UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXV CICLO

Settore Disciplinare INF/01 – INFORMATICA

TESI DI DOTTORATO

DATALOG WITH EXISTENTIAL QUANTIFIERS: AN OPTIMAL TRADE-OFF BETWEEN EXPRESSIVENESS AND SCALABILITY

PIERFRANCESCO VELTRI

Supervisori	**Coordinatore**
Prof. Nicola Leone	Prof. Nicola Leone
Prof. Giorgio Terracina	

A.A. 2011 – 2012

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXV CICLO

Settore Disciplinare INF/01 – INFORMATICA

TESI DI DOTTORATO

DATALOG WITH EXISTENTIAL QUANTIFIERS: AN OPTIMAL TRADE-OFF BETWEEN EXPRESSIVENESS AND SCALABILITY

[illegible]

Pierfrancesco Veltri

Supervisori

Prof. Nicola Leone [illegible]

Prof. Giorgio Terracina [illegible]

Coordinatore

Prof. Nicola Leone [illegible]

A.A. 2011 – 2012

Dedicated to my parents,
Francesco and Rosa,
for their loving support

Acknowledgments

First of all, I would like to express my deep and sincere gratitude to my supervisor Nicola Leone, whose constant support, and warm encouragement have made possible to achieve the results described in this thesis. His energy and enthusiasm in research gave me all the inspiration and motivation I needed to complete my doctoral studies. It has been a privilege and honour to work under his supervision during these years.

I am deeply grateful to Marco Manna and Giorgio Terracina for their constant help and brilliant advices; their steadfast support to my studies has been greatly needed and deeply appreciated.

I am also thankful to Marco Maratea and my supervisor Francesco Ricca for the precious advices they gave me, and for the opportunities for personal and professional growth they ensured me.

I would like to thank also the research group of the Department of Mathematics at Unical, which has been a source of friendships as well as good advice and collaboration. Among them, a sincere and grateful thanks goes to all the colleagues who shared the office with me, especially Onofrio Febbraro and Kristian Reale for their friendship and constant help.

I wish to thank also a very special person that came into my life recently, and that became extremely important for me. Her love helped me to get through tough moments.

Lastly, and most importantly, I owe my deepest gratitude to my mother, my father, my brothers, and my dog, Batista, for their care, moral support, and unconditional love.

Abstract

Ontologies and rules play a central role in the development of the Semantic Web. Recent research in this context focuses especially on highly scalable formalisms for the Web of Data, which may highly benefit from exploiting database technologies.

In particular, *Datalog*$^\exists$ is the natural extension of *Datalog*, allowing existentially quantified variables in rule heads. This language is highly expressive and enables easy and powerful knowledge-modeling, but the presence of existentially quantified variables makes reasoning over *Datalog*$^\exists$ undecidable, in the general case. The results in this thesis enable powerful, yet decidable and efficient reasoning (query answering) on top of *Datalog*$^\exists$ programs.

On the theoretical side, we define the class of parsimonious *Datalog*$^\exists$ programs, and show that it allows of decidable and efficiently-computable reasoning. Unfortunately, we can demonstrate that recognizing parsimony is undecidable. However, we single out *Shy*, an easily recognizable fragment of parsimonious programs, that significantly extends both *Datalog* and *Linear Datalog*$^\exists$. Moreover, we show that *Shy* preserves the same (data and combined) complexity of query answering over *Datalog*, although the addition of existential quantifiers.

On the practical side, we implement a bottom-up evaluation strategy for *Shy* programs inside the DLV system, enhancing the computation by a number of optimization techniques. The resulting system is called DLV$^\exists$– a powerful system for answering conjunctive queries over *Shy* programs, which is profitably applicable to ontology-based query answering. Moreover, we design a rewriting method extending the well-known Magic-Sets technique to any *Datalog*$^\exists$ program. We demonstrate that our rewriting method preserves query equivalence on *Datalog*$^\exists$, and can be safely applied to *Shy* programs. We therefore incorporate the Magic-Sets method in DLV$^\exists$. Finally, we carry out an experimental analysis assessing the positive impact of Magic-Sets on DLV$^\exists$, and the effectiveness of the enhanced DLV$^\exists$ system compared to a number of state-of-the-art systems for ontology-based query answering.

Sommario

Nello sviluppo del Semantic Web, attualmente, le ontologie giocano un ruolo fondamentale. Gli studi condotti nell'ultimo periodo in questo ambito si sono focalizzati maggiormente sulla ricerca di formalismi altamente scalabili per il Web of Data, che potrebbe beneficiare delle più recenti tecnologie del mondo dei database.

In questo contesto spicca il linguaggio *Datalog*$^\exists$, che è la naturale estensione di *Datalog* dove si possono rappresentare variabili quantificate esistenzialmente in testa alle regole. Questo linguaggio è altamente espressivo e consente di modellare domini di conoscenza in maniera semplice e potente. Purtroppo, però, la presenza delle variabili esistenziali rende indecidibile il ragionamento su *Datalog*$^\exists$, nel caso generale. I risultati riportati in questo lavoro di tesi ci consentono di fare ragionamento (e quindi query answering) su programmi *Datalog*$^\exists$ in maniera potente, nonché decidibile ed efficiente.

Da un punto di vista teorico, definiamo la classe dei programmi *Datalog*$^\exists$ "parsimoniosi" e mostriamo che fare ragionamento su programmi di questo tipo è decidibile ed anche efficiente. Sfortunatamente, però, possiamo dimostrare che riconoscere la proprietà di "parsimonia" è indecidibile. Così, individuiamo un frammento facilmente riconoscibile dei programmi parsimoniosi, chiamato *Shy*, che estende in maniera significativa sia *Datalog* che *Linear Datalog*$^\exists$. Inoltre, mostriamo che *Shy* preserva la stessa complessità (data e combined) del query answering su *Datalog*, nonostante l'aggiunta dei quantificatori esistenziali.

Dal punto di vista pratico, invece, implementiamo una strategia di valutazione bottom-up per programmi *Shy* all'interno del ben noto sistema DLV, ed arricchiamo la computazione attraverso una serie di tecniche di ottimizzazione. Il sistema ottenuto è stato chiamato DLV$^\exists$– un potente ragionatore in grado di rispondere a query congiuntive su programmi *Shy*, che è proficuamente applicabile al problema dell'ontology-based query answering. Inoltre, progettiamo un metodo di riscrittura che estende la ben nota tecnica Magis-Sets a qualsiasi programma *Datalog*$^\exists$. Dimostriamo che la nostra tecnica di riscrittura preserva l'equivalenza tra query su *Datalog*$^\exists$, e può essere tranquillamente applicata a programmi *Shy*. Quindi, integriamo questo metodo basato sui Magic-Sets in DLV$^\exists$. E per concludere, con-

duciamo un'analisi sperimentale che dimostra l'effetto positivo dell'impatto che i Magic-Sets hanno su DLV$^{\exists}$, ed anche l'efficacia del sistema evoluto, DLV$^{\exists}$, a confronto con una serie di sistemi che rappresentano lo stato dell'arte nell'ontology-based query answering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The adoption of ontologies and semantic technology in companies, governmental organizations, and academia is becoming nowadays more and more prominent, especially for knowledge representation and data management. Thanks to their expressive power and formal semantics, ontologies have also been adopted as high-level conceptual descriptions of the data in a database, often replacing traditional metadata and documentation such as data dictionaries, UML class-diagrams and E/R schemata. Recently, the relationship of ontologies and databases tightened, originating a new type of data management systems where a relational database is enriched by an ontological theory that enforces expressive constraints over the database. Such constraints go far beyond traditional integrity constraints and can be used to enable complex reasoning tasks over the database instances. However, the main task in an ontological database remains that of query answering. A number of commercial data management systems – such as Oracle[1], Ontotext[2] and Ontoprise[3] – provide ontological querying capabilities in their current solutions. Also, ontological reasoning is part of several research-based systems, such as QuOnto [2], FaCT++ [89], and Nyaya [43]. The main problem is how to couple these two different types of technologies smoothly and efficiently.

In knowledge representation community, the term "ontology" indicates the general domain knowledge – sometimes also called terminological knowledge – in order to be clearly distinguished from the assertional knowledge, called here data set. Given a *knowledge base* $KB = (\Sigma, D)$ composed of an ontology Σ and of data set D, and a query q, the ontology-based Query Answering (QA) problem consists in computing the set of answers to the query q on KB, while taking implicit knowledge represented in the ontology into account.

A key issue in ontology-based QA is the design of the language that is pro-

[1]See: `http://www.oracle.com/`
[2]See: `http://www.ontotext.com/`
[3]See: `http://www.ontoprise.de/`

vided for specifying the ontological theory Σ. This language should balance expressiveness and complexity, and in particular it should possibly be: (1) intuitive and easy-to-understand; (2) QA-decidable (i.e., QA should be decidable in this language); (3) efficiently computable; (4) powerful enough in terms of expressiveness; and (5) suitable for an efficient implementation.

In the Semantic Web, ontological knowledge is often represented with formalisms based on description logics (DLs). However, DLs traditionally focused on reasoning tasks about the ontology itself (the so-called TBox), for instance classifying concepts; querying tasks were restricted to ground atom entailment. Conjunctive query answering with classical DLs has appeared to be extremely complex (e.g., for the classical DL $\mathcal{ALCI}$, it is **2EXP**-complete, and still NP-complete in the size of the data). Hence, less expressive DLs specially devoted to conjunctive query answering on large amounts of data have been designed recently. A family of well-known DLs fulfilling this criterion is, e.g., the *DL-Lite* family [34, 81] (which has recently been further extended in [9, 10]). The following example briefly illustrates how queries can be posed and answered in *DL-Lite*.

Example 1.0.1. A DL knowledge base consists of a TBox and an ABox (the data set). For example, the knowledge that every conference paper is an article and that every scientist is the author of at least one paper can be expressed by the axioms $ConferencePaper \sqsubseteq Article$ and $Scientist \sqsubseteq \exists isAuthorOf$ in the TBox, respectively, while the knowledge that John is a scientist can be expressed by the axiom $Scientist(john)$ in the ABox. A simple Boolean conjunctive query (BCQ) asking whether John authors a paper is $\exists X isAuthorOf(john, X)$.

An ABox can be identified with an extensional database, while a TBox can be regarded as a set of integrity constraints involving, among others, functional dependencies and (possibly recursive) inclusion dependencies [47, 1]. An important result of [34, 81] is that the *DL-Lite* description logics, in particular, *DL-Lite*$_F$, *DL-Lite*$_R$, and *DL-Lite*$_A$, are not only decidable, but that answering (unions of) conjunctive queries for them is in **LOGSPACE**, and actually in **AC**$_0$, in the data complexity, and query answering in *DL-Lite* is FO-rewritable [34]. Notice that, in the context of DLs as well as in other contexts, two complexity measures are classically considered for query answering problem: the usual complexity, called *combined* complexity, and *data* complexity. We talk about combined complexity of query answering in general, and about data complexity of query answering, under the assumption that only the data, here the ABox, are considered as part of the input while both the query and the TBox are considered fixed.

Moreover, many alghoritms and systems for *DL-Lite* have been developed, such as Quonto (Acciarri et al. 2005), Owlgres (Stocker and Smith 2008), Requiem (Perez-Urbina, Motik, and Horrocks 2009), and Presto (Almatelli, and

Rosati 2010). Thus, *DL-Lite* is a well-consolidated formalism for ontology-based QA in the Semantic Web context.

On the other hand, querying large amounts of data is the fundamental task of databases. Therefore, the challenge in this domain is now to access data while taking ontological knowledge into account. The deductive database language *Datalog* allows to express some ontological knowledge. However, in *Datalog* rules, variables are range-restricted, i.e., all variables in the rule head necessarily occur in the rule body, which does not allow for value invention. This feature has been recognized as crucial in an open-world perspective, where it cannot be assumed that all individuals are known in advance. This motivated the recent extension of *Datalog* to TGDs (i.e., existential rules) which gave rise to Datalog$^{\pm}$, the family of *Datalog*-based languages proposed by Calì, Gottlob, and Lukasiewicz (2009) for tractable query answering over ontologies, that is arousing increasing interest [77] in the last period. This family of languages, that encompasses and generalizes all the description logics of the *DL-Lite* family, is mainly based on *Datalog*$^{\exists}$, the natural extension of *Datalog* that allows $\exists$-quantified variables in rule heads. For example, the following *Datalog*$^{\exists}$ rules

```
∃Y father(X,Y) :- person(X).
person(Y) :- father(X,Y).
```

state that if X is a person, then X must have a father Y, which has to be a person as well. However, more in general, the Datalog$^{\pm}$ family intends to collect all expressive extensions of *Datalog* which are based on *tuple-generating dependencies* (or TGDs, which are *Datalog*$^{\exists}$ rules with possibly multiple atoms in rule heads), *equality-generating dependencies* and *negative constraint*. 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.

A number of QA-decidable Datalog$^{\pm}$ languages have been defined in the literature. They rely on three main paradigms, called *weak-acyclicity* [48], *guardness* [27] and *stickiness* [29], depending on syntactic properties. But there are also QA-decidable “abstract” classes of *Datalog*$^{\exists}$ programs, called *Finite-Expansion-Sets*, *Finite-Treewidth-Sets* and *Finite-Unification-Sets*, depending on semantic properties that capture the three mentioned paradigms, respectively [77]. However, even if all known languages based on these properties enjoy the simplicity of *Datalog* and are endowed with a number of properties that are desired for ontology specification languages, none of them fully satisfy conditions (1)–(5) above (see Section 2.4). While *DL-Lite* is a well-consolidated formalism for ontology-based QA in the Semantic Web context, the Datalog$^{\pm}$ family has still some “weaknesses”. In particular, notwithstanding a number of Datalog$^{\pm}$ fragments have been already proposed, the evident gap emerging in this scenario is given by the lack of a language that offers a good efficiency whitout renouncing expressiveness. Thus, in this work, we focus on this framework and we intend to close this gap by singling 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 QA-decidable ontology specification language that combines positive aspects of different Datalog$^\pm$ languages. *Shy* represents an optimal trade-off between expressiveness and scalability in the scenario of *Datalog* with existential quantifiers (see Section 3.4). Indeed, 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 QA-decidable; (3) it is efficiently computable (tractable data complexity and limited combined-complexity); (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 of an efficient on-the-fly QA, as witnessed by the experimental results [4] reported in Chapter 5. From a technical viewpoint, the contribution of our work is the following.

▸ We propose a new semantic property called *parsimony*, and prove that on the abstract class of parsimonious *Datalog*$^\exists$ programs, called *Parsimonious*, (atomic) query answering is decidable and also efficiently computable.

▸ After showing that recognition of parsimony is undecidable (**coRE**-complete), we single out *Shy*, a subclass of *Parsimonious*, which guarantees both easy recognizability and efficient answering even to conjunctive queries.

▸ We demonstrate that both *Parsimonious* and *Shy* 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.

▸ We introduce a novel approach for conjunctive query answering, called *parsimonious-chase resumption*, which is sound and complete for query answering over *Shy*.

▸ We implement a bottom-up evaluation strategy for Shy programs inside the well-known DLV system, and enhance the computation by a number of optimization techniques (e.g. we implement a variant of the well-known magic-set optimization technique (Cumbo et al. 2004), that we adapted to *Datalog*$^\exists$), yielding DLV$^\exists$ – a powerful system for query answering over *Shy* programs, which is profitably applicable for ontology-based query answering. To the best of our knowledge, DLV$^\exists$ is the first system supporting the standard first-order semantics for unrestricted CQs with existential variables over ontologies with advanced properties (some of these beyond **AC**$_0$), such as, role transitivity, role hierarchy, role inverse,

[4]Intuitively, parsimonious inference generates no isomorphic atoms (see Section 3.1); while on-the-fly QA does not need any preliminary materialization or compilation phase (see Chapter 5), and is very well suited for QA against frequently changing ontologies.

and concept products [50].
▸ We perform an experimental analysis, comparing DLV$^\exists$ with a number of state-of-the-art systems for ontology-based QA. The positive results attained through this analysis 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.
▸ We analyze the Datalog$^\pm$ framework, providing a precise taxonomy of the QA-decidable *Datalog*$^\exists$ classes (see Chapter 2). It turns out that both *Parsimonious* and *Shy* strictly contain *Datalog* $\cup$ *Linear-Datalog*$^\exists$, while they are uncomparable to *Finite-Expansion-Sets*, *Finite-Treewidth-Sets*, and *Finite-Unification-Sets* (see Section 3.4).
▸ We analyze related work, providing a description of the basic *DL-Lite* classes, we observe that *Shy* encompasses and generalizes all the languages of the *DL-Lite* family (see Chapter 6).

The remainder of the thesis is structured as follows. First, syntax and semantics of *Datalog*$^\exists$ programs, as well as some preliminaries and useful notation, are fixed in Chapter 2. Moreover, in the same chapter, an overview of the Datalog$^\pm$ family is also provided, and strenghts and weaknesses of this framework are highlighted. Then, a new class of *Datalog*$^\exists$ programs, called *Parsimonious-Sets*, is introduced in Chapter 3, as well as some of its properties. In particular, recognizing *parsimony* is demonstrated to be undecidable, thus, the *Shy* language and its main properties are presented. Afterwards, a description of the DLV$^\exists$ system is provided in Chapter 4, where the variant of the Magic-Sets tecnique implemented in DLV$^\exists$ for *Shy* programs is also discussed. Moreover, experimental results are presented in Chapter 5. Finally, related work and conclusion are reported in Chapters 6-7.

Chapter 2

The Datalog$^{\pm}$ family

In this chapter we give an overview of the Datalog$^{\pm}$ family of languages. In Section 2.1, after some useful preliminaries, we introduce *Datalog*$^{\exists}$ programs and conjunctive queries (CQs), query answering and universal models and, finally, the chase procedure. In Section 2.2 we survey all notable Datalog$^{\pm}$ classes. Afterwards, in Section 2.3, we deal with complexity. Finally, in Section 2.4, we analyze relationships among Datalog$^{\pm}$ fragments, enlighting the lack of an optimal trade-off between expressiveness and scalability.

2.1 The framework

As discussed in the introduction, *Datalog*$^{\exists}$ is the natural extension of *Datalog* that allows existentially quantified variables in rule heads. In particular, *Datalog*$^{\exists}$ rules, or more in general TGDs, are an interesting fragment of first-order logic originally introduced in the context of the design of relational database schemas and suitable for expressing integrity constraints of databases [23]. Nowadays, as previously discussed, TGDs are also an important and convenient formalism for describing missing information in ontologies. However, in this context, a crucial question is how to interpret the combined information provided by an extensional database and an ontological theory. In the case of TGDs, given a database D and a set of dependencies Σ, the semantics of this pair is commonly given by the so-called *universal* model U of the logical theory $D \cup \Sigma$ (i.e., U contains D and also satisfies all TGDs of Σ) which can be homomorphically mapped to all other models of $D \cup \Sigma$. In terms of query evaluation, this entails, for example, that for a Boolean CQ q, $D \cup \Sigma \models q$ iff $U \models q$ [48, 44]. However, Calì, Gottlob, and Kifer (2008) have shown that CQ answering under general TGDs is logspace-reducible to CQ answering under *Datalog*$^{\exists}$ programs. Therefore, w.l.o.g., *Datalog*$^{\exists}$ can be always used instead of general TGDs for QA purposes.

A well-known procedure that computes a universal model for a *Datalog*$^{\exists}$ program is called *chase* [70, 60]. Due to its power, this technique, firstly introduced for enforcing the validity of a set of TGDs, has several important uses, such as *query equivalence* and *query optimization* [1]. However, there are cases where the universal model found by the chase is infinite and also there are cases where the problem of deciding whether a *Datalog*$^{\exists}$ program entails a query is undecidable [48, 44]. Finally, even if the QA problem is decidable, it could be computationally hard.

In this section, after some useful preliminaries, we provide more formally syntax of *Datalog*$^{\exists}$ programs and CQs. Next, we equip such structures with a formal semantics. Finally, we show the *chase*, a well-known procedure that allows of answering CQs [70, 60].

2.1.1 Preliminaries

The following notation will be used throughout the thesis. 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 t, a generic *term*; by $\mathtt{c}$, $\mathtt{d}$ and $\mathtt{e}$, constants; by φ, a null; by $\mathtt{X}$ and $\mathtt{Y}$, variables; by $\mathbf{X}$ and $\mathbf{Y}$, sets of variables; by Π an alphabet of *predicate symbols* each of which, say $\mathtt{p}$, has a fixed nonnegative arity, denoted by $\mathsf{arity}(\mathtt{p})$; by $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, *atoms* being expressions of the form $\mathtt{p}(t_1, \dots, t_k)$, where $\mathtt{p}$ is a predicate symbol and $t_1, \dots, t_k$ is a *tuple* of terms. Moreover, if the tuple of an atom consists of only constants and nulls, then this atom is called *ground*; if $T \subseteq \Delta_C \cup \Delta_N$, then $\mathsf{base}(T)$ denotes the set of all ground atoms that can be formed with predicate symbols in Π and terms from T; if $\mathbf{a}$ is an atom, then $\mathsf{pred}(\mathbf{a})$ denotes the predicate symbol of $\mathbf{a}$; if ς is any formal structure containing atoms, then $\mathsf{terms}(\varsigma)$ (resp., $\mathsf{dom}(\varsigma)$) denotes all the terms from Δ (resp., $\Delta_C \cup \Delta_N$) occurring in the atoms of ς; moreover, $\mathsf{vars}(\varsigma) = \mathsf{terms}(\varsigma) \setminus \mathsf{dom}(\varsigma)$ indicates all the variables appearing in the atoms of ς.

Mappings.

Given a *mapping* $\mu : S_1 \to S_2$, its *restriction to* a set S is the mapping $\mu|_S$ from $S_1 \cap S$ to S_2 s.t. $\mu|_S(s) = \mu(s)$ for each $s \in S_1 \cap S$. If μ' is a restriction of μ, then μ is called an *extension* of μ', also denoted by $\mu \supseteq \mu'$. Let $\mu_1 : S_1 \to S_2$ and $\mu_2 : S_2 \to S_3$ be two mappings. We denote by $\mu_2 \circ \mu_1 : S_1 \to S_3$ the *composite* mapping.

We call *homomorphism* any mapping $h : \Delta \to \Delta$ whose restriction $h|_{\Delta_C}$ is the identity mapping. In particular, h is an homomorphism from an atom $\mathbf{a} = \mathtt{p}(t_1, \dots, t_k)$ to an atom $\mathbf{b}$ if $\mathbf{b} = \mathtt{p}(h(t_1), \dots, h(t_k))$. With a slight abuse of notation, $\mathbf{b}$ is denoted by $h(\mathbf{a})$. Similarly, h is a homomorphism from a set of atoms S_1 to another set of atoms S_2 if $h(\mathbf{a}) \in S_2$, for each $\mathbf{a} \in S_1$. Moreover, $h(S_1) = \{h(\mathbf{a}) : \mathbf{a} \in S_1\} \subseteq S_2$. In particular, if $S_1 = \emptyset$, then $h(S_1) = \emptyset$. In case the domain of h is the empty set, then h is called *empty homomorphism* and it is denoted by $h_\emptyset$. In particular, $h_\emptyset(\mathbf{a}) = \mathbf{a}$, for each atom $\mathbf{a}$.

An *isomorphism* between two atoms (or two sets of atoms) is a bijective homomorphism. Given two atoms $\mathbf{a}$ and $\mathbf{b}$, we say that: $\mathbf{a} \leq \mathbf{b}$ iff there is a homomorphism from $\mathbf{b}$ to $\mathbf{a}$; $\mathbf{a} \simeq \mathbf{b}$ iff there is an isomorphism between $\mathbf{a}$ and $\mathbf{b}$; $\mathbf{a} < \mathbf{b}$ iff $\mathbf{a} \leq \mathbf{b}$ holds but $\mathbf{a} \simeq \mathbf{b}$ does not.

A *substitution* is a homomorphism σ from Δ to $\Delta_C \cup \Delta_N$ whose restriction $\sigma|_{\Delta_C \cup \Delta_N}$ is the identity mapping. Also, $\sigma_\emptyset = h_\emptyset$ denotes the empty substitution.

2.1.2 Programs and Queries

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

$$\forall \mathbf{X} \exists \mathbf{Y} \ \mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]} \leftarrow \mathbf{conj}_{[\mathbf{X}]} \tag{2.1}$$

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

Example 2.1.1. The following expression is a *Datalog*$^\exists$ rule where father is the head and person the only body atom.

```
∃Y father(X,Y) :- person(X).
```
□

Given a *Datalog*$^\exists$ program P, a *conjunctive query* (CQ) q over P is a first-order (FO) expression of the form:

$$\exists \mathbf{Y} \ \mathbf{conj}_{[\mathbf{X} \cup \mathbf{Y}]} \tag{2.2}$$

where $\mathbf{X}$ are its free variables, and $\mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$ is a conjunct containing only and all the variables in $\mathbf{X} \cup \mathbf{Y}$ and possibly some constants. To highlight the free variables, we write $q(\mathbf{X})$ instead of q. Query q is called *Boolean CQ* (BCQ) if $\mathbf{X} = \emptyset$. Moreover, q is called *atomic* if $\mathbf{conj}$ is an atom. Finally, $\mathsf{atoms}(q)$ denotes the set of atoms in $\mathbf{conj}$.

