shy programs behaves as follows: (1) it inherits the simplicity and naturalness of Datalog; (2) it is decidable; (3) it is efficiently computable (tractable data complexity and limited combined-complexity —no complexity overhead w.r.t. Datalog); (4) it offers a good expressive power being a strict superset of Datalog; and (5) it is suitable for an efficient implementation. Specifically, shy programs can be evaluated by parsimonious forward-chaining inference that allows for an efficient on-the-fly OBQA, as witnessed by our experimental results.[1] From a technical viewpoint, the contribution of the paper is the following:

(1) *Parsimonious chase*: We define the parsimonious chase procedure, which is sound and terminating over any Datalog$^{\exists}$ program. An infinite reapplication of this procedure ensures completeness also for conjunctive queries.
(2) *Parsimony*: We propose a new semantic property called parsimony, and we prove that on the abstract class of parsimonious Datalog$^{\exists}$ programs, called ps, atomic

[1] Intuitively, parsimonious inference generates no isomorphic atoms (see Section 3); while on-the-fly OBQA does not need any preliminary materialization or compilation phase (see Section 9), and is very well suited for query answering over frequently changing ontologies.

query answering is decidable and also efficiently computable via the parsimonious chase. After showing that conjunctive query answering over ps is undecidable, we focus on strongly parsimonious programs, or sps, to gain decidability also in the presence of conjunctive queries. In particular, it suffices to "reapply" the parsimonious chase a number of times that is linear in the size of the query. Moreover, we demonstrate that both ps and sps preserve the same (data and combined) complexity of Datalog for atomic query answering: the addition of existential quantifiers does not bring any computational overhead here.

(3) *Shyness*: Since the recognition of parsimony is undecidable (we prove that it is coRE-complete), we single out shy, a subclass of sps, which guarantees both easy recognizability and efficient answering even to conjunctive queries. In particular, shy generalizes Datalog as well as the class of linear existential programs (i.e., with at most one body-atom), while it is uncomparable to the other main classes ensuring decidability. Moreover, we demonstrate that shy preserves the same (data and combined) complexity of Datalog for both atomic and conjunctive query answering.

(4) *Implementation*: We implement a bottom-up evaluation strategy for shy programs inside the DLV system, and enhance the computation by a number of optimization techniques, yielding DLV$^\exists$ —a powerful system for query answering over shy programs, which is profitably applicable for OBQA. To the best of our knowledge, DLV$^\exists$ has been the first[2] system supporting the standard first-order semantics for unrestricted CQs with existential variables over ontologies with advanced properties (some of these beyond $\mathbf{AC}_0$), such as, role transitivity, role hierarchy, role inverse, and concept products.[3]

(5) *Comparison*: We analyze related work, providing a precise taxonomy of the QA-decidable Datalog$^\exists$ classes. It turns out that both sps and shy strictly contain both datalog and linear, while they are uncomparable to fes (finite expansion sets), bts (bounded treewidth sets), and fus (finite unification sets).

(6) *Experiments*: We perform an experimental analysis, comparing DLV$^\exists$ against a number of state-of-the-art systems for OBQA. The positive results attained through this analysis do give clear evidence that DLV$^\exists$ is definitely the most effective system for query answering in dynamic environments, where the ontology is subject to frequent changes, making pre-computations and static optimizations inapplicable. In particular, it turns out that DLV$^\exists$ outperforms all other systems in terms of number of solved queries, and it is faster in terms of running time. The DLV$^\exists$ system is freely available for experimenting with Datalog$^\pm$ and can be downloaded from https://www.mat.unical.it/dlve/.

1.3. Organization

The remaining of the paper is organized as follows. Section 2 formally fixes syntax and semantics of Datalog$^\exists$ programs, as well as some preliminaries and useful notation. Section 3, after defining the parsimonious chase and parsimonious programs, studies the decidability of atomic query answering over parsimonious programs. Section 4, after refining the concept of parsimony, studies the decidability of conjunctive query answering over strongly parsimonious programs. Section 5 presents the shy language and its main properties. Section 6 deals with computational complexity. Sections 7 describes the DLV$^\exists$ system. Section 8 surveys notable related works and compares the newly defined classes with the main decidable ones from the literature. Section 9 describes the experimental analysis carried out to evaluate the efficiency and the effectiveness of DLV$^\exists$. Section 10 concludes the paper.

[2]DLV$^\exists$ has been released for the first time by Leone et al. [2012].
[3]These properties can be even combined under suitable restrictions.

2. DATALOG WITH EXISTENTIAL QUANTIFIERS

In this section, after introducing syntax and semantics of Datalog with existential quantifiers, we fix the notion of ontological inference that we are going to consider in this paper, which is the problem of answering a conjunctive query over a set of Datalog rules extended with existential quantification in the head.

2.1. Preliminaries

The following notation will be used throughout the paper. We always denote by Δ_C, Δ_N and Δ_V, countably infinite domains of *terms* called *constants*, *nulls* and *variables*, respectively; by Δ, the union of these three domains; by φ, a null; by X, Y and Z variables; by $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$, sets (or tuples) of variables; by $\mathcal{R}$ a *relational schema* consisting of a set of *relational predicates* each of which has a fixed nonnegative *arity*; by $\underline{a}$, $\underline{b}$ and $\underline{c}$, *atoms* being expressions of the form $p(\mathbf{t})$, where p is a relational predicate, and $\mathbf{t} = t_1, \ldots, t_k$ is a *tuple* of terms.

A position $p[i]$ (in a schema $\mathcal{R}$) is identified by a predicate $p \in \mathcal{R}$ and its i-th argument (or attribute). For an atom $\underline{a}$, we denote by $pred(\underline{a})$ the relational predicate of $\underline{a}$. For a structure ς containing atoms, $atoms(\varsigma)$ denotes the set of atoms in ς, $terms(\varsigma)$ denotes the set of terms occurring in $atoms(\varsigma)$, and $arity(\varsigma)$ denotes the maximum arity over all the relational predicates occurring in ς.

If $\mathbf{X}$ is the set of variables in ς (namely, $\mathbf{X} = terms(\varsigma) \cap \Delta_V$), then ς is also denoted by $\varsigma_{[\mathbf{X}]}$. A structure $\varsigma_{[\emptyset]}$ is called *ground*. If $T \subseteq \Delta$ and $T \neq \emptyset$, then $base(T)$ denotes the set of all atoms that can be formed with predicates of $\mathcal{R}$ and terms from T. An *instance*, usually denoted by I, is any nonempty subset of $base(\Delta_C \cup \Delta_N)$.

A *substitution* is a total mapping from terms to terms. Consider a set $T \subseteq \Delta$ and a substitution $\sigma : \Delta \to \Delta$. The restriction of σ to T, denoted by $\sigma|_T$, is the substitution σ' such that $t \in T$ implies $\sigma'(t) = \sigma(t)$, and $t \notin T$ implies $\sigma'(t) = t$. In this case, we say that σ is an *extension* of σ', namely $\sigma \supseteq \sigma'$. The restriction $\sigma|_\emptyset$ defines the *empty substitution*, conventionally denoted by $\sigma_\emptyset$. The application of σ to T, denoted by $\sigma(T)$, is the set $\{\sigma(t) \mid t \in T\}$. For an atom $\underline{a} = p(t_1, \ldots, t_k)$, we have that $\sigma(\underline{a}) = p(\sigma(t_1), \ldots, \sigma(t_k))$. For a structure ς containing atoms, we denote by $\sigma(\varsigma)$ the structure obtained by replacing each atom $\underline{a}$ of ς with $\sigma(\underline{a})$. The *composition* of a substitution σ_1 with a substitution σ_2, denoted by $\sigma_2 \circ \sigma_1$, is the substitution associating each $t \in \Delta$ to $\sigma_2(\sigma_1(t))$.

Let ς_1 and ς_2 be two structures containing atoms. A *homomorphism* from ς_1 to ς_2 is a substitution h which satisfies the following conditions: (i) $c \in \Delta_C$ implies $h(c) = c$; (ii) $\varphi \in \Delta_N$ implies $h(\varphi) \in \Delta_C \cup \Delta_N$; and (iii) $h(\varsigma_1)$ is a substructure of ς_2 (for example, if ς_1 and ς_2 are sets of atoms, then $h(\varsigma_1) \subseteq \varsigma_2$).

2.2. Programs and Queries

An *(existential) rule* r is a finite expression of the form $\forall \mathbf{X}\ (\exists \mathbf{Y}\ \underline{a}_{[\mathbf{X}' \cup \mathbf{Y}]} \leftarrow \varsigma_{[\mathbf{X}]})$, where (i) $\mathbf{X}$ and $\mathbf{Y}$ are disjoint sets of variables (next called $\forall$-variables and $\exists$-variables, respectively); (ii) $\mathbf{X}' \subseteq \mathbf{X}$; (iii) $\underline{a}_{[\mathbf{X}' \cup \mathbf{Y}]}$ is an atom on the variables $\mathbf{X}' \cup \mathbf{Y}$; and (iv) $\varsigma_{[\mathbf{X}]}$ is a conjunction of (zero, one, or more) atoms on the variables $\mathbf{X}$. Constants may also occur in r. In the following, $head(r) = \underline{a}_{[\mathbf{X}' \cup \mathbf{Y}]}$, and $body(r) = atoms(\varsigma_{[\mathbf{X}' \cup \mathbf{Y}]})$. Universal quantifiers are usually omitted to lighten the syntax, while existential quantifiers are omitted only if $\mathbf{Y} = \emptyset$. In the second case, r coincides with a standard Datalog rule. If $body(r) = \emptyset$, then r is also called 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 $pred(P)$ the predicates occurring in P, by $data(P)$ all the atoms in the head of the ground facts of P, and by $dep(P)$ the *dependencies* of P, which are the rules of P being not ground facts.

Finally, without loss of generality, we assume that, for each pair $\langle r_1, r_2 \rangle$ of rules of P, $vars(r_1) \cap vars(r_2) = \emptyset$.

Example 2.1. Consider the program P consisting of the following rules.

$$\begin{array}{lrcl} r_1: & person(john) & \leftarrow & \\ r_2: & \exists Y_2\, hasFather(X_2, Y_2) & \leftarrow & person(X_2) \\ r_3: & person(Y_3) & \leftarrow & hasFather(X_3, Y_3) \end{array}$$

We have that r_1 is a ground fact, $data(P) = \{person(john)\}$, and $dep(P) = \{r_2, r_3\}$. □

A *conjunctive query* (*CQ*) q is a first-order (FO) expression of the form $\exists \mathbf{Y}\, \varsigma_{[\mathbf{X} \cup \mathbf{Y}]}$, where $\mathbf{X}$ are the free variables, $\mathbf{Y}$ are the existential variables, and $\varsigma_{[\mathbf{X} \cup \mathbf{Y}]}$ is a conjunction of atoms on the variables $\mathbf{X} \cup \mathbf{Y}$. Constants may also occur in q. To highlight the free variables, we write $q(\mathbf{X})$ instead of q. Moreover, q is called *atomic* if $\varsigma_{[\mathbf{X} \cup \mathbf{Y}]}$ is simply an atom. Finally, a *Boolean conjunctive query* (*BCQ*) is a conjunctive query without free variables.

Example 2.2. Both the following expressions are conjunctive queries.

$$\begin{array}{ll} q_1: & \exists Y \exists Z\, hasFather(X, Y), hasFather(Y, Z) \\ q_2: & \exists X\, hasFather(john, X) \end{array}$$

In particular, q_1 has a single free variable, namely X. Regarding q_2, it is Boolean since it contains no free variable, and it is atomic since it consists of only one atom. □

2.3. Semantics

Consider an instance I. We say that I *satisfies* a rule r if whenever there is a homomorphism h from $body(r)$ to I, there is a homomorphism $h' \supseteq h|_{vars(body(r))}$ from $head(r)$ to I. Moreover, I is a *model* of a program P, denoted by $I \models P$, if I satisfies each rule of P. Let $mods(P)$ denote the set of all the models of P.

Consider a BCQ q. We say that q is *true* over an instance I, denoted by $I \models q$, if there is a homomorphism $h = h|_{vars(q)}$ from $atoms(q)$ to I. Moreover, q is *true* over a program P, denoted by $P \models q$, if q is true with respect to each model of P.

Consider a conjunctive query $q(\mathbf{X})$. The answer to q over an instance I is the set $\mathsf{ans}(q, I) = \{\sigma|_{\mathbf{X}} \ : \ \sigma \text{ is a substitution and } I \models \sigma|_{\mathbf{X}}(q)\}$. Observe that, in case q is a BCQ, it holds that $\mathsf{ans}(q, I) = \{\sigma_\emptyset\}$ if and only if q is true over I. The answer to q over a program P is the set $\mathsf{ans}_P(q) = \{\sigma \ : \ \text{for each } I \in mods(P),\ \sigma \in \mathsf{ans}(q, I)\}$.

We now fix the computational problem studied in this paper.

Definition 2.3 (*Boolean Conjunctive Query Evaluation*). The problem BCQEVAL is defined as follows. *Given a program P, and a Boolean conjunctive query q, decide whether q is true over P.* □

Before concluding this section, we mention that the problem BCQEVAL can be carried out by using a universal model. Actually, a model U of P is called *universal* if, for each $M \in mods(P)$, there is a homomorphism from U to M. And such a property immediately implies the following proposition [Fagin et al. 2005b].

PROPOSITION 2.4. *Consider a program P, a universal model U of P, and a BCQ q. It holds that $P \models q$ if and only if $U \models q$.*

2.4. The Chase

As already mentioned, the *chase* [Maier et al. 1979; Johnson and Klug 1984] is definitely the prime procedure for constructing a universal model of a program. Four main variants of this procedure have been proposed in the literature, called *oblivious* (or

Input: A Datalog$^\exists$ program P.
Output: The universal model $chase(P)$.

```
1  I' := data(P);
2  I := I';
3  for each rule r of dep(P) do
4     for each unspent (w.r.t. r) firing homomorphism h for the pair ⟨r, I⟩ do
5        if ⟨r, h⟩ satisfies the fire condition with respect to I' then I' := I' ∪ {fire(r, h)};
6  if I ≠ I' then goto step 2;
   else return I;
```

Fig. 1. The chase procedure.

naive) [Calì et al. 2013b], *skolem* [Marnette 2009], *restricted* (or standard) [Fagin et al. 2005b], and *core* [Deutsch et al. 2008]. In what follows, the first and the third variants are described, which are those that are exploited later in the paper.

Consider a rule r and an instance I. A *firing* homomorphism for the pair $\langle r, I\rangle$ is any homomorphism h from $body(r)$ to I such that $h = h|_{vars(body(r))}$. The fire of r via h produces the atom $fire(r, h)$ that is obtained from $h(head(r))$ by replacing each $\exists$-quantified variable of r with a different null. After that, h is said to be *spent* with respect to r.

Depending on the variant of the chase under consideration, however, the fire of r via h may be subject to a specific *fire condition*. Consider an instance $I' \supseteq I$. In the case of the restricted chase, we say that the pair $\langle r, h\rangle$ satisfies the fire condition with respect to I' if there is no homomorphism $h' \supseteq h$ from $\{head(r)\}$ to I'. Differently, in the case of the oblivious chase, the pair $\langle r, h\rangle$ always satisfies the fire condition with respect to I'.

Example 2.5. Let P be a program consisting of the following rules:

$$\begin{array}{rrcl} r_1: & employee(john) & \leftarrow & \\ r_2: & hasManager(john, john) & \leftarrow & \\ r_3: & \exists Y_3\, worksFor(X_3, Y_3) & \leftarrow & employee(X_3) \\ r_4: & \exists Y_4\, hasManager(X_4, Y_4) & \leftarrow & employee(X_4) \\ r_5: & employee(X_5) & \leftarrow & hasManager(Y_5, X_5) \end{array}$$

The restricted chase produces the following output: $rchase(P) = data(P) \cup \{worksFor(john, \varphi_1)\}$, where $worksFor(john, \varphi_1)$ is the atom generated by the fire of r_3 via homomorphism $h_3 : \{X_3 \mapsto john\}$. It is worth noting that the pair $\langle r_4, h_4\rangle$, with $h_4 = \{X_4 \mapsto john\}$, does not satisfy the restricted fire condition with respect to $data(P) \cup \{worksFor(john, \varphi_1)\}$; therefore r_4 will not be fired. In particular, $fire(r_4, h_4)$ is not generated because there exists a homomorphism $h'_4 \supseteq h_4$ from $\{hasManager(X_4, Y_4)\}$ to $\{hasManager(john, john)\}$. Thus, the procedure terminates after a finite number of steps. Conversely, the oblivious chase produces infinitely many atoms: $ochase(P) = data(P) \cup S_1 \cup S_2$, where $S_1 = \{worksFor(john, \varphi_1),\ hasManager(john, \varphi_2)\}$, and $S_2 = \{employee(\varphi_i),\ hasManager(\varphi_i, \varphi_{i+1})\}_{i\geq 2}$. □

Starting from a program P, Figure 1 illustrates the overall chase procedure over P. The procedure consists of an exhaustive series of fires in a breadth-first (level-saturating) fashion, which leads as result to the (possibly infinite) universal model $chase(P)$. More precisely, as far as the restricted (resp., oblivious) chase is concerned, the output of Figure 1 can be also denoted by $rchase(P)$ (resp., $ochase(P)$). Importantly, different fires (of the same or different rules) always introduce different "fresh" nulls.

For simplicity of exposition and without loss of generality, we assume that nulls introduced at each fire functionally depend on the pair $\langle r, h\rangle$ that is involved in the fire. The last assumption has the immediate consequence that (regardless of the order in which the rules and the firing homomorphisms are processed) $ochase(P)$ can be

considered unique and that $rchase(P) \subseteq ochase(P)$. For example, such a behavior can be achieved as follows: given a pair $\langle r, h\rangle$ such that r (as given in Section 2.2) contains an existentially quantified variable Y, it suffices to pick $\varphi_{\langle Y,h(\mathbf{X})\rangle}$ as the fresh null replacing Y when the oblivious chase produces $\mathit{fire}(r, h)$.[4]

Finally, the *chase relation* $CR[P]$ of P is the maximal subset of $ochase(P) \times ochase(P)$ satisfying the following condition: if $\langle \underline{a}, \underline{b}\rangle$ belongs to $CR[P]$, then there exist a rule r and a homomorphism h such that: (i) $\underline{a} \in h(body(r))$, and (ii) $\underline{b} = \mathit{fire}(r, h)$ has been produced by the fire of r via h.

PROPOSITION 2.6. *[Fagin et al. 2005b; Deutsch et al. 2008] Given a program P, instance $chase(P)$ is a universal model of P.*

Unfortunately, the chase procedure does not always terminate.

PROPOSITION 2.7. *[Fagin et al. 2005b; Deutsch et al. 2008] The problem* BCQEVAL *is undecidable, even for atomic queries. In particular, it is RE-complete.*

3. DECIDABILITY OVER ATOMIC QUERIES: PARSIMONIOUS PROGRAMS

This section considers a novel semantic property, called parsimony, that guarantees decidability of atomic query answering in general, and of conjunctive query answering under some assumptions provided in the next section. To this end, we start by defining a variant of the chase procedure presented in Section 2.4, called parsimonious chase, that differs from the original version only for the adoption of a stricter fire condition.

Definition 3.1. Consider a rule r, an instance I, and a firing homomorphism h for $\langle r, I\rangle$. The pair $\langle r, h\rangle$ satisfies the *parsimonious fire condition* with respect to an instance $I' \supseteq I$ if there is no homomorphism from $\{h(head(r))\}$ to I'. □

Differently from the standard chase, a null in $h(head(r))$ can be mapped to $\Delta_C \cup \Delta_N$. The *parsimonious chase* is the procedure that is obtained from the one shown in Figure 1 by replacing the original fire condition (step 5) with the parsimonious one introduced in Definition 3.1. The new output is denoted by $pchase(P)$. We now compare, with the aid of the following example, the behavior of the parsimonious chase with that of the original one.

Example 3.2. Consider again program P given in Example 2.1. The parsimonious chase terminates after a finite number of steps and produces the following output: $pchase(P) = \{person(john), hasFather(john, \varphi_1)\}$, by firing rule r_2 only. In fact, the pair $\langle r_3, \{X_3 \mapsto john, Y_3 \mapsto \varphi_1\}\rangle$ does not satisfy the parsimonious fire condition since there is a homomorphism from $\{person(\varphi_1)\}$ to $pchase(P)$. Conversely, the original chase procedure does not stop and generates an infinite number of fathers. In particular, we have that $chase(P) = pchase(P) \cup S_1 \cup S_2$, where $S_1 = \{person(\varphi_i)\}_{i\in\mathbb{N}^+}$ and $S_2 = \{hasFather(\varphi_i, \varphi_{i+1})\}_{i\in\mathbb{N}^+}$. □

Note that, differently from $chase(P)$, the instance $pchase(P)$ might not be a model. However, as stated in the following proposition, the parsimonious chase is sound for query answering purposes.

PROPOSITION 3.3. *Consider a program P. It holds that $pchase(P) \subseteq ochase(P)$ and, therefore, that $ochase(dep(P) \cup pchase(P)) = ochase(P)$.*

[4]Since different rules share no variable, $\varphi_{\langle Y,h(\mathbf{X})\rangle}$ and $\varphi_{\langle Y,r,h(\mathbf{X})\rangle}$ are equivalent. Moreover, note that a standard Skolemization (as in the skolem chase introduced by Marnette [2009]) replacing Y in r by the functional term $f_Y(\mathbf{X}')$ would produce a different result since $\mathbf{X}' \subseteq \mathbf{X}$.

PROOF. The statement holds since we are assuming that the nulls introduced at each fire functionally depend on the pair $\langle r, h\rangle$ that is involved in the fire, and since the parsimonious fire condition is a stricter criterion than the oblivious one. In fact, given a rule r, an instance I, and a firing homomorphism h for $\langle r, I\rangle$, it holds that if there is no homomorphism from $\{h(head(r))\}$ to another instance $I' \supseteq I$, then there is also no homomorphism $h' \supseteq h$ from $\{head(r)\}$ to I'. Moreover, $ochase(dep(P) \cup pchase(P)) = ochase(P)$ holds since the chase is a monotone procedure. □

Based on Definition 3.1, we next define a new class of programs depending on a novel semantic property, called *parsimony*.

Definition 3.4 (*Parsimony*). A program P is called *parsimonious* if, for each atom $\underline{a}$ of $chase(P)$, there exists a homomorphism from $\{\underline{a}\}$ to $pchase(P)$. □

Observe that, according to Definition 3.4, the program discussed in Example 3.2 is parsimonious because for each $person(\varphi) \in S_1$, there exists homomorphism $h = \{\varphi \mapsto john\}$ such that $h(person(\varphi)) \in pchase(P)$, and for each $hasFather(\varphi, \varphi') \in S_2$, there exists a homomorphism $h' = \{\varphi \mapsto john,\ \varphi' \mapsto \varphi_1\}$ such that $h'(hasFather(\varphi, \varphi')) \in pchase(P)$.

Before proving that parsimony guarantees decidability of Boolean atomic query answering, we show that, for each program P, $pchase(P)$ can be computed in finite time, and therefore it is always of finite size.

PROPOSITION 3.5. *The parsimonious chase always terminates.*

PROOF. Consider a program P. Let $C = terms(data(P))$, $c = |C|$ be the number of constants of P, and $\omega = arity(P)$. We claim that $|pchase(P)| \leq |pred(P)| \cdot (c + \omega)^{\omega}$. Towards a contradiction, assume that $|pchase(P)| > |pred(P)| \cdot (c+\omega)^{\omega}$. By Proposition 3.3, it holds that $pchase(P) \subseteq ochase(P)$, and according to the chase procedure we have that $ochase(P) \subseteq base(C \cup \Delta_N)$. Hence, $pchase(P) \subseteq base(C \cup \Delta_N)$. However, any subset S of $base(C \cup \Delta_N)$ with $|S| > |pred(P)| \cdot (c+\omega)^{\omega}$ necessarily contains two atoms, say $\underline{a}$ and $\underline{a}'$, which are isomorphic, namely there is a homomorphism h from $\{\underline{a}\}$ to $\{\underline{a}'\}$ such that h^{-1} is a homomorphism from $\{\underline{a}'\}$ to $\{\underline{a}\}$. Therefore, $pchase(P)$ would necessarily contain at least two such isomorphic atoms $\underline{a}$ and $\underline{a}'$. Since $data(P)$ contains no pair of isomorphic atoms, it follows that $\underline{a}$ or $\underline{a}'$ is produced by the parsimonious chase. But this is not possible because the parsimonious fire condition would be violated. Finally, in the worst case —when the procedure generates one atom during each macro iteration beginning at step 2— the parsimonious chase stops after finitely many steps. □

We are now ready to show that parsimony guarantees decidability of Boolean atomic query answering.

THEOREM 3.6. *The problem* BCQEVAL *over parsimonious programs and atomic queries is decidable.*

PROOF. Consider a parsimonious program P and a Boolean atomic query $q = \exists \mathbf{Y}\, \underline{a}_{[\mathbf{Y}]}$. Since, by Proposition 3.5, the parsimonious chase always terminates after finitely many steps, to prove the statement it suffices to show that $ochase(P) \models q$ if, and only if, $pchase(P) \models q$. The "if" direction holds since $pchase(P) \subseteq ochase(P)$, by Proposition 3.3. Regarding the "only if" direction, we know that there is a homomorphism h from $\{\underline{a}\}$ to $ochase(P)$. But, by Definition 3.4, we also know that there is a homomorphism h' from $\{h(\underline{a})\}$ to $pchase(P)$. Hence, $h' \circ h$ is a homomorphism from $\{\underline{a}\}$ to $pchase(P)$. □

4. DECIDABILITY OVER CONJUNCTIVE QUERIES: STRONGLY PARSIMONIOUS PROGRAMS

Unfortunately, parsimony alone is not enough to guarantee also the decidability of Boolean conjunctive query evaluation, and therefore of conjunctive query answering.

THEOREM 4.1. *The problem* BCQEVAL *over parsimonious programs is undecidable.*

PROOF. To prove the statement, we define a family of parsimonious programs each of which simulates the behavior of a single-tape deterministic Turing machine that reads the empty string. The first part of the construction —inspired by the one provided by Calì et al. [2013b]— serves to simulate a given Turing machine on the empty string; the second one provides extra rules that do not alter the aforementioned simulation but do make the entire program parsimonious by "saturating" the output of the parsimonious chase with harmless atoms. More precisely, consider a single-tape deterministic Turing machine $M = \langle K, \Sigma, s_0, \delta \rangle$ where: (i) K is a finite set of states; (ii) Σ is the alphabet of M; (iii) $s_0 \in K$ is the initial state; and (iv) $\delta : K \times (\Sigma \cup \{\sqcup\}) \to (K \cup \{\mathsf{h}\}) \times (\Sigma \cup \{\sqcup\}) \times D$ is the (total) transition function of M, where $\sqcup$ is the blank symbol, h is the halting state, and $D = \{left, stay, right\}$ denotes the standard motion directions. We build a parsimonious program P and a BCQ q such that $M(\varepsilon)$ terminates if, and only if, $P \models q$.

For each transition of M of the form $\delta(s, a) = (s'a'd)$, we add to P the following fact:

$$trans(s, a, s', a', d) \leftarrow$$

To store the configurations of M on ε, we use the predicates $tape$, $cursor$ and $state$. In particular, an atom $tape(t, c, a)$ says that at time t the content of the tape-cell c is the symbol a. Similarly, $cursor(t, c)$ represents the fact that at time t the cursor is under the cell c. Finally, $state(t, s)$ says that at time t the state of the machine is s.

To consider the fact that M on ε can use infinitely many cells and, therefore, perform infinitely many steps, we add to P the following rules, which define a countably infinite well-ordered set of elements that will be used for both cells and steps:

$$\begin{aligned} index(0) &\leftarrow \\ \exists Y\ next(X, Y) &\leftarrow index(X) \\ index(Y) &\leftarrow next(X, Y) \end{aligned}$$

We are now ready to simulate the behavior of M over ε. To represent the initial configuration of the machine, we add to P the following facts:

$$\begin{aligned} cursor(0, 0) &\leftarrow \\ state(0, s_0) &\leftarrow \\ tape(0, 0, \sqcup) &\leftarrow \end{aligned}$$

saying that at time 0: the cursor is under cell 0, the state of M is s_0, and the content of tape-cell 0 is symbol $\sqcup$, respectively. For each configuration, the following rule collects in a single predicate $trigger$ the main information used by M to take a single step according to δ:

$$trigger(T, C, S, A) \leftarrow cursor(T, C), state(T, S), tape(T, C, A)$$

In particular, an atom $trigger(t, c, s, a)$ says that at time t: the cursor is under cell c, the state of M is s, and the content of c is a. To update the cursor position during a single step, we add to P the following rules, which consider the three possible motion directions of D:

$$\begin{aligned} cursor(T', C_r) &\leftarrow trigger(T, C, S, A),\ trans(S, A, \_, \_, right),\ next(C, C_r), next(T, T') \\ cursor(T', C_l) &\leftarrow trigger(T, C, S, A),\ trans(S, A, \_, \_, left),\ next(C_l, C), next(T, T') \\ cursor(T', C) &\leftarrow trigger(T, C, S, A),\ trans(S, A, \_, \_, stay),\ next(T, T') \end{aligned}$$

Similarly, we update the state of M and the content of the tape-cell under the cursor:

$$\begin{aligned} state(T', S') &\leftarrow trigger(T, C, S, A),\ trans(S, A, S', \_, \_),\ next(T, T') \\ tape(T', C, A') &\leftarrow trigger(T, C, S, A),\ trans(S, A, \_, A', \_),\ next(T, T') \\ write(T, C) &\leftarrow trigger(T, C, S, A),\ trans(S, A, \_, \_, \_) \end{aligned}$$

Notice that, predicate $write$ is used to mark, during each single step, the tape-cell that has been modified. Anyway, we need additional "inertia" rules to ensure that all other tape-cells preserve their previous value. To this end, we use two different markings: $keep_r$ for the tape positions that follow the one marked with $write$, and $keep_l$ for the preceding tape positions. In this way, we are able to ensure that, at every time instant t, every tape-cell c, such that $keep_l(t, c)$ or $keep_f(t, c)$ is true, keeps the same symbol at instant t' following t. Thus, the following rules propagate the aforementioned markings forward and backward, respectively, starting from the marked tape positions:

$$\begin{aligned} keep_r(T, C_r) &\leftarrow write(T, C),\ next(C, C_r) \\ keep_r(T, C_r) &\leftarrow keep_r(T, C),\ next(C, C_r) \\ keep_l(T, C_l) &\leftarrow write(T, C),\ next(C_l, C) \\ keep_l(T, C_l) &\leftarrow keep_l(T, C),\ next(C_l, C) \end{aligned}$$

To update the tape-content after each step of M, we use the following rules guaranteeing that the tape-cells marked by either $keep_r$ or $keep_l$ keep their previous values:

$$\begin{aligned} tape(T', C, A) &\leftarrow tape(T, C, A),\ keep_r(T, C),\ next(T, T') \\ tape(T', C, A) &\leftarrow tape(T, C, A),\ keep_l(T, C),\ next(T, T') \end{aligned}$$

Finally, the Boolean conjunctive query q is:

$$\exists T\, index(T),\ state(T, \mathsf{h})$$

This concludes the first part of our construction. To show that $M(\varepsilon) = h$ if and only if $P \models q$ holds, let us first assume that $M(\varepsilon) = h$. Then, there exists a sequence of machine transitions which starts from the initial configuration and terminates in the halting one. Notice that, both the initial configuration and the transition function of M have been encoded in P in such a way that they can simulate the behaviour of M step by step. Consequently, since the rules of P are deterministically applied according to the transition function, we have that, by construction, there exists a time step t such that $index(t), state(t, h) \in chase(P)$. Hence, in this case $P \models q$. Whereas, the "if" direction follows from the fact that if $P \models q$ then there exists a time step t such that $state(t, h)$ is produced by the chase. This implies that, going backward to the initial state in the chase, there exists a sequence of time steps which led M to the halting state. Notice that, such a sequence is produced by a deterministic application of the rules of P which encode the transition function of M. This means that, in this case, M over ε would terminate in the halting state. Thus, we can now assert that, by construction, $M(\varepsilon) = h$ if and only if $P \models q$.

Up to this point, however, program P might not be parsimonious in general. To make the program parsimonious, while preserving its properties of simulating M over ε, we add to P the following rules:

$$\begin{aligned} \exists T\, state(T, S) &\leftarrow trans(S, \_, \_, \_, \_) \\ \exists T\, state(T, \mathsf{h}) &\leftarrow \\ \exists T\, tape(T, 0, A) &\leftarrow trans(\_, A, \_, \_, \_) \\ \exists T \exists C\, trigger(T, C, S, A) &\leftarrow state(\_, S),\ tape(\_, \_, A) \end{aligned}$$

Let us now refer to this new program as P and to the first version as P^-. We recall that, a program P is called parsimonious if, for each atom $\underline{a}$ of $chase(P)$ there exists a homomorphism from $\{\underline{a}\}$ to $pchase(P)$. Thus, to prove that P is parsimonious, it suffices

to identify a superset X of $chase(P)$ such that there exists a homomorphism from each atom of X to $pchase(P)$. To this end, let $\Delta'_N = \Delta_N \cup \{0\}$ be a set of terms, and $\Sigma' = \Sigma \cup \{\sqcup\}$ be an alphabet; we construct X as the union of the following sets of atoms:

$$
\begin{aligned}
&X_{trans} = \{trans(s, a, s', a', d) \mid (s, a) \mapsto (s', a', d) \in \delta\}\\
&X_{index} = \{index(i) \mid i \in \Delta'_N\}\\
&X_{next} = \{next(i, i') \mid (i, i') \in \Delta'_N \times \Delta_N\}\\
&X_{tape} = \{tape(0, 0, \sqcup)\} \cup \{tape(t, c, a) \mid (t, c, a) \in \Delta_N \times \Delta'_N \times \Sigma'\}\\
&X_{cursor} = \{cursor(0, 0)\} \cup \{cursor(t, c) \mid (t, c) \in \Delta_N \times \Delta'_N\}\\
&X_{state} = \{state(0, s_0)\} \cup \{state(t, s) \mid (t, s) \in \Delta_N \times K\}\\
&X_{write} = \{write(0, 0)\} \cup \{write(t, c) \mid (t, c) \in \Delta_N \times \Delta'_N\}\\
&X_{keep_r} = \{keep_r(t, c) \mid (t, c) \in \Delta'_N \times \Delta_N\}\\
&X_{keep_l} = \{keep_l(t, c) \mid (t, c) \in \Delta_N \times \Delta'_N\}\\
&X_{trigger} = \{trigger(0, 0, s_0, \sqcup)\} \cup \{trigger(t, c, s, a) \mid (t, c, s, a) \in \Delta_N \times \Delta'_N \times K \times \Sigma'\}
\end{aligned}
$$

First of all, let us analyze $chase(P)$ to prove that it is a subset of X. Let $\mathcal{R} = \{p_1, p_2, ..., p_k\}$ be the relational schema consisting of the relational predicates of P, $chase(P)$ can be partitioned into k mutually disjoint subsets $S|p_1, S|p_2, ..., S|p_k$ such that $S|p_i = \{\underline{a} \mid \underline{a} \in chase(P) \ \wedge pred(\underline{a}) = p_i\}$ for each predicate $p_i \in \mathcal{R}$. Thus, to prove that $chase(P) \in X$ it suffices to show that $S|p_i \in X$ for each predicate $p_i \in \mathcal{R}$. Starting from $data(P)$ and considering all the possible ways of propagating a single term during the chase, we have that $S|p_i \subseteq X_{p_i}$ for each predicate $p_i \in \mathcal{R}$, which immediately implies that $chase(P) \subseteq X$.

Let us now consider the parsimonious chase in order to prove that there is a homomorphism from each atom of X to $pchase(P)$. In particular, $pchase(P)$ is composed by the union of the following sets of atoms:

$$
\begin{aligned}
&Y_{trans} = \{trans(s, a, s', a', d) \mid (s, a) \mapsto (s', a', d) \in \delta\}\\
&Y_{index} = \{index(0)\}\\
&Y_{next} = \{next(0, \varphi_1)\}\\
&Y_{tape} = \{tape(0, 0, \sqcup)\} \cup \{tape(\varphi_2, 0, a) \mid a \in \Sigma \cup \{\sqcup\}\}\\
&Y_{cursor} = \{cursor(0, 0)\}\\
&Y_{state} = \{state(0, s_0)\} \cup \{state(\varphi_3, s) \mid s \in K\}\\
&Y_{write} = \{write(0, 0)\}\\
&Y_{keep_r} = \{keep_r(0, \varphi_4)\}\\
&Y_{keep_l} = \{keep_l(\varphi_5, 0)\}\\
&Y_{trigger} = \{trigger(0, 0, s_0, \sqcup)\} \cup \{trigger(\varphi_6, \varphi_7, s, a) \mid s \in K \wedge a \in \Sigma \cup \{\sqcup\}\}
\end{aligned}
$$

Given a predicate $p \in R$, it is now straightforward to see that for each atom $\underline{a} \in X_p$ there exists a homomorphism from $\underline{a}$ to Y_p. This implies that there is a homomorphism from each atom of X to $pchase(P)$. Consequently, since we proved that $chase(P) \subseteq X$, we immediately get that P is parsimonious.

Now, to prove that $M(\varepsilon) = h$ if and only if $P \models q$ still holds despite the addition of the above rules, we show that the new rules do not interfer in the evaluation of q. Let $P' = P - P^-$ be the set of rules added to P^- in order to get parsimony. Let us denote by A' and A^- the disjoint sets of atoms generated during the chase by the fire of rules from P' and P^-, respectively. In particular, we have that $chase(P) = data(P) \cup A' \cup A^-$ where $A' \cap data(P) = \emptyset$ and $A^- \cap data(P) = \emptyset$. The set of predicates occurring in A' is referred to as $\mathcal{R}_{A'} = \{state, tape, trigger, write\}$. Notice that, $A' \not\models q$ holds because there are no atoms of predicate $index$ in A'. Hence, to prove that the addition of P' to P^- does not interfer in the evaluation of q, it suffices to show that there is no rule in P^- which admits a firing homomorphism that maps a body atom to an atom of A'. It is straightforward to see that the claim holds for the rules of P^- where a predicate from $R_{A'}$ does not occur in the body. Consider now the rest of the rules of P^-, denoted by

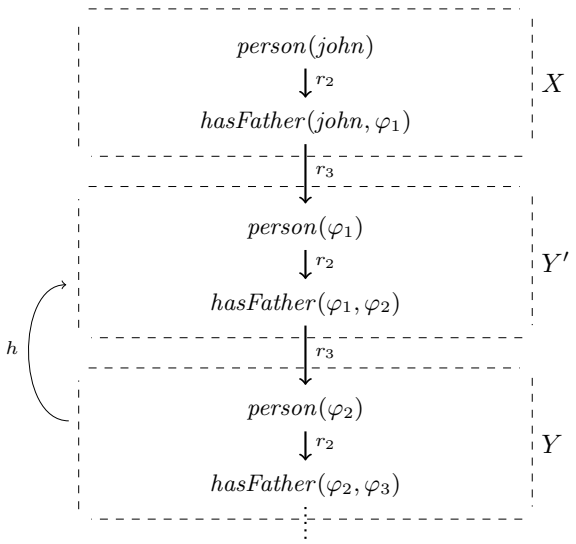

Fig. 2. A chase-fragment and a nucleus of the program considered in Example 4.3.