Example 2.1.2. The following expression is a CQ asking whether there exists a grandfather having `john` as nephew.

```
∃Y father('john',X),father(X,Y)
```
□

2.1.3 Query Answering and Universal Models

In the following, we equip *Datalog*$^{\exists}$ programs and queries with a formal semantics to result in a formal QA definition.

Given a set S of atoms and an atom $\mathbf{a}$, we say that $S \models \mathbf{a}$ (resp., $S \Vdash \mathbf{a}$) holds if there is a substitution σ s.t. $\sigma(\mathbf{a}) \in S$ (resp., a homomorphism h s.t. $h(\mathbf{a}) \in S$).

Let $P \in$ *Datalog*$^{\exists}$. A set $M \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$ is a *model* for P ($M \models P$, for short) if, for each $r \in P$ of the form (2.1), whenever there exists a substitution σ s.t. $\sigma(\mathsf{body}(r)) \subseteq M$, then $M \models \sigma|_{\mathbf{X}}(\mathsf{head}(r))$. (Note that, $\sigma|_{\mathbf{X}}(\mathsf{head}(r))$ contains only and all the $\exists$-variables $\mathbf{Y}$ of r.) The set of all the models of P are denoted by $\mathsf{mods}(P)$.

Let $M \in \mathsf{mods}(P)$. A BCQ q is *true* w.r.t. M ($M \models q$) if there is a substitution σ s.t. $\sigma(\mathsf{atoms}(q)) \subseteq M$. Analogously, the answer of a CQ $q(\mathbf{X})$ w.r.t. M is the set $\mathsf{ans}(q, M) = \{\sigma|_{\mathbf{X}} : \sigma \text{ is a substitution} \wedge M \models \sigma|_{\mathbf{X}}(q)\}$.

The answer of a CQ $q(\mathbf{X})$ w.r.t. a program P is the set $\mathsf{ans}_P(q) = \{\sigma : \sigma \in \mathsf{ans}(q, M)\ \forall M \in \mathsf{mods}(P)\}$. Note that, $\mathsf{ans}_P(q) = \{\sigma_{\emptyset}\}$ iff q is a BCQ. In this case, we say that q is *cautiously true* w.r.t. P or, equivalently, that q is *entailed* by P. This is denoted by $P \models q$, for short.

Let $\mathcal{C}$ be a class of *Datalog*$^{\exists}$ programs. The following definition formally fixes the computational problem studied in this thesis, concerning query answering.

Definition 2.1.3. $\textsc{QA}_{[\mathcal{C}]}$ is the following decision problem. Given a program P belonging to $\mathcal{C}$, an atomic query q, and a substitution σ for q, does σ belong to $\mathsf{ans}_P(q)$? □

In the following, a *Datalog*$^{\exists}$ class $\mathcal{C}$ is called QA-decidable if and only if problem $\textsc{QA}_{[\mathcal{C}]}$ is decidable. Finally, before concluding this section, we mention that QA can be carried out by using a universal model. Actually, a model U for P is called *universal* if, for each $M \in \mathsf{mods}(P)$, there is a homomorphism h s.t. $h(U) \subseteq M$.

Procedure 1 CHASE(P)

```
Input: $Datalog^{\exists}$ program $P$
Output: A Universal Model $\mathsf{chase}(P)$ for $P$
1.  $C := \mathsf{data}(P)$
2.  $NewAtoms := \varnothing$
3.  for each $r \in P$ do
4.    for each firing substitution $\sigma$ for $r$ w.r.t. $C$ do
5.      if ( $(C \cup NewAtoms) \nvDash \sigma(\mathsf{head}(r))$ )
6.        $\mathsf{add}(\hat{\sigma}(\mathsf{head}(r)), NewAtoms)$
7.  if ($NewAtoms \neq \varnothing$)
8.    $C := C \cup NewAtoms$
9.    go to step 2
10. return $C$
```

Proposition 2.1.4 (Fagin et al. (2005))**.** *Let U be a universal model for P. Then, (i) $P \vDash q$ iff $U \vDash q$, for each BCQ q; (ii) $\mathsf{ans}_P(q) \subseteq \mathsf{ans}(q, U)$ for each CQ q; and (iii) $\sigma \in \mathsf{ans}_P(q)$ iff both $\sigma \in \mathsf{ans}(q, U)$ and $\sigma : \Delta_V \to \Delta_C$.*

2.1.4 The Chase

As already mentioned, the chase is a well-known procedure for constructing a universal model for a *Datalog*$^{\exists}$ program. We are now ready to show how this procedure works, in one of its variants (although slightly revised).

First, we introduce the notion of *chase step*, which, intuitively, *fires* a rule r on a set C of atoms for inferring new knowledge. More precisely, given a rule r of the form (2.1) and a set C of atoms, a *firing* substitution σ for r w.r.t. C is a substitution σ on $\mathbf{X}$ s.t. $\sigma(\mathsf{body}(r)) \subseteq C$. Next, given a firing substitution σ for r w.r.t. C, the *fire* of r on C due to σ infers $\hat{\sigma}(\mathsf{head}(r))$, where $\hat{\sigma}$ is an extension of σ on $\mathbf{Y} \cup \mathbf{X}$ associating each $\exists$-variable in $\mathbf{Y}$ to a different null. Finally, Procedure 1 illustrates the overall *restricted chase procedure*. Importantly, we assume that different fires (on the same or different rules) always introduce different "fresh" nulls. The procedure consists of an exhaustive series of fires in a breadth-first (level-saturating) fashion, which leads as result to a (possibly infinite) $\mathsf{chase}(P)$.

The *level* of an atom in $\mathsf{chase}(P)$ is inductively defined as follows. Each atom in $\mathsf{data}(P)$ has level 0. The level of each atom constructed after the application of a restricted chase step is obtained from the highest level of the atoms in $\sigma(\mathsf{body}(r))$ plus one. For each $k \geq 0$, $\mathsf{chase}^k(P)$ denotes the subset of $\mathsf{chase}(P)$ containing only and all the atoms of level up to k. Actually, by Procedure 1, $\mathsf{chase}^k(P)$ is precisely the set of atoms which is inferred the k^{th}-time that the outer for-loop is ran.

The *chase graph* for P is the directed graph consisting of $\mathsf{chase}(P)$ as the set of nodes and having an arrow from $\mathtt{a}$ to $\mathtt{b}$ iff $\mathtt{b}$ is obtained from $\mathtt{a}$ and possibly other atoms by a one-step application of a rule $r \in P$.

Proposition 2.1.5. *[48, 44] Given a Datalog$^{\exists}$ program P,* CHASE *constructs a universal model for P.*

Unfortunately, CHASE does not always terminate.

Proposition 2.1.6. *[48, 44]* $\mathrm{QA}_{[Datalog^{\exists}]}$ *is undecidable even for atomic queries. In particular, it is **RE**-complete.*

Therefore, the general *Datalog*$^{\exists}$ language has to be somehow restricted in order to reach decidability of QA. In the next section we show the scenario of the most known QA-decidable fragments of *Datalog*$^{\exists}$.

2.2 Decidability landscape

Datalog [1] has been widely used as a database programming and query language for long time. It is rarely used directly as a query language in corporate application contexts. However, it is used as an inference engine for knowledge processing within several software tools, and has recently gained popularity in the context of various applications, such as web data extraction [21, 51], source code querying and program analysis [57], and modeling distributed systems [3]. At the same time, Datalog has been shown to be too limited to be effectively used to model ontologies and expressive database schemata, as explained in [78]. In this respect, the main missing feature in Datalog is the possibility of expressing existential quantification in the head; this was addressed in the literature by introducing Datalog with value invention [26, 71]. Unfortunately, as remarked in Section 2.1.4, QA over general *Datalog*$^{\exists}$ is undecidable even for atomic queries; therefore, some restrictions is needed to ensure decidability.

In this regard, Datalog$^{\pm}$, the family of *Datalog*-based languages proposed by Calì, Gottlob, and Lukasiewicz (2009) for tractable query answering over ontologies, is arousing increasing interest [77]. More in general, the Datalog$^{\pm}$ family intends to collect all expressive extensions of *Datalog* which are based on *tuple-generating dependencies* (or TGDs, which are *Datalog*$^{\exists}$ rules with possibly multiple atoms in rule heads), *equality-generating dependencies* and *negative constraint*. In particular, the "plus" symbol refers to any possible combination of these extensions, while the "minus" one imposes at least decidability.

A number of QA-decidable Datalog$^{\pm}$ languages have been defined in the literature. Decidable classes found in the literature are based on various syntactic properties of existential rules. In order to classify them, three abstract properties related to the behavior of reasoning mechanisms are considered in [15, 13]: the "forward chaining halts in finite time"; the "forward chaining may not halt but the atoms generated have a tree-like structure"; the "backward chaining mechanism halts in finite time". These properties yield three abstract classes of rules, respectively called *finite expansion sets*, *bounded treewidth sets* and *finite unification sets*. These classes are said to be abstract in the sense that they do not come with a syntactic property that can be checked on rules or programs. As a matter of fact, none of these classes is recognizable, i.e., the problem of determining whether a given program fulfills the abstract property is not decidable [13]. In the following, we first give some preliminary notions about these reasoning mechanisms and then we analyze these three abstract classes in more detail.

2.2.1 Forward and backward chaining mechanisms

In this section we give some preliminary notions about forward and backward chaining algorithm. In the sequel, let P be a *Datalog*$^{\exists}$ program, we denote by D the set, $\mathsf{data}(P)$, of ground facts of P and by Σ the set, $\mathsf{rules}(P)$, of rules of the program. Moreover, given a *Datalog*$^{\exists}$ rule r, the set of variables $fr(r) = \mathsf{vars}(\mathsf{head}(r)) \cap \mathsf{vars}(\mathsf{body}(r))$ is called the *frontier* of r.

Definition 2.2.1. Given a rule $r \in \Sigma$, r is *applicable* to D if there is a substitution h on $\mathsf{vars}(\mathsf{body}(r))$ s.t. $h(\mathsf{body}(r)) \subseteq D$; the application of r on D w.r.t. h is a set of ground atoms $\sigma(D, r, h) = D \cup h'(\mathsf{head}(r))$ where h' is an extension of h on $\mathsf{vars}(\mathsf{head}(r)) \cup \mathsf{vars}(\mathsf{body}(r))$, that replaces each $\mathtt{X} \in fr(r)$ with $h(\mathtt{X})$, and each other variable with a "fresh" null not introduced before; this application is said to be redundant if $\sigma(D, r, h) \equiv D$.

Now, we are ready to define a *derivation sequence*.

Definition 2.2.2. A Σ-derivation of D is a finite sequence $(D_0 = D), \ldots, D_k$ s.t. for all $0 \leq i < k$, there is $r_i \in \Sigma$ and a substitution h_i from $\mathsf{body}(r_i)$ to D_i s.t. $D_{i+1} = \sigma(D_i, r_i, h_i)$.

Below, we report a theorem stating the completeness of forward chaining.

Theorem 2.2.3. *Let q be a BCQ. Then $(D, \Sigma) \vDash q$ iff there exists an Σ-derivation $(D_0 = D), \ldots, D_k$ such that $D_k \vDash q$.*

It follows that a breadth-first forward chaining mechanism yields a positive answer in finite time when $(D, \Sigma) \vDash q$. This mechanism, called the *saturation* hereafter (and the *chase* in databases) works as follows. Let $D_0 = D$ be the initial set of ground atoms. Each step i ($i \geq 1$) consists of checking if q maps to D_{i-1}, and otherwise producing D_i from D_{i-1}, by computing all new substitutions from each rule body to D_{i-1}, then performing all corresponding rule applications. A substitution is said to be new if it has not been already computed at a previous step, i.e., it uses at least an atom added at step $i-1$ ($i \geq 2$). The result D_k obtained after the step k is called the k-saturation of D and is denoted by $\sigma_k(D, \Sigma)$.

The two classical ways of processing rules are forward chaining, introduced above, and *backward chaining*. Instead of using rules to enrich the facts, the backward chaining proceeds in the "reverse" manner: it uses the rules to *rewrite* the query in different ways with the aim of producing a query that maps to D. The key operation in this mechanism is the *unification* operation between part of a current goal (a conjunctive query in our framework) and a rule head. This mechanism is typically used in logic programming, with rules having a head restricted to a single atom, which is unified with an atom of the current goal. The operator that rewrites the query is denoted by β and is informally defined as follows (for a formal definition see [15, 17]): given a conjunctive query q, a rule $r \in \Sigma$ and a unifier μ of (part of) q with (the head of) r, $\beta(q, r, \mu) = q_\mu \cup B_\mu$ where q_μ is a specialization of the non-unified subset of q (determined by μ), and B_μ is a specialization of the body of r (also determined by μ). Now we define a *rewriting sequence*.

Definition 2.2.4. Let q be a boolean conjunctive query. An Σ-rewriting of q is a finite sequence $(q_0 = q), q_1, ..., q_k$ s. t. for all $0 \leq i < k$, there is $r_i \in \Sigma$ and a unifier μ of q_i with (the head of) r_i such that $q_{i+1} = \beta(q_i, r_i, \mu)$.

The soundness and completeness of the backward chaining mechanism can be proven via the following equivalence with the forward chaining: there is an Σ-rewriting from the query q to a query q' that maps to the initial set of ground atoms D iff there is a Σ-derivation from D to a set of ground atoms D' such that q maps to D'.

2.2.2 Abstract classes and their semantic properties

Now, we are ready to analyze, in more detail, the abstract classes *finite expansion sets*, *bounded treewidth sets* and *finite unification sets* introduced before. In particular, a set of *Datalog*$^{\exists}$ rules Σ is said to be a *finite expansion set (fes)* if, for every set of ground atoms D, there exists an integer k such that $\sigma_k(D, \Sigma) \equiv \sigma_{k+1}(D, \Sigma)$, i.e., all rule applications to $\sigma_k(D, \Sigma)$ are redundant [18]. Weaker versions, in the sense that they allow to stop in less cases, can be considered. For instance the halting condition may be $\sigma_k(D, \Sigma) = \sigma_{k+1}(D, \Sigma)$, i.e., no new rule application can be performed on $\sigma_k(D, \Sigma)$; a forward-chaining algorithm with this halting condition corresponds to the so-called *oblivious* chase in databases (note that the *restricted* chase is still weaker than *fes*). If Σ is a *fes*, then the termination is guaranteed for any forward chaining that (1) builds a derivation sequence until the halting condition is satisfied (the order in which rules are applied does not matter), then (2) checks if the query maps to the obtained result. Bounded-treewidth sets of rules form a more general class, which was essentially introduced in [27]. The following definition of the *treewidth* of a set of ground atoms corresponds to the usual definition of the treewidth of a graph, where the considered graph is the primal graph of the hypergraph of the set of atoms (this graph has the same set of nodes as the hypergraph and there is an edge between two nodes if they belong to the same hyperedge).

Definition 2.2.5. Let D be a set of ground atoms. A *tree decomposition* of D is an undirected tree T with set of nodes $\mathcal{X} = \{X_1, ..., X_k\}$ where:

1. $\bigcup_i X_i = \mathsf{terms}(D)$;
2. for each atom $\mathbf{a}$ in D, there is $X_i \in \mathcal{X}$ s.t. $\mathsf{terms}(\mathbf{a}) \subseteq X_i$;
3. for each term t in D, the subgraph of T induced by the nodes X_i that contain t is connected ("running intersection property"). The width of a tree decomposition T is the size of the largest node in T, minus 1. The *treewidth* of D is the minimal width among all its possible tree decompositions.

A set of *Datalog*$^\exists$ rules Σ is called a *bounded treewidth set (bts)* if for any set of ground atoms D there exists an integer b such that, for any D' that can be Σ-derived from D (for instance with the chase algorithm), $treewidth(D') \leq b$. A *fes* is a *bts*, since the finite chase graph generated by a *fes* has a treewidth bounded by its size. Proving the decidability of QA with *bts* is not as immediate as with *fes*. Indeed, the proof relies on a theorem from Courcelle [39], that states that classes of first-order logic having the bounded treewidth model property are decidable. This proof does not (at least not directly) provide a halting algorithm. Very recently, a subclass of *bts* has been defined, namely *greedy bts (gbts)*, which is equipped with a halting algorithm [19]. This class is defined as follows. A derivation is said to be *greedy* if, for every rule application in this derivation, all the frontier variables (notice that the frontier variables of a rule are the universally quantified variables appearing in the head of the rule) not being mapped to terms of the initial set of ground atoms are jointly mapped to terms added by a single previous rule application. The third class, *finite unification set (fus)* [15], requires that the number of rewritings of q using the rules Σ is finite for any set of ground atoms D. More precisely, one considers only the "most general" rewritings of q, the other rewritings being useless for the querying task. Indeed, let q_1 and q_2 be two rewritings such that q_1 maps to q_2 (i.e., q_1 is "more general" than q_2): if q_1 does not map to D, neither does q_2. A set of *Datalog*$^\exists$ rules Σ is called a *fus* if for every set of ground atoms D, there is a finite set $\mathcal{Q}$ of Σ-rewritings of q such that, for any Σ-rewriting q' of q, there is an Σ-rewriting q'' in $\mathcal{Q}$ that maps to q'. Note that it may be the case that the set of the most general rewritings is finite while the set of rewritings is infinite. If Σ is a *fus*, then a backward chaining algorithm that builds rewritings of q in a breadth-first way, while maintaining a set $\mathcal{Q}$ of the most general Σ-rewritings built, and answers yes if an element of $\mathcal{Q}$ maps to D, necessarily halts in finite time. The *fes* and *fus* classes are not comparable, neither are *bts* (resp. *gbts*) and *fus*.

2.2.3 Actual languages and their syntactic conditions

Let us now enumerate the main concrete classes. Most of them implement one of the three preceding abstract behaviors; however, some concrete classes that are not *bts* neither *fus* have been exhibited very recently [29], we will mention them in this section. The typical *fes* concrete class is plain *Datalog*, where rules do not have any existential variable in their head, i.e., for any *Datalog* rule r, $\mathsf{vars}(\mathsf{head}(r)) \subseteq \mathsf{vars}(\mathsf{body}(r))$. Other names for this class are *range-restricted rules (rr)* [1], *full* implicational dependencies [37] and *total* tuple-generating dependencies [24]. These rules typically allow to express specialization relationships between concepts or relations in ontological languages, as well as properties of relations such as reflexivity, symmetry or transitivity. A special class is that of *disconnected (disc)* rules, which have an empty frontier [18]. A *disconnected* rule needs to be applied only once: any further application of it is redundant; this is why these rules are both *fes* and *fus*. The body and the head of a *disc*-rule may share constants, which allows to express knowledge about specific individuals. Apart from this use, this class is mostly useful in technical constructions. Other *fes* cases are obtained by restricting possible interactions between rules. These interactions have been encoded in two different directed graphs: a graph encoding variable sharing between positions in predicates and a graph encoding dependencies between rules. In the first graph, called *(position) dependency graph* [48], the nodes represent positions in predicates, i.e., the node $(\mathtt{p}, i)$ represents a position i in predicate $\mathtt{p}$. Then, for each rule r and each variable $\mathtt{X}$ in $\mathsf{body}(r)$ occurring in position $(\mathtt{p}, i)$, edges with origin $(\mathtt{p}, i)$ are built as follows: if $\mathtt{X} \in fr(r)$, there is an edge from $(\mathtt{p}, i)$ to each position of $\mathtt{X}$ in $\mathsf{head}(r)$; furthermore, for each existential variable $\mathtt{Y}$ in $\mathsf{head}(r)$ (i.e., $\mathtt{Y} \in \mathsf{vars}(\mathsf{head}(r)) \setminus fr(r)$) occurring in position $(\mathtt{r}, j)$, there is a special edge from $(\mathtt{p}, i)$ to $(\mathtt{r}, j)$. A set of *Datalog*$^{\exists}$ rules is said to be *weakly acyclic (wa)* if its position dependency graph has no circuit passing through a special edge. Intuitively, such a circuit means that the introduction of a null in a given position may lead to create another null in the same position, hence an infinite number of null terms. The weak-acyclicity property is a sufficient condition (but of course not a necessary condition) for the forward chaining to be finite [48, 45]. Recently, weak-acyclicity has been independently generalized in various ways, namely *safety* [74], *super-weak-acyclicity* [72] and *joint-acyclicity* [65], while keeping the forward chaining finiteness property. Note that *joint-acyclicity (ja)* is obtained by simply shifting the focus from positions to existential variables, hence replacing the position dependency graph by the *existential dependency graph*,where the nodes are the existential variables occurring in rules; this yields a finer analysis of potentially infinite creations of nulls. In the second graph, called *graph of rule dependencies* (GRD), the nodes represent rules and the edges represent dependencies between rules. Intuitively, we say that a rule r_2 depends on a rule r_1 if r_1 may bring knowledge that leads to a new application of r_2. GRD encodes dependencies between rules from a program P. It is a directed graph with P as the set of nodes and an edge (r_i, r_j) if r_j depends on r_i. The acyclicity of the GRD, noted $aGRD$ in Figure 2.1 ensures that the forward chaining, as well as the backward chaining, is finite, thus $aGRD$ is both a *fes* and a *fus* class. More generally, when all strongly connected components of the GRD have the property of being weakly-acyclic sets of rules (noted $wa\text{-}GRD$), then the forward chaining is finite; this class corresponds to the notion of a stratified chase graph in [44]. Let us now review *gbts* classes, which, intuitively, ensure that the derived facts have a tree-like structure that can be built in a greedy way. The notion of a *guarded* rule is inspired from guarded logic [8]. A rule r is *guarded (g)* if there is an atom a in its body (called a *guard*) that contains all variables from the body, i.e., $\mathsf{vars}(\mathsf{body}(r)) \subseteq \mathsf{vars}(\mathsf{a})$. A generalization of *guarded* rules is obtained by relaxing the guardedness property: a set of rules is *weakly guarded (wg)* if, for each rule *r*, there is $\mathsf{a} \in \mathsf{body}(r)$ (called a weak guard) that contains all *affected* variables from $\mathsf{body}(r)$. The notion of an affected variable is relative to the rule set: a variable is affected if it occurs only in affected predicate positions, which are positions that may contain a new null generated by forward chaining. The important property is that a rule application necessarily maps non-affected variables to terms from Δ_C.

A rule r is *frontier-one (fr1)* if its frontier is of size one (note that rules restricted to a frontier of size two still lead to undecidability). By noticing that the shape of derived facts depends only on how the frontier of rules is mapped (and not on how the whole body is mapped, since only the images of the frontier are used to apply a rule), one obtains a generalization of both $fr1$ and *guarded* rules: a rule r is *frontier-guarded (fg)* if there is an atom a in its body that contains all variables in its frontier, i.e., $\mathsf{vars}(fr(r)) \subseteq \mathsf{vars}(\mathsf{a})$. The same remark as for *guarded* rules can be made: only affected variables need to be guarded.One then obtains a generalization of both *wg* and *fg*: a set of rules is *weakly-frontier-guarded (wfg)* if, for each rule r, there is $\mathsf{a} \in \mathsf{body}(r)$ that contains all affected variables from $fr(r)$. In a very recent paper [65], the class $w(f)g$ is further generalized into *jointly-(frontier)-guarded (j-(f)g)*, by refining the notion of affected variable. Interestingly, [65] exhibits a class that is *bts* but neither *fes* nor *gbts*, namely *glutfrontier-guarded (glut-fg)*. This class generalizes both notions

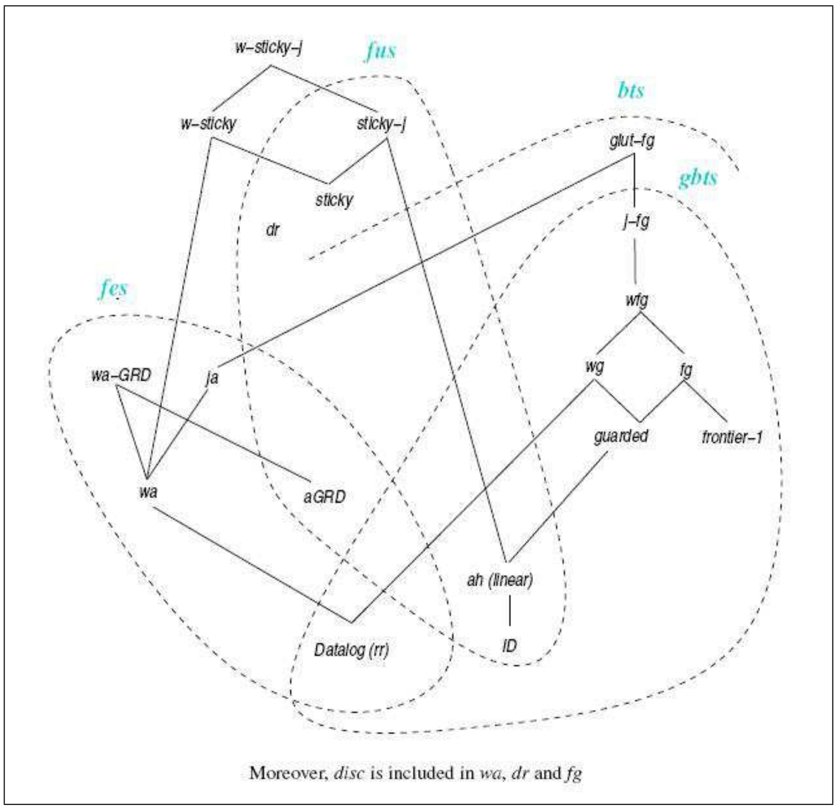


Figure 2.1: Inclusions between decidable cases

of *joint-acyclicity* (which itself generalizes *weak-acyclicity*) and *joint-(frontier)-guardedness*: a set of rules is *glut-(frontier)-guarded* if each rule has an atom in its body that contains all *glut* variables (occurring in its frontier). This class relies on a special method for eliminating existential quantifiers; instead of being replaced by functional terms as in skolemization, existential quantifiers are replaced by "flattened" functional terms encoded as additional arguments in predicates. Briefly, the *glut* variables are the variables that remain affected after this rule rewriting.