$P^-_{[R_{A'}]} = \{r \mid r \in P^- \wedge \exists \underline{a} \in body(r)\, s.t.\, pred(\underline{a}) = \mathcal{R}_{A'}\}$. Let N' and N^- be the sets of nulls appearing in A' and A^-, respectively. More specifically, $N' = terms(A') \cap \Delta_N$ and $N^- = terms(A^-) \cap \Delta_N$. Notice that, by construction, a null from N' occurs in the first position of each atom $\underline{a} \in A'$. Moreover, it is easy to verify that, for each rule $r \in P^-_{[R_{A'}]}$ the following holds: given an atom $\underline{a} \in body(r)$ such that $pred(\underline{a}) \in \mathcal{R}_{A'}$, (i) there exists an atom $\underline{b} \in body(r)$ such that the first position of $\underline{a}$ is in join with the first position of $\underline{b}$ and (ii) just nulls from N^- may occur in the first position of $\underline{b}$ in $chase(P)$. Since, by construction, $N' \cap N^- = \emptyset$, it thus follows that atoms from A' cannot be used to produce firing homomorphisms involving the rules of P^-. Then, the addition of P' to P^- does not affect the evaluation of q. This implies that $P \models q$ if and only if $P^- \models q$. Thus, we immediately get that $M(\varepsilon)$ terminates if and only if $P \models q$, where P is parsimonious.

Hence, the claim follows because we have reduced the halting problem to the problem of answering boolean conjunctive queries over parsimonious programs. The latter problem is thus undecidable. □

We next refine the concept of parsimony to guarantee also the decidability of conjunctive query answering. To this aim, we first introduce the notion of "nucleus".

Definition 4.2 (*Nucleus*). Consider a program P. A set $X \subseteq ochase(P)$ is a *nucleus* of $ochase(P)$ if, for each $Y \subseteq ochase(P)$ containing nulls, there exists a homomorphism from Y to $ochase(P)$ that maps at least one null of Y to a term of X.

This is a key notion which will be illustrated by means of the following example.

Example 4.3. Let us consider again program P of Example 2.1. Starting from $data(P) = \{person(john)\}$, Figure 2 illustrates the behaviour of the oblivious chase procedure over P. Every edge $(\underline{a}, \underline{b})$ with label r denotes that $\underline{b}$ is the atom generated by the fire of rule r via a homomorphism that maps $body(r)$ to $\underline{a}$.

We claim that the set X of atoms depicted in Figure 2 is a nucleus of $ochase(P)$. Consider, for instance, the set Y in the figure. Clearly, it can be mapped to Y' via a homomorphism h that associates null φ_2 to null φ_1. And we know that φ_1 is a term

of X. However, to prove formally that X is a nucleus of $ochase(P)$ one has to consider every possible subset of $ochase(P)$. To this end, let us decompose $ochase(P)$ as follows:

$$ochase(P) = X \cup \{person(\varphi_i) \mid i \in \mathbb{N}\} \cup \{parent(\varphi_i, \varphi_{i+1}) \mid i \in \mathbb{N}\}.$$

Consider now an arbitrary subset Y of $ochase(P)$. Let $K = \{i \mid \varphi_i \in terms(Y)\}$, and $k = \min(K)$. There always exists a homomorphism $h : terms(Y) \to terms(ochase(P))$ from Y to $ochase(P)$ such that, for each $\varphi_i \in terms(Y)$, $h(\varphi_i) = \varphi_{i-k+1}$. And, in particular, h maps null φ_k to φ_1, which we recall is a term of X. □

At this point, we are ready to define the notion of "strong parsimony".

Definition 4.4 (*Strong Parsimony*). A program P is *strongly parsimonious* if the following conditions —respectively called *uniformity* and *compactness*— are satisfied:

(1) $P \cup F$ is parsimonious, for each set F of facts; and
(2) $pchase(P)$ is a nucleus of $ochase(P)$.

This new class of programs is called *strongly parsimonious sets* (sps, for short). □

Clearly, according to uniformity, every strongly parsimonious program is trivially parsimonious, by choosing $F = \emptyset$. Regarding compactness, we refer again to Example 4.3. Actually, the nucleus X depicted in Figure 2 coincides with $pchase(P)$. This means that every subset of $ochase(P)$ containing nulls can be embedded into $ochase(P)$ itself via a homomorphism that maps some null into a term of $pchase(P)$.

The following proposition states an important property of strongly parsimonious programs, namely that they are closed under data expansions.

PROPOSITION 4.5. *Consider a strongly parsimonious program P. For each set F of facts, the program $P \cup F$ is still strongly parsimonious.*

PROOF. Let F be a set of facts, and $P' = P \cup F$. Program P' enjoys uniformity because, for each set F' of facts, $P' \cup F'$ is parsimonious. In fact, this is true because P enjoys uniformity and therefore $P \cup F \cup F'$ is parsimonious. By using a similar argument, it is possible to show that also $P \cup F$ enjoys compactness. □

To prove that conjunctive query answering over strongly parsimonious programs is decidable, we introduce a technique called *parsimonious-chase resumption*. Intuitively, assume that we have a program P such that $pchase(P) = \{p(c, \varphi), r(c, e), s(d, e)\}$, and that $ochase(P) = pchase(P) \cup \{s(\varphi, e)\}$. Consider the BCQ $q = \exists X \exists Y \exists Z\, p(X, Y), s(Y, Z)$. Clearly, $pchase(P) \not\models q$ even if $P \models q$. Let us both "promote" φ to a constant in Δ_C, and "resume" the parsimonious chase execution at step 2, in the same state in which it had stopped after returning the set I at step 6 in the else-branch. But, now, since φ can be considered as a constant, then there is no homomorphism from $\{s(\varphi, e)\}$ to $pchase(P)$. Thus, $s(\varphi, e)$ can be inferred by the algorithm and used to prove that q is true over P.

We call *freezing* the act of promoting a null from Δ_N to a novel constant in Δ_C. Also, given an instance I, we denote by $\lceil I \rfloor$ the set obtained from I after freezing all of its nulls. The following definition formalizes the notion of *parsimonious-chase resumption* after freezing actions.

Definition 4.6. Consider a program P. The output of the parsimonious chase after $k \geqslant 0$ resumptions is defined as follows:

(1) $pchase(P, 0) = data(P)$;
(2) $pchase(P, k) = pchase(dep(P) \cup \lceil pchase(k-1) \rfloor)$.

Clearly, it holds that $pchase(P, 1) = pchase(dep(P) \cup data(P)) = pchase(P)$. □

According to the chase procedure, the sequence $\{pchase(P,k)\}_{k\in\mathbb{N}}$ is monotonically increasing; the limit of this sequence is denoted by $pchase(P,\infty)$. The next proposition states that the proposed resumption technique is sound for query answering purposes.

PROPOSITION 4.7. *Given a program P, it holds that $pchase(P,\infty) \subseteq ochase(P)$.*

PROOF. This is proved by induction on the number of resumptions. The base case, for $k = 0$, is obviously true since $data(P) \subseteq ochase(P)$. Let us assume that the claim holds for some $k > 0$, namely that $pchase(P,k) \subseteq ochase(P)$. We now show that $pchase(P,k+1) \subseteq ochase(P)$ still holds. To this end, let $P' = P \cup \lceil pchase(P,k) \rfloor$. By Proposition 3.3, we know that $pchase(P') \subseteq ochase(P')$. However, since we are assuming that $pchase(P,k) \subseteq ochase(P)$, by the monotonicity and the uniqueness of the oblivious chase, we immediately have that $ochase(P') = ochase(P)$ and therefore that $pchase(P') \subseteq ochase(P)$. Since, $pchase(P,k) \supseteq data(P)$, by Definition 4.6, we have that $pchase(P') = pchase(dep(P) \cup \lceil pchase(P,k) \rfloor) = pchase(P,k+1)$. Hence, $pchase(P,k+1) \subseteq ochase(P)$. □

Actually, an infinite application of the proposed resumption technique ensures also completeness of query answering. In fact, even if $pchase(P,\infty)$ might not coincide with $ochase(P)$, it is still a universal model of P. However, since also $pchase(P,\infty)$ is generally infinite, its universality does not imply decidability. Hence, we skip its proof in favor of a stronger property, which is the main result of this section.

PROPOSITION 4.8. *Consider a strongly parsimonious program P and a Boolean conjunctive query q. If $P \models q$, then $pchase(P, |vars(q)|+1) \models q$.*

PROOF. Let $n = |vars(q)|$ denote the number of variables occurring in q. Let us consider a homomorphism h from $atoms(q)$ to $ochase(P)$. We claim that there exists a homomorphism h_n from $h(atoms(q))$ to $ochase(P)$ that associates all the nulls occurring in $h(atoms(q))$ with terms occurring in $pchase(P,n)$. Observe that the nulls occurring in $h(atoms(q))$ are at most n. To this end, we proceed by induction on the number of resumptions. The base case, for $k = 1$ resumptions, is clearly true since P is strongly parsimonious and therefore, by Definition 4.4, P enjoys compactness, namely there exists a homomorphism h_1 from $h(atoms(q))$ to $ochase(P)$ that associates at least a null occurring in $h(atoms(q))$ with a term occurring in $pchase(P) = pchase(P,1)$. Let us assume that the claim holds for some $k > 0$, namely that there exists a homomorphism h_k from $h(atoms(q))$ to $ochase(P)$ that associates at least k nulls occurring in $h(atoms(q))$ with terms occurring in $pchase(P,k)$. We now show that there exists a homomorphism h_{k+1} from $h(atoms(q))$ to $ochase(P)$ that associates at least $k+1$ nulls occurring in $h(atoms(q))$ with terms occurring in $pchase(P,k+1)$. Let $P_k = P \cup \lceil pchase(P,k) \rfloor$, and $q_k = h_k(h(q))$, where also the nulls occurring in $pchase(P,k)$ are considered frozen. By Proposition 4.5, we know that also P_k is strongly parsimonious. Hence, there is a homomorphism h' from $atoms(q_k)$ to $ochase(P_k) = ochase(P)$ that associates at least a null occurring in $atoms(q_k)$ with a term occurring in $pchase(P_k) = pchase(P,k+1)$. Therefore, $h_{k+1} = h' \circ h_k$ is a homomorphism from $h(atoms(q))$ to $ochase(P)$ that associates at least $k+1$ nulls occurring in $h(atoms(q))$ with terms occurring in $pchase(P,k+1)$.

To conclude the proof, it suffices to observe that the uniformity of P implies that also $P \cup \lceil pchase(P,n) \rfloor$ is parsimonious. In fact, since all the nulls occurring in $h_n(h(q))$ also occur in $pchase(P,n)$ then, after another resumption, we can generate all the atoms with constants and such nulls. Hence, $pchase(P,n+1) \models q$. □

By combining Proposition 4.7 (soundness) with Proposition 4.8 (completeness), the following result follows.

THEOREM 4.9. BCQEVAL *over strongly parsimonious programs is decidable.*

Input: A Datalog$^\exists$ program P, and a Boolean atomic query q.

1 **if** P is parsimonious **then goto** step 2;
else goto step 3;
2 **if** $pchase(P) \models q$ **then** *accept*;
else *reject*;
3 $P := P \cup efact(completion(P))$;
4 **goto** step 1;

Fig. 3. The AmazingEvaluation procedure.

PROOF. In order to prove the statement, we are going to show that, for each strongly parsimonious program P and for each Boolean conjunctive query q, it holds that $P \models q$ if and only if $pchase(P, |vars(q)| + 1) \models q$.

Let $n = |vars(q)|$ denote the number of variables occurring in q. Soundness, namely $pchase(P, n+1) \models q$ implies $P \models q$, directly follows by Proposition 4.7. For completeness, namely $P \models q$ implies $pchase(P, n + 1) \models q$, follows by Proposition 4.8. □

After showing that strong parsimony guarantees the decidability of conjunctive query answering, we have to close this section with a negative result: (strongly) parsimony is an undecidable property. This fact, although not too surprising, is formally stated in the following theorem and it is the main reason for the identification of sufficient syntactic conditions at the basis of a recognizable class of strongly parsimonious programs, which is the subject of the next section.

THEOREM 4.10. *Deciding whether a program is strongly parsimonious is not decidable. In particular, checking parsimony alone is already* ***coRE****-complete.*

PROOF. For the membership, we show that the complementary problem is in **RE**, namely we show how to semi-decide whether a program is not parsimonious. To this end, consider a program P. First, we compute $pchase(P)$. Second, we run the oblivious chase over $dep(P) \cup pchase(P)$ but, whenever there is a firing homomorphism h for a pair $\langle r, I \rangle$, we perform an extra-check before adding $fire(r, h)$ to I'. In particular, we check whether there is a homomorphism from $\{fire(r, h)\}$ to I'. In case P is not parsimonious, by Definition 3.4 and Proposition 3.5, the chase necessarily generates in a finite number of steps an atom $fire(r, h)$ that cannot be mapped homomorphically to I'. Hence, our procedure *accepts*.

For the hardness, we proceed by contradiction. More precisely, we assume that the problem of deciding whether a program is parsimonious is decidable, and we provide a decision procedure, called *AmazingEvaluation*, for the problem BCQEVAL over atomic queries, which is known to be **RE**-*complete* by Proposition 2.7. To this end, given an atom $\underline{a}$ with terms from $\Delta_C \cup \Delta_N$, we call $efact(\underline{a})$ the existential fact $\exists \mathbf{X}\, \underline{a}' \leftarrow$, where $\underline{a}'$ is obtained from $\underline{a}$ via a substitution that maps each null occurring in $\underline{a}$ into a different variable of $\mathbf{X}$. For example, if $\underline{a} = p(c_1, \varphi_1, c_2, \varphi_2, \varphi_1)$, then $efact(\underline{a}) = \exists X \exists Y\, p(c_1, X, c_2, Y, X) \leftarrow$. Moreover, given a program P which is not parsimonious, we denote by $completion(P)$ the earliest atom $\underline{a}$ that is generated by the oblivious chase during the construction of $ochase(P)$ such that there is no homomorphism from $\{\underline{a}\}$ to $pchase(P)$. The AmazingEvaluation procedure is shown in Figure 3. Clearly, the procedure terminates because steps 1 and 2 never fall in a loop since we are assuming that deciding whether a program is parsimonious is decidable, and since we know, by Theorem 3.6, that atomic query answering over parsimonious programs is decidable. Moreover, step 3 is performed finitely many times since P evolves to a parsimonious program and the output of the parsimonious chase is always finite, as guaranteed by Proposition 3.5. The algorithm is sound since every atom $completion(P)$

is an atom of $ochase(P)$, and because the addition of $efact(completion(P))$ to P leads to the generation of an atom which is less "vital" than $completion(P)$. Finally, the algorithm is complete since we add to P enough rules to satisfy all possible Boolean atomic queries that are true over P. Hence, AmazingEvaluation would be always a terminating procedure computing BCQEVAL over atomic queries, which is a clear contradiction. □

5. RECOGNIZABLE PARSIMONIOUS PROGRAMS

In this section, we work on the identification of sufficient syntactic conditions that guarantee strongly parsimony in order to result in a new class of programs, called shy.

5.1. Shy Programs

The first step towards the design of shy is to generalize the existing notion of *affected position* of a relational predicate with respect to a program. Such a notion has been proposed by Calì et al. [2008] to separate positions where the chase can introduce only constants from those where nulls might appear. However, this notion suffers from two major drawbacks: (i) it is not so informative to reveal whether two atoms of the chase cannot host the same null (in the same or different positions); and (ii) it may mark as affected positions that actually can never contain nulls.

Definition 5.1 (*Invaded Positions*). Consider a program P, an $\exists$-variable Y of P, a predicate p of arity k, and an index $i \in \{1, \ldots, k\}$. We say that position $p[i]$ is *invaded* by Y if there is a rule r of P such that $head(r) = p(t_1, \ldots, t_k)$ and either $t_i = Y$, or t_i is a $\forall$-variable which occurs in the body of r only in positions that are invaded by Y. □

According to the above definition, we have that if a position is invaded by some variable, then it is affected; however, the converse is not true. The following example compares the notion of invaded positions with that of affected positions.

Example 5.2. Consider the program P consisting of the following rules.

$$\begin{array}{lrcl} r_1: & person(john) & \leftarrow & \\ r_2: & \exists Y_2\, hasFather(X_2, Y_2) & \leftarrow & person(X_2) \\ r_3: & man(Y_3) & \leftarrow & hasFather(X_3, Y_3) \\ r_4: & \exists Y_4\, hasMother(X_4, Y_4) & \leftarrow & person(X_4) \\ r_5: & woman(Y_5) & \leftarrow & hasMother(X_5, Y_5) \\ r_6: & special(X_6) & \leftarrow & man(X_6), woman(X_6) \end{array}$$

The affected positions are: $hasFather[2]$, $man[1]$, $hasMother[2]$, $woman[1]$, and $special[1]$. However, $hasFather[2]$ and $man[1]$ are invaded by Y_2, while $hasMother[2]$ and $woman[1]$ are invaded by Y_4. Hence, according to Definition 5.1, we have that $special[1]$ is not invaded by any variable, and therefore it is not "really" affected. □

Our second step is to partition the variables occurring in a conjunction of atoms as attacked (by a certain variable) or protected. To this end, consider a conjunction $\varsigma_{[\mathbf{X}]}$ of atoms, and a variable $X \in \mathbf{X}$. We say that X is *attacked* in ς by a variable Y if X occurs in ς only in positions that are invaded by Y. Conversely, we say that X is *protected* in ς if it is attacked by no variable. By considering again Example 5.2, variable Y_3 is attacked in $body(r_3)$ by Y_2, while variable Y_5 is attacked in $body(r_5)$ by Y_4. Moreover, since $man[1]$ is invaded only by Y_2 and $woman[1]$ is invaded only by Y_4, we have that X_6 is protected in $body(r_6)$. We are now ready to define the new class of programs.

Definition 5.3 (*Shy Programs*). Consider a program P. A rule r of P is called *shy* with respect to P if the following conditions are both satisfied:

(1) If a variable X occurs in more than one body atom, then X is protected in $body(r)$;

(2) If two distinct $\forall$-variables are not protected in $body(r)$ but occur both in $head(r)$ and in two different body atoms, then they are not attacked by the same variable.

Moreover, program P is called *shy* if every rule of P is shy with respect to P. Finally, the class of shy programs is hereafter denoted by shy. $\square$

After noticing that a program is shy regardless its ground facts, we point out that program P of Example 5.1 is shy because rules r_1–r_5 have at most one body atom, and because variable X_6 is protected in $body(r_6)$. Intuitively, the key idea behind this class is as follows: *During the execution of the chase over a shy program, nulls propagated body-to-head do not meet each other to join.* Before proving that shy enjoys strong parsimony, we provide another example to better appreciate its syntactic properties.

Example 5.4. Consider the following rules

$$\begin{array}{lrcl} r_1: & \exists Y_1\, p(X_1, Y_1) & \leftarrow & s(X_1) \\ r_2: & r(X_2, Y_2) & \leftarrow & p(X_2, Y_2), u(Y_2) \\ r_3: & \exists Y_3\, u(Y_3) & \leftarrow & t(X_3) \end{array}$$

Let $P = \{r_1, r_2, r_3\}$. Clearly, r_1 and r_3 are shy rules w.r.t. P, since they are rules with one single body atom, which cannot violate any of the two shy conditions. Moreover, rule r_2 is also shy w.r.t. P as the positions $p[2]$ and $u[1]$ are invaded by disjoint sets of existential variables. Indeed, $p[2]$ is invaded by the existential variable Y_1 of the first rule, and $u[1]$ is invaded by the existential variable Y_3 of the third rule. Therefore, P is a shy program. Consider now the further three rules

$$\begin{array}{lrcl} r_4: & \exists Y_4\, p(Y_4, X_4) & \leftarrow & u(X_4) \\ r_5: & \exists Y_5\, p(X_5, Y_5) & \leftarrow & u(X_5) \\ r_6: & v(X_6) & \leftarrow & r(X_6, X_6) \end{array}$$

Let $P' = P \cup \{r_4\}$. It is easy to see that r_1, r_3 and r_4 are shy w.r.t. P'. However, r_2 is not shy w.r.t. P', as property (1) is not satisfied. Indeed, variable Y_2 occurring in two body atoms in $body(r_2)$ is not protected, as the position $p[2]$ and $u[1]$ (the only positions in which Y_2 occurs) are invaded by the same existential variable, namely Y_3. Therefore, P' is not shy. Let P'' be the program $P \cup \{r_5, r_6\}$. Again, r_1, r_3, r_5 and r_6 are trivially shy w.r.t. P''; and again r_2 is not shy w.r.t. P''. However, this time, r_2 is not shy because property (2) is not satisfied. Indeed, the universal variables X_2 and Y_2, occurring in two different body atoms and in $head(r_2)$, are not protected in $body(r_2)$, as the position $p[1]$ and $u[1]$ (in which X_2 and Y_2 occur, respectively) are attacked by the same variable Y_3. Therefore, P'' is not shy. $\square$

Essentially, during every possible chase step, condition (1) guarantees that each variable occurring in more than one body atom is always mapped into a constant. Although this is the key property behind shy, we now explain the role played by condition (2) and its importance. To this aim, we exploit again P'', as introduced in the previous example, and we reveal why this second condition, in a sense, turns into the first one. Indeed, rule r_6 bypasses the propagation of the same null in r_2 via different variables. However, one can observe that rules r_2 and r_6 imply the rule $r'_6 : v(X_6) \leftarrow p(X_6, Y_6), u(X_6)$, which of course does not satisfy condition (1). Actually, it is not difficult to see that the rules $dep(P)$ of every program P can be rewritten (independently from $data(P)$) into an en equivalent (w.r.t. query answering) set of rules that satisfy condition (1). As an example, consider the following rule r

$$\exists W_1\, t(X_1, Z_1, W_1) \leftarrow p(X_1, Y_1), r(Y_1, Z_1), u(Z_1, Y_1),$$

and assume that it belongs to a program P and that it is not shy w.r.t. P since it violates condition (1) only. Let us now construct P' as $P \setminus \{r\}$ plus the following two rules:

$$aux_\rho(X_1, Y_1, Y_1', Z_1, Z_1', Y_1'') \leftarrow p(X_1, Y_1), r(Y_1', Z_1), u(Z_1', Y_1'')$$
$$\exists W_2\, t(X_2, Z_2, W_2) \leftarrow aux_\rho(X_2, Y_2, Y_2, Z_2, Z_2, Y_2).$$

Both the new rules satisfy now condition (1) w.r.t. P'. Moreover, it is not difficult to see that, for every set of facts F and for every BCQ q, it holds that $F \cup P \models q$ if and only if $F \cup P' \models q$. However, since r does not satisfy condition (1), this immediately implies that the first new rule does not satisfy condition (2). Before concluding the section, we show that shy is a recognizable class.

THEOREM 5.5. *Checking whether a program P is shy is decidable. In particular, this check is doable in polynomial-time.*

PROOF. In order to check whether P belongs to shy, we need to construct the set of invading variables for each position. The procedure used to build these sets is monotone and stops as soon as a fixpoint is reached. Since the number n of variables that may invade a position is fixed by the occurrences of $\exists$-variables in P, such a fixpoint is always reached in finite time. Let k be the number of atoms occurring in P. To conclude the proof, it is enough to observe that there are at most $k \cdot arity(P)$ positions to be checked in P, and, each of which may be invaded by at most n different variables. □

5.2. Decidability

To show that BCQEVAL over shy is decidable we show that every shy program is indeed strongly parsimonious.

THEOREM 5.6. *shy $\subseteq$ sps.*

PROOF. To prove the statement, we show separately that a shy program P enjoys both uniformity and compactness.

(*Uniformity.*) Consider an arbitrary set F of facts. We have to show that there is a homomorphism from each singleton $\{\underline{b}\} \subseteq ochase(P \cup F)$ to $pchase(P \cup F)$. By Proposition 3.3, we know that $ochase(P \cup F)$ coincides with $ochase(dep(P \cup F) \cup pchase(P \cup F))$, which is also equivalent to $ochase(dep(P) \cup pchase(P \cup F))$. Let $P' = dep(P) \cup pchase(P \cup F)$. We prove uniformity by induction on the number of fires performed by the oblivious chase over P'. To this end, let O_i denote the subset of $ochase(P')$ containing only and all the atoms generated during the first i fires. We are going to show that, for each $i \geqslant 0$, there is a homomorphism from each singleton of O_i to $pchase(P \cup F)$.

The base case, for $i = 0$, is obviously true since $O_0 = data(P') = pchase(P \cup F)$. Assume the claim holds for some $i > 0$. We prove it at fire $i + 1$. Let $\langle r, h \rangle$ be the pair involved in the $(i + 1)$th fire. We now show that there is a homomorphism from $\{fire(r, h)\} = O_{i+1} \setminus O_i$ to $pchase(P \cup F)$. Let $body(r) = \{a_1, \ldots, a_k\}$. Since $h(body(r)) \subseteq O_i$, by hypothesis we have that, for each $j \in [1..k]$, there exists a homomorphism h_j from $\{h(a_j)\}$ to $pchase(P \cup F)$. However, since P is shy, by Definition 5.3, h may map a variable into a null only if such a variable does not appear in two different atoms. Therefore, $f = (h_1 \circ h) \circ \ldots \circ (h_k \circ h)$ is a homomorphism from $body(r)$ to $pchase(P \cup F)$. This means that the pair $\langle r, f \rangle$ is considered during the parsimonious chase. If $\langle r, f \rangle$ satisfies (resp., does not satisfy) the parsimonious fire condition, then $fire(r, f)$ is (resp., is not) added to $pchase(P \cup F)$. In both cases, we can say that there exists a homomorphism from $fire(r, f)$ to $pchase(P \cup F)$. It remains to show why necessarily there is a homomorphism g from $fire(r, h)$ to $fire(r, f)$. Let $h' \supseteq h$ such that $h'(head(r)) = fire(r, h)$ and $f' \supseteq f$ such that $f'(head(r)) = fire(r, f)$. We define g as follows: for each variable X of $head(r)$, $g(h'(X)) = f'(X)$. To prove that g is indeed a homomorphism we

have to guarantee that, for each pair of $\forall$-variables X and Y such that $h'(X) = h'(Y)$, we also have $f'(X) = f'(Y)$. If X and Y belong to the same atom in $body(r)$, say $\underline{a}_j$, this is guaranteed by the fact that f maps $\{a_j\}$ to $pchase(P \cup F)$. If X and Y belong to different atoms in $body(r)$, say $\underline{a}_j$, we rely on the second condition of Definition 5.3 which guarantees h cannot map X and Y to the same null.

(*Compactness.*) Consider a shy program P. We are going to restrict the chase relation $CR[P]$. For any set $S \subseteq ochase(P)$, we define $G[P,S] = (nodes(G[P,S]), arcs(G[P,S]))$ as the direct graph inductively defined as follows: $\underline{a} \in S$ implies $\underline{a} \in nodes(G[P,S])$; $\underline{a} \in nodes(G[P,S])$ and $(\underline{a},\underline{b}) \in CR[P]$ imply $\underline{b} \in nodes(G[P,S])$ and $(\underline{a},\underline{b}) \in arcs(G[P,S])$. Intuitively, by interpreting $CR[P]$ also as a graph, $G[P,S]$ is its maximal subgraph induced by the nodes of S plus those reachable from nodes of S.

Consider a set $Y_B \subseteq ochase(P)$ of atoms containing nulls. We show that there is a homomorphism from Y_B to $ochase(P)$ associating at least one of the nulls of Y_B with a term of $pchase(P)$. Let $N = \Delta_N \cap terms(Y_B)$, and $B = \{\underline{b}_1, \ldots, \underline{b}_n\}$ be the set of atoms of $ochase(P)$ where the nulls of N have been introduced for the first time; $j > i$ means that $\underline{b}_j$ is generated after $\underline{b}_i$. Also, let $\{N_1, \ldots, N_n\}$ be the partition of N where each N_i contains exactly the nulls introduced for the first time in $\underline{b}_i$.

From $G[P,B]$, we construct a set $A = \{\underline{a}_1, \ldots, \underline{a}_n\} \subseteq ochase(P)$, its associated graph $G[P,A]$, and a homomorphism $h = h|_N$ with the following properties:

– there is $h' \supseteq h$ such that $h'(\underline{b}_1) = \underline{a}_1 \in pchase(P)$;

– for each $\underline{b} \in nodes(G[P,B])$, there is $h' \supseteq h$ such that $h'(\underline{b}) \in nodes(G[P,A])$;

Therefore, since $Y_B \subseteq nodes(G[P,B])$, we have that $h(Y_B) \subseteq nodes(G[P,A])$. We proceed by induction on the sequence of fires generating (in parallel) the atoms of $G[P,B]$ and $G[P,A]$. For $x \in \{A,B\}$, at fire $i > 0$, $G_i[P,x]$ contains the atoms of $G[P,x]$ generated after the first i fires, and h_i is the partial homomorphism defined after the first i fires.

Base case: For $i = 1$, $G_1[P,B]$ contains only atom $\underline{b}_1$ and no arc. Since P is shy, then there is a homomorphism g from $\{\underline{b}_1\}$ to $pchase(P)$. Hence, let $h_1 = g|_{N_1}$, let $\underline{a}_1 = g(\underline{b}_1)$, and let $G_1[P,A]$ contain only atom $\underline{a}_1$.

Inductive hypothesis: After considering the first i fires we assume that for each $\underline{b} \in nodes(G_i[P,B])$, there is $h'_i \supseteq h_i$ such that $h'_i(\underline{b}) \in nodes(G_i[P,A])$;

Inductive step: Let $\underline{b} = fire(r, \tilde{h})$ be the atom of $G[P,B]$ produced at fire $i+1$. We will define a homomorphism g which determines the atom $g(\underline{b})$ that will be added to $G_i[P,A]$ to obtain $G_{i+1}[P,A]$. We distinguish two cases: (1) $\underline{b}$ has no ingoing arc in $G[P,B]$. Hence, $\underline{b} \in B$. Since P is shy, let g be any possible homomorphism from $\{\underline{b}\}$ to $pchase(P)$. (2) $\underline{b}$ has some ingoing arc in $G[P,B]$. Let $body(r) = \{\beta_1, \ldots, \beta_k, \beta_{k+1}, \ldots, \beta_\ell\}$ such that $\tilde{h}$ maps all and only the first $k > 0$ atoms to $nodes(G_i[P,B])$. For each $j \in [1..\ell]$, let $\tilde{h}_j = \tilde{h}|_{terms(\beta_j)}$. Also, for each $j \in [1..k]$, let f_j be the homomorphism that maps $\{\tilde{h}_j(\beta_j)\}$ to $nodes(G_i[P,A])$ such that $f_j = f_j|_{terms(\beta_j)}$. Since P is a shy program, then $e = (f_1 \circ \tilde{h}_1) \circ \ldots \circ (f_k \circ \tilde{h}_k) \circ \tilde{h}_{k+1} \circ \ldots \circ \tilde{h}_\ell$ is indeed a homomorphism that maps $body(r)$ to $ochase(P)$ and in particular $\{\beta_1, \ldots, \beta_k\}$ to $G_i[P,A]$. Hence, there is necessarily $e' \supseteq e$ such that $fire(r, e') \in ochase(P)$, and $g \supseteq f_1 \circ \ldots \circ f_k$ such that $g(\underline{b}) = fire(r, e')$. In both cases, after defining g we need to update our structures. Atom $g(\underline{b})$ is added to $G_{i+1}[P,A]$, possibly together with some arcs connecting it to atoms of $G_i[P,A]$. If $\underline{b} = \underline{b}_j$ for some $j \in [2..n]$, then $h_{i+1} = g|_{N_j} \circ h_i$ and and $\underline{a}_j = g(\underline{b}_j)$; $h_{i+1} = h_i$, otherwise. □

By combining the above result with Proposition 4.7 and Proposition 4.8 we obtain:

COROLLARY 5.7. *Problem* BCQEVAL *over shy is decidable. In particular, given a shy program P and a BCQ q, it holds that $P \models q$ if, and only if, $pchase(P, |vars(q)|+1) \models q$.*

The following example, after defining a shy program P, shows that P requires the computation of $pchase(P,3)$ to prove (after two resumptions) that a BCQ q containing two atoms and two variables is true over P.

Example 5.8. Let P denote the following shy program:

$$\begin{array}{rrcl} r_1: & admires(mary, john) & \leftarrow & \\ r_2: & hasFather(luke, tim) & \leftarrow & \\ r_3: & \exists Z_3\, mother(Z_3) & \leftarrow & hasFather(X_3, Y_3) \\ r_4: & \exists Y_4\, hasFather(X_4, Y_4) & \leftarrow & mother(X_4) \\ r_5: & admires(X_5, Z_5) & \leftarrow & mother(X_5), admires(Y_5, Z_5) \\ r_6: & admires(X_6, W_6) & \leftarrow & admires(X_6, Y_6), hasFather(Z_6, W_6). \end{array}$$

By applying three times the parsimonious chase over P we have:

$$\begin{array}{rcl} pchase(P,0) & = & data(P) = \{admires(mary, john),\ hasFather(luke, tim)\}, \\ pchase(P,1) & = & pchase(P,0) \cup \{mother(\varphi_1),\ admires(mary, tim)\}, \\ pchase(P,2) & = & pchase(P,1) \cup \{hasFather(\varphi_1, \varphi_2),\ admires(\varphi_1, john),\ admires(\varphi_1, tim)\}, \\ pchase(P,3) & = & pchase(P,2) \cup \{admires(mary, \varphi_2),\ admires(\varphi_1, \varphi_2)\}, \end{array}$$

Consider now the BCQ $q = \exists X \exists Y\, admires(X,Y), hasFather(X,Y)$. Clearly, q is true in $pchase(P,3)$ but that it is false in $pchase(P,2)$. □

5.3. Shy Programs vs. Strongly Parsimonious Programs

As stated by Theorem 5.6, shy is contained in sps. However, there are strongly parsimonious programs that are not shy. The following result holds.

THEOREM 5.9. *shy* $\neq$ *sps*.

PROOF. Let us consider the following set R of rules:

$$\begin{array}{rrcl} r_1: & \exists Y_1\, aux_1(X_1, Y_1) & \leftarrow & child(X_1) \\ r_2: & hasFather(X_2, Y_2) & \leftarrow & aux_1(X_2, Y_2) \\ r_3: & man(Y_3) & \leftarrow & aux_1(X_3, Y_3) \\ r_4: & child(X_4) & \leftarrow & hasFather(X_4, Y_4), man(Y_4) \end{array}$$

Positions $aux_1[2]$, $hasFather[2]$ and $man[1]$ are invaded by variable Y_1. Thus, differently from the other rules, r_4 is not shy since variable Y_4, occurring in two body atoms, is attacked by Y_1. Anyway, given a set of facts I, let us analyze the behaviour of the chase over the program $P = R \cup I$. In particular, let $I' \supseteq I$ be a subset of $ochase(P)$, we observe that each atom $child(c) \in I'$ would trigger the generation of atoms $aux(c, \varphi)$, $hasFather(c, \varphi)$ and $man(\varphi)$ via rules r_1, r_2 and r_3, respectively. Thus, any firing homomorphism h for the pair $\langle r_4, I' \rangle$ mapping Y_4 to a null value is such that the fire of r_4 via h would always produce an atom $fire(r_4, h) = child(c)$ that is already in I'. Hence, we can state that P is in sps because the contribution of these homomorphisms is redundant. □

By combining the above result with Theorem 5.6 we obtain that:

COROLLARY 5.10. *shy* $\subset$ *sps*.

6. COMPUTATIONAL COMPLEXITY

In this section, we study the complexity of BCQEVAL for atomic and conjunctive queries over three classes of programs: parsimonious, strongly parsimonious, and shy. In particular, we distinguish between data and combined complexity. Consider a program P. The data complexity of our problem is calculated by taking only $data(P)$ as input, while the query and the set rules(P) are considered fixed. The combined complexity is

Table I. Complexity of problem BCQEVAL.

	Data complexity for		*Combined complexity* for	
	atomic queries	*conjunctive queries*	*atomic queries*	*conjunctive queries*
shy	PTIME-complete LB: Thm. 6.4	PTIME-complete	EXPTIME-complete LB: Thm. 6.4	EXPTIME-complete UB: Thm. 6.3
sps	PTIME-complete	PTIME-complete UB: Thm. 6.2	EXPTIME-complete	in 2EXPTIME UB: Thm. 6.2
ps	PTIME-complete UB: Thm. 6.1	*Undecidable* Thm. 4.1	EXPTIME-complete UB: Thm. 6.1	*Undecidable* Thm. 4.1

the complexity calculated considering as input, together with $data(P)$, also the query and the set rules(P). A summary of our results is reported in Table I.

Each row corresponds to a class of programs, while each column corresponds to a different setting of the problem. In each cell of the table, we have indicated where to find the corresponding results; UB and LB stand for upper bound and lower bound, respectively. Note that missing references for the upper bounds are inherited from one of the closest lower-right cell in which an UB-reference is given. Conversely, missing references for the lower bounds are inherited from one of the closest upper-left cell in which an LB-reference is given.

To simplify the analysis, we first normalize our rules. Consider a program P. A rule of P is in *normal form* if it contains at most one existentially quantified variable which occurs at the last position of the head-atom. Then, we say that P is in normal form if each of its rules is in normal form. Let us now define a function $\mathcal{N}$ that transforms any rule r of P in a set of rules in normal form. More precisely, if r is already in normal form, then $\mathcal{N}(r) = \{r\}$; otherwise, assuming that $head(r) = \underline{a}$, $\mathbf{X} = \Delta_V \cap (terms(body(r)) \cap terms(head(r)))$, and $Z_1, \ldots, Z_m$ are the existentially quantified variables of r, let $\mathcal{N}(r)$ be the set of rules

$$\begin{array}{rrcl}
r_1: & \exists Z_1^r\, p_1^r(\mathbf{X}, Z_1^r) & \leftarrow & body(r) \\
r_2: & \exists Z_2^r\, p_2^r(\mathbf{X}, Z_1^r, Z_2^r) & \leftarrow & p_\sigma^1(\mathbf{X}, Z_1^r) \\
 & & \cdots & \\
r_m: & \exists Z_m^r\, p_m^r(\mathbf{X}, Z_1^r, \ldots, Z_m^r) & \leftarrow & p_{m-1}^r(\mathbf{X}, Z_1^r, \ldots, Z_{m-1}^r) \\
r: & & \underline{a} & \leftarrow\ p_m^r(\mathbf{X}, Z_1^r, \ldots, Z_m^r)
\end{array}$$

where, for each $i \in \{1, \ldots, m\}$, we enforce that p_i^r is an $(|\mathbf{X}| + i)$-ary auxiliary predicate not occurring in $pred(P)$. Let $\mathcal{N}(P) = \bigcup_{r \in P} \mathcal{N}(r)$. It is straightforward, and also well-known, that $\mathcal{N}(P)$ can be computed in polynomial time, and that P and $\mathcal{N}(P)$ are semantically equivalent. Namely, for each BCQ q, it holds that $P \models q$ if, and only if, $\mathcal{N}(P) \models q$. In particular, if we restrict $ochase(\mathcal{N}(P))$ to the predicates of P we precisely obtain $ochase(P)$.