Whether the *gbts* class is concrete, i.e. recognizable, is not known yet. Note that *guarded* and *weakly guarded* rules were already provided with an algorithm [27, 28], but that it was not the case for *fr1*-rules and their generalizations up to *(j-(f)g)*-rules, which can now benefit from the algorithm for *gbts*. A *glut-fg* set of rules can be translated into an exponentially large *j-fg* set of rules, thus the *glut-fg* class is also provided with an algorithm.

About *fus* concrete cases, two classes are exhibited in [15]. The first class is that of *atomic-hypothesis rules (ah)* - where "hypothesis" stands for "body" - whose body is restricted to a single atom; these rules are also called *linear* TGDs

[1]. Since *ah*-rules are *fus*, there is a halting algorithm based on backward chaining, but, since they are also special guarded rules, there is also a halting algorithm based on forward chaining. *Atomic-hypothesis* rules are useful to express necessary properties of concepts or relations in ontological languages, without any restriction on the form of the head, i.e., by rules of the form $\mathbf{h} \leftarrow \texttt{c(X)}$ or $\mathbf{h} \leftarrow \texttt{r}(\texttt{X}_1, ..., \texttt{X}_k)$, where `c` is a concept, `r` a k-ary relation and $\mathbf{h}$ any head atom. Specific ah-rules translate the so-called *inclusion dependencies (ID)* in databases: the body and the head of these rules are each composed of a single atom, whose arguments are pairwise distinct variables.

The second class of rules, *domain-restricted rules (dr)*, constrains the form of the head: the head atom contains all or none of the variables in the body. For instance, a domain-restricted rule can express the so-called *concept-product*, argued to be a useful constructor for description logics in [86]: this operator allows to compute the cartesian product of two concepts by rules of the form `r(X,Y)` $\leftarrow$ `p(X),q(Y)` (e.g., `biggerThan(X,Y)` $\leftarrow$ `elephant(X),mouse(Y)`).

In [29], another concrete *fus* class is defined: *sticky* rules, which are incomparable with ah-rules and dr-rules. The *stickyness* property restricts multiple occurrences of variables (in the same atom or in distinct atoms, i.e., in joins) in the rule bodies. Variables that occur in rule bodies are marked according to the following procedure: (1) for each rule r, for each variable `X` in $\mathsf{body}(r)$, if there is an atom in $\mathsf{head}(r)$ that does not contain `X`, then mark every occurrence of `X` in $\mathsf{body}(r)$; (2) repeat the following step until a fixpoint is reached: for each rule r, if a marked variable in $\mathsf{body}(r)$ appears at position i of an atom whose predicate is `p` then, for every rule r' (including $r = r'$) and every variable `X` appearing in position i of an atom whose predicate is `p` in $\mathsf{head}(r')$, mark every occurrence of `X` in $\mathsf{body}(r')$. A program P is said to be sticky if there is no rule $r \in P$ such that a marked variable occurs in $\mathsf{body}(r)$ more than once. The above mentioned concept-product rule is obviously sticky since no variable occurs twice in the rule body.

Several generalizations of *sticky* rules are defined in [30]. All these classes are obtained by more sophisticated variable-marking techniques. *Weakly-sticky (w-sticky)* sets are a generalization of both *weakly-acyclic* sets and *sticky* sets: intuitively, if a marked variable occurs more than once in a rule body, then at least one of these positions has to be safe, i.e., only a finite number of terms can appear in this position during the forward chaining. *Sticky join (sticky-j)* sets generalize *sticky* sets. Finally, *weakly-sticky-join (w-sticky-j)* sets generalize both *weakly-sticky* sets and *sticky-join* sets. These classes are still incomparable with *dr*.

Figure 2.1 synthesizes inclusions between the preceding concrete decidable classes. All inclusions are strict and classes not related in the schema are indeed incomparable. Each class belongs to at least one of the abstract classes *fes*, *fus*, *gbts* and *bts*, except for the two recent classes *weakly-sticky* and *weakly-sticky-join*: indeed, they generalize both a *fes* but not *fus* nor *gbts* class, namely *wa*, and

Class $\mathcal{C}$	*Data Complexity*	*Combined Complexity*
glut-fg	**EXP**-hard	**3EXP**-complete
gbts	**EXP**-complete	**3EXP**-complete
j-fg	**EXP**-complete	**2EXP**-complete
wfg	**EXP**-complete	**2EXP**-complete
wg	**EXP**-complete	**2EXP**-complete
fg	**P**-complete	**2EXP**-complete
fr1	**P**-complete	**2EXP**-complete
guarded	**P**-complete	**2EXP**-complete
j-a	**P**-complete	**2EXP**-complete
wa, wa-GRD	**P**-complete	**2EXP**-complete
Datalog	**P**-complete	**EXP**-complete
linear	in $\mathbf{AC}_0$	**PSPACE**-complete
sticky, sticky-j	in $\mathbf{AC}_0$	**EXP**-complete
w-sticky, w-sticky-j	in $\mathbf{AC}_0$	**2EXP**-complete

Table 2.1: Combined and Data Complexities for the main concrete decidable classes

a *fus* but not *bts* class, namely *sticky*.

2.3 Complexity

In this section we deal with the complexity of $\text{QA}_{[\mathcal{C}]}$ problem as Definition 2.1.3, given a program P belonging to a class C of those introduced in Section 2.2.3. Two complexity measures are classically considered for $\text{QA}_{[\mathcal{C}]}$ problem: the usual complexity, called *combined* complexity, and *data* complexity. We talk about combined complexity of $\text{QA}_{[\mathcal{C}]}$ in general, and about data complexity of $\text{QA}_{[\mathcal{C}]}$, under the assumption that only the data, here $\mathsf{data}(P)$, are considered as part of the input while both q and $\mathsf{rules}(P)$ are considered fixed.

The latter complexity is relevant when the data size is much larger than the size of the rules and the query. Table 2.1 summarizes the combined and data complexity results for the main concrete classes mentioned in Section 2.2.3. Note that combined complexity is here without bound of the predicate arity (putting an upper-bound on the arity of predicates may decrease the complexity). By definition, all *fus* classes have polynomial data complexity, since the number of rewritten queries is not related to the data size. They are even *first-order rewritable*, which means that every query q can be rewritten as a first-order query q' using the set of rules of the program P, such that the evaluation of q produces the same set of answers as the evaluation of q' on P (i.e. $\mathsf{ans}_P(q) = \mathsf{ans}_P(q')$). An interest of first-order queries is that they can be encoded in SQL, which allows to use relational database systems, thus benefiting from their optimizations. Obviously, any Boolean query over a fus class can be rewritten as a first-order query. It is well-known in databases that deciding whether a first-order query is entailed by a database belongs to the class $\mathbf{AC}_0$ in data complexity. Several non-fus classes have polynomial data complexity: some $gbts$ classes, namely fg (and its subclasses $fr1$ and $guarded$), some fes classes, namely *wa-GRD* and ja (and subclasses $aGRD$, wa and *Datalog*) and some non-bts classes, namely *w-sticky-j* (and its subclass *w-sticky*). Note that relaxing guardedness into weak-guardedness leads to **EXP**-complete data complexity.

In this thesis, every class C of the Datalog$^\pm$ family will be indifferently denoted by C Datalog$^\pm$ or C *Datalog*$^\exists$. In general, we prefer the first notation when we refer to C as a class of the Datalog$^\pm$ framework, and the second when we want to remark the presence of existential quantifiers as an extension of standard *Datalog*. Moreover, we indifferently use both database and logic programming notations. In particular, we refer to a *Datalog*$^\exists$ program P as the union of a database D and a set of TGDs Σ, and vice versa.

Notice that, two additional features, that are important for representing ontologies, could be added to Datalog$^\pm$ without increasing complexity. These features are: *negative constraints* and a limited form of *equality-generating dependencies (EGDs)*. A negative constraint is a Horn clause whose body is not necessarily guarded and whose head is the truth constant *false* which we denote by $\perp$. For example, the requirement that a person *ID* cannot simultaneously appear in the `employee(ID,Name)` and in the `retired(ID,Name)` relation can be expressed by:

$$\perp \leftarrow \mathbf{employee}(\mathtt{X},\mathtt{Y}), \mathbf{retired}(\mathtt{X},\mathtt{Z}) \tag{2.3}$$

While negative constraints do add expressive power to Datalog$^\pm$, they are actually very easy to handle, and we next show that the addition of negative constraints does not increase the complexity of query answering (see [28] for a detailed analysis). It is also allowed a limited form of equality-generating dependencies, namely, *keys*, to be specified, but it is required that these keys be - in a precise sense - not conflicting with the existential rules of the Datalog$^\pm$ program. Because it had been shown that the interaction of TGDs and general EGDs leads to undecidability of query answering. In [28], a result from [31] about non-key-conflicting inclusion dependencies is lifted to the setting of arbitrary TGDs to prove that the keys that we consider do not increase the complexity. With these additions we have a quite expressive and still efficient version of Datalog$^{\pm}$. Notice that, these two features are not properly admitted by *Datalog*$^{\exists}$ syntax, but, in the following two sections we show how they can be integrated into two particular classes, *Guarded* and *Linear*, of the Datalog$^{\pm}$ framework without increasing complexity of query answering.

2.3.1 Adding (negative) constraints

In this section, we recall the extension of Datalog$^{\pm}$ with (negative) constraints, which are an important ingredient, in particular, for representing ontologies. A *negative constraint* (or simply *constraint*) is a first-order formula of the form:

$$\bot \leftarrow \mathbf{conj}_{[\mathbf{X}]} \tag{2.4}$$

where $\mathbf{conj}_{[\mathbf{X}]}$ is a (not necessarily guarded) conjunction of atoms. It is often also written as $\neg\mathbf{p}(\mathbf{X}) \leftarrow \mathbf{conj}'_{[\mathbf{X}]}$, where $\mathbf{conj}'_{[\mathbf{X}]}$ is obtained from $\mathbf{conj}_{[\mathbf{X}]}$ by removing the atom $\mathbf{p}(\mathbf{X})$. We implicitly assume that all sets of constraints are finite here.

Example 2.3.1. If the two unary predicates `male` and `female` represent two classes (also called concepts in DLs), we may use the constraint $\bot \leftarrow$ `male(X), female(X)` (or alternatively $\neg$`female(X)` $\leftarrow$ `male(X)`). to assert that the two classes have no common instances. Similarly, if additionally the binary predicate `mother` represents a relationship (also called a role in DLs), we may use $\bot \leftarrow$ `male(X),mother(X,Y)` to enforce that no member of the class `male` is the mother of another individual. Furthermore, if the two binary predicates `mother` and `father` represent two relationships, we may use the constraint $\bot \leftarrow$ `mother(X,Y),father(X,Y)` to express that the two relationships are disjoint.

Query answering on a database D under a set of TGDs Σ_T (as well as a set of EGDs Σ_E as introduced in the next section) and a set of constraints Σ_C can be done effortless by additionally checking that every constraint $\sigma = \bot \leftarrow \mathbf{conj}_{[\mathbf{X}]} \in \Sigma_C$ is satisfied in D and Σ_T , each of which can be done by checking that the BCQ $q_\sigma = \mathbf{conj}_{[\mathbf{X}]}$ evaluates to false on D and Σ_T . We write $D \cup \Sigma_T \models \Sigma_C$ iff every q_σ with $\sigma \in \Sigma_C$ evaluates to false in D and Σ_T. Thus, a BCQ q is true in D and Σ_T and Σ_C, denoted $D \cup \Sigma_T \cup \Sigma_C \models q$, iff (i) $D \cup \Sigma_T \models q$ or (ii) $D \cup \Sigma_T \not\models \Sigma_C$ (as usual in DLs). Therefore, $D \cup \Sigma_T \cup \Sigma_C \models q$ iff (i) $D \cup \Sigma_T \models q$ or (ii) $D \cup \Sigma_T \models q_\sigma$ for some $\sigma \in \Sigma_C$. As an immediate consequence, we have that constraints do not increase the data complexity of answering BCQs in the *guarded* (resp., *linear*) case.

2.3.2 Adding Equality-Generating Dependencies (EGDs) and Keys

In this section, we recall the addition of equality-generating dependencies (EGDs) to *Guarded* (and *Linear*) *Datalog*$^{\exists}$, which are also important when representing ontologies. Note that EGDs generalize *functional dependencies (FDs)* and, in particular, *key dependencies (or keys)* [1]. In [28] a result by [31] about *non-key-conflicting (NKC)* inclusion dependencies is transferred to the more general setting of *Guarded* Datalog$^{\pm}$. However, while adding negative constraints is effortless from a computational perspective, adding EGDs is more problematic: The interaction of TGDs and EGDs leads to undecidability of query answering even in simple cases, such that of functional and inclusion dependencies [36], or keys and inclusion dependencies (see, e.g., [31], where the proof of undecidability is done in the style of Vardi as in [61]). It can even be seen that a *fixed* set of EGDs and guarded TGDs can simulate a universal Turing machine, and thus query answering and even propositional ground atom inference is undecidable for such dependencies. For this reason, we consider a restricted class of EGDs, namely, *non-conflicting key dependencies (or NC keys)*, which show a controlled interaction with TGDs (and negative constraints), such that they do not increase the complexity of answering BCQs. Nonetheless, this class is sufficient for modeling ontologies (e.g., *in DL-Lite*, see Chapter 6). An *equality-generating dependency (or EGD)* σ is a first-order formula of the form:

$$\mathtt{X}_i = \mathtt{X}_j \leftarrow \mathbf{conj}_{[\mathbf{X}]} \tag{2.5}$$

where $\mathbf{conj}_{[\mathbf{X}]}$, called the body of σ, denoted $\mathsf{body}(\sigma)$, is a (not necessarily guarded) conjunction of atoms, and $\mathtt{X}_i$ and $\mathtt{X}_j$ are variables from Δ_V. We call $\mathtt{X}_i = \mathtt{X}_j$ the head of σ, denoted $\mathsf{head}(\sigma)$. Given a database schema R, such σ is satisfied in a database D for R iff, whenever there exists a homomorphism h such that $h(\mathbf{conj}_{[\mathbf{X}]}) \subseteq D$, it holds that $h(\mathtt{X}_i) = h(\mathtt{X}_j)$.

Example 2.3.2. The following formula σ is an equality-generating dependency:

$$\mathtt{Y} = \mathtt{Z} \leftarrow \mathtt{r}_1(\mathtt{X},\mathtt{Y}), \mathtt{r}_2(\mathtt{Y},\mathtt{Z}). \tag{2.6}$$

The database $D = \{\mathtt{r}_1(\mathtt{a},\mathtt{b}), \mathtt{r}_2(\mathtt{b},\mathtt{b})\}$ satisfies σ because every homomorphism h mapping the body of σ to D is such that $h(\mathtt{Y}) = h(\mathtt{Z})$. On the contrary, the database $D = \{\mathtt{r}_1(\mathtt{a},\mathtt{b}), \mathtt{r}_2(\mathtt{b},\mathtt{c})\}$ does not satisfy σ.

An EGD σ on R of the form $\mathtt{X}_i = \mathtt{X}_j \leftarrow \mathbf{conj}_{[\mathbf{X}]}$ is applicable to a database D for R iff there exists a homomorphism $\eta : \mathbf{conj}_{[\mathbf{X}]} \to D$ such that $\eta(\mathtt{X}_i)$ and $\eta(\mathtt{X}_j)$ are different and not both constants. If $\eta(\mathtt{X}_i)$ and $\eta(\mathtt{X}_j)$ are different constants in $\mathsf{dom}(D)$, then there is a *hard violation* of σ, and the chase *fails*. Otherwise, the result of the application of σ to D is the database $h(D)$ obtained from D by replacing every occurrence of a non-constant element $\mathtt{e} \in \{\eta(\mathtt{X}_i), \eta(\mathtt{X}_j)\}$ in D by the other element $\mathtt{e}'$ (if $\mathtt{e}$ and $\mathtt{e}'$ are both nulls, then $\mathtt{e}$ precedes $\mathtt{e}'$ in the lexicographic order). Note that h is a homomorphism, but not necessarily an endomorphism of D, since $h(D)$ is not necessarily a subset of D. But for the special class of TGDs and EGDs that we recall in this section, h is actually an endomorphism of D. The *chase* of a database D, in the presence of two sets Σ_T and Σ_E of TGDs and EGDs, respectively, denoted $\mathsf{chase}(D, \Sigma_T \cup \Sigma_E)$, is computed by iteratively applying (1) a single TGD once, according to the standard order and (2) the EGDs, as long as they are applicable (i.e., until a fixpoint is reached).

Example 2.3.3. Consider the following set of TGDs and EGDs $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$:

```
σ1:  ∃Z s(X,Y,Z)   ←   r(X,Y).
σ2:  Y = Z  ←   s(X,Y,Z).
σ3:  X = Y  ←   r(X,Y), s(Z,Y,Y).
```

Let D be the database $\{\mathtt{r(a,b)}\}$. In the computation of $\mathsf{chase}(D, \Sigma)$, we first apply σ_1 and add the fact $\mathtt{s(a,b,z_1)}$, where $\mathtt{z_1}$ is a null. Then, the application of σ_2 on $\mathtt{s(a,b,z_1)}$ yields $\mathtt{z_1} = \mathtt{b}$, thus turning $\mathtt{s(a,b,z_1)}$ into $\mathtt{s(a,b,b)}$. Now, we apply σ_3 on $\mathtt{r(a,b)}$ and $\mathtt{s(a,b,b)}$, and by equating $\mathtt{a} = \mathtt{b}$, the chase fails; this is a hard violation, since both $\mathtt{a}$ and $\mathtt{b}$ are constants in $\mathsf{dom}(D)$.

The following definition generalizes the notion of separability originally introduced in [31] to Datalog$^{\pm}$. Intuitively, the semantic notion of *separability* for EGDs formulates a controlled interaction of EGDs and TGDs/(negative) constraints, so that the EGDs do not increase the complexity of answering BCQs.

Definition 2.3.4. Let R be a relational schema, and Σ_T and Σ_E be sets of TGDs and EGDs on R, respectively. Then, Σ_E is *separable* from Σ_T iff for every database D for R, the following conditions (1) and (2) are both satisfied:

1. (i) If there is a hard violation of an EGD of Σ_E in $\mathsf{chase}(D, \Sigma_T \cup \Sigma_E)$, then there is also a hard violation of some EGD of Σ_E in D.

2. (ii) If there is no chase failure, then for every BCQ q, it holds that $\mathsf{chase}(D, \Sigma_T \cup \Sigma_E) \vDash q$ iff $\mathsf{chase}(D, \Sigma_T) \vDash q$.

Note that (2) is equivalent to: (2′) if there is no chase failure, then for every CQ q, it holds that $\mathsf{ans}_{D,\Sigma_T \cup \Sigma_E}(q) = \mathsf{ans}_{D,\Sigma_T}(q)$. Here, (2′) implies (2), since (2) is a special case of (2′), and the converse holds, since a tuple t over Δ is an answer for a CQ q to D and Σ iff the BCQ q_t to D and Σ evaluates to true, where q_t is obtained from q by replacing each free variable by the corresponding constant in t.

In [28], it is showed that adding separable EGDs to TGDs and constraints does not increase the data complexity of answering BCQs in the guarded and linear case. It follows immediately from the fact that the separability of EGDs implies that chase failure can be directly evaluated on D. Here, for disjunctions of BCQs $\mathcal{Q}$, $D \cup \Sigma \models \mathcal{Q}$ iff $D \cup \Sigma \models q$ for some BCQ q in $\mathcal{Q}$.

We next recall a sufficient syntactic condition of separability of EGDs introduce in [31]. We assume that the reader is familiar with the notions of a functional dependency (FD) (which informally encodes that certain attributes of a relation functionally depend on others) and a key (dependency) (which is informally a tuple-identifying set of attributes of a relation) [1]. Clearly, FDs are special types of EGDs. A key k of a relation r can be written as a set of FDs that specify that k determines each other attribute of r. Thus, keys can be identified with sets of EGDs. It will be clear from the context when we regard a key as a set of attribute positions, and when we regard it as a set of EGDs. The following definition generalizes the notion of "non-key-conflicting" dependency relative to a set of keys, introduced in [31], to the context of arbitrary TGDs.

Definition 2.3.5. Let k be a key, and σ be a TGD of the form $\exists \mathbf{Y}\ \mathbf{r}_{[\mathbf{X}' \cup \mathbf{Y}]} \leftarrow \mathbf{conj}_{[\mathbf{X}]}$ Then, k is *non-conflicting (NC)* with σ iff either (i) the relational predicate on which k is defined is different from $\mathbf{r}$, or (ii) the positions of k in $\mathbf{r}$ are not a proper subset of the $\mathbf{X}'$-positions in $\mathbf{r}$ in the head of σ, and every variable in $\mathbf{Y}$ appears only once in the head of σ. We say k is *non-conflicting (NC)* with a set of TGDs Σ_T iff k is NC with every $\sigma \in \Sigma_T$. A set of keys Σ_K is *non-conflicting (NC)* with Σ_T iff every $k \in \Sigma_K$ is NC with Σ_T .

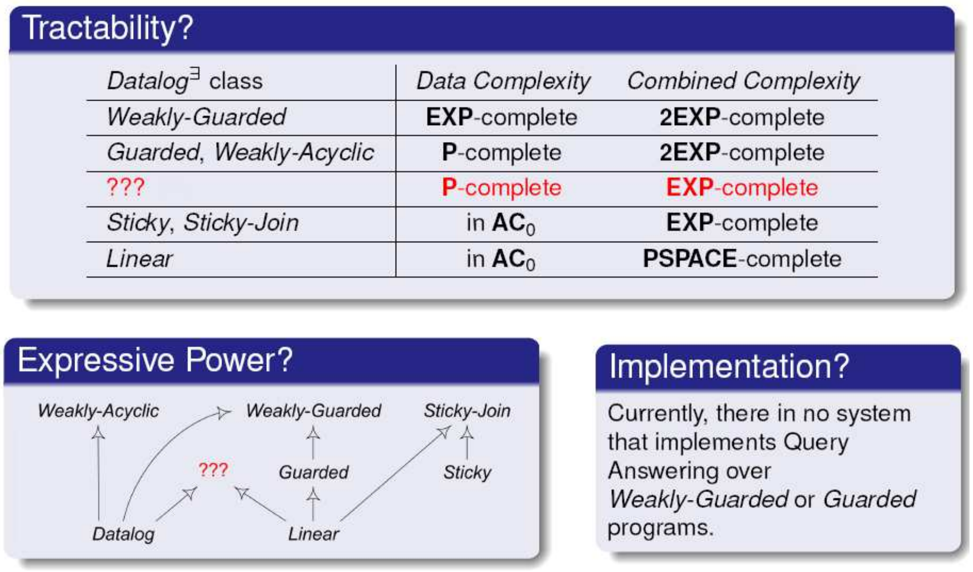
Example 2.3.6. Consider the four keys k_1, k_2, k_3, k_4 defined by the key attribute sets $K_1 = \{\texttt{r}[1], \texttt{r}[2]\}, K_2 = \{\texttt{r}[1], \texttt{r}[3]\}, K_3 = \{\texttt{r}[3]\}, K_4 = \{\texttt{r}[1]\}$, respectively, and the TGD $\sigma = \exists Z \texttt{r(X,Y,Z)} \leftarrow \texttt{p(X,Y)}$. Then, the head predicate of σ is $\texttt{r}$, and the set of positions in $\texttt{r}$ with universally quantified variables is $H = \{\texttt{r}[1], \texttt{r}[2]\}$. Observe that all keys but k_4 are NC with σ, since only $K_4 \subset H$. Roughly, every atom added in a chase by applying σ would have a fresh null in some position in K_1, K_2, and K_3, thus never firing k_1, k_2, and k_3, respectively.

It has been showed that the property of being NC between keys and TGDs implies their separability. The main idea behind the proof can be roughly described as follows. The NC condition between a key k and a TGD σ assures that either (a) the application of σ in the chase generates an atom with a fresh null in a position of k, and so the fact does not violate k (see also Example 2.3.6), or (b) the $\mathbf{X}'$-positions in the predicate $\mathbf{r}$ in the head of σ coincide with the key positions of k in $\mathbf{r}$, and thus any newly generated atom must have fresh distinct nulls in all but the key position, and may eventually be eliminated without violation. It then follows that the full chase does not fail. Since new nulls are all distinct, it also contains a homomorphic image of the TGD chase. Therefore, the full chase is in fact homomorphically equivalent to the TGD chase.

We conclude this section by recalling that in the NC case, keys do not increase the data complexity of answering BCQs under guarded (resp., linear) TGDs and constraints. However, we refer the reader to [28] for a detailed analysis of the addition of negative constraints and keys to Datalog$^{\pm}$.

2.4 Comparative analysis

In this section we show some local weaknesses of the Datalog$^{\pm}$ framework, that motivated our work. As introduced before, a key issue in ontology-based QA is the design of the language that is provided for specifying the ontological theory Σ. This language should balance expressiveness and complexity, and in particular it should possibly be: (1) intuitive and easy-to-understand; (2) QA-decidable (i.e., QA should be decidable in this language); (3) efficiently computable; (4) powerful enough in terms of expressiveness; and (5) suitable for an efficient implementation.