6.1. Upper Bounds

We start this section by showing the following result:

THEOREM 6.1. *Problem* BCQEVAL *for atomic queries over parsimonious programs is in* EXPTIME *in combined complexity, and in* PTIME *in data complexity.*

PROOF. Consider a parsimonious program P. Let c be the number of distinct constants occurring in P, let ω denote $arity(P)$, and β be the maximum number of body atoms over all the rules of P. Clearly, each rule of P admits no more than $|pchase(P)|^\beta$

distinct firing homomorphisms. Hence, since in the proof of Proposition 3.5 we have shown that $|pchase(P)| \leq |pred(P)| \cdot (c+\omega)^{\omega}$, we obtain that all the distinct firing homomorphisms are no more than $|dep(P)| \cdot |pred(P)|^{\beta} \cdot (c+\omega)^{\omega \cdot \beta}$. And this is also the number of times the parsimonious fire condition has to be checked. Since such a check asks for a homomorphism from the head of a rule to $pchase(P)$, the overall number of checks is bounded by $|pchase(P)| \cdot |dep(P)| \cdot |pred(P)|^{\beta} \cdot (c+\omega)^{\omega \cdot \beta}$ which, in turn, is bounded by $|dep(P)| \cdot |pred(P)|^{\beta+1} \cdot (c+\omega)^{\omega(\beta+1)}$. □

We now generalize the above result by focusing on conjunctive queries.

THEOREM 6.2. *Problem* BCQEVAL *for CQs over strongly parsimonious programs is in* 2EXPTIME *in combined complexity, and in* PTIME *in data complexity.*

PROOF. Consider a parsimonious program P already in normal form, and a Boolean conjunctive query q. Let c be the number of distinct constants occurring in program P, β be the maximum number of body atoms over all the rules of P, $\omega = arity(P)$, $\alpha = \max\{|dep(P)|, |pred(P)|, \omega+1, \beta+2\}$, and $n = |vars(q)|$ denote the number of variables occurring in q. We claim that to compute $pchase(P, n+1)$ the number of steps performed by the parsimonious chase after n resumptions, call it $C(n)$, is at most

$$(n+1)(c+\alpha)^{\alpha^{2(n+1)}},$$

which is also an upper bound for the cardinality of $pchase(P, n+1)$. We proceed by induction on the number n of resumptions.

Base case: If $n = 0$, then we have to show that $C(0) \leqslant (c+\alpha)^{\alpha^2}$. In the proof of Theorem 6.1 we have shown that the $C(0) \leqslant |dep(P)| \cdot |pred(P)|^{\beta+1} \cdot (c+\omega)^{\omega(\beta+1)}$. Hence, $C(0) \leqslant \alpha^{\beta+2} \cdot (c+\alpha)^{\omega(\beta+1)} \leqslant (c+\alpha)^{\alpha} \cdot (c+\alpha)^{\omega(\alpha)} \leqslant (c+\alpha)^{(\omega+1)(\alpha)} \leqslant (c+\alpha)^{\alpha^2}$.

Inductive hypothesis: Assume that $C(n-1) \leqslant n(c+\alpha)^{\alpha^{2n}}$.

Inductive step: Since P is in normal form, $C(n-1)$ is also an upper bound for the number of distinct nulls introduced by the parsimonious chase during the first $n-1$ resumptions. Hence, $C(n) \leqslant |dep(P)| \cdot |pred(P)|^{\beta+1} \cdot (c+\omega+C(n-1))^{\omega(\beta+1)}$. Hence, $C(n) \leqslant \alpha^{\beta+2} \cdot (c+\alpha+n(c+\alpha)^{\alpha^{2n}})^{\omega(\beta+1)} \leqslant (c+\alpha)^{\alpha} \cdot ((c+\alpha)^{\alpha^{2n}} + n(c+\alpha)^{\alpha^{2n}})^{\omega\alpha}$. Therefore, $C(n) \leqslant (c+\alpha)^{\alpha} \cdot ((n+1)(c+\alpha)^{\alpha^{2n}})^{\omega\alpha} \leqslant ((n+1)(c+\alpha)^{\alpha^{2n}})^{\alpha} \cdot ((n+1)(c+\alpha)^{\alpha^{2n}})^{\omega\alpha}$. Thus, $C(n) \leqslant ((n+1)(c+\alpha)^{\alpha^{2n}})^{(\omega+1)\alpha} \leqslant ((n+1)(c+\alpha)^{\alpha^{2n}})^{\alpha^2} \leqslant (n+1)(c+\alpha)^{\alpha^{2(n+1)}}$.

Clearly, in data complexity, both n and α are considered fixed. Then, $C(n)$ is polynomial in c and therefore in $data(P)$. □

Finally, we show a better upper bound for the combined complexity over shy.

THEOREM 6.3. *Problem* BCQEVAL *for conjunctive queries over shy programs is in* EXPTIME *in combined complexity.*

PROOF. Consider a parsimonious program P, and a Boolean conjunctive query q. Figure 4 shows a resolution-based alternating algorithm, called Shy-BCQans, for solving BCQEVAL over the pair (P, q). Intuitively, the algorithm guesses a homomorphism from the query to $ochase(P)$ and applies resolution-based inference to reach $data(P)$, which involves both nondeterministic and universal moves. To guarantee correctnesses, universal branches have to remember some common knowledge. Let us now describe one by one the steps of Shy-BCQans. *Step_1.* Rewrites P in normal form. *Step_2.* Collects, in C, all the constants occurring in (P, q). *Step_3.* Guesses a homomorphism h from $atoms(q)$ to $ochase(P)$; w.l.o.g., we assume that the nulls in $h(atoms(q))$ belong to the set $\{\varphi_1, \ldots, \varphi_{|vars(q)|}\}$. *Step_4.* Denotes $h(atoms(q))$ by Q and the nulls of Q by $N \subseteq \{\varphi_1, \ldots, \varphi_{|vars(q)|}\}$; moreover, this step predisposes a number $k = arity(P) \cdot (|N|+2)$ of extra nulls $\{\star_1, \ldots, \star_k\}$, denoted by S, disjoint from N, which are used during the

Input: A shy program P, and a BCQ q.

1 $P := \mathcal{N}(P)$;
2 $C := (terms(P) \cup terms(q)) \cap \Delta_C$;
3 **guess** a homomorphism $h : vars(q) \to C \cup \{\varphi_1, \ldots, \varphi_{|vars(q)|}\}$;
4 $Q := h(atoms(q))$; $N := terms(Q) \setminus C$; $k := arity(P) \cdot (|N| + 2)$; $S := \{\star_1, \ldots, \star_k\}$;
5 **for** each null φ in N **do**
6 **guess** an atom $\underline{a}_\varphi \in base(C \cup N \cup S)$ such that φ is the rightmost term of $\underline{a}_\varphi$;
7 **universally select** every $\underline{b} \in Q$ and **do**
8 **if** $\underline{b} \in data(P)$ **then** *accept*;
9 **else**
10 **guess** a rule $r \in dep(P)$, and let t denote the rightmost term of $head(r)$;
11 **guess** a homomorphism $h : vars(r) \to C \cup N \cup S$;
12 **if** $(h(head(r)) \neq \underline{b}) \vee$
(t is an $\exists$-variable of $r \wedge h(t) \in N \wedge \underline{b} \neq \underline{a}_{h(t)}) \vee$
(h violates at least one of the two shyness conditions) **then** *reject*;
13 **else**
14 $Q := h(body(r))$;
15 **goto** step 7;

Fig. 4. The alternating algorithm Shy-BCQans.

algorithm (the reasons why such a number k is sufficient to guarantee correctness will be explained subsequently). *Steps_5-6.* For each null φ of N, guess the atom $\underline{a}_\varphi$ of $ochase(P)$ where such a null has been introduced for the first time (since P is in normal form, the null can be introduced in the rightmost position only). The maximum number of terms in these atoms are $|N| \cdot arity(P)$, which is also an upper bound for the number of distinct extra nulls from S they contain. Note that each $\underline{a}_\varphi$ is actually guessed from the set $base(C \cup N \cup S)$. Hence, $|S| \geqslant |N| \cdot arity(P)$. *Step_7.* It universally branches to prove "in parallel" that each atom $\underline{b}$ of Q really belongs to $ochase(P)$. *Step_8.* It checks whether $\underline{b}$ belongs to $data(P)$; if so, this branch accepts. *Step_9.* It states that a resolution-based action has to be performed since $\underline{b}$ does not belong to $data(P)$; this will be specified in the following steps. *Step_10.* It guesses a rule r of P, and denotes by t the rightmost term of $head(r)$. *Step_11.* It guesses a homomorphism h from $atoms(r)$ to $ochase(P)$. *Step_12.* It rejects whether one of the following occurs: (i) the guessed homomorphism does not map the head of r to $\underline{b}$; (ii) $h(head(r)) = \underline{b}$, t is an existentially quantified variable of r, $h(t) \in N$, but $\underline{b}$ is not the atom where $h(t)$ has been invented; (iii) h violates shyness; more precisely h maps a body variable occurring in two different atoms into a null of $N \cup S$, or h maps two different variables occurring in different atoms and also in the head to the same null. *Step_13.* It states that the resolution-based is correct, namely that h represent a firing homomorphism for r in $ochase(P)$, and therefore, a new universal move can be performed to prove separately the atoms of $h(body(r))$. If so, $h(head(r))$ may contain at most $arity(P)$ nulls from S, which in the worst case are completely different from the extra nulls already used in the set of atoms $\{\underline{a}_\varphi \mid \varphi \in N\}$. Hence, $|S| \geqslant |N| \cdot arity(P) + arity(P) = arity(P) \cdot (|N| + 1)$. Consider now the set $M = \Delta_N \cap (terms(body(t)) \setminus terms(head(r)))$ containing the variables of r occurring in the body of r but not in the head. Because of shyness, if h maps a variable $X \in M$ to a null, this means that X occurs in exactly one body atom. Moreover, if two variables $X, Y \in M$ occurring in different atoms are mapped to a null, then $h(X)$ and $h(Y)$ might even be the same, but this fact can be ignored in the universal move that will prove the atoms of $h(body(r))$. Therefore, the number of extra nulls that are strictly needed in $h(M)$ are again $arity(P)$. Finally, to guarantee correctness, $|S| \geqslant arity(P) \cdot (|N| + 1) + arity(P) = arity(P) \cdot (|N| + 2)$, which corresponds to the value

k computed in step 4. *Step_14.* Denotes $h(body(r))$ by Q. *Step_15.* Jumps to step 7 to branch universally.

Shy-BCQans runs in alternating polynomial space, since the space it uses at each step has to keep only the following knoledge: $\mathcal{N}(P)$, C, N, S, $\{\underline{a}_\varphi \mid \varphi \in N\}$, the current r, the current h, and the current Q. And each of these objects is of polynomial size with respect to the input pair (P, q). □

6.2. Lower Bounds

We now consider lower bounds:

THEOREM 6.4. *Problem* BCQEVAL *for atomic queries over parsimonious programs is* EXPTIME*-hard in combined complexity, and* PTIME*-hard in data complexity.*

PROOF. By Theorem 8.1, every Datalog program is also parsimonious. Hence, the considered problem inherits both lower-bounds from instance checking in Datalog [Dantsin et al. 2001]. □

7. IMPLEMENTATION AND OPTIMIZATIONS

We implemented the parsimonious-chase resumption tecnique introduced in Section 4 inside the well-known Answer Set Programming (ASP) system DLV [Leone et al. 2006]. This implementation is referred to as DLV$^\exists$ and it constitutes a powerful system for answering (unrestricted) conjunctive queries over strongly parsimonious programs. Following the DLV philosophy, it has been designed as an in-memory system. Basically, we extended the DLV parser and the rule safety check because of the presence of existentially quantified variables. Moreover, we evolved the DLV fixpoint computation so that it can be now resumed an arbitrary number of times. Finally, we developed an optimized homomorphism checker which exploits some of the standard routines implemented in DLV. To speed-up the computation, a number of optimization techniques has been introduced. In particular, we implemented a rewriting strategy to avoid the evaluation of the rules that are not relevant for answering the input query. Before parsing the input facts we find out the list of predicates that are significant for the query at hand and, consequently, we load only facts belonging to such predicates. To compute the number of times the parsimonious chase needs to be resumed, we execute an optimized version of the algorithm induced by Theorem 4.9. In the following subsections, after showing the overall architecture of the main components of the system, we give a detailed description of the implemented tecniques and optimizations. All data and encodings used in our evaluation, along with the Unix executable of the system, are publicly available.[5]

7.1. System Architecture

The overall architecture of the main components of DLV$^\exists$ and their interconnections is depicted in Figure 5. DLV$^\exists$ has been implemented as an in-memory system. Thus, it needs to load input data in memory before the computation can start. The input of the system consists of a pair $\langle P, q\rangle$ where q is an unrestricted conjunctive query (with possibly $\exists$-variables) to be executed over a program P. In particular, the computation starts from the Loader which firstly parses only $dep(P)$ and q. After that, $dep(P)$ and q are passed to the Rewriter which produces a set of rules R that is equivalent to $dep(P)$ with respect to the task of answering q — more details are given in the following subsections. At this point, the control goes back to the Loader which receives R from the Rewriter and parses $data(P)$. Actually, before parsing $data(P)$, the Loader singles out the

[5] See https://www.mat.unical.it/dlve/.

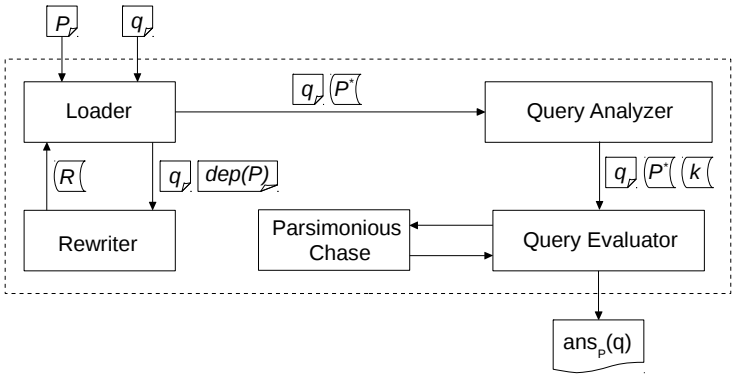

Fig. 5. DLV$^\exists$ architecture components

list of predicates which are relevant for answering q by recursively traversing rules in R (head-to-body) starting from the query predicates. This information is crucial because it is exploited to filter out, at loading time, all the facts belonging to predicates certainly irrelevant. Let us denote by D the loaded subset of $data(P)$. After the loading phase is completed, the QueryAnalyzer is invoked. This component takes as input the pair $\langle P^*, q\rangle$ where $P^* = R \cup D$ and returns the number k of parsimonious chase resumptions that are needed for answering q. Finally, the triple $\langle P^*, q, k\rangle$ is passed to the QueryEvaluator which computes $pchase(P^*, k)$ and returns $\mathsf{ans}_P(q)$. In the following subsections, a low-level description of the main components of the architecture is given.

Loading. The input of the system has to be organized as follows: files with the `.rul` extension for queries and rules and, for each predicate p, a file named `p.data` consisting of all the facts which belong to p. Note that, the system uses a special syntax for non-ASCII symbols. For example, the rule "$\exists X \exists Y\ p(Z, X, W, Y) \leftarrow s(Z, W), r(W, T)$" has to be given as "`#exists{X,Y} p(Z,X,W,Y) :- s(Z,W), r(W,T).`", the fact "$r(1, 2) \leftarrow$" as "`r(1,2).`" and the query "$\exists X \exists Y p(Z, X, W, Y)$" as "`#exists{X,Y} p(Z,X,W,Y)?`". The Loader firstly parses files with the `.rul` extension. Then, it calls the rewriting module and it gets an equivalent set of rules, R. Notice that, R is such that $\mathsf{ans}_{dep(P)\cup D}(q) = \mathsf{ans}_{R\cup D}(q)$ for any set of facts D. After the rewriting step is concluded, the loading module singles out the list of predicates which are relevant for answering q by recursively traversing the rules of R (head-to-body) starting from the query predicates. After that, only a subset of $data(P)$ containing facts which are relevant for q is loaded in memory; we call such a set D. Program $P^* = R \cup D$ is stored in two distinct data structures: the set of rules and the set of facts. Finally, the set of facts is indexed by predicate name. Notice that, the built-in predicate $\#const$ introduced in Section 5.3 belongs to the syntax accepted by the system. This predicate can be exploited by the user to enforce the protection of a specific variable in a rule body. In particular, given an instance I, if an atom $\#const(\mathbf{X})$ occurs in the body of a rule r, any homomorphism h from $body(r)$ to I such that $h = h|_{vars(body(r))}$ is considered by DLV$^\exists$ as a firing homomorphism for the pair $\langle r, I\rangle$ only if it maps all the variables in $\mathbf{X}$ to constant values.

Rewriting. The input program is subject to different rewriting steps. The first one performs the skolemization of $\exists$-variables in rule heads. The skolemization $sk(r)$ of a Datalog$^\exists$ rule r (as given in Section 2.2) is obtained by replacing each variable $Y \in \mathbf{Y}$

in $head(r)$ by a skolem term $f_Y(\mathbf{X}')$ where f_Y is a fresh *skolem function symbol* of arity $|\mathbf{X}'|$. Every rule in P with $\exists$-variables is skolemized in this way, and skolemized terms are interpreted as functional symbols [Calimeri et al. 2010] within DLV$^\exists$. After that, the rewriting techniques for positive logic programs with function symbols implemented in DLV are performed. For a detailed description of these algorithms see [Leone et al. 2006] and [Calimeri et al. 2010].

Query analysis. This task is performed to find out the number of parsimonious chase resumptions that are sufficient for answering q. Notice that, the upper bound given by Theorem 4.9 can be further reduced. The implemented procedure relies on the following idea. Let S be the set of substitutions mapping the free variables of q to constants of Δ_C. According to the given semantics (see Section 2.3), S is an upper bound of $\mathsf{ans}_P(q)$. Moreover, it is well-known that $\mathsf{ans}_P(q)$ can be even defined as follows: $\mathsf{ans}_P(q) = \{\sigma \in S \mid P \models \sigma(q)\}$. Since, by definition, the number k of $\exists$-variables of q is such that $k = |vars(\sigma(q))|$ for each $\sigma \in S$, by Theorem 4.9, we can infer that k parsimonious chase resumptions are sufficient to answer $\sigma(q)$ for each $\sigma \in S$ and, consequently, to compute $\mathsf{ans}_P(q)$.

Query evaluation. After the initialization process is terminated, the system computes $pchase(P^*, k)$. Since $\exists$-variables are skolemized, the rules are safe and can be evaluated in the usual bottom-up way. The semi-naive implementation of the DLV instantiator has been extended in order to simulate an execution of the parsimonious chase procedure. To this aim, a homomorphism verification is performed for each derivable atom. The implemented homomorphism check relies on the DLV indexing technique. In particular, the strategy adopted by DLV to materialize input data allows us to retrieve every facts of a given predicate with a certain term (or a tuple of terms) in a specific position. In this way, given a new head atom $\underline{a}$ the homomorphism check is performed by considering a limited set of atoms B such that for each atom $\underline{b} \in B$ (i) $pred(\underline{b}) = pred(\underline{a})$ and (ii) every constant in $terms(\underline{a}) \cap \Delta_C$ occurs both in $\underline{a}$ and $\underline{b}$ at the same positions. Moreover, in order to reproduce the resumption tecnique introduced in Section 4 the fixpoint computation has been extended so that it can be now resumed an arbitrary number of times. More precisely, it is reiterated until something has been derived in the last iteration and $pchase(P^*, k)$ has not been computed yet. To restart the fixpoint computation, every null (skolem term) derived previously is *frozen* (see Section 4) and considered as a standard constant, i.e. it is virtually moved to the set of constants of P. In our implementation, this is done by attaching a "level" to each skolem term, representing the fixpoint reiteration where it has been derived. This is important because homomorphism verification must consider as nulls only skolem terms produced in the current resumption-phase; while previously introduced skolem terms must be interpreted as constants.

7.2. Optimizations

The aim of the most relevant optimization techniques implemented in DLV$^\exists$ is to limit the loading of data that are redundant for the given query. This should automatically bring another advantage, i.e., reducing the space needed to materialize the output of the parsimonious chase. Thus, on the one hand, we refined the basic implementation of the rewriting algorithm described in Section 7.1. On the other hand, we improved the query analyzer in order to find out a better upper bound of the number of resumptions needed for the given query.

Rewriting. The DLV$^\exists$ computation is further optimized by "pushing-down" the bindings coming from possible query constants. To this end, the program is rewritten by a variant of the well-known magic-set optimization technique [Cumbo et al. 2004],

that we adapted to Datalog$^\exists$ by avoiding to propagate bindings through "attacked" argument-positions (since $\exists$-quantifiers generate "unknown" constants). The result is a smaller program, being equivalent to P for the given query, that can be evaluated more efficiently. This optimization provides a substantial advantage even in terms of input loading since the list of relevant predicates induced by the rewritten program is generally more restricted.

Query decomposition. As already stated, the number of parsimonious chase resumptions that is necessary and sufficient for answering q is not known. An upper bound is given by Theorem 4.9. In Section 7.1, we showed that such bound can be improved by ignoring the free variables of q. Anyway, this bound can be further optimized by also neglecting the protected variables of q. Indeed, since no substitution mapping a protected variable into a null can be part of $\mathsf{ans}_P(q)$, the intuition given in Section 7.1 can be easily extended to the set of protected variables. Furthermore, if we consider the query hypergraph with no free or protected variables we may get a number of independent components which could be evaluated separately. We show the above intuition with the aid of an example.

Example 7.1. Let us consider the following BCQ q over a program P:

$$q: \ \exists X \exists Y \exists Z \, r(X,Y), s(Y,Z)$$

where Y is protected. According to Theorem 4.9, we need $pchase(P,4)$ for answering q. Following the intuition discussed before, we can stop at $pchase(P,3)$ because Y is protected. Anyway, $P \models q$ if, and only if, there exists a constant c of P such that both $P \models \exists X \, r(X,c)$ and $P \models \exists Z \, s(c,Z)$ hold. Clearly, $\exists X \, r(X,c)$ and $\exists Z \, s(c,Z)$ can be evaluated separately against $pchase(P,2)$ since each of them contains exactly one variable. Finally, one could even observe that $\exists X \, r(X,c)$ and $\exists Z \, s(c,Z)$ are atomic, and therefore they can be evaluated correctly over $pchase(P)$. □

To generalize the above intuition, we recall the definition of query hypergraph.

Definition 7.2. Consider a CQ q. Let $\hat{q}$ denote the query obtained from q by replacing both free and protected variables with some constant of Δ_C. The *query hypergraph* of q is defined as $H_q = \langle V, E \rangle$ where $V = vars(\hat{q})$ and $E = \{vars(\underline{a}) \mid \underline{a} \in atoms(\hat{q})\}$.

Example 7.3. Consider again query q as in Example 7.1. According the previous definition, $\hat{q} = \exists X \exists Z \, r(X,\hat{c}), s(\hat{c},Z)$ and $H_q = \langle \{X,Z\}, \{\{X\},\{Z\}\} \rangle$. □

We are now ready to give the general statement underlying the optimization technique implemented in DLV$^\exists$ to find out the number of needed resumptions. Consider a query q over a program P. In case, each variable of $\hat{q}$ appears in a single atom only, then $\mathsf{ans}_P(q)$ can be computed by constructing $pchase(P)$. Conversely, $\mathsf{ans}_P(q)$ can be computed by constructing $pchase(P, k+1)$, where k is the cardinality of the largest connected component of H_q.

8. RELATED WORK

In this section we compare our newly defined classes with the main decidable ones from the literature. Moreover, we survey a number of tools designed and developed in the context of ontology-based data access.

8.1. Datalog$^\pm$ Languages

We overview the main QA-decidable subclasses of Datalog$^\exists$ defined in the literature. Then, we provide their precise taxonomy and the complexity of BCQEVAL over them.

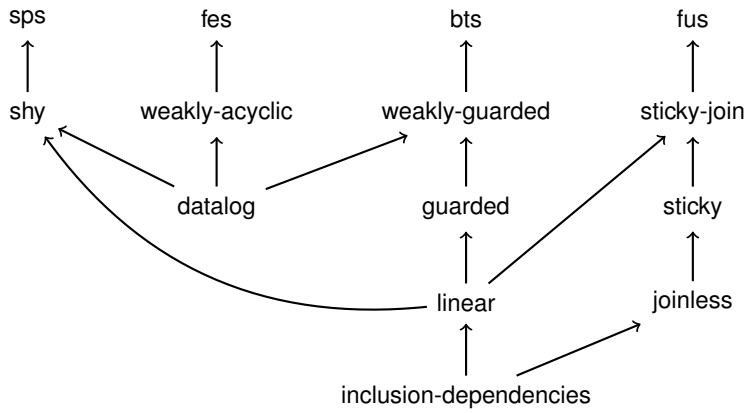

Fig. 6. Taxonomy of the basic Datalog$^{\pm}$ classes.

The best-known QA-decidable class is clearly datalog [Abiteboul et al. 1995], which collects all and only the programs with no existential quantifier. Notably, it admits a unique and yet finite (universal) model enabling efficient QA.

Three abstract QA-decidable classes have been singled out, namely, fes (finite expansion sets), bts (bounded/finite treewidth sets), and fus (finite unification sets) [Baget et al. 2009; Baget et al. 2010b]. Intuitively, the semantic properties behind these classes rely on a "forward-chaining inference that halts in finite time", a "forward-chaining inference that generates a tree-shaped structure", and a "backward-chaining inference that halts in finite time", respectively.

Syntactic subclasses of bts, of increasing complexity and expressivity, have been defined by Calì et al. [2008]. They are: (i) linear where at most one body atom is allowed in each rule; (ii) guarded where each rule needs at least one body atom that covers all $\forall$-variables; and (iii) weakly-guarded extending both guarded by allowing unaffected "unguarded" variables (see Section 5 for the meaning of unaffected). The first one generalizes the well known inclusion-dependencies class [Johnson and Klug 1984; Abiteboul et al. 1995], with no computational overhead; while only the last one is a superset of datalog, but at the price of a drastic increase in complexity. In general, to be complete with respect to QA, the chase ran on a program belonging to one of the latter two classes requires the generation of a very high number of isomorphic atoms, so that no (efficient) implementation has been realized yet.

More recently, the class called sticky has been defined by Calì et al. [2010a]. It enjoys very good complexity, encompasses inclusion-dependencies, but it does not capture datalog. Intuitively, if a program is sticky, then all the atoms that are inferred (by the chase) starting from a given join contain the term of this join. Several generalizations of stickiness have been defined by Calì et al. [2010b]. For example, sticky-join preserving the same complexity of sticky by also generalizing linear. Conversely, Gogacz and Marcinkowski [2013] introduced joinless, the subclass of sticky collecting all and only the programs where each body contains no repeated variable. Clearly, joinless captures inclusion-dependencies. Finally, joinless, sticky, and sticky-join are subclasses of fus.

Finally, in the context of data exchange, where a finite universal model is required, weakly-acyclic, a subclass of fes, has been introduced [Fagin et al. 2005b]. Intuitively, a program is weakly-acyclic if the presence of a null occurring in an inferred atom at a given position does not trigger the inference of an infinite number of atoms (with the same relational predicate) containing several nulls in the same position. This class

Table II. Complexity of problem BCQEVAL under the main sytactic classes.

	Data complexity for		*Combined complexity* for	
	atomic queries	*conjunctive queries*	*atomic queries*	*conjunctive queries*
weakly-guarded	EXPTIME-c	EXPTIME-c	2EXPTIME-c	2EXPTIME-c
guarded, weakly-acyclic	PTIME-c	PTIME-c	2EXPTIME-c	2EXPTIME-c
shy, datalog	PTIME-c	PTIME-c	EXPTIME-c	EXPTIME-c
sticky, sticky-join	in AC_0	in AC_0	EXPTIME-c	EXPTIME-c
linear	in AC_0	in AC_0	PSPACE-c	PSPACE-c

Note: For any complexity class C, to improve readability, C-c is used as a shorthand for C-complete.

both includes and has much higher complexity than datalog, but misses to capture even inclusion-dependencies. A number of extensions, techniques and criteria for checking chase termination have been recently proposed in this context [Deutsch et al. 2008; Marnette 2009; Meier et al. 2009; Greco et al. 2011].

Figure 6 provides a precise taxonomy of the considered classes; while Table II summarizes the complexity of BCQEVAL, by varying $\mathcal{C}$ among the main syntactic classes. In both diagrams, only datalog is intended to be $\exists$-free.

In the following, given two classes $\mathcal{C}_1$ and $\mathcal{C}_2$, we write $\mathcal{C}_1 \subseteq \mathcal{C}_2$ to indicate that $\mathcal{C}_1$ is a subclass of $\mathcal{C}_2$, $\mathcal{C}_1 \subset \mathcal{C}_2$ to indicate that $\mathcal{C}_1$ is a proper subclass of $\mathcal{C}_2$, and $\mathcal{C}_1 \,\|\, \mathcal{C}_2$ to indicate that $\mathcal{C}_1$ and $\mathcal{C}_2$ are uncomparable.

THEOREM 8.1. *For each pair $\mathcal{C}_1$ and $\mathcal{C}_2$ of classes represented in Figure 6, the following hold: (i) there is a direct path from $\mathcal{C}_1$ to $\mathcal{C}_2$ if, and only if, $\mathcal{C}_1 \subset \mathcal{C}_2$; (ii) $\mathcal{C}_1$ and $\mathcal{C}_2$ are not linked by any directed path if, and only if, $\mathcal{C}_1 \| \mathcal{C}_2$.*

PROOF. Relationships among known classes are pointed out by [Mugnier 2011; Gogacz and Marcinkowski 2013].

Regarding new containments, by Corollary 5.10, it immediately holds that shy $\subset$ sps. Moreover, it also holds that datalog $\subset$ shy since datalog $\subseteq$ shy because all the variables of every datalog program are protected, and since datalog $\neq$ shy because datalog does not admit existential quantifiers in rules heads. Finally, it holds that linear $\subset$ shy since linear $\subseteq$ shy because every linear program satisfies condition (i) in Definition 5.3, and because linear $\neq$ shy as shy admits rule-bodies with multiple atoms.

We now consider new uncomparability results. For each $\mathcal{C}_1 \in \{\text{shy}, \text{sps}\}$ and for each $\mathcal{C}_2 \in \{\text{weakly-acyclic}, \text{fes}\}$, to show that $\mathcal{C}_1 \| \mathcal{C}_2$ we prove that there is a program $P_1 \in$ shy such that $P_1 \notin \mathcal{C}_2$ and there is a program $P_2 \in$ weakly-acyclic such that $P_2 \notin \mathcal{C}_1$. As far as P_1 is concerned, it suffices to choose $P_1 \in$ linear since it is well-known that linear $\|$ fes and therefore $P_1 \notin \mathcal{C}_2$. Regarding P_2, we choose the program provided by Calì et al. [2010b] (Theorem 1) which proves that BCQEVAL for atomic queries over weakly-acyclic programs is 2EXPTIME-hard in combined complexity. Since BCQEVAL

for atomic queries over sps programs is in EXPTIME by Theorem 6.1, this immediately implies that $P_2 \notin \mathcal{C}_1$.

For each $\mathcal{C}_1 \in \{\mathsf{shy}, \mathsf{sps}\}$ and for each $\mathcal{C}_2 \in \{\mathsf{guarded}, \mathsf{weakly\text{-}guarded}, \mathsf{bts}\}$, to show that $\mathcal{C}_1 \| \mathcal{C}_2$ we prove that there is a program $P_1 \in \mathsf{shy}$ such that $P_1 \notin \mathcal{C}_2$ and there is a program $P_2 \in \mathsf{guarded}$ such that $P_2 \notin \mathcal{C}_1$. As far as P_1 is concerned, we choose the following program:

$$\begin{array}{lrcl} r_1: & set_1(a,a) & \leftarrow & \\ r_2: & set_2(b,b) & \leftarrow & \\ r_3: & \exists Z_3\, set_1(Y_3,Z_3) & \leftarrow & set_1(X_3,Y_3) \\ r_4: & \exists Z_4\, set_2(Y_4,Z_4) & \leftarrow & set_2(X_4,Y_4) \\ r_5: & graphK(U_5,V_5) & \leftarrow & set_1(U_5,X_5), set_2(V_5,Y_5) \end{array}$$

whose chase relation $CR[P]$ has no finite treewidth since it contains a complete bipartite graph $K_{n,n}$ of $2n$ vertices —the treewidth of which is n [Kloks 1994]— where n is not finite. Regarding P_2, we choose the program provided by Calì et al. [2013b] (Theorem 6.2) which proves that BCQEVAL for atomic queries over guarded programs is 2EXPTIME-hard in combined complexity. Since BCQEVAL for atomic queries over sps programs is in EXPTIME by Theorem 6.1, this immediately implies that $P_2 \notin \mathcal{C}_1$.

For each $\mathcal{C}_1 \in \{\mathsf{shy}, \mathsf{sps}\}$ and for each $\mathcal{C}_2 \in \{\mathsf{joinless}, \mathsf{sticky}, \mathsf{sticky\text{-}join}, \mathsf{fus}\}$, to show that $\mathcal{C}_1 \| \mathcal{C}_2$ we prove that there is a program $P_1 \in \mathsf{shy}$ such that $P_1 \notin \mathcal{C}_2$ and there is a program $P_2 \in \mathsf{joinless}$ such that $P_2 \notin \mathcal{C}_1$. As far as P_1 is concerned, it suffices to choose $P_1 \in \mathsf{datalog}$ since it is well-known that $\mathsf{datalog} \| \mathsf{fus}$ and therefore $P_1 \notin \mathcal{C}_2$. Regarding P_2, we choose the following program:

$$\begin{array}{lrcl} r_1: & p(0) & \leftarrow & \\ r_2: & s(1) & \leftarrow & \\ r_3: & t(X_3,Y_3) & \leftarrow & p(X_3), s(Y_3) \\ r_4: & \exists Y_4\, r(Y_4) & \leftarrow & p(X_4) \\ r_5: & p(X_5) & \leftarrow & r(X_5) \\ r_6: & s(X_6) & \leftarrow & r(X_6) \end{array}$$

together with the atomic Boolean query $q = \exists X\, t(X,X)$. By inspection, it is easy to see that P_2 is joinless. However, $P \models q$ since $ochase(P) = \{p(0), s(1), t(0,1)\} \cup \{r(\varphi_i), p(\varphi_i), s(\varphi_i), t(\varphi_i,\varphi_i)\}_{i \in \mathbb{N}}$ but we have that $pchase(P) \not\models q$ since $pchase(P) = \{p(0), s(1), t(0,1), r(\varphi_1)\}$. □

We care to notice that the program explicitly provided in the proof of Theorem 8.1 use the so called *concept product*. A natural and common example is given by the rule

$$biggerThan(X,Y) \leftarrow elephant(X), mouse(Y)$$

that is expressible in shy providing that $elephant$ and $mouse$ are disjoint concepts. However, such a relationship between concepts cannot be expressed in guarded, for example, and can be only simulated by a very expressive ontology language for which no tight worst-case complexity is known [Rudolph et al. 2008].

8.2. State-of-the-art Tools

Systems suitable for ontology-based data access can be classified in four groups: *query-rewriting*, *tableau*, *forward-chaining*, and *hybrid approaches*. However, as discussed in Section 9, some of them might not be complete when dealing with conjunctive queries.

Query rewriting. Systems belonging to this category are: QuOnto [Acciarri et al. 2005], Presto [Rosati and Almatelli 2010], Quest [Rodriguez-Muro and Calvanese 2011a], Mastro [Calvanese et al. 2011], OBDA [Rodriguez-Muro and Calvanese 2011b], Requiem [Pérez-Urbina et al. 2009], Rapid [Chortaras et al. 2011], Ontop [Calvanese

Table III. Systems vs. Benchmarks.

	DLV$^\exists$	PAGOdA	Clipper	Graal$_B$	Graal$_F$
LUBM	sps	OWL2/DL	$\mathcal{ELHI}$-Tr	–	weakly-acyclic
Deep	shy	–	–	linear	weakly-acyclic
Adolena	shy	–	DL-Lite$_R$	linear	–
Stock Exchange	shy	OWL2/DL	DL-Lite$_R$	linear	–
Vicodì	shy	$\mathcal{ELHO}^r_\bot$	DL-Lite$_R$	linear	weakly-acyclic
Path5	shy	$\mathcal{ELHO}^r_\bot$	DL-Lite$_R$	linear	weakly-acyclic

Note: Symbol "–" under a certain system indicates that it cannot (or, at least, there is no evidence that it can) be tested on the corresponding benchmark. Conversely, a class of ontologies under a certain system indicates that the corresponding benchmark falls in this class and that the system is sound and complete over this class. In particular, PAGOdA can be tested on LUBM and Stock Exchange since all the variables in the provided queries are "free", namely not existentially quantified.

et al. 2017], Graal [Baget et al. 2015] (see also Section 9), and Clipper [Eiter et al. 2012] (see also Section 9). They rewrite a given ontological query into an equivalent first-order (resp., Datalog) query against the underlying extensional database. After that, most of them delegate the answer computation to an RDBMS (resp., a Datalog reasoner). Every system in this category supports the standard first-order semantics for unrestricted CQs. The expressiveness of their languages is limited to AC_0 (and excludes, for instance, transitivity property or concept products) except for Clipper whose expressive power goes up to PTIME.

Tableau. Systems based on tableau calculi are: FaCT++ [Tsarkov and Horrocks 2006], RacerPro [Haarslev and Möller 2001], Pellet [Sirin et al. 2007], and HermiT [Motik et al. 2009]. They materialize all inferences at loading-time, implement very expressive description logics, but they do not support the standard first-order semantics for CQs [Glimm et al. 2008]. Actually, the Pellet system enables first-order CQs but only in the acyclic case.

Forward chaining. GraphDB [Bishop et al. 2011] (also known as OWLIM in erlier versions), and KAON2 [Hustadt et al. 2004] are based on forward-chaining.[6] Similar to tableau-based systems, they perform full-materialization and implement expressive DLs, but they still miss to support the standard first-order semantics for CQs [Glimm et al. 2008]. Graal [Baget et al. 2015] (see also Section 9) implements also a variant of the standard chase and it supports the standard first-order semantics for CQs. However, this algorithm might not stop if the restricted chase over the input program does not terminate.

Hybrid approaches. PAGOdA [Zhou et al. 2015] (see also Section 9) implements an hybrid approach to answer conjunctive queries over arbitrary OWL 2 DL ontologies.[7] This system exploits a forward-chaining reasoner to compute lower bound and upper bound answers to the input query. If lower and upper bounds match a sound and complete answer is returned. Otherwise, correctness of the tuples in the gap is checked by means of a fully-fledged OWL 2 reasoner. Such an approach showed good performances. However, if the aforementioned bounds do not match, then the standard first-order semantics for CQs is not supported.

9. EXPERIMENTS

In this section we report on some experiments we carried out to evaluate the efficiency and the effectiveness of DLV$^\exists$. In particular, we compared our system against a number of state-of-the-art systems for OBQA over a bunch of significant benchmark domains (see Table III). In the following, we first illustrate the criteria which led us to select the concurrent systems and the benchmark domains. Afterwards, we talk about the overall architecture of the machine where the experiments were run. Finally, we introduce domain by domain the attained results along with final considerations.

9.1. Systems

In the last years, a number of systems for OBQA have been introduced in literature. They mostly come from both the Database community and the Knowledge Representation world. However, many of them are still research prototypes and, in some cases, they are not suitable for a fair comparison with self-consistent OBQA systems as DLV$^\exists$. In order to set up a fair competition, we selected reasoners whose nature is compliant with the following prerequisites:

(1) *self-consistent execution*, the system should be able to run the input query over the input knowledge base without the aid of any external tool;
(2) *specific ability for OBQA*, the system should be designed expressly for answering conjunctive queries over ontologies and, in particular, the implemented algorithm should be proved to be sound and complete over a well defined class of ontologies.