Tractability?

Datalog$^{\exists}$ class	*Data Complexity*	*Combined Complexity*
Weakly-Guarded	**EXP**-complete	**2EXP**-complete
Guarded, Weakly-Acyclic	**P**-complete	**2EXP**-complete
???	**P**-complete	**EXP**-complete
Sticky, Sticky-Join	in **AC**$_0$	**EXP**-complete
Linear	in **AC**$_0$	**PSPACE**-complete

Expressive Power?

$$\begin{tikzcd}
\text{Weakly-Acyclic} & & \text{Weakly-Guarded} & \text{Sticky-Join} \\
& ??? & \text{Guarded} \arrow[u] & \text{Sticky} \arrow[u] \\
\text{Datalog} \arrow[uu] \arrow[uurr, bend left] \arrow[ur] & & \text{Linear} \arrow[ul] \arrow[u] \arrow[uur] &
\end{tikzcd}$$

Implementation?

Currently, there in no system that implements Query Answering over *Weakly-Guarded* or *Guarded* programs.

Figure 2.2: Local weaknesses/shortcomings of the Datalog$^{\pm}$ family

In particular, if we consider the languages of the Datalog$^{\pm}$ family (introduced in Section 2.2), all of them satisfy the first two properties above (1 and 2). But, currently, each Datalog$^{\pm}$ language is missing at least one of the last three properties (3, 4 and 5). Indeed, analyzing Figure 2.1 and Table 2.1, we can observe that it is still missing a language that is suitable for an efficient implementation and that offers a good efficiency without renouncing to the expressiveness. This language should possibly be expressive enough to extend at least *Datalog* and *Linear* Datalog$^{\pm}$, while preserving tractable data complexity of QA. These two languages are important because the former offers some useful expressive axioms as for example “transitivity” and “concept product”, while the latter is very efficient and extends all the description logics of the *DL-Lite* family, as showed in [27, 28].

Figure 2.2 shows the weaknesses of the Datalog$^{\pm}$ family in more detail. Notice that, each language of the Datalog$^{\pm}$ framework showed in Figure 2.2 is the most representative language of the abstract class to which it belongs. The other classes of the family have been excluded because they are in fact combinations of these representative languages (see Figure 2.1). We have to say that *Linear* and *Sticky-Join* are very efficiently computable and extend all the description logics of the *DL-Lite* family. but they are not expressive enough to represent “transitivity” and “concept product” and to generalize *Datalog*. *Weakly-Guarded* Datalog$^{\pm}$ is very expressive, because it extends both *Linear* Datalog$^{\pm}$ and standard *Datalog*, but it is exponential even in data complexity. Moreover, to be complete w.r.t. QA, the CHASE ran on a program belonging to *Guarded* or *Weakly-Guarded* Datalog$^{\pm}$ requires the generation of a very high number of isomorphic atoms, therefore no (efficient) implementation has been realized yet.

Thus, it is clear that there is a gap in the Datalog$^{\pm}$ family, because if we need efficiency we have to renounce to the expressiveness and vice versa. In the work reported in this thesis, we introduce a new Datalog$^{\pm}$ fragment, called *Shy* (see Chapter 3), that combines positive aspects of different Datalog$^{\pm}$ languages and closes this gap. *Shy* represents an optimal trade-off between expressiveness and scalability in the scenario of *Datalog* with existential quantifiers. In fact, with respect to properties (1)–(5) above, this new class of programs behaves as follows: (1) it inherits the simplicity and naturalness of *Datalog*; (2) it is QA-decidable; (3) it is efficiently computable (tractable data complexity and limited combined-complexity); (4) it offers a good expressive power being a strict superset of *Datalog* and *Linear* Datalog$^{\pm}$; and (5) it is suitable for an efficient implementation, indeed we implement a bottom-up evaluation strategy for *Shy* programs inside the well-known DLV system. This extension of DLV is called DLV$^{\exists}$. We carried out an experimental analysis comparing a number of state-of-the-art systems for ontology-based QA with DLV$^{\exists}$. The results of our analysis confirm the effectiveness of DLV$^{\exists}$, which outperforms all other systems in the benchmark domain.

Chapter 3

Parsimonious and *Shy* programs

In this chapter we present the main theoretical contribution of this thesis, that is, the design of two novel *Datalog*$^\exists$ classes, called *Parsimonious* and *Shy*. First, we slightly modify the CHASE in order to define a novel semantic property, called *parsimony*. Next, we define a new class of *Datalog*$^\exists$ programs depending on the parsimony property, and we show interesting properties of this class. But, we prove that recognizing *parsimony* is undecidable. Therefore, we define a novel syntactic *Datalog*$^\exists$ class, called *Shy*, and we prove that this class enjoys the parsimony property. *Shy* significantly extends *Datalog* and *Linear Datalog*$^\exists$, while preserving the same (data and combined) complexity of query answering over *Datalog*, although the addition of existential quantifiers.

This chapter is structured as follows: in Section 3.1 we introduce the class of *Parsimonious* programs, as well as some of its properties; in Section 3.2 we present *Shy* and we show that conjunctive query answering over *Shy* is decidable; in Section 3.3 we deal with complexity and, finally, in Section 3.4 we compare *Shy* against other classes of the Datalog$^\pm$ family.

3.1 *Parsimony*: A novel semantic property ensuring decidability

In this section we introduce a new class of *Datalog*$^\exists$ programs, called *parsimonious programs*. Intuitively, the key idea behind this class is as follows. As already mentioned, the chase is a well-known procedure for constructing a universal model for a *Datalog*$^\exists$ program. But, unfortunately, this procedure does not always terminate. Thus, we first modify the standard version of the algorithm in order to get a new procedure that terminates on any *Datalog*$^\exists$ program. Next, we define the *parsimony* property that relies on this new version of the chase. In particular, we say that a *Datalog*$^\exists$ program P enjoys the *parsimony* property if every atom of $\mathsf{chase}(P)$ has an homomorphic representative atom in $\mathsf{pChase}(P)$. This property is very powerful because it allows us to carry out atomic QA against *Parsimonious* programs by considering only this new version of the chase procedure. It follows that atomic QA over *Parsimonious* programs is decidable.

Thus, first of all, we slightly modify the standard CHASE procedure introduced in Section 2.1.4.

Definition 3.1.1. For any *Datalog*$^\exists$ program P, *parsimonious chase* (PARSIM-CHASE(P) for short) is the procedure resulting by the replacement of operator $\not\models$ by $\not\Vdash$ in the condition of the if-instruction at step 5 in Procedure 1 CHASE(P). The output of PARSIM-CHASE(P) is denoted by $\mathsf{pChase}(P)$. □

Note that, differently from $\mathsf{chase}(P)$, here $\mathsf{pChase}(P)$ might not be a model any more. Based on Definition 3.1.1, we next define a new class of *Datalog*$^\exists$ programs depending on a novel semantic property, called *parsimony*.

Definition 3.1.2. A *Datalog*$^\exists$ program P is called *parsimonious* if $\mathsf{pChase}(P) \Vdash \mathbf{a}$, for each $\mathbf{a} \in \mathsf{chase}(P)$. *Parsimonious* next denotes the class of all parsimonious programs. □

We next show that atomic QA against a *Parsimonious* program can be carried out by the PARSIM-CHASE algorithm.

Proposition 3.1.3. *Algorithm* PARSIM-CHASE *over parsimonious programs is sound and complete w.r.t. atomic QA.*

Proof. Soundness follows, by Definition 3.1.1, since $\mathsf{pChase}(P) \subseteq \mathsf{chase}(P)$ holds. In fact, since each substitution is a homomorphism, then, given a set of atoms S and an atom $\mathbf{a}$, $S \models \mathbf{a}$ always entails $S \Vdash \mathbf{a}$. Conversely, $S \not\Vdash \mathbf{a}$ always entails $S \not\models \mathbf{a}$. Finally, $\mathsf{ans}(q, \mathsf{pChase}(P)) \subseteq \mathsf{ans}_P(q)$, for each CQ q.

For completeness, let P be a parsimonious program and q be an atomic query. To prove that $\mathsf{ans}_P(q) \subseteq \mathsf{ans}(q, \mathsf{pChase}(P))$ we observe that whenever $\sigma \in \mathsf{ans}_P(q)$, then $\mathsf{chase}(P) \models \sigma(q)$, namely there is a substitution σ' such that $\sigma'(\sigma(q)) \in \mathsf{chase}(P)$. But, by Definition 3.1.2, $\mathsf{pChase}(P) \Vdash \sigma'(\sigma(q))$, namely there is a homomorphism h such that $h(\sigma'(\sigma(q))) \in \mathsf{pChase}(P)$. Now, since each substitution is a homomorphism and since composition of homomorphisms is a homomorphism, we call h' the homomorphism $h \circ \sigma'$. Thus, $h'(\sigma(q)) \in \mathsf{pChase}(P)$. But, since $h' = h \circ \sigma'$ is actually a substitution, then $\mathsf{pChase}(P) \models \sigma(q)$, namely $\sigma \in \mathsf{ans}(q, \mathsf{pChase}(P))$. □

Now, before proving one of the main results of this section concerning decidability of atomic query answering against parsimonious programs, we show that the cardinality of $\mathsf{pChase}(P)$ is finite as well as the number of levels reached by PARSIM-CHASE.

Lemma 3.1.4. *Let P be a Datalog$^\exists$ program, α be the maximum arity over all predicate symbols in P, and Φ be a set of α nulls. Then, there is a one-to-one correspondence μ between $\mathsf{pChase}(P)$ and a subset of $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$ such that $\mathbf{a} \simeq \mu(\mathbf{a})$, for each $\mathbf{a} \in \mathsf{pChase}(P)$.*

Proof. First we observe that each atom in $\mathsf{pChase}(P)$, say $\mathbf{a}$, has at most α different nulls. Thus, after replacing the nulls of $\mathbf{a}$ with different nulls from Φ we obtain an isomorphic atom belonging to $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$. Now assume that two atoms $\mathbf{a}_1 \neq \mathbf{a}_2$ in $\mathsf{pChase}(P)$ had one common isomorph $\mathbf{b} \in \mathsf{base}(\mathsf{dom}(P) \cup \Phi)$, namely $\mathbf{a}_1 \simeq \mathbf{b}$ and $\mathbf{a}_2 \simeq \mathbf{b}$. This would clearly entail that $\mathbf{a}_1 \simeq \mathbf{a}_2$. But this is not possible since $\mathsf{data}(P)$ contains no pair of isomorphic atoms, and because PARSIM-CHASE (due to the introduction of operator $\Vdash$) does not allow any addition to $\mathsf{pChase}(P)$ of an isomorphic atom. Consequently, μ can be built by associating to each atom in $\mathsf{pChase}(P)$ one of its isomorphic atoms in $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$. □

Corollary 3.1.5. *Let P be a Datalog$^\exists$ program, and α be the maximum arity over all predicate symbols in P. Then, $|\mathsf{pChase}(P)| \leq |\mathsf{preds}(P)| \cdot (|\mathsf{dom}(P)| + \alpha)^\alpha$.*

Proof. This upperbound directly follows from Lemma 3.1.4 by considering the cardinality of $\mathsf{base}(\mathsf{dom}(P) \cup \Phi)$, where Φ is a set of α nulls. □

The following theorem claims that parsimony makes atomic query answering decidable.

Theorem 3.1.6. *Atomic query answering against* Parsimonious *programs is decidable.*

Proof. Proposition 3.1.3 ensures that atomic QA is sound and complete against $\mathsf{pChase}(P)$. Corollary 3.1.5 ensures that the cardinality of $\mathsf{pChase}(P)$ is finite, entailing that both PARSIM-CHASE stops after computing no more that $|\mathsf{pChase}(P)|$ levels, and the number of firing substitutions considered at step 4 of the algorithm is always finite. □

However, we now show that, unfortunately, recognizing parsimony is undecidable. Later, we introduce a recognizable fragment of parsimonious programs.

Theorem 3.1.7. *Checking whether a program is parsimonious is not decidable. In particular, it is **coRE**-complete.*

Proof. For the membership, given a *Datalog*$^\exists$ program P, we show that one can semi-decide whether P is not parsimonious. In fact, in such a case, there must exist by definition a level k such that, for each atom $\mathbf{a} \in \mathsf{chase}^k(P)$, $\mathsf{chase}^{k-1}(P) \Vdash \mathbf{a}$ but there is an atom $\mathbf{a}' \in \mathsf{chase}^{k+1}(P)$ such that $\mathsf{chase}^k(P) \nVdash \mathbf{a}'$. Thus, if a program is not parsimonious, then we can discover that by running the CHASE.

Algorithm 2 ORACLE-QA(P, q)

```
Input: Datalog^∃ program $P \wedge$ Boolean atomic query $q$
Output: true ∨ false
1. if (IS-PARSIMONIOUS$(P)$)
2.   return (pChase$(P) \models q$)
3. else
4.   $k :=$ firstAwakeningLevel$(P)$
5.   $P' := P \cup (\mathsf{chase}^k(P) - \mathsf{chase}^{k-1}(P))$
6.   return ORACLE-QA$(P', q)$
```

For the hardness part, we use Algorithm 2, called ORACLE-QA, that would solve the $\text{QA}_{[Datalog^\exists]}$ problem (which, by Proposition 2.1.6, is **RE**-complete) if the problem of checking whether a program is parsimonious was decidable.

In particular, given a *Datalog*$^\exists$ program P, we denote by IS-PARSIMONIOUS the Boolean terminating function deciding whether P is parsimonious or not; and by firstAwakeningLevel(P) the lowest level k reached by the CHASE such that $\mathsf{pChase}(P) \Vdash \mathrm{a}$ for each $\mathrm{a} \in \mathsf{chase}^{k-1}(P)$, and $\mathsf{pChase}(P) \nVdash \mathrm{a}$ for at least one $\mathrm{a} \in \mathsf{chase}^k(P)$. Finally, it is enough to show that the algorithm: (i) is sound, since P' only contains atoms from $\mathsf{chase}(P)$; (ii) is complete, since P' evolves to a parsimonious program after each execution of instruction 5 adding to P' at least one atom a such that $\mathsf{chase}^{k-1}(P) \nVdash \mathrm{a}$; (iii) terminates, since the cardinality of $\mathsf{pChase}(P)$ is finite (where P denotes the initial program), entailing that at most $|\mathsf{pChase}(P)|$ recursive calls can be activated. □

3.2 *Shyness*: A syntactic property guaranteeing *parsimony*

We next define a novel syntactic *Datalog*$^\exists$ class: *Shy*. Later, we prove that this class enjoys the parsimony property.

3.2.1 The *Shy* language: Definition and main properties

Calì, Gottlob, and Kifer (2008) introduced the notion of "affected position" (see Chapter 2) to know whether an atom with a null at a given position might belong to the output of the CHASE. Specifically, let a be an atom of arity k with a variable X occurring at position $i \in [1..k]$. Position i of a is marked as *affected* w.r.t. P if there is a rule $r \in P$ s.t. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathrm{a})$ and X is either an $\exists$-variable, or a $\forall$-variable s.t. $\mathtt{X}$ occurs in the body of r in affected positions only. Otherwise, position i is definitely marked as *unaffected*. However, this procedure might mark as affected some position hosting a variable that can never be mapped to nulls.

To better detect whether a program admits a firing substitution that maps a $\forall$-variable into a null, we introduce the notion of *null-set* of a position in an atom. More precisely, $\varphi^r_{\mathtt{X}}$ denotes the "representative" null that can be introduced by the $\exists$-variable $\mathtt{X}$ occurring in rule r. (If $(r, \mathtt{X}) \neq (r', \mathtt{X}')$, then $\varphi^r_{\mathtt{X}} \neq \varphi^{r'}_{\mathtt{X}'}$.)

Definition 3.2.1. Let P be a *Datalog*$^{\exists}$ program, $\mathbf{a}$ be an atom, and $\mathtt{X}$ a variable occurring in $\mathbf{a}$ at position i. The *null-set* of position i in $\mathbf{a}$ w.r.t. P, denoted by $\mathsf{nullset}(i, \mathbf{a})$, is inductively defined as follows. If $\mathbf{a}$ is the head atom of some rule $r \in P$, then $\mathsf{nullset}(i, \mathbf{a})$ is: (1) either the set $\{\varphi^r_{\mathtt{X}}\}$, if $\mathtt{X}$ is $\exists$-quantified in r; or (2) the intersection of every $\mathsf{nullset}(j, \mathbf{b})$ s.t. $\mathbf{b} \in \mathsf{body}(r)$ and $\mathtt{X}$ occurs at position j in $\mathbf{b}$, if $\mathtt{X}$ is $\forall$-quantified in r. If $\mathbf{a}$ is not a head atom, then $\mathsf{nullset}(i, \mathbf{a})$ is the union of $\mathsf{nullset}(i, \mathsf{head}(r))$ for each $r \in P$ s.t. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathbf{a})$. $\square$

Note that $\mathsf{nullset}(i, \mathbf{a})$ may be empty. A representative null φ *invades* a variable $\mathtt{X}$ that occurs at position i in an atom $\mathbf{a}$ if φ is contained in $\mathsf{nullset}(i, \mathbf{a})$. A variable $\mathtt{X}$ occurring in a conjunct $\mathbf{conj}$ is *attacked* in $\mathbf{conj}$ by a null φ if each occurrence of $\mathtt{X}$ in $\mathbf{conj}$ is invaded by φ. A variable $\mathtt{X}$ is *protected* in $\mathbf{conj}$ if it is attacked by no null. Clearly, each attacked variable is affected but the converse is not true.

We are now ready to define the new *Datalog*$^{\exists}$ class.

Definition 3.2.2. A rule r of a *Datalog*$^{\exists}$ program P is called *shy* w.r.t. P if the following conditions are both satisfied:

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

Finally, *Shy* denotes the class of all *Datalog*$^{\exists}$ programs containing only shy rules. $\square$

After noticing that a program is *Shy* regardless its ground facts, we give an example of program being not *Shy*.

Example 3.2.3. Let P be the following *Datalog*$^{\exists}$ program:

```
r1:  ∃Y u(X,Y) :- q(X).
r2:  v(X,Y,Z) :- u(X,Y), p(X,Z).
r3:  p(X,Y) :- v(X,Y,Z).
r4:  u(Y,X) :- u(X,Y).
```

Let $\mathbf{a}_1, \ldots, \mathbf{a}_9$ be the atoms of P in left-to-right/top-to-bottom order.
First, $\mathsf{nullset}(2, \mathbf{a}_1) = \{\varphi_{\mathrm{Y}}^{r_1}\}$. Next, this singleton is propagated (head-to-body) to $\mathsf{nullset}(2, \mathbf{a}_4)$ and $\mathsf{nullset}(2, \mathbf{a}_9)$. At this point, from $\mathbf{a}_9$ the singleton is propagated (body-to-head) to $\mathsf{nullset}(1, \mathbf{a}_8)$, and from $\mathbf{a}_4$ to $\mathsf{nullset}(2, \mathbf{a}_3)$, and so on, according to Definition 3.2.1. Finally, even if `X` is protected in r_2 since it is invaded only in $\mathbf{a}_4$, rule r_2, and therefore P, is not shy due to `Y` and `Z` that are attacked by $\varphi_{\mathrm{Y}}^{r_1}$ and occur in $\mathsf{head}(r_2)$. Moreover, it is easy to verify that P plus any fact for `q` does not belong to *Parsimonious*. □

Intuitively, the key idea behind this class is as follows. If a program is shy then, during a CHASE execution, nulls do not meet each other to join but only to propagate. Moreover, a null is propagated, during a given fire, from a single atom only. Hence, the *shyness* property, which ensures parsimony.

Theorem 3.2.4. *Shy* $\subset$ *Parsimonious.*

Proof. Let P be a *Shy* program. Assume that there exists a level k such that $\mathsf{pChase}(P) \Vdash \mathbf{a}$ for each $\mathbf{a} \in \mathsf{chase}^{k-1}(P)$, and $\mathsf{pChase}(P) \nVdash \mathbf{b}$ for at least one atom $\mathbf{b} \in \mathsf{chase}^{k}(P)$. Let $j < k - 1$ be the level where PARSIM-CHASE has stopped. Since $\mathbf{b} \in \mathsf{chase}^{k}(P) - \mathsf{chase}^{k-1}(P)$, then there must be at least one atom in $\mathsf{chase}^{k-1}(P) - \mathsf{chase}^{k-2}(P)$ that is necessary for firing a rule r to $\mathsf{chase}^{k-1}(P)$ to infer $\mathbf{b}$. Let σ be the firing substitution for r w.r.t. $\mathsf{chase}^{k-1}(P)$ used for inferring $\mathbf{b}$, and $\mathbf{a}_1, \ldots, \mathbf{a}_n$ be the body atoms of r. Clearly, $\mathsf{pChase}(P) \Vdash \sigma(\mathbf{a}_i)$ for each $i \in [1..n]$. Now, since P is shy then, by Definition 3.1.2, σ may map a variable into a null only if such a variable does not appear in two different atoms, and two different variables appearing in the head cannot be mapped to the same null. This means that if we consider the n homomorphisms $h_1, \ldots, h_n$ such that $h_i(\sigma(\mathbf{a}_i)) \in \mathsf{pChase}(P)$ for each $i \in [1..n]$, then we can take the union h of their restrictions on the $\exists$-variables of r without generating any conflict. But this is not possible because $h \circ \sigma$ is also a firing substitution for r on $\mathsf{pChase}(P)$ entailing the existence of an homomorphism from $\sigma(\mathsf{head}(r))$ to $h(\sigma(\mathsf{head}(r)))$. Finally, this entails an homomorphism from $\mathbf{b}$ to the atom inferred by the extension of $h \circ \sigma$. □

Corollary 3.2.5. *Atomic QA over Shy is decidable.*

We now show that recognizing parsimony is decidable.

Theorem 3.2.6. *Checking whether a program P is shy is decidable. In particular, it is doable in polynomial-time.*

Proof. First, the occurrences of $\exists$-variables in P fix the number h of nulls appearing in the null-sets of P. Next, let k be the number of atoms occurring in P, and α be the maximum arity over all predicate symbols in P. It is enough to observe that P allows at most $k * \alpha$ null-sets each of which of cardinality no greater than h. Finally, the statement holds since the null-set-construction is monotone and stops as soon as a fixpoint has been reached. $\square$

3.2.2 Conjunctive query answering over *Shy*

In this section we show that conjunctive QA against *Shy* programs is also decidable. To manage CQs, we next describe a technique called parsimonious-chase resumption, which is sound for any *Datalog*$^{\exists}$ program P, and also complete over *Shy*. Before proving formal results, we give a brief intuition of this approach. Assume that $\mathsf{pChase}(P)$ consists of the atoms $\mathtt{p}(\mathtt{c},\varphi)$, $\mathtt{q}(\mathtt{d},\mathtt{e})$, $\mathtt{r}(\mathtt{c},\mathtt{e})$. It is definitely possible that $\mathsf{chase}(P)$ contains also $\mathtt{q}(\varphi,\mathtt{e})$, which, of course, cannot belong to $\mathsf{pChase}(P)$ due to $\mathtt{q}(\mathtt{d},\mathtt{e})$. Now consider the CQ $q = \exists \mathtt{Y}\ \mathtt{p}(\mathtt{X},\mathtt{Y}), \mathtt{q}(\mathtt{Y},\mathtt{Z})$. Clearly, $\mathsf{pChase}(P)$ does not provide any answer to q even if P does. Let us both "promote" φ to constant in Δ_C, and "resume" the PARSIM-CHASE execution at step 3, in the same state in which it had stopped after returning the set C at step 10. But, now, since φ can be considered as a constant, then there is no homomorphism from $\mathtt{q}(\varphi,\mathtt{e})$ to $\mathtt{q}(\mathtt{d},\mathtt{e})$. Thus, $\mathtt{q}(\varphi,\mathtt{e})$ may be now inferred by the algorithm and used to prove that $\mathsf{ans}_P(q)$ is nonempty.

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

Definition 3.2.7. Let $P \in$ *Datalog*$^{\exists}$. The set $\mathsf{pChase}(P,0)$ denotes $\mathsf{data}(P)$, while the set $\mathsf{pChase}(P,k)$ denotes $\mathsf{pChase}(\mathsf{rules}(P) \cup \lceil \mathsf{pChase}(k-1) \rfloor)$, for each $k > 0$. $\square$

Clearly, the sequence $\{\mathsf{pChase}(P,k)\}_{k\in\mathbb{N}}$ is monotonically increasing; the limit of this sequence is denoted by $\mathsf{pChase}(P,\infty)$. The next lemma states that the proposed resumption technique is always sound w.r.t. QA, and that its infinite application also ensures completeness.

Lemma 3.2.8. $\mathsf{pChase}(P,\infty) = \mathsf{chase}(P)\ \forall P \in$ *Datalog*$^{\exists}$.

Proof. The statement holds since operator $\Vdash$ in PARSIM-CHASE behaves, on freezed nulls, as $\models$ in the CHASE. $\square$

Before proving that the PARSIM-CHASE algorithm over *Shy* programs is complete w.r.t. CQ answering after a finite number of resumptions, we need to introduce some more notation. The *chase-graph* for a *Datalog*$^\exists$ program P is the directed acyclic graph $G_P = \langle \mathsf{chase}(P), A\rangle$ where $(\mathbf{a}, \mathbf{b}) \in A$ iff $\mathbf{b}$ has been inferred by the CHASE through a firing substitution σ for a rule r where $\mathbf{a} \in \sigma(\mathsf{body}(r))$. Moreover, for a given set $S \in \mathsf{chase}(P)$, G_P^S denotes the maximal subgraph of G_P where a node may have no ingoing arc only if it belongs to S.

Lemma 3.2.9. *Let P be a Shy program, q be a CQ, $\sigma_a \in \mathsf{ans}_P(q)$, σ be a substitution proving that $P \vDash \sigma_a(q)$ holds, and $\mathbf{Y}_N$ be only and all the $\exists$-variables of q mapped by σ to nulls. Then, there is a substitution σ', proving that $P \vDash \sigma_a(q)$ holds, that maps at least one variable in $\mathbf{Y}_N$ to a term occurring in $\mathsf{pChase}(P)$.*

Proof. Let Φ contain the nulls occurring in $\sigma(q)$, B contain the atoms in $\mathsf{chase}(P)$ where the nulls of Φ have been introduced for the first time, and $\mathbf{b}_1, \ldots, \mathbf{b}_n$ be the atoms of B listed in the same order they have been inferred by the CHASE. Moreover, let Φ_i denote, for each $i \in [1..n]$, the subset of Φ of only and all the nulls that have been introduced for the first time in $\mathbf{b}_i$, and $\mathbf{a}_1$ an atom form $\mathsf{pChase}(P)$ such that $\mathbf{a}_1 \leq \mathbf{b}_1$. We build σ' in such a way that at least one variable in $\mathbf{Y}_N$ is mapped to some term occurring in $\mathbf{a}_1$. In particular, we build a set $A \subseteq \mathsf{chase}(P)$ and a homomorphism $h : \Phi \to \mathsf{terms}(A)$ such that $\mathbf{a}_1 \in A$ and, for each atom $\mathbf{b}$ in G_P^B containing at least a null from Φ, there is $h' \supseteq h$ such that $h'(\mathbf{b})$ belongs to G_P^A. Finally, $\sigma' = h \circ \sigma$.

We proceed by induction. More precisely, we construct A, G_P^A and h by progressively considering all the atoms of G_P^B in the same order they have been inferred by the CHASE. Initially, $A = \{\mathbf{a}_1\}$, G_P^A contains only node $\mathbf{a}_1$, and h maps each constant in $\sigma(q)$ to itself, and each null in Φ_1 occurring at position i in $\mathbf{b}_1$ to the i^{th} term of $\mathbf{a}_1$.

Base case: Let $\mathbf{b}$ be the first atom in G_P^B inferred by the CHASE, via a rule r, after $\mathbf{b}_1$. Let σ_r be the firing substitution for r used by the CHASE whose extension $\hat{\sigma_r}$ has produced $\mathbf{b}$. If $\sigma_r(\mathsf{body}(r))$ does not involve $\mathbf{b}_1$, then $\mathbf{b} = \mathbf{b}_2$ and we can choose any $\mathbf{a} \leq \mathbf{b}$ to extend A. On the contrary, if $\sigma_r(\mathsf{body}(r))$ involves $\mathbf{b}_1$, since P is shy, then there is also a firing substitution σ_r' for r, where $\mathbf{a}_1 \in \sigma_r'(\mathsf{body}(r))$ and $\sigma_r(\mathsf{body}(r)) - \{\mathbf{b}_1\} = \sigma_r'(\mathsf{body}(r)) - \{\mathbf{a}_1\}$. (Note that also in this case, $\mathbf{b} = \mathbf{b}_2$ might hold.) Clearly, if σ_r can be extended to infer a new atom $\mathbf{b}$, then either σ_r' can be extended to infer a new atom $\mathbf{a}$ or there is already some $\mathbf{a}$ such that $\{\mathbf{a}\} \vDash \sigma_r'(\mathsf{head}(r))$. But since a null in $\mathbf{a}$ but not in $\mathbf{b}$ either comes from $\mathbf{a}_1$ or it is fresh, then $\mathbf{a} \leq \mathbf{b}$. Finally, $\mathbf{a}$ is added to A, G_P^A is updated and, only in case $\mathbf{b} = \mathbf{b}_2$, h is updated according to $\mathbf{a}$.

Inductive hypothesis: After considering the first k atoms in G_P^B inferred by the CHASE, we assume that, for each such an atom $\mathbf{b}$ containing at least a null from Φ, there is $h' \supseteq h$ such that $h'(\mathbf{b})$ belongs to G_P^A.

Inductive step: Let $\mathbf{b}$ be the $(k+1)^{th}$ atom in G_P^B inferred by the CHASE, via a rule r. By using the same argument that was used in the Base case, we can extend A and G_P^A by an atom $\mathbf{a} \leq \mathbf{b}$. Moreover, if $\mathbf{b} = \mathbf{b}_i$ for some $i \in [2..n]$, then h is updated according to $\mathbf{a}$. The only difference here is that $\mathbf{b}$ may require more than one atom among the first k already inferred. $\square$

Therefore, a conjunctive query q may be not *Shy*. Nulls could meet each other to join in the positions of existentially quantified variables. Thus, PARSIM-CHASE is not sufficient for answering conjunctive queries. But, in light of Lemma 3.2.9, we prove that, given a conjunctive query q with n different $\exists$-variables, we can answer q by considering $\mathsf{pChase}(P, n+1)$.

Lemma 3.2.10. *Let $P \in$ Shy and q be a CQ with n different $\exists$-variables. Then, $\mathit{ans}_P(q) \subseteq \mathit{ans}(q, \mathit{pChase}(P, n+1))$.*

Proof. In Light of Lemma 3.2.9, in the worst case, to be sure that all the nulls involved by σ' are generated, we claim that it is enough to compute $\mathsf{pChase}(P, n)$ where n is the number of $\exists$-variables of q. With respect to Lemma 3.2.9, let $\mathtt{Y}$ be one of the variable in $\mathbf{Y}_N$ mapped by σ' to a term occurring in $\mathsf{pChase}(P)$. Assume that this term is a null say φ. After freezing φ, we could replace $\mathtt{Y}$ in q by φ to obtain q'. Clearly, $P \vDash \sigma_a(q)$ iff $P \vDash \sigma_a(q')$. However, The BCQ $\sigma_a(q')$ has an $\exists$-variable less than the BCQ $\sigma_a(q)$. Thus, we can use again the statement of Lemma 3.2.9 after replacing $\mathsf{pChase}(P)$ by $\mathsf{pChase}(P, 2)$ and q by q'. We can reiterate this process until the query has no $\exists$-variable, namely after $n-1$ resumptions producing $\mathsf{pChase}(P, n)$. Finally, by Definition 3.1.2, we are sure that $\mathsf{pChase}(P, n+1)$ contains all the atoms appearing in $\sigma_a(\sigma'(q))$. $\square$

Now, we are ready to demonstrate the decidability of conjunctive QA over *Shy*.

Theorem 3.2.11. *Conjunctive QA over Shy is decidable.*

Proof. Soundness follows by Lemma 3.2.8, completeness by Lemma 3.2.10, while termination by combining Theorem 3.1.6 and Definition 3.2.7. $\square$

The following example, after defining a *Shy* program P, shows that P imposes the computation of $\mathsf{pChase}(P, 3)$ to prove (after two resumptions) that a BCQ q containing two atoms and two variables is entailed by P.

Example 3.2.12. Let P denote the following *Shy* program.

`p(a,b). u(c,d).` r_1: `∃Z v(Z) :- u(X,Y).` r_2: `∃Y u(X,Y) :- v(X).`
r_3: `p(X,Z) :- v(X), p(Y,Z).` r_4: `p(X,W) :- p(X,Y), u(Z,W).`

Consider the BCQ q = `∃X,Y p(X,Y),u(X,Y)`. Figure 3.1 shows that q cannot be proved before two freezing.

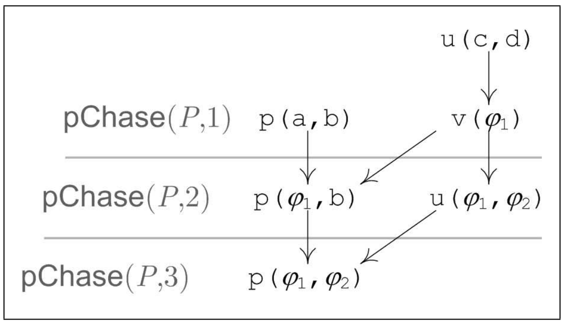


Figure 3.1: Snapshot of $\mathsf{pChase}(P,3)$ w.r.t. Example 3.2.12

3.3 Complexity analysis

In this section we study the complexity of *Parsimonious* and *Shy* programs. Moreover, let $\mathcal{C}$ be one of these classes, we talk about *combined complexity* of $\mathrm{QA}_{[\mathcal{C}]}$ in general, and about *data complexity* of $\mathrm{QA}_{[\mathcal{C}]}$ under the assumption that $\mathsf{data}(P)$ are the only input while both q and $\mathsf{rules}(P)$ are considered fixed. The results obtained from our analysis have been then compared, in Section 3.4, with those already proved for some representative Datalog$^{\pm}$ languages. We start with upper bounds.

Theorem 3.3.1. $\mathrm{QA}_{[\mathit{Parsimonious}]}$ *is in* ***P*** *(resp.,* ***EXP****) in data complexity (resp., combined complexity).*

Proof. Let P be a parsimonious program, α be the maximum arity over all predicate symbols in P, and β be the maximum number of body atoms over all rules in P. Since $|\mathsf{pChase}(P)| \leq |\mathsf{preds}(P)| \cdot (|\mathsf{dom}(P)| + \alpha)^{\alpha}$ by Corollary 3.1.5, then each rule admits at most $|\mathsf{pChase}(P)|^{\beta}$ different firing substitutions. Thus, all the firing substitutions are no more that $|P - \mathsf{data}(P)| \cdot |\mathsf{preds}(P)|^{\beta} \cdot (|\mathsf{dom}(P)| + \alpha)^{\alpha \cdot \beta}$. Moreover, for each firing substitution σ for a rule r, the algorithm has to check whether there is an homomorphism from $\hat{\sigma}(\mathsf{head}(r))$ to $\mathsf{pChase}(P)$. These checks are no more than $|P - \mathsf{data}(P)| \cdot |\mathsf{preds}(P)|^{2 \cdot \beta} \cdot (|\mathsf{dom}(P)| + \alpha)^{2 \cdot \alpha \cdot \beta}$. □

We now consider lower bounds, and thus completeness.

Theorem 3.3.2. *Both* $\mathrm{QA}_{[\mathit{Shy}]}$ *and* $\mathrm{QA}_{[\mathit{Parsimonious}]}$ *are* ***P****-complete (resp.,* ***EXP****-complete) in data complexity (resp., combined complexity).*

Proof. Since, by Theorem 3.2.4, a shy program is also parsimonious, then (i) upper-bounds of Theorem 3.3.1 hold for *Shy* programs as well; (ii) lower-bounds for $\mathrm{QA}_{[\mathit{Datalog}]}$ [41] also hold both for *Shy* and *Parsimonious* programs, by Theorem 3.4.1. □

3.4 *Shy* vs. other Datalog$^{\pm}$ classes

In this section we present a comparison between *Shy* and other Datalog$^{\pm}$ classes. Thus, we first recall the most representative QA-decidable subclasses of *Datalog*$^{\exists}$, discussed in Section 2.4. Then, we provide their precise taxonomy and the complexity of QA in each class, highlighting the differences to *Parsimonious* and *Shy*.

The best-known QA-decidable subclass of *Datalog*$^{\exists}$ is clearly *Datalog*, the largest $\exists$-free *Datalog*$^{\exists}$ class [1] which, notably, admits a unique and yet finite (universal) model enabling efficient QA.

As showed in Chapter 2, three abstract QA-decidable classes have been singled out, namely, *Finite-Expansion-Sets*, *Finite-Treewidth-Sets*, and *Finite-Unification-Sets* [16, 14]. Syntactic subclasses of *Finite-Treewidth-Sets*, of increasing complexity and expressivity, have been defined by Calì, Gottlob, and Kifer (2008). They are: (i) *Linear-Datalog*$^{\exists}$ where at most one body atom is allowed in each rule; (ii) *Guarded-Datalog*$^{\exists}$ where each rule needs at least one body atom that covers all $\forall$-variables; and (iii) *Weakly-Guarded-Datalog*$^{\exists}$ extending *Guarded* by allowing unaffected "unguarded" variables (see Section 3.2.1 for the meaning of unaffected). The first one generalizes the well known *Inclusion-Dependencies* class [60, 1], 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 w.r.t. 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, another class of *Datalog*$^{\exists}$, called *Sticky*, has been defined by Calì, Gottlob, and Pieris (2010a). Such a class enjoys very good complexity, encompasses *Inclusion-Dependencies*, but since it is FO-rewritable, it has limited expressive power and, clearly, does not include *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, Gottlob, and Pieris (2010b). For example, the *Sticky-Join* class preserves the sticky-complexity by also including *Linear-Datalog*$^{\exists}$. Both *Sticky* and *Sticky-Join* are subclasses of *Finite-Unification-Sets*.

Finally, in the context of data exchange, where a finite universal model is required, *Weakly-Acyclic-Datalog*$^{\exists}$, a subclass of *Finite-Expansion-Sets*, has been introduced [48]. 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 predicate symbol) containing several nulls in the same position. This class 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

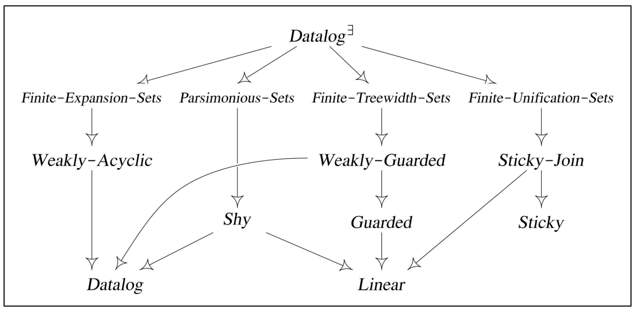


Figure 3.2: Taxonomy of representative Datalog$^{\pm}$ languages

recently proposed in this context [44, 72, 74, 53].

Figure 3.2 provides a precise taxonomy of the considered classes; while Table 3.1 summarizes the complexity of $\texttt{QA}_{[\mathcal{C}]}$, by varying $\mathcal{C}$ among the syntactic classes. In both diagrams, only *Datalog* is intended to be $\exists$-free; while *Datalog*$^{\exists}$ is the only undecidable language in the figure.

Theorem 3.4.1. *For each pair $\mathcal{C}_1$ and $\mathcal{C}_2$ of classes represented in Figure 3.2, the following hold: (i) there is a direct path from $\mathcal{C}_1$ to $\mathcal{C}_2$ iff $\mathcal{C}_1 \supset \mathcal{C}_2$; (ii) $\mathcal{C}_1$ and $\mathcal{C}_2$ are not linked by any directed path iff they are uncomparable.*

Proof. Relationships among known classes are pointed out by Mugnier (2011). *Shy* $\subset$ *Parsimonious* holds by Theorem 3.2.4. *Shy* $\supset$ *Datalog* $\cup$ *Linear* holds since *Datalog* programs only admit protected positions, while *Linear* ones only bodies with one atom. However, since there are both *Weakly-Acyclic* and *Sticky* programs being not *Parsimonious*, then both *Shy* and *Parsimonious* are uncomparable to *Finite-Expansion-Sets*, *Weakly-Acyclic*, *Finite-Unification-Sets*, *Sticky-Join* and *Sticky*. Now, to prove that *Shy* $\notin$ *Finite-Treewidth-Sets* we use the shy program

```
set1(a,a).    ∃V′ set1(V,V′) :- set1(X,V).
set2(b,b).    ∃V′ set2(V,V′) :- set2(X,V).
graphK(V1,V2) :- set1(V1,X), set2(V2,Y).
```

whose chase-graph G_P has no finite treewidth [27] since it contains a complete bipartite graph $K_{n,n}$ of $2n$ vertices – the treewidth of which is n [62] – where n is not finite. Finally, since there are *Guarded* programs that are not *Parsimonious*, then both *Shy* and *Parsimonious* are uncomparable to *Finite-Treewidth-Sets*, -*Weakly-Guarded* and *Guarded.* □

We care to notice that the proof of Theorem 3.4.1 uses the so called *concept product* to generate a complete and infinite bipartite graph. A natural and common example is

```
biggerThan(X,Y) :- elephant(X), mouse(Y).
```

Class $\mathcal{C}$	*Data Complexity*	*Combined Complexity*
Weakly-Guarded	**EXP**-complete	**2EXP**-complete
Guarded Weakly-Acyclic	**P**-complete	**2EXP**-complete
Datalog, *Shy* (*Parsimonious*)	**P**-complete	**EXP**-complete
Sticky, *Sticky-Join*	in $\mathbf{AC}_0$	**EXP**-complete
Linear	in $\mathbf{AC}_0$	**PSPACE**-complete

Table 3.1: Complexity of the $\texttt{QA}_{[\mathcal{C}]}$ problem

that is expressible in *Shy* if `elephant` and `mouse` are disjoint concepts. However, such a concept cannot be expressed in *Finite-Treewidth-Sets* and can be only simulated by a very expressive ontology language for which no tight worst-case complexity is known [86].

Thus, if we consider Figure 2.2 we can conclude that *Shy* closes that gap, resolving the weaknesses of the Datalog$^{\pm}$ framework enlighted in Section 2.4. *Shy* represents an optimal trade-off between expressiveness and scalability in the Datalog$^{\pm}$ framework. It offers a high expressive power, being a strict superset of both *Datalog* and *Linear Datalog*$^{\exists}$. It supports the standard first-order semantics for unrestricted CQs with existential variables, and it provides advanced properties (some of these beyond $\mathbf{AC}_0$), such as, *role transitivity*, *role hierarchy*, *role inverse*, and *concept products*. Notwithstanding its high expressive power, *Shy* preserves the same complexity as *Datalog* for QA (tractable data complexity and limited combined complexity) and it inherits the simplicity and naturalness of logic programing. Moreover, *Shy* is well suitable for an efficient implementation. Indeed, we implemented a bottom-up evaluation strategy for *Shy* programs (see Chapter 4) inside the well-known DLV system. We performed an experimental analysis (see Chapter 5), comparing our system, DLV$^{\exists}$, to a number of state-of-the-art systems for ontology-based query answering. The results evidence that DLV$^{\exists}$ is the most effective system in the benchmark domain for query answering in dynamic environments. In fact in this field the ontology is subject to frequent changes, making pre-computations and static optimizations inapplicable.

Chapter 4

A DLV-based implementation of QA over *Shy*

In this chapter we present the main practical contribution of this thesis, that is, the implementation of a bottom-up evaluation strategy for *Shy* programs inside the DLV system, whose computation is enhanced by a number of optimization techniques that we developed specifically for *Shy*. The resulting system is called DLV$^{\exists}$- a system for QA over Shy programs, which is profitably applicable for ontology-based QA.

This chapter is structured as follows: we first introduce the DLV system and its general architecture in Section 4.1; next, in Section 4.2, we analyze in more detail the implementation of DLV$^{\exists}$, showing all of its features; finally, in Section 4.3, we present the query-driven optimizations that we implemented in DLV$^{\exists}$.

4.1 The DLV system

DLV [68] is a deductive database system, based on *Disjunctive Logic Programming (DLP)* [38, 82, 75, 73, 49, 90, 69], which offers front-ends to several advanced KR formalisms. It has been conceived by an Italian-Austrian research team (of the University of Calabria and the Vienna University of Technology).

The DLV system embeds important results obtained in the fields of Artificial Intellingence, Databases and Computational Logic, and it is founded on a solid theoretical basis. It supports, beyond the classical constructs of DLP, the specification of "strong" constraints (conditions that have to be necessarily satisfied) and "weak" constraints (conditions that have to be possibly satisfied). It allows to resolve very hard problems by simply specifying, in a declarative way, the desired solutions. DLV is based on the extension of the Gelfond-Lifschitz semantics to the disjunctive case [49].

The first release of the system became available in 1997, after several years of theoretical research. It has been significantly improved over and over in the last years, incorporating new features and relevant optimisation techniques in all modules of the engine. Nowadays it represents the state of the art among Knowledge Representation and Reasoning (KRR) systems. Thanks to a long lasting theoretical and implementation effort, the language is now supported by an efficient run-time system that exploits techniques developed throughout the years by the scientific and industrial database community. By analyzing the nature of its specific input, the system is able to apply the techniques that better reflect the complexity of the problem at hand, so that easy problems are solved fast, while only harder problems involve methods of higher computational cost.

4.1.1 General architecture

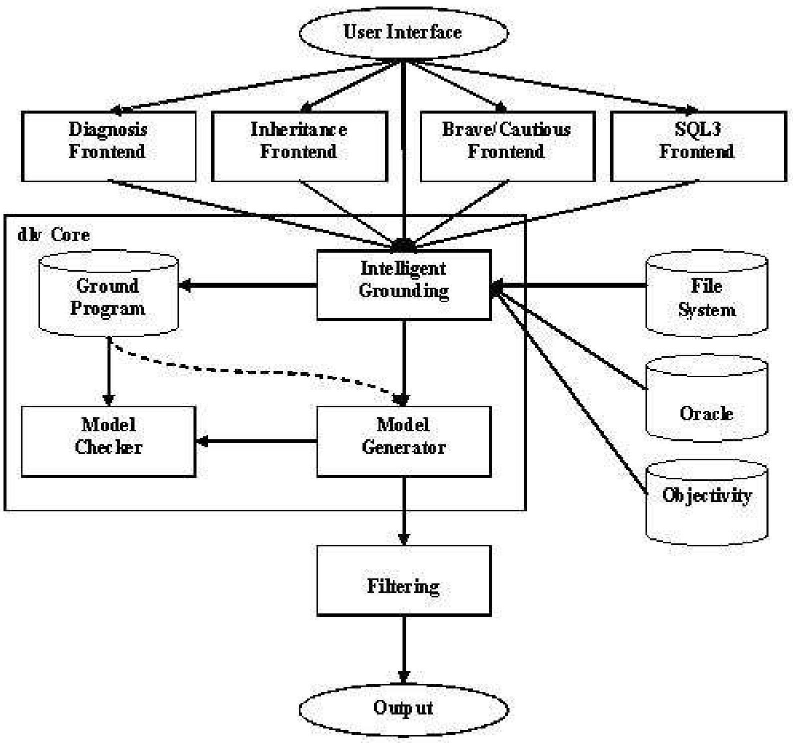


Figure 4.1: Architecture of the DLV system

The heart of DLV is the *DLV Core* module. It controls the execution of the entire system. This module pre-processes input programs and post-processes computed models.

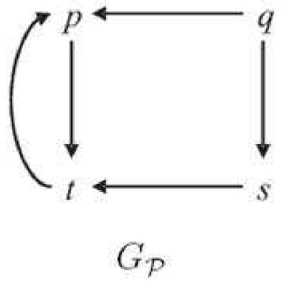
The system takes input data from user (mainly via command-line) and from file-system and/or from database systems. At the start the input (possibly transformed by a frontend) is send to the *Intelligent Grounding* module. This module produces a propositional program *P'* having the same answer sets of *P*. In general, *P'* is significantly smaller than *Ground(P)*.

Afterwards the *Model Generator (MG)* and *Model Checker (MC)* modules are invoked. They both work over the propositional program *P'*. The former produces an interpretation that is candidate to be an answer set, the latter checks whether it is really an answer set. This process is reiterated until there are no other answer sets or it has been computed the requested number of answer sets.

Notice that *Datalog* (and, in particular, *Datalog*$^{\exists}$) programs are solved directly by the Intelligent Grounding, because they have no negation and disjunction. Thus, the other two modules (Model Generator and Model Checker) of the DLV Core are not involved in the evaluation of *Shy* programs. In fact, the PARSIM-CHASE algorithm defined in Section 3.1 and the resumption technique introduced in Section 3.2.2 were integrated in the Intelligent Grounding. In the following section we illustrate the algorithm of the original Intelligent Grounding of DLV, and next, in Section 4.2, we show how we modified the original implementation in order to deal with *Shy* programs.

4.1.2 Intelligent grounding

Given an input DLP program *P*, the Intelligent Grounding module efficiently generates a ground instantiation which have the same answer sets of the standard instantiation, but, in general, it is much smaller than the latter. Notice that the size of the instantiation is a crucial aspect for the efficiency, because the computation takes exponential time (in the worst case) with respect to the size of the ground input program (generated by the instantiator).



$$\begin{tikzcd} p \arrow[d] & q \arrow[l] \arrow[d] \\ t \arrow[u, bend left] & s \arrow[l] \end{tikzcd}$$

$G_{\mathcal{P}}$

Figure 4.2: Dependency Graph

In order to generate a small ground program equivalent to *P*, the instantiation module of DLV produces ground instances of the rules containing only atoms that can be probably derived from *P*, so avoiding the combinatorial explosion that could be obtained by using a trivial algorithm that would consider all the atoms of the Herbrand Base. This can be done by taking into account some structural informations of the input program, concerning the dependencies between *IDB* predicates. Note that a predicate p is an *extensional database* predicate (*EDB* predicate) if all defining rules[1] for p are facts; otherwise, p is an *intensional database* predicate (*IDB* predicate). Now it will be given the definition of the *Dependency Graph* of *P*, that, intuitively, describes how predicates depends on each other.