The prototypes that enjoy the above criteria can be further subdivided in two distinct categories relying on the way of managing input data: systems running on top of a persistent storage layer, and systems that need to load input data in memory before the process can start. We decided to exclude reasoners based on persistent storage mechanisms because they usually rely on execution paradigms completely different from the one of DLV$^\exists$. For example, they expect the source instance to be preloaded into a persistent storage layer and, moreover, they execute a number of updating techniques employed in classical database management systems when data or rules change. On the contrary, in-memory systems like DLV$^\exists$ typically read their (possibly evolving) input from files at the beginning of the process. Such a practice may hardly affect overall performances of these systems, especially for huge volumes of input data. Even though, this paradigm does not need to be supported by any updating strategy because (possibly changed) input data are reloaded every time the process is executed. Consequently, the first category of systems is more suitable for querying massive volumes of data, whereas the second one can be profitably used when the context is subject to frequent changes. Anyway, comparing such different behaviors would have been unfair. Hence, having the above criteria in mind, we decided to compare DLV$^\exists$ against the following systems: Clipper [Eiter et al. 2012], Graal [Baget et al. 2015] and PAGOdA [Zhou et al. 2015]. In the following lines, a brief description of each system is provided. The other reasoners that are listed in Section 8.2 have been excluded from our comparison because they violate at least one of the above preconditions.

Clipper. This system has been designed for answering conjunctive queries over Horn-SHIQ [Hustadt et al. 2005] ontologies. Such approach relies on a query rewriting tecnique that transforms a given Horn-SHIQ ontology and a CQ into an equivalent Datalog program that can be evaluated over any input set of data. Clipper is written

[6] Actually, KAON2 first translates the ontology to a disjunctive Datalog program, on which forward inference is then performed.
[7] See https://www.w3.org/TR/owl2-overview/.

in Java and it accepts as input arbitrary Horn-SHIQ ontologies and datasets in the RDF/XML format and CQs in SPARQL[8]. The initial step of the implemented algorithm is devoted to check whether the input ontology is Horn-SHIQ. In affirmative case, the aforementioned rewriting step is performed and an equivalent Datalog program P is produced. Afterwards, the input dataset D is loaded in memory and the evaluation of P over D is demanded to a Datalog reasoner between DLV and Clingo [Leuschel and Schrijvers 2014].

Graal. Graal is an open-source toolkit mainly developed for computing certain answers to queries under dependencies, that is composed by the following tools. Basically, there is a common input manager that allows to choose between an in-memory execution and a persistent storage mapping. Moreover, on top of the above layer, two different reasoning algorithms have been implemented. The former relies on a query rewriting technique using *piece-unifiers* [Mugnier 2011] as resolution operator. The latter is given by a *saturation procedure* that can be seen as a variant of the standard chase: it applies rules to the data in breadth-first, forward-chaining manner. In our experimental evaluation, the in-memory versions of the above approaches are both considered and are referred to as Graal$_B$ and Graal$_F$, respectively. Regarding the task of conjunctive query answering, Graal$_B$ is sound and complete over first-order rewritable ontologies; Graal$_F$ is always sound and complete but it terminates (for a given ontological query in input) only if the restricted chase would stop after finitely many steps. Notice that, the system accepts as input ontologies, data sets and queries in a proprietary format, called *dlgp*, that has been defined to model Datalog$^{\pm}$ rules with predicate namespaces. Anyway, Graal comes with a tool converting OWL ontologies and SPARQL queries in the *dlgp* format.

PAGOdA. PAGOdA is a highly optimized system for answering conjunctive queries over arbitrary OWL 2 DL ontologies. This implementation relies on a hybrid approach that combines a Datalog reasoner (currently RDFox [Nenov et al. 2015]) with a fully-fledged OWL 2 reasoner (currently HermiT [Glimm et al. 2014]) to provide scalable performance. In particular, PAGOdA uses the Datalog reasoner to compute lower bound (sound but possibly incomplete) and upper bound (complete but possibly unsound) answers to the input query. If lower bound and upper bound coincide, it returns a sound and complete answer. Otherwise, it resorts to the OWL 2 reasoner to check correctness of the tuples in the gap between lower and upper bounds. In order to lighten the computational cost of this check, a number of optimization techniques has been implemented aiming at reducing the workload on the OWL 2 reasoner. This system supports the standard first-order semantics for CQs only if the input ontology is in $\mathcal{ELHO}^r_\bot$ (or in OWL 2 RL). Otherwise, sound and complete answers are returned only if lower and upper bounds coincide. If such bounds do not match, queries are evaluated under the ground semantics. However, it is worth pointing out that if the input ontology is not in $\mathcal{ELHO}^r_\bot$, correctness of the expected answers cannot be verified a priori but it can be checked only at running time because it may depend on the input data (in addition to the input ontology and query). To conclude, notice that PAGOdA accepts as input arbitrary OWL 2 DL ontologies, datasets in Turtle[9] format and CQs in SPARQL.

9.2. Benchmarks

Since the ontological query answering in the context of Datalog programs with existential rules is a relatively recent area of research, there are no well-established benchmarks for this task. We therefore imported a number of scenarios from some related

[8] See https://www.w3.org/TR/rdf-sparql-query/.
[9] See https://www.w3.org/TR/turtle/.

Table IV. Query analysis in terms of number of atoms ($\#a$), number of join variables ($\#j$), number of free variables ($\#f$) and number of bounded variables ($\#b$). Notice that, query identifiers are acronyms whose left-hand sides indicate the benchmark IDs.

query	$\#a$	$\#j$	$\#f$	$\#b$	*query*	$\#a$	$\#j$	$\#f$	$\#b$	*query*	$\#a$	$\#j$	$\#f$	$\#b$
deep-100					**deep-200**					**lubm-10** and **lubm-20**				
D1:01	2	1	1	6	D2:01	2	1	1	6	LB:01	2	1	1	0
D1:02	2	1	1	6	D2:02	2	1	1	6	LB:02	6	3	3	0
D1:03	2	1	1	6	D2:03	2	1	1	6	LB:03	2	1	1	0
D1:04	2	1	1	6	D2:04	2	1	1	6	LB:04	5	1	4	0
D1:05	2	1	1	6	D2:05	2	1	1	6	LB:05	2	1	1	0
D1:06	3	1	1	9	D2:06	3	1	1	9	LB:06	1	0	1	0
D1:07	3	1	1	6	D2:07	3	1	1	9	LB:07	4	2	2	0
D1:08	3	1	1	6	D2:08	3	1	1	9	LB:08	5	2	3	0
D1:09	3	1	1	6	D2:09	3	1	1	9	LB:09	6	3	3	0
D1:10	3	1	1	6	D2:10	3	1	1	9	LB:10	2	1	1	0
D1:11	5	1	1	15	D2:11	5	1	1	15	LB:11	2	1	1	0
D1:12	5	1	1	15	D2:12	5	1	3	13	LB:12	4	2	2	0
D1:13	5	2	3	13	D2:13	5	1	2	14	LB:13	2	1	1	0
D1:14	5	1	1	15	D2:14	5	1	1	15	LB:14	1	0	1	0
D1:15	5	1	1	15	D2:15	5	1	3	13	**path5**				
D1:16	8	1	1	24	D2:16	8	1	1	24	P5:01	1	0	1	1
D1:17	8	1	1	24	D2:17	8	1	2	23	P5:02	2	1	1	2
D1:18	8	1	2	23	D2:18	8	1	4	21	P5:03	3	2	1	3
D1:19	8	1	1	24	D2:19	8	1	3	2	P5:04	4	3	1	4
D1:20	8	2	5	20	D2:20	8	2	3	22	P5:05	5	4	1	5
adolena					**stock-exchange**					**vicodì**				
AD:01	2	1	1	1	SE:01	1	0	1	0	VD:01	1	0	1	0
AD:02	3	2	1	1	SE:02	3	2	2	0	VD:02	3	1	2	1
AD:03	5	3	1	2	SE:03	5	3	3	0	VD:03	3	2	2	0
AD:04	3	2	1	1	SE:04	5	3	3	0	VD:04	3	2	2	0
AD:05	5	3	1	2	SE:05	7	4	4	0	VD:05	7	4	1	3

areas. The benchmark suite proposed in this paper consists of six domains (*LUBM, Deep, Adolena, Stock Exchange, Vicodì*, and *Path5*), each comprising at least an ontology and a set of conjunctive queries. Some of these domains are also endowed with input data sets (or generators). However, for those scenarios that are missing input instances, we used SyGENiA (Synthetic GENerator of instance Axioms) [Cuenca Grau et al. 2012] to generate meaningful sets of data.[10]

As discussed in Section 9.1, the tested systems take a wide range of input formats. Whenever possible, the translation between various formats has been carried out with the aid of automated tools. Rdf2Rdf[11] has been used to produce the Turtle format, OWL2Dlgp[12] for the dlgp format and, finally, Aspide [Febbraro et al. 2013] for the DLV$^{\exists}$ format. Notice that, not all reasoners are able to perform all tests: for each benchmark B and for each systems S, we decided to run S over B only if it is known a priori that S provides sound and complete answers to the benchmark queries over any sets of data. However, in the following lines we give some details about the benchmarks and, for each of them we indicate the list of the involved systems. Details on input queries for

[10] SyGENiA is prototypical tool for the automatic generation of ontology parts for the purpose of testing and evaluating Semantic Web reasoners. Given an ontology and a query, SyGENiA is able to produce a set of test data that can be profitably used for testing OBQA systems (the data generated are not random).
[11] See http://www.l3s.de/~minack/rdf2rdf/.
[12] See https://graphik-team.github.io/graal/owl2dlgp.

Table V. Running times (sec) for LUBM queries.

	lubm-10			lubm-20		
	DLV$^{\exists}$	*PAGOdA*	*Clipper*	*DLV*$^{\exists}$	*PAGOdA*	*Clipper*
LB:01	1.12	10.97	46.64	2.41	21.43	193.26
LB:02	1.40	10.97	45.97	3.01	21.66	187.79
LB:03	0.99	11.48	44.47	2.11	21.65	194.87
LB:04	5.16	11.09	47.22	10.94	21.51	201.29
LB:05	4.76	10.97	49.32	10.03	21.64	196.50
LB:06	8.55	11.62	50.04	18.59	22.06	212.50
LB:07	4.41	11.54	48.98	9.41	21.90	194.75
LB:08	5.77	12.26	46.86	10.97	21.87	196.48
LB:09	9.18	11.09	49.02	19.54	22.48	189.94
LB:10	6.22	11.11	44.93	12.23	23.73	193.44
LB:11	0.04	10.80	46.01	0.07	21.71	194.01
LB:12	4.23	11.90	45.12	8.86	21.65	203.39
LB:13	4.36	12.51	44.39	9.24	21.62	197.52
LB:14	0.40	11.28	48.23	0.75	21.95	204.37
#	14	14	14	14	14	14
avg	2.35	11.39	46.91	4.78	21.91	197.06

Note: The times of Graal$_F$ are not shown since the system timed out on every benchmark query. On the contrary, Graal$_B$ has not been involved because the LUBM ontology is not first-order rewritable.

each benchmark are given in Table IV in terms of number of atoms, number of join variables, number of free variables and number of bounded variables. The table shows that, overall, queries involved by the various benchmarks span over heterogeneous sets of proposed parameters; as a consequence, testing scenarios should be sufficiently variegate to assess different facets of tested systems.

LUBM. The Lehigh University Benchmark (LUBM) has been specifically developed to facilitate the evaluation of Semantic Web reasoners in a standard and systematic way. In fact, the benchmark is intended to evaluate the performance of those reasoners with respect to extensional queries over large data sets that commit to a single realistic ontology. It consists of a university domain OWL ontology (expressible in Datalog$^{\exists}$) with customizable and repeatable synthetic data and a set of 14 input SPARQL queries. The LUBM ontology provides a wide range of axioms that are aimed at testing different capabilities of the reasoning systems. The ontology describes (among others) universities, departments, students, professors and relationships among them. Data generation has been carried out by the LUBM data generator tool whose main generation parameter is the number of universities to consider. In order to perform scalability tests, we produced two data sets of increasing sizes: *lubm-10* and *lubm-20*, where the '10' and the '20' in these acronyms indicate the number of universities used as parameter to generate the data. The number of statements (both individuals and assertions) stored in the data sets vary from about 1M for lubm-10 to about 3M for lubm-20. In this comparison, we decided to involve the following systems: DLV$^{\exists}$, PAGOdA, Clipper and Graal$_F$. Graal$_B$ has been excluded because LUBM is equipped with an axiom (transitivity) that is not first-order rewritable. Whereas, PAGOdA has not been excluded, despite LUBM is not a $\mathcal{ELHO}^{r}_{\bot}$ ontology, because every query of the benchmark has only free variables (and sometimes constant terms); hence, in this case the ground semantics coincides with the standard first-order one.

Deep. The Deep benchmark has been recently developed by Benedikt et al. [2017] as a pure stress test for chase-based systems. Actually, Deep is part of a benchmark

Table VI. Running times (sec) for DEEP queries.

	deep-100				deep-200		
	$DLV^{\exists}$	$Graal_B$	$Graal_F$		$DLV^{\exists}$	$Graal_B$	$Graal_F$
D1:01	0.36	2.17	13.61	D2:01	0.37	3.01	↗
D1:02	0.54	4.55	12.08	D2:02	0.36	2.28	↗
D1:03	0.79	94.95	11.08	D2:03	0.36	1.95	↗
D1:04	0.36	3.01	13.40	D2:04	0.36	2.56	↗
D1:05	0.36	1.63	13.29	D2:05	1.01	18.10	↗
D1:06	0.38	10.92	12.72	D2:06	0.36	2.36	↗
D1:07	0.36	8.67	13.64	D2:07	0.96	124.74	↗
D1:08	0.35	3.87	12.55	D2:08	0.44	103.04	↗
D1:09	0.55	57.94	13.13	D2:09	0.36	3.82	↗
D1:10	0.35	6.51	13.80	D2:10	0.38	4.81	↗
D1:11	0.37	↗	13.05	D2:11	1.04	↗	↗
D1:12	0.39	↗	12.33	D2:12	0.40	↗	↗
D1:13	0.36	↗	12.75	D2:13	3.03	↗	↗
D1:14	0.50	↗	13.89	D2:14	0.40	↗	↗
D1:15	0.41	↗	13.49	D2:15	1.05	↗	↗
D1:16	0.43	↗	13.19	D2:16	1.27	↗	↗
D1:17	0.40	↗	12.37	D2:17	0.42	↗	↗
D1:18	0.43	↗	13.14	D2:18	1.03	↗	↗
D1:19	0.43	↗	12.36	D2:19	0.40	↗	↗
D1:20	0.52	↗	13.27	D2:20	3.32	↗	↗
#	20	10	20	#	20	10	0
avg	0.42	47.73	12.94	avg	0.64	46.29	↗

Note: If used in place of a running time, symbol "↗" indicates that the running time exceeds the time limit (300 seconds). Otherwise, it means that the average running time has not been computed since the corresponding system timed out on every benchmark query. To compute geometric means, we used the time limit (300 secs) in case of timeout. Systems Clipper and PAGOdA have not been considered since Deep has been designed as a Datalog$^{\exists}$ ontology and, as such, it is composed by a number of predicates and rules that cannot be translated in OWL.

consisting of five different scenarios, also including LUBM. (The remaining three scenarios of the original benchmark turned out to be unsuitable in our setting since either they do not provide any input query or their ontologies are not shy.) Deep comes with three different domains, referred to as deep-100, deep-200 and deep-300, each provided with a rule set of increasing size (1100, 1200 and 1300 rules, respectively), a set of facts (composed by just one fact per predicate) and a set of twenty conjunctive queries. Notice that, every ontology of the benchmark is linear and weakly-acyclic. Thus, we decided to involve in this run the following systems: $DLV^{\exists}$, $Graal_F$ and $Graal_B$. In this case, we did not consider Clipper and PAGOdA because Deep has been designed as a Datalog$^{\exists}$ ontology and, as such, it is composed by a number of predicates and rules that cannot be translated in OWL. Although the limited size of the input sets of facts, the complexity of the rules is such that the restricted chase produces for example over 500M atoms on the largest deep-300 scenario. Notice that, we considered in our evaluation only deep-100 and deep-200 because deep-300 is still unavailable.

Adolena, Stock Exchange, Vicodì and Path5. These domains have been derived from a well-established benchmark for DL-based query rewriting systems used, e.g., in [Pérez-Urbina et al. 2010]. These ontologies are expressed in the description logic *DL-Lite*$_R$ [Calvanese et al. 2007], each provided with a different set of five SPARQL queries. Notice that, each ontology of the benchmark is linear, whereas only Vicodì and

Table VII. Running times (sec) over Adolena, Stock Exchange, Vicodì and Path5 ontologies.

	stock-exchange			
	DLV$^{\exists}$	*PAGOdA*	*Clipper*	*Graal*$_B$
SE:01	0.31	6.21	3.29	5.21
SE:02	1.96	7.66	8.02	↗
SE:03	12.16	18.61	↗	↗
SE:04	4.05	10.86	29.67	↗
SE:05	61.93	↗	↗	↗
#	5	4	3	1
avg	4.50	19.59	37.11	133.38

	vicodì				
	DLV$^{\exists}$	*PAGOdA*	*Clipper*	*Graal*$_B$	*Graal*$_F$
VD:01	0.25	4.53	4.67	5.98	27.07
VD:02	0.59	4.91	5.91	↗	↗
VD:03	0.44	5.92	6.23	↗	28.43
VD:04	0.41	4.29	5.10	↗	26.56
VD:05	0.37	5.53	4.42	↗	109.62
#	5	5	5	1	4
avg	0.40	5.00	5.22	137.10	58.28

	adolena		
	DLV$^{\exists}$	*Clipper*	*Graal*$_B$
AD:01	0.95	5.71	5.75
AD:02	1.27	4.71	49.60
AD:03	1.26	5.03	↗
AD:04	1.12	4.25	↗
AD:05	3.03	5.01	↗
#	5	5	2
avg	1.39	4.92	94.92

	path5				
	DLV$^{\exists}$	*PAGOdA*	*Clipper*	*Graal*$_B$	*Graal*$_F$
P5:01	0.36	5.93	2.94	5.53	14.91
P5:02	0.89	6.92	4.40	262.43	10.29
P5:03	2.50	10.67	7.12	↗	35.13
P5:04	20.75	142.23	50.49	↗	↗
P5:05	↗	↗	↗	↗	↗
#	4	4	4	2	3
avg	5.48	28.47	16.94	131.39	54.60

Note: Symbol "↗" indicates that the running time exceeds the time limit (300 seconds). To compute geometric means, we used the time limit (300 secs) in case of timeout. PAGOdA has not been involved over Adolena because its ontology is not $\mathcal{ELHO}^r_\bot$ and its queries are not bound. Graal$_F$ has not been considered over Adolena and Stock Exchange because their ontologies are not weakly-acyclic and there is no evidence that the system can stop after finitely many steps.

Path5 are weakly-acyclic and $\mathcal{ELHO}^r_\bot$. Hence, we run DLV$^{\exists}$, Clipper and Graal$_B$ on every domain; PAGOdA on Stock Exchange (its queries have no bound variables), Vicodì and Path5; Graal$_F$ on Vicodì and Path5. This benchmark is not provided with an input data generator. We therefore used Sygenia in order to produce a meaningful set of data for each domain. In particular, the generated instances are composed by 30k assertions and 5k different individuals. In order to get the proper formats, we used the aforementioned conversion tools. In the following lines, we give a brief description of each ontology:

— *Adolena* (Abilities and Disabilities OntoLogy for ENhancing Accessibility) has been developed for the South African National Accessibility Portal. It describes abilities, disabilities and devices.
— *Stock Exchange* is an ontology of the domain of financial institution within the EU.
— *Vicodì* is an ontology of European history, developed within the Vicodì project.[13]
— *Path5* is a synthetic ontology encoding graph structures, and used to generate an exponential blow-up of the size of the rewritten queries.