Definition 4.1.1. Let P be a disjunctive logic program. The *Dependency Graph* of P is a directed graph $G_P =< N, E >$, where N is the set of nodes and E is the set of arcs. N contains a node for each IDB predicate of P, while E contains an arc $e = (p, q)$ if there is a rule r in P such that q occurs in the head of r while p appears in a positive literal of the body of r.

The graph G_P induces a subdivision of P in different subprograms (also called *modules*) so allowing a modular evaluation of the program itself. We say that a rule $r \in P$ defines a predicate p if p appears in the head of r.
For each *strictly connected component* (SCC[2]) C of G_P, the set of the rules defining all the predicates in C is called *module* of C and it is denoted by P_C[3].
In more detail, a rule r occurring in a module of P_C (i.e. r defines some predicate $q \in C$) is called *recursive* if there is some predicate $p \in C$ occurring in the positive body of r, otherwise, r is considered as an *exit rule*.

Example 4.1.2. Consider the following program P, where a is an EDB predicate:

r_1: $p(X,Y) \lor s(Y) \leftarrow q(X), q(Y), not\ t(X,Y).$
r_2: $q(X) \leftarrow a(X).$
r_3: $p(X,Y) \leftarrow q(X), t(X,Y).$
r_4: $t(X,Y) \leftarrow p(X,Y), s(Y).$

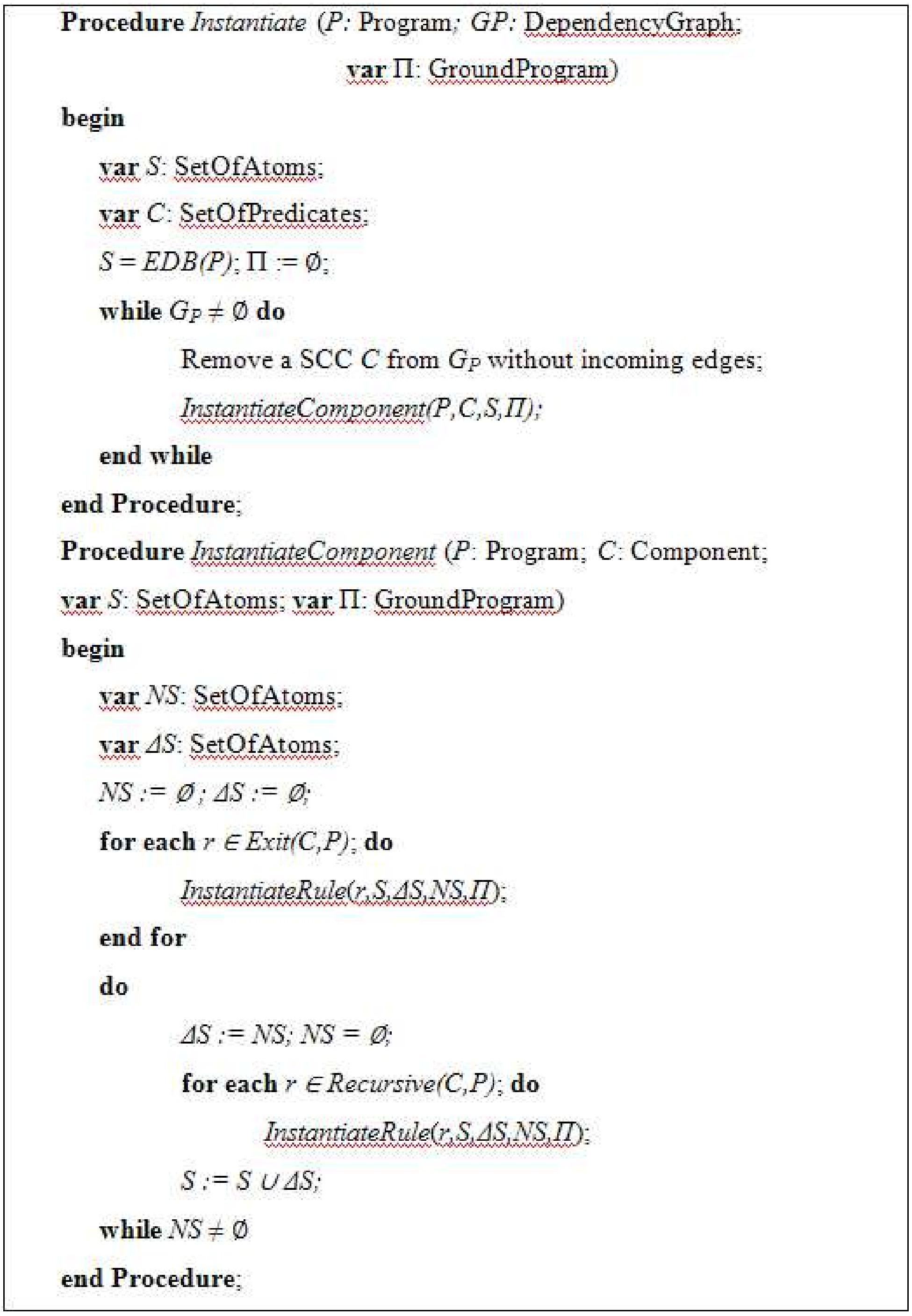
The graph G_P is illustrated in Figure 4.2; the strictly connected components of G_P are $\{s\}$, $\{q\}$, $\{p,t\}$. They correspond to the following 3 modules:

$\{\ p(X,Y) \lor s(Y) \leftarrow q(X), q(Y), not\ t(X,Y).\ \}$
$\{\ q(X) \leftarrow a(X).\ \}$
$\{\ p(X,Y) \leftarrow q(X), t(X,Y).$
$p(X,Y) \lor s(Y) \leftarrow q(X), q(Y), not\ t(X,Y).$
$t(X,Y) \leftarrow p(X,Y), s(Y).\ \}$

[1] A *defining rule* for a predicate p is a rule $r \in P$ such that some atom $\mathbf{p}(\mathbf{t})$ belongs to $head(r)$.

[2] We remember that a strictly connected component of a directed graph is a maximal subset of nodes, such that each node is reachable from each other.

[3] Notice that, since integrity constraints are considered as rules with the same head (i.e., a special symbol that does not appear anywhere in the program), they all belong to the same module.



```
Procedure Instantiate (P: Program; GP: DependencyGraph;
                              var Π: GroundProgram)
begin
    var S: SetOfAtoms;
    var C: SetOfPredicates;
    $S = EDB(P)$; $\Pi := \emptyset$;
    while $G_P \neq \emptyset$ do
            Remove a SCC C from $G_P$ without incoming edges;
            InstantiateComponent(P,C,S,Π);
    end while
end Procedure;
Procedure InstantiateComponent (P: Program; C: Component;
var S: SetOfAtoms; var Π: GroundProgram)
begin
    var NS: SetOfAtoms;
    var ΔS: SetOfAtoms;
    $NS := \emptyset$; $\Delta S := \emptyset$;
    for each $r \in Exit(C,P)$; do
            InstantiateRule(r,S,ΔS,NS,Π);
    end for
    do
            $\Delta S := NS$; $NS = \emptyset$;
            for each $r \in Recursive(C,P)$; do
                    InstantiateRule(r,S,ΔS,NS,Π);
            $S := S \cup \Delta S$;
    while $NS \neq \emptyset$
end Procedure;
```

Figure 4.3: DLV's instantiating procedure

Moreover, the first two modules contain no recursive rules, while the third one contains an exit rule, that is $p(X,Y) \vee s(Y) \leftarrow q(X), q(Y), not\ t(X,Y)$, and two recursive rules.

The dependency graph[4] induces a *partial ordering* on its SSC_s, defined as follows: for each pair A, B of SCC_s of G_P, it is said that B *directly depends* on A (denoted by $A \prec B$) if there is an arc outgoing from a predicate of A and going into a predicate of B; and, B depends on A if $A \prec_S B$, where $\prec_S$ denotes the transitive closure of the relation $\prec$.

Example 4.1.3. Consider the dependency graph illustrated in Figure 4.2; it can be noted that component $\{p,t\}$ depends on components $\{s\}$ and $\{q\}$, while $\{s\}$ depends only on $\{q\}$.

This ordering can be exploited for choosing an ordered sequence $C_1, ..., C_n$ of SCC_s of G_P (that is not unique, in general) such that whenever C_j depends on C_i, C_i precedes C_j in the sequence (namely $i < j$). Intuitively, this partial ordering allows to evaluate the program a module at a time, so that all data necessary for instantiating a module C_i have been already generated during the instantiation of the modules preceding C_i.

Now, it will be given a description of the instantiation process based on this principle. In the sequel, let P be a DLP program, $ANS(P)$ and $EDB(P)$ denote the set of all answer sets of P and the set of initial ground facts of P, respectively.

The procedure *Instantiate* showed in Figure 4.3 takes as input both the program P and its dependency graph G_P, and then it returns in output a set Π of ground rules containing only atoms which can be probably derived starting from P, such that $ANS(P) = ANS(\Pi \cup EDB(P))$. As already mentioned, the input program P is divided into modules corresponding to SCC_s of the dependency graph G_P. These modules are evaluated one at a time following the ordering induced by the dependency graph. The algorithm creates a new set of atoms S, that will contain the subset of *Herbrand Base* that is significant for the instantiation. Initially, $S = EDB(P)$ and $\Pi = \emptyset$. Then, a strictly connected component C with no ingoing arcs, is removed by G_P and the module corresponding to C is evaluated by the function *InstantiateComponent*. This ensures that modules are evaluated one at a time so that whenever $C_1 \prec_S C_2$, P_{C_1} is evaluated before P_{C_2}. The procedure *Instantiate* remains active until all the components of G_P have not been evaluated.

Example 4.1.4. Let P be the program of Example 4.1.2. The only component of G_P having no ingoing arc is $\{q\}$. Thus the module P_q is evaluated for first. Then, after $\{q\}$ is removed by G_P, $\{s\}$ becomes the only component of G_P having no

[4]We remember that dependency graph does not take into account negative dependencies.

ingoing arc, thus it is evaluated. After the evaluation and the removal of $\{s\}$ by G_P, $\{p, t\}$ is processed, so completing the instantiation process.

While the procedure *InstantiateComponent* takes as input both the component C that has to be instantiated and S, and then for each atom a belonging to C and for each rule r defining a, it computes the ground instance of r containing only atoms that can be probably derived from P. At the same time, the procedure updates the set S by adding atoms appearing in the head of the rules of Π. Finally, each rule r in C is processed by the procedure *InstantiateRule*. This procedure, starting from the set of atoms that have been demonstrated to be significant so far, builds all the ground instances of r, it adds them to Π and it marks as significant the atoms of the head of the new ground rules generated. It is worth noting that a disjunctive rule r could appear in modules of different components. Thus, before processing r, *InstantiateRule* verifies whether r has been already "grounded" during the instantiation process of another component. This ensures that a rule is processed only once inside a unique module of the program. Concerning recursive rules, they are processed many times following a semi-na$\ddot{i}$ve tecnique, where, during a generic iteration n, only significant informations derived during the iteration $n - 1$ are used. This is implemented by using a significant partitioning of atoms into 3 sets: $\Delta S, S$ and NS. NS is filled by atoms derived during the current iteration (n); ΔS contains atoms computed during the previous iteration ($n - 1$); finally S contains atoms obtained previously (namely from iteration $n - 2$ backwards). Initially ΔS and NS are empty and exit rules contained in the module C are evaluated through a single call to procedure *InstantiateRule*; finally, recursive rules are evaluated through a *do-while* cycle. At the beginning of each iteration, NS is assigned to ΔS, i.e., new informations derived during the iteration n are considered significant for the next iteration ($n + 1$). Then, the method *InstantiateRule* is invoked for each recursive rule r and, at the end of each iteration, ΔS (since it has been already exploited) is added to S. The procedure stops when no new informations are derived during the current iteration (namely when $NS = \emptyset$).

Theorem 4.1.5. *Let P be a disjunctive logic program, and Π be the ground program generated by algorithm* Instantiate*. We have that $ANS(P) = ANS(\Pi \cup EDB(P))$ (i.e., P and $\Pi \cup EDB(P)$ have the same answer sets).*

4.2 DLV$^\exists$: Design and implementation

We implemented a system for answering conjunctive queries over *Shy* programs (it actually works on any parsimonious program) on top of the DLV system introduced in Section 4.1. The system, called DLV$^\exists$, efficiently integrates the PARSIM-CHASE algorithm defined in Section 3.1 and the resumption technique introduced in Section 3.2.2, in the DLV system. Following the DLV philosophy, it has been designed as an in-memory reasoning system.

To answer a CQ q against a *Shy* program P, DLV$^{\exists}$ carries out the following steps.

Skolemization.

$\exists$-variables in rule heads are managed by skolemization. Given a head atom $\mathbf{a} = \texttt{p}(t_1, \ldots, t_k)$, let us denote by $\mathsf{fpos}(\texttt{Y}, \mathbf{a})$ the position of the first occurrence of variable Y in $\mathbf{a}$. The skolemized version of $\mathbf{a}$ is obtained by replacing in $\mathbf{a}$ each $\exists$-variable Y by $f^{\mathtt{p}}_{\mathsf{fpos}(\texttt{Y},\mathbf{a})}(t'_1, \ldots, t'_k)$ where, for each $i \in [1..k]$, t'_i is either $\#_{\mathsf{fpos}(t_i,\mathbf{a})}$ or t_i according to whether t_i is an $\exists$-variable or not, respectively. Every rule in P with $\exists$-variables is skolemized in this way, and skolemized terms are interpreted as functional symbols [33] within DLV$^{\exists}$.

Example 4.2.1. The *Datalog*$^{\exists}$ rule

```
∃X,Y p(Z,X,W,Y) :- s(Z,W).
```

is skolemized in

$\texttt{p(Z,}t_1\texttt{,W,}t_2\texttt{) :- s(Z,W).}$

where $t_1 = f^{\mathtt{p}}_2(\texttt{Z},\#_2,\texttt{W},\#_4)$, $t_2 = f^{\mathtt{p}}_4(\texttt{Z},\#_2,\texttt{W},\#_4)$. □

The skolemization task is carried out at parsing time. Every rule is first rewritten according to the criteria above, and, then, it is stored. Thus, the parser of DLV has been extended in order to accomodate existential quantifiers in rule heads. In the following, we illustrate the syntax of existential quantifiers in DLV$^{\exists}$ by showing a list of example programs executable by the system.

Employee and department

Consider an employee database, which stores information about managers, employees, and departments, where managers may supervise employees and direct departments, and employees may work in a department. The relational schema R consists of the unary predicates $manager$ and $employee$ as well as the binary predicates $directs$, $supervises$, and $works\_in$ with obvious semantics. In this context, a *Shy* program encoding this domain could consist of the following rules:

- every manager is an employee:

```
employee(M) :- manager(M).
```

- every manager directs at least one department:

```
#exists{P}  directs(M,P) :- manager(M).
```

- every employee supervising a manager is a manager:

```
manager(E) :- employee(E), supervises(E,E′), manager(E′).
```

- every employee who directs a department is a manager, and supervises at least another employee who works in the same department:

```
#exists{E′} aux(E,E′,P) :- employee(E), directs(E,P).
manager(E) :- aux(E,E′,P).
supervises(E,E′) :- aux(E,E′,P).
works_in(E′,P) :- aux(E,E′,P).
```

Father and person

Consider a family database, which stores information about fathers and persons, where every person must have a father, which has to be a person as well. In this context, a *Shy* program implementing this issue could be:

- every person has a father:

```
#exists{X} father(X,Y) :- person(Y).
```

- every father has to be a person as well:

```
person(X) :- father(X,Y).
```

- the individual “pierfrancesco” is a person:

```
person(``pierfrancesco'').
```

Data loading and filtering.

Since DLV$^\exists$ is an in-memory system, it needs to load input data in memory before the reasoning process can start. In order to optimize the execution, the system first singles out the set of predicates which are needed to answer the input query, by recursively traversing top-down (head-to-body) the rules in P, starting from the query predicates. This information is used to filter out, at loading time, all the facts belonging to predicates certainly irrelevant for answering the input query.

Notice that, the application of this optimization tecnique is optional. In particular, data filtering is automatically activated only if the input to DLV$^\exists$ is structured according to the following guidelines:

- input files are distinguished between rule files (the so-called TBox in ontologies) and data files (the ABox); the extension ".rul" is used for TBox's files, and ".data" for ABox's ones;
- there is a file, “p.data”, for each predicate name “p” of the ABox storing all and only the assertions regarding “p” (notice that, the file has to be named with the same name of the predicate);
- an eventual query in input is expressed in a rule file.

If the input is organized in this way, DLV$^{\exists}$ first scans the rule files, and then (by considering the input query and the TBox) it singles out the set of the predicates that have to be considered in order to answer the input query. Thus, it loads only data files whose names are in this predicate set. The other part of the ABox is totally ignored by DLV$^{\exists}$ parser. However, this organization of the input data files is not mandatory. But, if the previous suggestions are not followed the optimization on the data loading task will not be applied.

pChase Computation.

After the skolemization, and loading phases, the system computes $\mathsf{pChase}(P)$ as defined in Section 3.1. Since $\exists$-variables have been skolemized, the rules are safe and can be evaluated in the usual bottom-up way; but, according to $\mathsf{pChase}(P)$, the generation of homomorphic atoms should be avoided. To this end, each time a new head-atom $\mathbf{a}$ is derivable, the system verifies whether an homomorphic atom had been previously derived, where each skolem term is considered as a null for the sake of homomorphisms verification. In the negative case, $\mathbf{a}$ is derived; otherwise it is discarded.

If the input query is atomic, then $\mathsf{pChase}(P)$ is sufficient to provide an answer (see Proposition 3.1.3); otherwise, the fixpoint computation should be resumed several times (see Lemma 3.2.10). In this case, every null (skolem term) derived in previous reiterations is *freezed* (see Section 3.2.2) and considered as a standard constant; in our implementation, this is implemented 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.

The PARSIM-CHASE procedure is integrated in the Intelligent Grounding module of DLV. The homomorphism verification (see Section 2.1.1) is implemented in the function *InstantiateRule* (see Figure 4.3). At the beginning of the function, we verify whether an homomorphism of the head atom of the current rule had been previously derived.

The resumption mechanism is implemented as follows. The procedure *Instantiate* (see Figure 4.3) is invoked as many times as the number k of resumptions stated by Lemma 3.2.10. In particular, we attach a “resumption-phase”, k_{curr}, to the grounding module, representing the number of times that *Instantiate* has been invoked so far.

During each execution of the procedure *Instantiate*, the skolem terms that play the role of nulls are identified by verifying whether the level of the skolem term is equal to current resumption-phase. If they are equal, the skolem term will be considered as a null in homomorphism verifications of that iteration, otherwise as a constant.

Query answering.

After the fixpoint is resumed k times, the answers to query q are given by $\mathsf{ans}(q, \mathsf{pChase}(P, k+1))$.

In DLV$^{\exists}$ syntax, queries consist of one or more literals, which must be separated by commata and terminated by a question mark. Note that if you want to express some existentially quantified variables in the query you have to follow the syntax seen above for program rules. Only one query per program is considered. If you specify more than one query, only the last one will be considered and DLV$^{\exists}$ will issue an appropriate message.

Example 4.2.2. The following three are ground queries,

```
a ?
b, c ?
a2(h,i,j,k), b1(1,2,3) ?
```

while the next three are non-ground queries.

```
a(X) ?
#exists{X,Y} a(X), b(X,Y), c(Y,Z) ?
#exists{X} a2(Z,i,X,k), b1(X,Z,3) ?
```

4.3 Optimizations

In this section, we present some additional optimization tecniques implemented in DLV$^{\exists}$. The query is the central point in every system for QA and optimizations are designed and developed around it. Thus, we integrate two different (query-oriented) optimization tecniques in DLV$^{\exists}$: (1) we further optimize the number of times that the PARSIM-CHASE has to be resumed for answering conjunctive queries, in fact, we reduce the bound stated in Lemma 3.2.10 by considering the

Algorithm 3 RESUMPTION-LEVEL(q, P)

```
Input: A CQ $q = \exists \mathbf{Y}\ \mathbf{conj}_{[\mathbf{X} \cup \mathbf{Y}]}$ and a program $P$
Output: The number of needed resumptions for $q$ and $P$.
1. $\mathbf{Y}_* := \mathbf{Y}$
2. for each $\mathtt{Y} \in \mathbf{Y}$ do
3.   if $\mathtt{Y}$ is protected in $q$ OR $\mathtt{Y}$ occurs in only one atom of $q$
4.     remove$(\mathtt{Y}, \mathbf{Y}_*)$
5. return $|\mathbf{Y}_*|$
```

structure of the query; (2) we optimize the DLV$^\exists$ computation by “pushing-down” the bindings coming from possible query constants, in order to exclude some facts and rules which are not needed for the query at hand. To this end, the program is rewritten by a variant of the well-known magic-set optimization technique [40], 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 program, being equivalent to P for the given query, that can be evaluated more efficiently. In the following, P denotes the program that has been rewritten by magic-sets. A more detailed description of the Magic-Sets tecnique implemented in DLV$^\exists$ is reported in Section 4.3.2.

4.3.1 Optimal resumption level

If the input query is atomic, then $\mathsf{pChase}(P)$ is sufficient to provide an answer (see Proposition 3.1.3); otherwise, the fixpoint computation should be resumed several times.

In Section 3.2.2, we proved that resuming the computation k times is “sufficient” for answering a generic CQ with k different $\exists$-variables (see Lemma 3.2.10). However, this number of resumptions is not always “necessary”. In fact, in our implementation, this number is further reduced by Algorithm 3 considering the structure of the query w.r.t. P.

Lemma 4.3.1. *Let $P \in$ Shy, q be a CQ and k be the number of needed resumptions for q on P returned by Algorithm 3. Then, $\mathsf{ans}_P(q) \subseteq \mathsf{ans}(q, \mathsf{pChase}(P, k+1))$.*

Proof (Sketch). Let $P \in$ *Shy*, q be a CQ, $\sigma_a \in \mathsf{ans}_P(q)$, σ be a substitution proving that $P \models \sigma_a(q)$ holds, and $\mathbf{X}$ be only and all the $\exists$-variables of q mapped by σ to nulls. Then, there is a substitution σ', proving that $P \models \sigma_a(q)$ holds, that maps at least one variable in $\mathbf{X}$ to a term occurring in $\mathsf{pChase}(P)$. Thus, in the worst case, to be sure that all the nulls involved by σ' are generated, it is enough to compute $\mathsf{pChase}(P, n)$ where n is the number of $\exists$-variables of q. Moreover,

$\mathsf{pChase}(P, n+1)$ contains the atoms for σ'. In the sequel, let $\mathtt{X}$ be an existential variable of the query. If $\mathtt{X}$ occurs in only one atom of q and $\mathtt{X} \notin \mathbf{X}$, σ maps $\mathtt{X}$ to a constant value. This can be resolved by $\mathsf{pChase}(P)$. Otherwise, if $\mathtt{X}$ occurs in only one atom of q and $\mathtt{X} \in \mathbf{X}$, generating the specific null mapped on $\mathtt{X}$ by σ is not necessary, because that null does not meet other nulls to join. Finally, if $\mathtt{X}$ occurs in more than one atom of q but it is protected in q, then $\mathtt{X}$ can be mapped by σ only to a constant value. Note that, joins on constant values are resolved by $\mathsf{pChase}(P)$. Thus, we are sure that all the nulls involved by σ' are generated already by $\mathsf{pChase}(P, n-1)$ in every case considered above. $\square$

4.3.2 Magic-Sets

In this section we focus on a particular optimization we developed in DLV$^{\exists}$, the Magic-Sets tecnique. The original Magic-Sets technique was introduced for *Datalog* [20]. Many authors have addressed the issue of extending Magic-Sets to broader languages, including nonmonotonic negation [46], disjunctive heads [52, 4], and uninterpreted function symbols [32, 5]. In order to bring Magic-Sets to the more general framework of *Datalog*$^{\exists}$, two main difficulties must be faced: the first is, obviously, the presence of existentially quantified variables; the second regards the correctness proof of a Magic-Sets rewriting. In fact, while a *Datalog* program can be associated with a universal model that comprises finitely many atoms, the universal model of a *Datalog*$^{\exists}$ program comprises in general infinitely many atoms. In this work, we designed a Magic-Sets rewriting algorithm handling existential quantifiers, and thus suitable for *Datalog*$^{\exists}$ programs in general. We demonstrated that our Magic-Sets algorithm preserves query equivalence for any *Datalog*$^{\exists}$ program and we show how Magic-Sets can be safely applied to Shy programs. Moreover, we implemented the Magic-Sets strategy in DLV$^{\exists}$ and we performed an experimental analysis. The results evidenced the optimization potential provided by Magic-Sets. A more detailed discussion about this analysis is reported in Chapter 5.

Magic-Sets for *Datalog*$^{\exists}$

The original Magic-Sets technique was introduced for *Datalog* [20]. In order to bring it to the more general framework of *Datalog*$^{\exists}$, we have to face two main difficulties. The first is that originally the technique was defined to handle $\forall$-variables only. How does the technique have to be extended to programs containing $\exists$-variables? The second difficulty, which is eventually due to the first one, concerns how to establish the correctness of an extension of Magic-Sets to *Datalog*$^{\exists}$. In fact, any *Datalog* program is characterized by a unique universal model of finite size. In this case, the correctness of Magic-Sets can be established by proving that the universal model of the rewritten program (modulo auxiliary predicates) is a subset of the universal model of the original program and contains all the answers for the input query. On the other hand, a *Datalog*$^\exists$ program may have in general many universal models of infinite size. Due to this difference, it is more difficult to prove the correctness of a Magic-Sets technique.

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

Magic-Sets Algorithm

Magic-Sets stem from SLD-resolution, which roughly acts as follows: Each rule r s.t. $\sigma(\mathsf{head}(r)) = \sigma'(q)$, where σ and σ' are two substitutions, is considered in a first step. Then, the atoms in $\sigma(\mathsf{body}(r))$ are taken as subqueries, and the procedure is iterated. During this process, if a (sub)query has some arguments bound to constant values, this information is used to limit the range of the corresponding variables in the processed rules, thus obtaining more targeted subqueries when processing rule bodies. Moreover, bodies are processed in a certain sequence, and processing a body atom may bind some of its arguments for subsequently considered body atoms. The specific propagation strategy adopted in a top-down evaluation scheme is called *sideways information passing strategy* (SIPS). Roughly, a SIPS is a strict partial order over the atoms of each rule which also specifies how the bindings originate and propagate [22].

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

Definition 4.3.2 (Adornments)**.** Let p be a predicate of arity k. An adornment for p is a string $\alpha = \alpha_1 \cdots \alpha_k$ defined over the alphabet $\{b, f\}$. The i-th argument of p is considered *bound* if $\alpha_i = b$, or *free* if $\alpha_i = f$ $(i \in [1..k])$.

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

Definition 4.3.3 (SIPS)**.** Let r be a *Datalog*$^\exists$ rule and α an adornment for - $\mathsf{pred}(\mathsf{head}(r))$. A SIPS for r w.r.t. α is a pair $(\prec_r^\alpha, f_r^\alpha)$, where: $\prec_r^\alpha$ is a strict partial order over $\mathsf{atoms}(r)$ s.t. $\mathbf{a} \in \mathsf{body}(r)$ implies $\mathsf{head}(r) \prec_r^\alpha \mathbf{a}$; f_r^α is a function assigning to each atom $\mathbf{a} \in \mathsf{atoms}(r)$ the subset of the variables in $\mathbf{a}$ that

Algorithm 4 $\mathrm{MS}(q, P)$

```
Input: An atomic query $q = \mathtt{g}(u_1, \dots, u_k)$ and a Datalog$^\exists$ program $P$
Output: an optimized Datalog$^\exists$ program
1. begin
2.   $\alpha := \alpha_1 \cdots \alpha_k$, where $\alpha_i = b$ if $u_i \in \Delta_C$, and $\alpha_i = f$ otherwise ($i \in [1..k]$);
3.   $S := \{\langle \mathtt{g}, \alpha \rangle\}$;  $D := \varnothing$;  $R^{mgc} := \{\mathsf{mgc}(q, \alpha) \leftarrow \}$;  $R^{mod} := \varnothing$;
4.   while $S \neq \varnothing$ do
5.     $\langle \mathtt{p}, \alpha \rangle :=$ any element in $S$;  $S := S \setminus \{\langle \mathtt{p}, \alpha \rangle\}$;  $D := D \cup \{\langle \mathtt{p}, \alpha \rangle\}$;
6.     foreach $r \in \mathsf{rules}(P)$ s.t. $\mathsf{head}(r) = \mathtt{p}(t_1, \dots, t_n)$ and
                 $t_i \in \Delta_\exists$ implies $\alpha_i = f$ ($i \in [1..k]$) do
         //  $\mathbf{a} := \mathtt{p}(t_1, \dots, t_n)$
7.       $R^{mod} := R^{mod} \cup \{\mathsf{head}(r) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge \mathsf{body}(r)\}$;
8.       foreach $\mathtt{q}(s_1, \dots, s_m) \in \mathsf{body}(r)$ s.t. $\mathtt{q} \in \mathsf{idb}(P)$ do
           //  $\mathbf{b} := \mathtt{q}(s_1, \dots, s_m)$
9.         $B := \{\mathbf{c} \in \mathsf{body}(r) \mid \mathbf{c} <_r^\alpha \mathbf{b}\}$;
10.        $\beta := \beta_1 \cdots \beta_m$, where $\beta_i = b$ if $s_i \in \Delta_C \cup f_r^\alpha(B)$, and
                 $\beta_i = f$ otherwise ($i \in [1..k]$);
11.        $R^{mgc} := R^{mgc} \cup \{\mathsf{mgc}(\mathbf{b}, \beta) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge B\}$;
12.        if $\langle \mathtt{q}, \beta \rangle \notin D$ then $S := S \cup \{\langle \mathtt{q}, \beta \rangle\}$;
13. return $R^{mgc} \cup R^{mod} \cup \{\mathbf{a} \leftarrow \ \mid \mathbf{a} \in \mathsf{data}(P)\}$;
```

are made bound after processing $\mathbf{a}$; f_r^α must guarantee that $f_r^\alpha(\mathsf{head}(r))$ contains only and all the variables of $\mathsf{head}(r)$ corresponding to bound arguments according to α.

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

Definition 4.3.4 (Magic Atoms)**.** Let $\mathbf{a} = \mathtt{p}(t_1, \dots, t_k)$ be an atom and α be an adornment for $\mathtt{p}$. We denote by $\mathsf{mgc}(\mathbf{a}, \alpha)$ the magic atom $\mathtt{mgc\_p}^\alpha(\bar{t})$, where: $\bar{t}$ contains all terms in $t_1, \dots, t_k$ corresponding to bound arguments according to α; and $\mathtt{mgc\_p}^\alpha$ is a new predicate symbol (we assume that no standard predicate in P has the prefix "mgc_").

We are now ready to describe the MS algorithm (see Algorithm 4), associating each atomic query q over a *Datalog*$^\exists$ program P with a rewritten and optimized program MS(q,P). (More complex queries can be encoded by means of auxiliary rules.) The algorithm uses two sets, S and D, to store pairs of predicates and adornments to be propagated and already processed, respectively. Magic and modified rules are stored in the sets R^{mgc} and R^{mod}, respectively. The algorithm starts by producing the adornment associated with the query (line 1), which is paired with the query predicate and put into S (line 2). Moreover, the algorithm stores a ground fact named *query seed* into R^{mgc} (line 2). Sets D and R^{mod} are initially empty (line 2).

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

Binding information from α are then passed to body atoms according to a specific SIPS (lines 7–11). Specifically, for each body atom $\mathbf{b} = \mathrm{q}(\bar{s})$, the algorithm determines the set B of predecessor atoms in the SIPS (line 8), from which an adornment string β for q is built (line 9). B and β are then used to generate a *magic rule* whose head atom is $\mathsf{mgc}(\mathbf{b}, \beta)$, and whose body comprises $\mathsf{mgc}(\mathbf{a}, \alpha)$ and atoms in B (line 10). Moreover, the pair $\langle \mathrm{q}, \beta \rangle$ is added to S unless it was already processed in a previous iteration (that is, unless $\langle \mathrm{q}, \beta \rangle \in D$; line 11). Finally, the algorithm terminates returning the program obtained by the union of R^{mgc}, R^{mod} and $\{\mathbf{a} \leftarrow \ \mid \mathbf{a} \in \mathsf{data}(P)\}$ (line 12).

Example 4.3.5. The next rules belong to a *Datalog*$^{\exists}$ program hereafter called *P-Jungle*:

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

This program describes a funny scenario where an escaping, yet fast animal X may induce many other animals to be afraid. We now use *P-Jungle* for showing an example of the application of the Algorithm 4. In particular, we consider SIPS s.t. atoms are totally ordered from left-to-right and binding information is propagated whenever possible. In this setting, the Algorithm 4 run on query afraid(antelope) and *P-Jungle* yields the following rewritten program:

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

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

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

A detailed description is reported in [7]. □

Query Equivalence Result

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

Definition 4.3.6 (Magic Variant)**.** Let $I \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$, and $\{\mathsf{var}_i(I)\}_{i \in \mathbb{N}}$ be the following sequence: $\mathsf{var}_0(I) = I$; for each $i \geq 0$, $\mathsf{var}_{i+1}(I) = \mathsf{var}_i(I) \cup \{\mathbf{a} \in I \mid \exists \alpha$ s.t. $\mathsf{mgc}(\mathbf{a}, \alpha) \in \mathsf{var}_i(I)\} \cup \{\mathsf{mgc}(\mathbf{a}, \alpha) \mid \exists r, \sigma$ s.t. $r \in R^{mgc} \wedge \sigma(\mathsf{head}(r)) = \mathsf{mgc}(\mathbf{a}, \alpha) \wedge \sigma(\mathsf{body}(r)) \subseteq \mathsf{var}_i(I)\}$. The fixpoint of this sequence is denoted by $\mathsf{var}(I)$.

We point out that the magic variant of a set of atoms I comprises magic atoms and a subset of I. Intuitively, these atoms are enough to achieve a model of MS(q,P) if I is a model of P. This intuition is formalized below and proven in [7].

Lemma 4.3.7. *If $M \models P$, then $\mathsf{var}(M) \models \mathrm{MS}(q, P)$.*

The soundness of the Algorithm 4 w.r.t. QA can be now established.

Theorem 4.3.8 (Soundness)**.** *If $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$, then $\sigma \in \mathsf{ans}_P(q)$.*

Proof. Assume $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$. Let $M \models P$. By Lemma 4.3.7, $\mathsf{var}(M) \models \mathrm{MS}(q, P)$. Since $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$ by assumption, $\sigma(q) \in \mathsf{var}(M)$. Thus, $\sigma(q) \in M$ because $\mathsf{var}(M)$ comprises magic atoms and a subset of M by construction. □ □

To prove the completeness of the Algorithm 4 w.r.t. QA we identify a set of atoms that are not entailed by the rewritten program but not due to the presence of magic atoms.

Definition 4.3.9 (Killed Atoms)**.** Let $M \models \mathrm{MS}(q, P)$. The set $\mathsf{killed}(M)$ is defined as follows: $\{\mathbf{a} \in \mathsf{base}(\Delta) \setminus M \mid$ either $\mathsf{pred}(\mathbf{a}) \in \mathsf{edb}(P)$, or $\exists \alpha$ s.t. $\mathsf{mgc}(\mathbf{a}, \alpha) \in M\}$.

Since the falsity of killed atoms is not due to the Magic-Sets rewriting, one expects that their falsity can also be assumed in the original program. This intuition is formalized below and proven in [7].

Lemma 4.3.10. *If $M \models MS(q,P)$, $M' \models P$ and $M' \supseteq M$, then $M' \setminus \mathit{killed}(M) \models P$.*

We can finally prove the completeness of the Algorithm 4 w.r.t. QA, which then establishes the correctness of Magic-Sets for queries over *Datalog*$^{\exists}$ programs.

Theorem 4.3.11 (Completeness)**.** *If $\sigma \in \mathit{ans}_P(q)$, then $\sigma \in \mathit{ans}(q, MS(q,P))$.*

Proof. Assume $\sigma \in \mathsf{ans}_P(q)$. Let $M \models \mathrm{MS}(q,P)$. Let $M' \models P$ and be s.t. $M' \supseteq M$. By Lemma 4.3.10, $M' \setminus \mathsf{killed}(M) \models P$. Since $\sigma \in \mathsf{ans}_P(q)$ by assumption, $\sigma(q) \in M' \setminus \mathsf{killed}(M)$. Note that all instances of the query which are not in M are contained in $\mathsf{killed}(M)$ because the query seed belongs to M. Thus, $\sigma(q) \in M$ holds. □ □

Preserving Shyness in the Magic-Sets Rewriting

In Section 4.3.2, the correctness of MS has been established for *Datalog*$^{\exists}$ programs in general. Our goal now is to preserve the desirable shyness property in the rewritten of a *Shy* program.

In fact, shyness is not preserved by MS per sé. Resuming Example 4.3.5, MS run on query `afraid(antelope)` and program *P-Jungle* may produce from r_4 a rule `mgc_hungry`^{b}`(Y)` $\leftarrow$ `mgc_afraid`^{b}`(X), pursues(Y,X)`, which assumes `hungry(`φ`)` relevant whenever some `pursues(`φ`,X)` is derived, for any $\varphi \in \Delta_N$. However, shyness guarantees that any extension of this substitution for r_4 is actually annihilated by `strongerThan(Y,X)`, which thus enforces protection on `Y`. Unfortunately, SIPS cannot represent this kind of information in general, and thus MS may yield a non-shy program. Actually, the rewritten program in Example 4.3.5 is not shy because it contains rule `hungry(Y)` $\leftarrow$ `mgc_hungry`^{b}`(Y), pursues(Y,X), fast(X).`

The problem described above originates by the inability to represent in SIPS that no join on nulls is required to evaluate *Shy* programs. We thus explicitly encode this information in rules by means of the following transformation strategy: Let r be a rule of the form (2.1) in a program P, and #dom be an auxiliary predicate not occurring in P. We denote by $r^{\star}$ the rule obtained from r by adding a body atom #dom(`X`) for each protected variable `X` in $\mathsf{body}(r)$. Moreover, we denote by $P^{\star}$ the program comprising each rule $r^{\star}$ s.t. $r \in P$, and each fact #dom(`c`) $\leftarrow$ s.t. `c` $\in \mathsf{dom}(P)$. (Note that the introduction of these facts is not really required because #dom can be treated as a built-in predicate, thus introducing no computational overhead.)

Proposition 4.3.12. *If P is Shy, then $P^\star$ is shy as well and $\mathsf{mods}(P) = \mathsf{mods}(P^\star)$.*

Now, for an atomic query q over a *Shy* program P, in order to preserve shyness, we apply the Algorithm 4 to $P^\star$ and force SIPS to comply with the following restriction: Let $r \in P^\star$ and α be an adornment. For each $\mathbf{a}, \mathbf{b} \in \mathsf{body}(r)$ s.t. $\mathbf{a} \prec_r^\alpha \mathbf{b}$, and for each variable $\mathtt{X}$ occurring in both $\mathbf{a}$ and $\mathbf{b}$, SIPS $(\prec_r^\alpha, f_r^\alpha)$ is s.t. $\mathbf{a} \prec_r^\alpha \mathsf{\#dom}(\mathtt{X}) \prec_r^\alpha \mathbf{b}$. (See [7] for an example.)

Theorem 4.3.13. *Let q be an atomic query. If P is Shy, then $\mathrm{MS}(q, P^\star)$ is Shy.*

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

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

Chapter 5

Experimental analysis

In this chapter we report on some experiments we carried out in order to evaluate the efficiency and the effectiveness of DLV$^{\exists}$. In particular, our analysis is divided into three main categories: (1) the experiments where we compare DLV$^{\exists}$ to a number of very expressive QA systems, (2) the experiments where we confront DLV$^{\exists}$ to some highly scalable (but less expressive) QA systems, and (3) the experiments on the Magic-Sets optimization implemented in DLV$^{\exists}$. In the first category, we compare DLV$^{\exists}$ against a number of state-of-the-art systems for ontology-based query answering, i.e., Pellet [88], OWLIM-SE [25] and OWLIM-Lite [25]. Then, we take into account several systems for ontological query answering which are based on the query rewriting tecnique, i.e., Requiem [79] and Presto [85]. On the one hand, the expressivity of the languages of these systems is limited to $\mathbf{AC}_0$, thus it is much more restricted than the expressivity of *Shy* (see Chapter 6). On the other hand, these systems are very highly scalable. In the third category of experiments, we compare the standard DLV$^{\exists}$ system against DLV$^{\exists}$ without the Magic-Sets optimization. Notice that, we do not include any experiments on the optimal resumption tecnique presented in Chapter 4, because no query of the considered benchmark suite requires a level of resumption greater than 2. Thus, in this case, the optimization on the number of resumptions does not considerably affect performance of the system. This aspect will be dealt in the future.

This chapter is structured as follows: first of all, we present the results of the comparison between DLV$^{\exists}$ and the others systems for ontology-based QA. In particular, DLV$^{\exists}$ is first compared against Pellet, OWLIM-SE and OWLIM-Lite in Section 5.1, and then against Presto and Requiem in Section 5.2. Finally, in Section 5.3 we discuss the experiments on the Magic-Sets tecnique implemented in DLV$^{\exists}$.

5.1 DLV$^{\exists}$ vs. expressive ontology-based QA systems

Benchmark Focus.

The focus of thesetests is on rapidly changing and evolving ontologies (rules or data). In fact, in many contexts data frequently vary, even within hours, and there is the need to always provide the most updated answers to user queries. One of these contexts is e-commerce; another example is the university context, where data on exams, courses schedule and assignments may vary on a frequent basis. Benchmark framework from university domain and obtained results are discussed next.

Compared Systems.

As it will be pointed out in Section 6.2, ontology reasoners mainly rely on three categories of inference, namely: tableau, forward-chaining, and query-rewriting. Systems belonging to the latter category are still research prototypes and a comparison with them is reported in Section 5.2. Here, we compare DLV$^{\exists}$ with the following systems, being representatives of the first two categories.
▸ Pellet [88] is an OWL 2 reasoner which implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering).
▸ OWLIM-SE [25] is a commercial product which supports the full set of valid inferences using RDFS semantics; it's reasoning is based on forward-chaining. This system is oriented to massive volumes of data and, as such, based on persistent storage manipulation and reasoning.
▸ OWLIM-Lite [25], sharing the same inference mechanisms and semantics with OWLIM-SE, is another product of the OWLIM family designed for medium data volumes; reasoning and query evaluation are performed in main memory.

Data Sets.

We concentrated on a well known benchmark suite for testing reasoners over ontologies, namely LUBM [55].

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 ontology with customizable and repeatable synthetic data. The LUBM ontology schema and its data generation tool are quite complex and their description is out of the scope of this thesis.

We used the Univ-Bench ontology that describes (among others) universities, departments, students, professors and relationships among them; we considered the entire set of rules in Univ-Bench, except for equivalences with restrictions on roles, which cannot be expressed in *Shy* in some cases; these have been transformed in subsumptions. Data generation is carried out by the Univ-Bench data generator tool (UBA) whose main generation parameter is the number of universities to consider. The interested reader can find all information in [55].

In order to perform scalability tests, we generated a number of increasing data sets named: `lubm-10`, `lubm-30`, and `lubm-50`, where right-hand sides of 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 7M for `lubm-50`

LUBM incorporates a set of 14 queries aimed at testing different capabilities of the systems. A detailed description of rules and queries is provided at `http://www.mat.unical.it/kr2012`.

Data preparation.

LUBM is provided as owl files. Each owl class is associated with a unary predicate in *Datalog*$^{\exists}$; each individual of a class is represented by a *Datalog*$^{\exists}$ fact on the corresponding predicate. Each role is translated in a binary *Datalog*$^{\exists}$ predicate with the same name. Finally, assertions are translated in suitable *Shy* rules. The following example shows some translations where the DL has been used for clarity.

Example 5.1.1. The assertions

```
AdministrativeStaff ⊑ Employee
subOrgOf⁺
```

are translated in the following rules:

```
Employee(X) :- AdministrativeStaff(X).
subOrgOf(X,Z) :-subOrgOf(X,Y),subOrgOf(Y,Z).
```

where `subOrgOf` stands for `subOrganizationOf`. □

The complete list of correspondences between DL, OWL, and *Datalog*$^{\exists}$ rules and queries is provided at `http://www.mat.unical.it/kr2012`.

	Q_{all}	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	# solved	Geom. Avg time
lubm-10																	
DLV$^\exists$	17	5	4	2	4	6	1	6	4	8	5	<1	1	6	2	14	2.87
Pellet	27	82	84	84	82	80	88	81	89	95	82	82	89	82	84	14	84.48
OWLIM-Lite	33	33	–	33	33	33	33	4909	70	–	33	33	33	33	33	12	53.31
OWLIM-SE	105	105	105	105	105	105	105	105	106	106	105	105	105	105	105	14	105.14
lubm-30																	
DLV$^\exists$	55	16	13	7	14	21	3	21	12	25	18	<1	5	23	8	14	9.70
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	106	107	–	107	106	107	106	–	528	–	107	106	106	107	106	11	123.18
OWLIM-SE	323	323	328	323	323	323	323	323	323	326	323	323	323	323	323	14	323.57
lubm-50																	
DLV$^\exists$	93	27	23	12	23	35	6	34	22	42	31	<1	9	33	14	14	16.67
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	187	188	–	190	187	189	188	–	1272	–	189	187	187	189	187	11	223.79
OWLIM-SE	536	536	547	536	536	536	537	536	536	542	536	536	536	536	537	14	537.35

Table 5.1: Running times for LUBM queries (sec).

Results and Discussion.

Tests have been carried out on an Intel Xeon X3430, 2.4 GHz, with 4 Gb Ram, running Linux Operating System; for each query, we allowed a maximum running time of 7200 seconds (two hours).

Table 5.1 reports the times taken by the tested systems to answer the 14 LUBM queries. Since, as previously pointed out, we are interested in evaluating a rapidly changing scenario, each entry of the table reports the *total* time taken to answer the respective query by a system (including also loading and reasoning). In addition, the first column (labeled Q_{all}) shows the time taken by the systems to compute all atomic consequences of the program; this roughly corresponds to loading and inference time for Pellet, OWLIM-Lite, and OWLIM-SE and to parsing and first fixpoint computation for DLV$^\exists$.

The results in Table 5.1 show that DLV$^\exists$ clearly outperforms the other systems as an on-the-fly reasoner. In fact, the overall running times for DLV$^\exists$ are significantly lower than the corresponding times for the other systems. Pellet shows, overall, the worst performances. In fact, it has not been able to complete any query against `lubm-30` and `lubm-50`, and is also slower than competitors for the smallest data sets.

For both OWLIM-Lite and OWLIM-SE, most of the total time is taken for loading/inference (Q_{all}), as the reconstruction of the answers from the materialized inferences is a trivial task, often taking less than one second. However, as previously stated, this behavior is unsuited for reasoning on frequently changing ontologies, where previous inferences and materialization cannot be re-used, and loading must be repeated or time-consuming updates must be performed. As expected, loading/inference times (Q_{all}) for OWLIM-SE are higher than for OWLIM-Lite, but OWLIM-SE is faster than OWLIM-Lite in the reconstruction of the answers from the materialized inferences (this time is basically obtainable by subtracting Q_{all}). Because of this inefficiency in answers-reconstruction OWLIM-Lite has not been able to answer some queries in the time-limit that we set for the experiments (two hours); these queries involve many classes and roles.

We carried out some tests also on ontology updates; just to show an example, deleting 10% of `lubm-50` individuals imposed OWLIM-SE 152 seconds of update activities, which is sensibly higher than the highest query time needed by DLV$^\exists$ (42 seconds for Q_9) on the same data set. OWLIM-Lite was even worse on updates, since it required 133 seconds for the deletion of just one individual.

It is worth pointing out that DLV$^\exists$ is the only of the tested systems for which the times needed for answering single queries ($Q_1 \ldots Q_{14}$) are significantly smaller than those required for materializing all atomic consequences (Q_{all}). This result highlights the effectiveness of the query-oriented optimizations implemented in DLV$^\exists$ (magic sets and filtering, in particular), and confirms the suitability of the system for on-the-fly query answering. Interestingly, even if DLV$^\exists$ is specifically designed for query answering, it outperformed the competitors also for the computation of *all* atomic consequences (query Q_{all}). Indeed, on each of the three ontologies, DLV$^\exists$ took, respectively, about 17% and 51% of the time taken by OWLIM-SE and OWLIM-Lite.

5.2 DLV$^\exists$ vs. highly scalable QA systems

Compared systems.

In this section we report on another experimental analysis we performed, comparing DLV$^\exists$ against two of state-of-the-art systems for ontology-based QA relying on the query-rewriting category of inference. In particular, we compared DLV$^\exists$ with the following systems: Requiem [79] and Presto [85]. Thus, we first introduce the query rewriting tecnique, and we next overview the main features of these systems.

More precisely, query answering by *query rewriting* is performed by first computing a rewriting of the query with respect to the intensional part of the ontology (TBox), thus obtaining a so-called *perfect reformulation* of the initial query. Such a perfect reformulation is then evaluated over the extensional part of the ontology (ABox) only. The expressivity of the languages (mainly DLs) of the systems belonging to this category of inference is limited to $\mathbf{AC}_0$ and excludes, for instance, transitivity property or concept product. But, a distinguishing feature of *DL-Lite* with respect to the other DLs is that the perfect reformulation of conjunctive queries can be expressed by first-order queries. This property, also called *first-order rewritability* of conjunctive queries, is extremely important, because it allows to delegate the management of the ABox to a relational database system (RDBMS) and to solve query answering by shipping the perfect reformulation of the initial query (expressed in the SQL language) to the RDBMS. This implementation strategy actually allows to handle ABoxes of very large size (comparable to the size of a database).

However, the bottleneck of these algorithms and systems is constituted by the fact that the perfect reformulation computed increases exponentially with the number of atoms of the conjunctive query.

Two systems that tries to overcome the above limitation are: Presto and Requiem. In the following, we report a brief description of the algorithms implemented by them.