9.3. Results

The machine used for testing is a MSI GE60-2PE with 4 dual-core Intel i7-4710HQ processors at 2.50GHz (8 cores in total), running Linux Ubuntu v14.04 x86-64. The machine is equipped with 16GB of RAM. Resource usage was limited to 300 seconds and 8GB of RAM in each execution.

[13] See http://www.vicodi.org.

Table VIII. Overall results on percentage of solved queries and average running time.

		$DLV^{\exists}$	*PAGOdA*	*Clipper*	$Graal_B$	$Graal_F$
lubm-10	*# solved queries (%)*	100%	100%	100%	–	0%
	average time	2.35	11.39	46.91	–	↗
lubm-20	*# solved queries (%)*	100%	100%	100%	–	0%
	average time	4.78	21.91	197.06	–	↗
deep-100	*# solved queries (%)*	100%	–	–	50%	100%
	average time	0.42	–	–	47.73	12.94
deep-200	*# solved queries (%)*	100%	–	–	50%	0%
	average time	0.64	–	–	46.29	↗
adolena	*# solved queries (%)*	100%	–	100%	40%	–
	average time	1.39	–	4.92	94.92	–
stock-exchange	*# solved queries (%)*	100%	80%	60%	20%	–
	average time	4.50	19.59	37.11	133.38	–
vicodì	*# solved queries (%)*	100%	100%	100%	20%	80%
	average time	0.40	5.00	5.22	137.10	58.28
path5	*# solved queries (%)*	80%	80%	80%	40%	60%
	average time	5.48	28.47	16.94	131.39	54.60

Note: Symbol "–" under a certain system indicates that it cannot (or there is no evidence that it can) be tested on the corresponding benchmark. Moreover, symbol "↗" under a certain system indicates that its average running time has not been computed since it timed out on every query of the benchmark.

Detailed results on running times for each query benchmark and system are shown in Tables V–VII[14], whereas in Table VIII we provide an overall picture of percentage of solved queries, and geometric mean of running times for each benchmark and each system. It is worth pointing out that, to compute geometric means we used the maximum available time (300 seconds) in case of timeout. By considering actual times, gaps might be greater.

From the analysis of these tables, it is possible to observe that $DLV^{\exists}$ outperforms all the other tested systems on both running times and number of solved queries. In particular, while each system has missing results for some benchmark/query either because of expressiveness or timeouts, $DLV^{\exists}$ is always capable to answer the queries within the timeout, with one only exception for query P5:05 which is analyzed next. As previously pointed out, Path5 benchmark is designed to stress the system by exponentially increasing the complexity of queries, from P5:01 to P5:05. Moreover, from a theoretical analysis of the queries, we may expect that an increasing number of join variables (see Table IV) increases the complexity of query answering if these variables are attacked. In our experiments, none of the tested systems have been able to answer P5:05 but if we look at the trend in time increase from P5:01 to P5:04 we may observe that $DLV^{\exists}$ increase is smoother than the other systems. In order to better analyze this case, we carried out a more thorough analysis on P5:05, trying to identify how far each system is from the limit. As a consequence, we let the system run over the time limit. It turned out that $Graal_B$ stops for out-of-memory issues after about 1200 seconds,

[14]It is worth pointing out that $Graal_F$ is not listed in Table V even if it could run the corresponding queries, since it timed out on every query of both data sets. We also tested it with only 5 universities (`lubm-5`) obtaining the same time outs.

PAGOdA requires about 2404 seconds to answer the query, whereas Clipper needs about 890 seconds and Graal$_F$ has been stopped after two hours. Finally, DLV$^\exists$ provides query result in about 551 seconds, confirming the smoother trend with respect to the other systems on this benchmark.

As far as running times are concerned, DLV$^\exists$ is always faster than the other systems and, in some cases, improvements are up to orders of magnitude. There is no leading systems among the competitors, where each system shows better results on different benchmarks.

We then considered the best performing system among competitors for each benchmark, based on the average running times. It turned out that (see Table VIII): PAGOdA is the best on LUBM, Stock Exchange and Vicodì; Graal$_F$ is the best on deep-100; Graal$_B$ is the best on deep-200; finally, Clipper turned out to perform best on Adolena and Path5.

If we consider now the ratio between the average running time of DLV$^\exists$ for each benchmark and the average running time of the best performing system on the same benchmark, we may observe that in the worst case DLV$^\exists$ takes on average only 32% of the time needed by Clipper on Path5, whereas in the best case, which occurs on deep-200, DLV$^\exists$ takes on average only 1.38% of the time needed by Graal$_B$.

10. CONCLUSION

We have provided a new variant of the chase procedure, called parsimonious, which is sound and terminating over any Datalog$^\exists$ program. Based on the parsimonious chase, we have isolated a new semantic property, called parsimony, which ensures decidability of atomic query answering in general, and of conjunctive query answering whenever two further conditions —respectively called uniformity and compactness— are satisfied. Since the recognition of parsimony is undecidable, we have singled out shy, an easily recognizable class of parsimonious programs enjoying both uniformity and compactness. In particular, shy generalizes plain Datalog as well as the class of linear existential programs, while it is uncomparable to the other main classes ensuring decidability.

From a computational point of view, we have demonstrated that shy preserves the same (data and combined) complexity of Datalog for both atomic and conjunctive query answering. By exploiting our results, we have implemented a bottom-up evaluation strategy for shy programs inside the DLV system, and enhanced the computation by a number of optimization techniques, yielding DLV$^\exists$ —a powerful system for a fully-declarative ontology-based query answering. The experiments confirm the efficiency of DLV$^\exists$. Summing up, it turns out that DLV$^\exists$ is the first system supporting the standard first-order semantics for unrestricted CQs with existential variables over ontologies with advanced properties (such as, role transitivity, role hierarchy, role inverse and concept products), which can be even combined under suitable restrictions.

In the future, it would be relevant and interesting to refine and adapt our techniques to deal, also in an efficient way, with equality-generating dependencies, negation, and disjunction.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc.

Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. 2005. QUONTO: querying ontologies. In *Proc. of the 20th AAAI Conf. on AI*, Vol. 4. 1670–1671. http://dl.acm.org/citation.cfm?id=1619566.1619608

Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. 2009. The DL-Lite Family and Relations. *J. Artif. Intell. Res.* 36 (2009), 1–69. DOI:http://dx.doi.org/10.1613/jair.2820

Franz Baader, Sebastian Brandt, and Carsten Lutz. 2005. Pushing the EL Envelope. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. 364–369.

Jean-François Baget, Michel Leclère, and Marie-Laure Mugnier. 2010a. Walking the Decidability Line for Rules with Existential Variables. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. http://aaai.org/ocs/index.php/KR/KR2010/paper/view/1216

Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On rules with existential variables: Walking the decidability line. *Artif. Intell.* 175, 9-10 (2011), 1620–1654. DOI:http://dx.doi.org/10.1016/j.artint.2011.03.002

Jean-François Baget, Michel Leclère, and Marie-Laure Mugnier. 2010b. Walking the Decidability Line for Rules with Existential Variables. In *Proc. of the 12th KR Int. Conf.* 466–476. http://aaai.org/ocs/index.php/KR/KR2010/paper/view/1216

Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. 2015. Graal: A toolkit for query answering with existential rules. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*. Springer, 328–344.

Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2009. Extending Decidable Cases for Rules with Existential Variables. In *Proc. of the 21st IJCAI*. 677–682. http://ijcai.org/papers09/Papers/IJCAI09-118.pdf

Vince Bárány, Georg Gottlob, and Martin Otto. 2014. Querying the Guarded Fragment. *Logical Methods in Computer Science* 10, 2 (2014). DOI:http://dx.doi.org/10.2168/LMCS-10(2:3)2014

Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. (2017).

Meghyn Bienvenu, Balder ten Cate, Carsten Lutz, and Frank Wolter. 2014. Ontology-Based Data Access: A Study through Disjunctive Datalog, CSP, and MMSNP. *ACM Trans. Database Syst.* 39, 4 (2014), 33:1–33:44. DOI:http://dx.doi.org/10.1145/2661643

Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semant. Web* 2 (2011), 33–42. Issue 1. http://dl.acm.org/citation.cfm?id=2019470.2019472

Pierre Bourhis, Marco Manna, Michael Morak, and Andreas Pieris. 2016. Guarded-Based Disjunctive Tuple-Generating Dependencies. *ACM Trans. Database Syst.* 41, 4, Article 27 (Nov. 2016), 45 pages. DOI:http://dx.doi.org/10.1145/2976736

Sebastian Brandt. 2004. Polynomial Time Reasoning in a Description Logic with Existential Restrictions, GCI Axioms, and - What Else?. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*. 298–302.

Andrea Calì, Georg Gottlob, and Michael Kifer. 2008. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In *Proc. of the 11th KR Int. Conf.* 70–80. http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf Revised version: http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf.

Andrea Calì, Georg Gottlob, and Michael Kifer. 2013a. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *J. Artif. Intell. Res. (JAIR)* 48 (2013), 115–174. DOI:http://dx.doi.org/10.1613/jair.3873

Andrea Calì, Georg Gottlob, and Michael Kifer. 2013b. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *J. Artif. Intell. Res. (JAIR)* 48 (2013), 115–174. DOI:http://dx.doi.org/10.1613/jair.3873

Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009a. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*. 14–30. DOI:http://dx.doi.org/10.1145/1514894.1514897

Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009b. Tractable Query Answering over Ontologies with Datalog+/-. In *Proceedings of the 22nd International Workshop on Description Logics (DL 2009), Oxford, UK, July 27-30, 2009*. http://ceur-ws.org/Vol-477/paper_46.pdf

Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2012. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Sem.* 14 (2012), 57–83. DOI:http://dx.doi.org/10.1016/j.websem.2012.03.001

Andrea Calì, Georg Gottlob, and Andreas Pieris. 2010. Advanced Processing for Ontological Queries. *PVLDB* 3, 1 (2010), 554–565. http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R49.pdf

Andrea Calì, Georg Gottlob, and Andreas Pieris. 2010a. Advanced Processing for Ontological Queries. *PVLDB* 3, 1 (2010), 554–565. http://www.comp.nus.edu.sg/$\sim$vldb2010/proceedings/files/papers/R49.pdf

Andrea Calì, Georg Gottlob, and Andreas Pieris. 2010b. Query Answering under Non-guarded Rules in Datalog$^{\pm}$. In *Proc. of the 4th RR Int. Conf.*, Vol. 6333. 1–17. http://dx.doi.org/10.1007/978-3-642-15918-3_1

Andrea Calì, Georg Gottlob, and Andreas Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.* 193 (2012), 87–128. DOI:http://dx.doi.org/10.1016/j.artint.2012.08.002

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. 2010. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In *Proc. of the 24th AAAI Conf. on AI*. 1666–1670. http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1563

Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8, 3 (2017), 471–487.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family. *J. Autom. Reasoning* 39, 3 (2007), 385–429. DOI:http://dx.doi.org/10.1007/s10817-007-9078-x

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2013. Data complexity of query answering in description logics. *Artif. Intell.* 195 (2013), 335–360. DOI:http://dx.doi.org/10.1016/j.artint.2012.10.003

Diego Calvanese, Giuseppe Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *J. Autom. Reason.* 39 (2007), 385–429. Issue 3. DOI:http://dx.doi.org/doi:10.1007/s10817-007-9078-x

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. 2011. The MASTRO system for ontology-based data access. *Semant. Web* 2, 1 (2011), 43–53. http://dblp.uni-trier.de/db/journals/semweb/semweb2.html#CalvaneseGLLPRRRS11

Alexandros Chortaras, Despoina Trivela, and Giorgos Stamou. 2011. Optimized query rewriting for OWL 2 QL. In *International Conference on Automated Deduction*. Springer, 192–206.

Cristina Civili and Riccardo Rosati. 2012. A Broad Class of First-Order Rewritable Tuple-Generating Dependencies. In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*. 68–80. DOI:http://dx.doi.org/10.1007/978-3-642-32925-8_8

Bernardo Cuenca Grau, Boris Motik, Giorgos Stoilos, and Ian Horrocks. 2012. Completeness guarantees for incomplete ontology reasoners: Theory and practice. *Journal of Artificial Intelligence Research* 43 (2012), 419–476.

Chiara Cumbo, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. 2004. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In *Proc. of the 20th ICLP*, Vol. 3132. 371–385.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33 (2001), 374–425. Issue 3. DOI:http://dx.doi.org/doi:10.1145/502807.502810

Alin Deutsch, Alan Nash, and Jeff Remmel. 2008. The Chase Revisited. In *Proc. of the 27th PODS Symp.* 149–158. DOI:http://dx.doi.org/doi:10.1145/1376916.1376938

Thomas Eiter, Magdalena Ortiz, Mantas Simkus, Trung-Kien Tran, and Guohui Xiao. 2012. Query Rewriting for Horn-SHIQ Plus Rules.. In *AAAI*.

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005a. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124. DOI:http://dx.doi.org/10.1016/j.tcs.2004.10.033

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005b. Data exchange: semantics and query answering. *Theoret. Comput. Sci.* 336, 1 (2005), 89–124. DOI:http://dx.doi.org/doi:10.1016/j.tcs.2004.10.033

O. Febbraro, N. Leone, F. Ricca, G. Terracina, and P. Veltri. 2013. A graphic tool for ontology reasoning under Datalog$^{\exists}$. *21st Italian Symposium on Advanced Database Systems, SEBD 2013* (2013), 51–62. https://www.scopus.com/inward/record.uri?eid=2-s2.0-84903515165&partnerID=40&md5=c37522a917c9112ee2224c82a765c462

Birte Glimm, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. 2008. Conjunctive query answering for the description logic SHIQ. *JAIR* 31, 1 (2008), 157–204. http://dl.acm.org/citation.cfm?id=1622655.1622660

Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. 2014. HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning* 53, 3 (2014), 245–269.

Tomasz Gogacz and Jerzy Marcinkowski. 2013. Converging to the Chase - A Tool for Finite Controllability. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. 540–549. DOI:http://dx.doi.org/10.1109/LICS.2013.61

Georg Gottlob, Stanislav Kikot, Roman Kontchakov, Vladimir V. Podolskii, Thomas Schwentick, and Michael Zakharyaschev. 2014. The price of query rewriting in ontology-based data access. *Artif. Intell.* 213 (2014), 42–59. DOI:http://dx.doi.org/10.1016/j.artint.2014.04.004

Georg Gottlob, Marco Manna, and Andreas Pieris. 2013. Combining decidability paradigms for existential rules. *TPLP* 13, 4-5 (2013), 877–892. DOI:http://dx.doi.org/10.1017/S1471068413000550

Georg Gottlob, Giorgio Orsi, and Andreas Pieris. 2014. Query Rewriting and Optimization for Ontological Databases. *ACM Trans. Database Syst.* 39, 3 (2014), 25:1–25:46. DOI:http://dx.doi.org/10.1145/2638546

Georg Gottlob, Andreas Pieris, and Lidia Tendera. 2013. Querying the Guarded Fragment with Transitivity. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*. 287–298. DOI:http://dx.doi.org/10.1007/978-3-642-39212-2_27

Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. 2011. Stratification Criteria and Rewriting Techniques for Checking Chase Termination. *PVLDB* 4, 11 (2011), 1158–1168. http://www.vldb.org/pvldb/vol4/p1158-greco.pdf

V. Haarslev and R. Möller. 2001. RACER System Description. In *Proc. of the 6th IJCAR*. 701–705.

U. Hustadt, B. Motik, and U. Sattler. 2004. Reducing SHIQ- Descrption Logic to Disjunctive Datalog Programs. In *Proc. of the 9th KR Int. Conf.* 152–162.

Ullrich Hustadt, Boris Motik, and Ulrike Sattler. 2005. Data complexity of reasoning in very expressive description logics. In *IJCAI*, Vol. 5. 466–471.

D.S. Johnson and A. Klug. 1984. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.* 28, 1 (1984), 167–189. DOI:http://dx.doi.org/doi:10.1016/0022-0000(84)90081-3

Ton Kloks. 1994. *Treewidth, Computations and Approximations*. LNCS, Vol. 842. Springer. DOI:http://dx.doi.org/doi:10.1007/BFb0045375

Markus Krötzsch and Sebastian Rudolph. 2011. Extending Decidable Existential Rules by Joining Acyclicity and Guardedness. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*. 963–968. DOI:http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-166

Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. 2012. Efficiently Computable Datalog∃ Programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4521

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL* 7, 3 (2006), 499–562.

M. Leuschel and T. Schrijvers (Eds.). 2014. *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*. Vol. 14(4-5). Theory and Practice of Logic Programming, Online Supplement.

David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst.* 4, 4 (1979), 455–469. DOI:http://dx.doi.org/doi:10.1145/320107.320115

Bruno Marnette. 2009. Generalized schema-mappings: from termination to tractability. In *Proc. of the 28th PODS Symp.* 13–22. DOI:http://dx.doi.org/doi:10.1145/1559795.1559799

Michael Meier, Michael Schmidt, and Georg Lausen. 2009. On Chase Termination Beyond Stratification. *PVLDB* 2, 1 (2009), 970–981. http://www.vldb.org/pvldb/2/vldb09-295.pdf

Boris Motik, Rob Shearer, and Ian Horrocks. 2009. Hypertableau Reasoning for Description Logics. *JAIR* 36 (2009), 165–228.

Marie-Laure Mugnier. 2011. Ontological query answering with existential rules. In *Proc. of the 5th RR Int. Conf.* 2–23. http://dl.acm.org/citation.cfm?id=2036949.2036951

Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*. Springer, 3–20.

Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. 2009. Efficient query answering for OWL 2. *The Semantic Web-ISWC 2009* (2009), 489–504.

Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. 2010. Tractable query answering and rewriting under description logic constraints. *Journal of Applied Logic* 8, 2 (2010), 186–209.

Hctor Prez-Urbina, Boris Motik, and Ian Horrocks. 2010. Tractable query answering and rewriting under description logic constraints. *Journal of Applied Logic* 8, 2 (2010), 186 – 209.

DOI:http://dx.doi.org/10.1016/j.jal.2009.09.004 Selected papers from the Logic in Databases Workshop 2008.

Mariano Rodriguez-Muro and Diego Calvanese. 2011a. Dependencies: Making Ontology Based Data Access Work in Practice. In *Proc. of the 5th AMW on Foundations of Data Management*, Vol. 477.

Mariano Rodriguez-Muro and Diego Calvanese. 2011b. Dependencies to Optimize Ontology Based Data Access. In *Description Logics*, Vol. 745. CEUR-WS.org. http://dblp.uni-trier.de/db/conf/dlog/dlog2011.html#Rodriguez-MuroC11

Riccardo Rosati. 2007. On Conjunctive Query Answering in EL. In *Proceedings of the 2007 International Workshop on Description Logics (DL2007), Brixen-Bressanone, near Bozen-Bolzano, Italy, 8-10 June, 2007*. http://ceur-ws.org/Vol-250/paper_83.pdf

R. Rosati and A. Almatelli. 2010. Improving Query Answering over DL-Lite Ontologies. In *Proc. of the 12th KR Int. Conf.* 290–300.

Sebastian Rudolph, Markus Krötzsch, and Pascal Hitzler. 2008. All Elephants are Bigger than All Mice. In *Proc. of the 21st DL Int. Workshop*, Vol. 353. http://ceur-ws.org/Vol-353/RudolphKraetzschHitzler.pdf

Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. 2007. Pellet: A practical OWL-DL reasoner. *Web Semant.* 5, 2 (2007), 51–53. DOI:http://dx.doi.org/doi:10.1016/j.websem.2007.03.004

D. Tsarkov and I. Horrocks. 2006. FaCT++ Description Logic Reasoner: System Description. In *Proc. of the 3rd IJCAR*, Vol. 4130. 292–297.

Yujiao Zhou, Bernardo Cuenca Grau, Yavor Nenov, Mark Kaminski, and Ian Horrocks. 2015. PAGOdA: Pay-As-You-Go Ontology Query Answering Using a Datalog Reasoner. *J. Artif. Intell. Res.(JAIR)* 54 (2015), 309–367.