Presto

Presto is an algorithm for the perfect reformulation of unions of conjunctive queries over *DL-Lite* ontologies. Presto is based on the following ideas: (i) it does not generate a union of conjunctive queries, but a non-recursive datalog program. In fact, the use of a disjunctive normal form is one of the reasons for the exponential blow-up of previous techniques, which can thus be avoided by Presto; (ii) the query expansion rules (based on resolution) used by previous techniques are deeply optimized in Presto. In particular, Presto applies expansion rules driven by the goal of *eliminating existential joins* from the query based on the computation of *most general subsumees* of concept and role expressions, which turns out to be a much smarter strategy than previous approaches. As a consequence of the above innovations, the query produced by Presto is not exponential anymore with respect to the number of atoms of the initial conjunctive query, but is only exponential with respect to the number of *eliminable existential join variables* of the query: such variables are a subset of the join variables of the query, and are typically much less than the number of atoms of the query.

Requiem

Requiem algorithm takes as input a conjunctive query q and a *DL-Lite* TBox T, and produces a union of conjunctive queries q' that is a rewriting of q w.r.t. T.

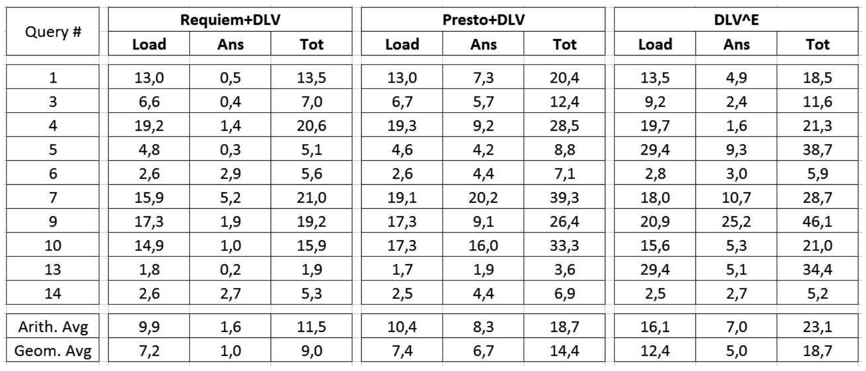
The algorithm first transforms q and T into clauses, and then computes the rewriting by using a resolution-based calculus to derive new clauses from the initial set. The rewriting is computed in four steps: clausification, saturation, unfolding, and pruning. The algorithm starts by transforming q and T into a set of clauses $\Sigma(T \cup \{q\})$. The expression $\Sigma(T)$ denotes the set of clauses obtained from axioms in T. Then, the algorithm keeps producing clauses in the saturation step until no other clause unique up to variable renaming can be produced. After the saturation step, the resulting clauses without existential variables are unfolded (a formal description of the unfolding step can be found in [80]). In the last step, every clause that does not have the same head predicate as q is dropped.

Data preparation

We concentrated on the LUBM benchmark suite introduced before. LUBM Tbox and queries are given as input to Requiem and Presto. `lubm-10` dataset and rewritings produced by the two systems are given as input to standard DLV in order to obtain answers to the queries. Actually, the syntax of these rewritings had been adapted to the syntax of DLV. Obviously, without altering original rewritings.

Results and discussion

Tests have been performed on an Intel Dual T2300, 1.6 GHz, with 1 Gb Ram, running Linux Operating System; for each query, we allowed a maximum running time of 7200 seconds (two hours) and a maximum memory usage of 1 Gb (gigabyte).



Query #	Requiem+DLV			Presto+DLV			DLV^E		
	Load	Ans	Tot	Load	Ans	Tot	Load	Ans	Tot
1	13,0	0,5	13,5	13,0	7,3	20,4	13,5	4,9	18,5
3	6,6	0,4	7,0	6,7	5,7	12,4	9,2	2,4	11,6
4	19,2	1,4	20,6	19,3	9,2	28,5	19,7	1,6	21,3
5	4,8	0,3	5,1	4,6	4,2	8,8	29,4	9,3	38,7
6	2,6	2,9	5,6	2,6	4,4	7,1	2,8	3,0	5,9
7	15,9	5,2	21,0	19,1	20,2	39,3	18,0	10,7	28,7
9	17,3	1,9	19,2	17,3	9,1	26,4	20,9	25,2	46,1
10	14,9	1,0	15,9	17,3	16,0	33,3	15,6	5,3	21,0
13	1,8	0,2	1,9	1,7	1,9	3,6	29,4	5,1	34,4
14	2,6	2,7	5,3	2,5	4,4	6,9	2,5	2,7	5,2
Arith. Avg	9,9	1,6	11,5	10,4	8,3	18,7	16,1	7,0	23,1
Geom. Avg	7,2	1,0	9,0	7,4	6,7	14,4	12,4	5,0	18,7

Figure 5.1: Running times for LUBM queries (sec.) over `lubm-10` dataset

Figure 5.1 reports the times taken by the tested systems to answer the 14 LUBM queries. For each system we have 3 columns: loading time, query answering time, total time (loading + answering). Loading and answering times for Requiem+DLV and Presto+DLV are the time taken by DLV for loading tuples from dataset and the time for evaluating the programs generated by the rewriters, respectively. Notice that, since rewritings produced by the two systems considered could involve different predicates, loading times could be different because of data filtering optimization introduced before. We observed that query rewriting time, in this case, is unrelevant for all the LUBM queries. Thus, we did not consider it in this analysis.

Obviously, we did not expect that performances of DLV$^{\exists}$ could be better than performances of the other two systems in the benchmark domain. Presto and Requiem are very highly scalable systems for query anwering (by query rewriting) over ontologies expressed in languages whose expressivity is much more limited than the expressivity of *Shy* (see Chapter 6).

However, we are very satisfied because DLV$^\exists$ is always in the middle between Requiem (that results the best on every query) and Presto if queries 4, 9 and 13 are not considered. Unfortunately, the delay of DLV$^\exists$ for these three queries is high and it affects the average total time. But, concerning average answering time, we observe that DLV$^\exists$ is faster than Presto. This is not still valid if we look at the average loading time. In fact, Magic-Sets rewritings takes into account more predicates than the ones involved in Presto's rewritings. Therefore, the effectiveness of the input filter optimization for DLV$^\exists$ is less evident than for Presto+DLV.

5.3 Impact of the proposed Magic-Sets optimization

The empirical evidence of the effectiveness of the Magic-Sets optimization implemented in DLV$^\exists$ is provided by means of the following experimental analysis on LUBM benchmark suite. Tests have been carried out on an Intel Xeon X3430, 2.4 GHz, with 4 Gb Ram, running Linux Operating System. For each query, we allowed 7200 seconds (two hours) or running time and 2 Gb of memory.

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

Table 5.2: Query evaluation time (seconds) of DLV$^\exists$ and improvements (IMP) of Magic-Sets

We evaluated the impact of Magic-Sets on DLV$^\exists$. Specifically, we measured the time taken by DLV$^\exists$ to answer the 14 LUBM queries with and without the application of Magic-Sets. Results are reported in Table 5.2, where times do not include data parsing and loading as they are not affected by Magic-Sets. On the considered queries, Magic-Sets reduce running time of 50% in average, with a peak of 87% on q_{13}. If only queries with no constants are considered, the average improvement of Magic-Sets is 37%, while the average improvement rises up to 55% for queries with at least one constant. We also point out that the average improvement provided by Magic-Sets is always greater than 25% if q_{12} and q_{14}

are not considered. Regarding these two queries, Magic-Sets do not provide any improvement because the whole data sets are relevant for their evaluation.

Chapter 6

Related work

In this chapter, we give a brief overview of several related approaches in the literature. The main motivation behind our research is the Semantic Web. We recall that the vision of the Semantic Web has led in the recent years to a new way of conceiving information systems, deeply integrated into the Web and its semantics, where Web information is annotated, so as to be machine-readable; in this way, such information can be integrated and especially queried in information systems, and not merely searched by keywords. This requires a precise sharing of terms by means of an ontology, so that the semantics of terms across different sources is clear. Moreover, by using ontologies, it is possible to perform automated reasoning tasks in order to infer new knowledge from the raw information residing on theWeb. Underneath the ontology, a data layer represents the raw data present on the Web, in an inherently heterogeneous way. The World Wide Web Consortium (W3C) defines several standards, including the Resource Description Framework (RDF) for the data layer, the Web Ontology Language (OWL) (based on description logics (DLs)) for the ontology layer, and the currently being standardized Rule Interchange Format (RIF) for the rule layer. As for the latter, rather than providing a common semantics, RIF aims at offering a common exchange format for rules, given that numerous languages already exist. The remainder of this chapter is structered as in the following. We start by discussing the features of DLs employed in Semantic Web reasoning. Next, we highlight the *DL-Lite* family and we show syntax and semantics of the base classes of this family. Afterwards, we recall a useful result of [28] about the relationship between *DL-Lite* and *Linear* Datalog$^{\pm}$, and we generalize this result to the more expressive language *Shy*. Finally, we close by reviewing some ontology reasoning systems.

6.1 Description Logics

In the Semantic Web, the ontology layer is highly important, and has led to a vast corpus of literature. DLs have been playing a central role in ontology reasoning; they are decidable fragments of first-order logic, based on concepts (classes of objects) and roles (binary relations on concepts); several variants of them have been thoroughly investigated, and a central issue is the trade-off between expressive power and computational complexity of the reasoning services. In DL reasoning, a knowledge base usually consists of a TBox (terminological component, i.e., ontology statements on concepts and roles) and an ABox (assertional component, i.e., ontology statements on instances of concepts and roles); the latter corresponds to a data set.

The description logic $\mathcal{SROIQ}$ [58] is one of the most expressive DLs, which is underlying OWL 2 [1], a new version of OWL [2]. Reasoning in $\mathcal{SROIQ}$ is computationally expensive, and several more tractable languages have been proposed in the Semantic Web community. Among such languages, we now discuss the *DL-Lite* family [34, 81], $\mathcal{EL}^{++}$ [11], and *DLP* [54], which are underlying the OWL 2 profiles QL, EL, and RL [3], respectively, as well as ELP [66], $\mathcal{SROEL}(\mathsf{x})$ [63], and $\mathcal{SROELV}_3(\sqcap, \mathsf{x})$ [64]. The DL-Lite family of description logics [34, 86] focuses on conjunctive query answering under a database and a set of axioms that constitute the ontology; query answering is in $\mathbf{AC}_0$ in the data complexity, due to FO-rewritability of all languages in the DL-Lite family (note that query answering in the extended *DL-Lite* family introduced in [9, 10] may also be more complex ($\mathbf{P}$ and $\mathbf{coNP}$)). The description logic *DL-Lite*$_R$ of the *DL-Lite* family provides the logical underpinning for the OWL 2 QL profile. Note here that the unique name assumption can be given up in *DL-Lite*$_R$ and OWL 2 QL, as it has no impact on the semantic consequences of a *DL-Lite*$_R$ and an OWL 2 QL ontology.

The description logic $\mathcal{EL}^{++}$ [11] is an extension of $\mathcal{EL}$ [12, 11] by the bottom element $\bot$, nominals, concrete domains, and role inclusions (between concatenations of abstract roles and atomic abstract roles); reasoning in $\mathcal{EL}^{++}$ is $\mathbf{P}$-complete, while conjunctive query answering in $\mathcal{EL}^{++}$ is undecidable. The OWL 2 EL profile is based on $\mathcal{EL}^{++}$; reasoning and conjunctive query answering in OWL 2 EL are both $\mathbf{P}$-complete in the data complexity. OWL 2 EL allows for stating the transitivity of atomic roles.

DLP [54] is a Horn fragment of OWL, i.e., a set of existential-free rules and negative constraints, without unique name assumption. The OWL 2 RL profile is an (existentialfree) extension of DLP, which aims at offering tractable reasoning services while keeping a good expressive power, enough to enhance RDF Schema

[1]See: `http://www.w3.org/TR/owl2-overview/`
[2]See: `http://www.w3.org/TR/owl-features/`
[3]See: `http://www.w3.org/TR/owl2-profiles/`

with some extra expressiveness from OWL 2. Compared to DLP, OWL 2 RL can in particular additionally encode role transitivity.

The rule-based tractable language ELP [66] generalizes both $\mathcal{EL}^{++}$ and DLP. In particular, it extends $\mathcal{EL}^{++}$ with local reflexivity, concept products, universal roles, conjunctions of simple roles, and limited range restrictions.

A closely related extension of $\mathcal{EL}^{++}$ is the DL $\mathcal{SROEL}(\mathsf{x})$ [63], which provides efficient rule-based inferencing for OWL 2 EL, and which is in turn extended by the DL $\mathcal{SROELV}_3(\sqcap, \mathsf{x})$ [64]. The latter introduces so-called *nominal schemas*, which allow for *variable nominals*, which are expressions that may appear in more than one conjunct in a concept expression, and such that all occurrences of the same variable nominal bind to the same individual.

In the following we report a more detailed analysis on the *DL-Lite* family, and a comparison between *DL-Lite* and Datalog$^{\pm}$.

6.1.1 The *DL-Lite* family

The base classes of the family are: *DL-Lite*$_F$, *DL-Lite*$_R$ and *DL-Lite*$_A$. All the other languages are constituted by restricting and combining these classes. In fact, in addition to these languages, we have (i) *DL-Lite*$_{core}$, which is the intersection of *DL-Lite*$_F$ and *DL-Lite*$_R$, (ii) *DL-Lite*^+_A, which is obtained from *DL-Lite*$_A$ by adding role attributes and identification constraints, and (iii) *DL-Lite*$_{F,\sqcap}$, *DL-Lite*$_{R,\sqcap}$, and *DL-Lite*$^+_{A,\sqcap}$, which are obtained from *DL-Lite*$_F$, *DL-Lite*$_R$, and *DL-Lite*^+_A, respectively, by additionally allowing conjunctions in the left-hand sides of inclusion axioms (without increase of complexity, which is related to the addition of Boolean role constructors in some popular description logics, as explored in [87]). Furthermore, each above description logic (with binary roles) *DL-Lite*$_X$ has a variant, denoted *DLR-Lite*$_X$, which additionally allows for n-ary relations, along with suitable constructs to deal with them.

In [28], an important result about the relationship between *DL-Lite* and *Linear* Datalog$^{\pm}$ is reported. In particular, it has been proved that the description logics *DL-Lite*$_F$, *DL-Lite*$_R$ and *DL-Lite*$_A$ (the base classes of the *DL-Lite* family) are reducible to *Linear Datalog*$^{\exists}$ with *(negative) constraints* and *non-conflicting (NC) keys*, called Datalog$^{\pm}_0$. Moreover, the former are strictly less expressive than the latter. The other description logics of the *DL-Lite* family can be similarly translated into Datalog$^{\pm}_0$.

Note that *DL-Lite*$_R$ is able to fully capture the (DL fragment of) RDF Schema [4], the vocabulary description language for RDF; see [42] for a translation. Hence, Datalog$^{\pm}_0$ is also able to fully capture (the DL fragment of) RDF Schema.

[4]See: http://www.w3.org/TR/rdf-schema

In the next section, we generalize these results to the more expressive language *Shy*.

6.1.2 *Shy* vs. *DL-Lite* languages

In Section 3.4 we showed that *Shy* encompasses and generalizes *Linear Datalog*$^{\exists}$. Adding (negative) constraints and non-conflicting (NC) keys to *Shy*, as showed in Sections 2.3.1 and 2.3.2 for *Linear Datalog*$^{\exists}$, does not increase the data complexity of answering BCQs under *Shy* TGDs and constraints.

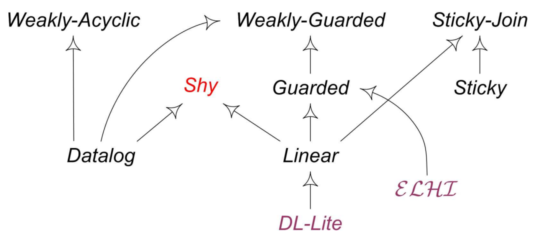


Figure 6.1: Relationships between DLs, Datalog$^{\pm}$ and *Shy*

Let *Shy*$^{+}$ denote the extension of *Shy* equipped with (negative) constraints and non-conflicting (NC) keys, we can say that *Shy*$^{+}$ is strictly more expressive than all the description logics of the *DL-Lite* family. The scenario of the relationships among these description logics, *Shy* and the other Datalog$^{\pm}$ languages is depicted in Figure 6.1.

DL Axiom	Shy Rule
$A \sqsubseteq B$	$p_A(X) \rightarrow p_B(X)$
$A \sqcap B \sqsubseteq C$	$p_A(X), p_B(X) \rightarrow p_C(X)$
$A \sqsubseteq \exists R.B$	$p_A(X) \rightarrow \exists Y\, p_R(X,Y), p_B(Y)$
$\exists R.A \sqsubseteq B$	$p_R(X,Y), p_A(Y) \rightarrow p_B(X)$
$R \sqsubseteq S$	$p_R(X,Y) \rightarrow p_S(X,Y)$
$R \sqsubseteq S^{-}$	$p_R(X,Y) \rightarrow p_S(Y,X)$
$R+$	$p_R(X,Y), p_R(Y,Z) \rightarrow p_R(X,Z)$

Table 6.1: DL VS *Shy*; A, B, C are concept names, R, S are role names.

Furthermore, we report in Table 6.1.2 the main DL constructs that can be expressed in *Shy*. for better understanding the expressive power of *Shy*. For each of them, we have also reported its translation in *Shy*.

6.2 Ontology reasoners

To the best of our knowledge, there is only one ongoing research work directly supporting ∃-quantifiers in *Datalog*, namely Nyaya [43]. This system, based on an SQL-rewriting, allows a strict subclass of *Shy* called *Linear-Datalog*$^{\exists}$, which does not include, for instance, transitivity and concept products.[5]

Since the system we developed enables ontology reasoning, existing ontology reasoners are also related. They can be classified in three groups: *query-rewriting*, *tableau* and *forward-chaining*.

The systems QuOnto [2], Presto [85], Quest [83], Mastro [35] and OBDA [84] belong to the query-rewriting category. They rewrite axioms and queries to SQL, and use RDBMSs for answers computation. Such systems support standard first-order semantics for unrestricted CQs; but the expressivity of their languages is limited to $\textbf{AC}_0$ and excludes, for instance, transitivity property or concept products.

The systems FaCT++ [89], RacerPro [56], Pellet [88] and HermiT [76] are based on tableau calculi. They materialize all inferences at loading-time, implement very expressive description logics, but they do not support the standard first-order semantics for CQs [50]. Actually, the Pellet system enables first-order CQs but only in the acyclic case.

OWLIM [25] and KAON2 [59] 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 [50].

Summing up, it turns out that DLV$^{\exists}$ is the first system supporting the standard first-order semantics for unrestricted CQs with ∃-variables over ontologies with advanced properties (some of these beyond $\textbf{AC}_0$), such as, role transitivity, role hierarchy, role inverse, and concept products. The experiments confirm the efficiency of DLV$^{\exists}$, which constitutes a powerful system for a fully-declarative ontology-based query answering.

[5]We could not compare DLV$^{\exists}$ with Nyaya since, as a research prototype, Nyaya provides no API for data loading and querying.

[6]Actually, KAON2 first translates the ontology to a disjunctive *Datalog* program, on which forward inference is then performed.

Chapter 7

Conclusion

In the field of data and knowledge management, query answering over ontologies (QA) is becoming more and more a challenging task. In this context, a conjunctive query (CQ) q is not merely evaluated on a extensional relational database D, but over a logical theory combining D with an ontology Σ describing rules for inferring intensional knowledge from D. A key issue here is the design of the language provided for specifying Σ. This language should balance expressiveness and complexity.

In the Semantic Web community, *DL-Lite* is a well-consolidated formalism for ontology-based QA. It is based on a solid theoretical fundation. Its expressivity is limited but it is specially devoted to conjunctive query answering on large amounts of data. In fact, query answering in *DL-Lite* is FO-rewritable. Moreover, recently, many efficient alghoritms and systems for *DL-Lite* have been developed and proposed in literature.

On the other hand, accessing data while taking ontological knowledge into account is becoming a challenging task also in databases. In this domain, Datalog$^{\pm}$, the family of *Datalog*-based languages recently proposed for tractable QA, is arousing more and more interest. This family, that generalizes well known ontology specification languages (e.g. *DL-Lite*), is mainly based on *Datalog*$^{\exists}$, which is the natural extension of *Datalog* that allows $\exists$-quantified variables in rule heads.

However, even if all known QA-decidable Datalog$^{\pm}$ languages maintain the simplicity of *Datalog* and are endowed with properties that are desired for ontology languages, none of them fully satisfy the following conditions: (1) efficient computability, (2) sufficient expressivity, and (3) suitability for an efficient implementation. For instance, *Linear* Datalog$^{\pm}$ is very efficiently computable. Moreover, an efficient rewriting algorithm has been proposed and developed for it. This system is the only one ongoing research work directly supporting $\exists$-quantifiers in *Datalog*, namely Nyaya [43].

But the expressivity of *Linear* Datalog$^{\pm}$ is very limited. In fact, it does not include, for example, transitivity and concept products.

Guarded fragment is a strict supersets of *Linear* Datalog$^{\pm}$, having tractable data complexity. But the chase ran on a program belonging to *Guarded* class requires the generation of a very high number of isomorphic atoms, therefore no (efficient) implementation has been developed yet.

Weakly-Guarded offers a good expressive power, it is in fact more expressive than both *Linear* and *Guarded* Datalog$^{\pm}$, but relaxing guardedness into weak-guardedness leads to **EXP**-complete data complexity.

Similar considerations can be extended to all other Datalog$^{\pm}$ classes. However, a detailed analysis of the entire Datalog$^{\pm}$ family is reported in Chapter 2.

Thus, this family of languages has still some "weaknesses". In particular, notwithstanding a number of Datalog$^{\pm}$ fragments has been already proposed, the evident gap emerging in this scenario is given by the lack of a language that offers a good efficiency without renouncing to the expressiveness.

Thus, in this work, we focused on this framework and we closed this gap by singling out a new class of *Datalog*$^{\exists}$ programs, called *Shy*. This satisfies a new semantic property called *parsimony* and results in a powerful and yet QA-decidable ontology specification language that combines positive aspects of different Datalog$^{\pm}$ languages. *Shy* is an optimal trade-off between expressiveness and scalability in the scenario of *Datalog* with existential quantifiers (see Section 3.4).

Indeed, the results obtained from our research can be summarized in the following terms. We proposed a new semantic property called *parsimony*. We proved that (atomic) query answering is decidable and also efficiently computable on the abstract class of parsimonious *Datalog*$^{\exists}$ programs, called *Parsimonious*. After showing that recognition of parsimony is undecidable (**coRE**-complete), we singled out *Shy*, a subclass of *Parsimonious*, which guarantees both easy recognizability and efficient answering even to conjunctive queries. We demonstrated that both *Parsimonious* and *Shy* preserve the same (data and combined) complexity of *Datalog* for atomic query answering. This shows that the addition of existential quantifiers does not bring any computational overhead here.

We introduced a new approach for conjunctive query answering, called *parsimonious-chase resumption*, which is sound and complete for query answering over *Shy*.

We implemented a bottom-up evaluation strategy for *Shy* programs inside the well-known DLV system. We enhanced the computation by a number of optimization techniques (among the improvements implemented in DLV$^{\exists}$, we highlighted the variant of the well-known magic-set optimization technique (Cumbo et al. 2004), adapted to *Datalog*$^{\exists}$ and implemented in DLV$^{\exists}$), yielding DLV$^{\exists}$ – a powerful system for query answering over *Shy* programs. This system is profitably applicable for ontology-based query answering. To the best of our knowledge, DLV$^\exists$ is the first 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 [50]. We performed an experimental analysis, comparing DLV$^\exists$ to a number of state-of-the-art systems for ontology-based QA. The positive results attained through this analysis demonstrates a 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.

Finally, we analyzed the Datalog$^\pm$ framework, providing a precise taxonomy of the QA-decidable *Datalog*$^\exists$ classes (see Chapter 2). It turned out that both *Parsimonious* and *Shy* strictly contain *Datalog* $\cup$ *Linear-Datalog*$^\exists$, while they are uncomparable to *Finite-Expansion-Sets*, *Finite-Treewidth-Sets*, and *Finite-Unification-Sets* (see Section 3.4). We analyzed related work, providing a description of the basic *DL-Lite* classes. After that, we observed that *Shy* encompasses and generalizes all the languages of the *DL-Lite* family (see Chapter 6).

In conclusion, we would like to remark that the work presented in this thesis is also the subject of the following papers: [67, 6]

Bibliography

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

[2] Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QUONTO: querying ontologies. In *Proc. of the 20th national conference on Artificial intelligence*, volume 4, pages 1670–1671. AAAI Press, 2005.

[3] Peter Alvaro, Dmitriy Ryaboy, and Divyakant Agrawal. Towards scalable architectures for clickstream data warehousing. In *Databases in Networked Information Systems*, volume 4777, pages 154–177. Springer Berlin / Heidelberg, 2007.

[4] Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. *Artificial Intelligence.* Elsevier, 187–188:156–192, 2012.

[5] Mario Alviano, Wolfgang Faber, and Nicola Leone. Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming.* Cambridge University Press, 10(4–6):497–512, July 2010.

[6] Mario Alviano, Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Magic-sets for datalog with existential quantifiers. In *Datalog*, pages 31–43, 2012.

[7] Mario Alviano, Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Magic-Sets for Datalog with Existential Quantifiers (Extended Version). Technical report, Department of Mathematics, University of Calabria, Italy, June 2012. See `www.mat.unical.it/datalog-exists/pub/12dl2.pdf`.

[8] Hajnal Andréka, Johan Van Benthem, and István Németi. Modal languages and bounded fragments of predicate logic, 1996.

[9] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. Dl-lite in the light of first-order logic. In *IN PROC. OF THE 22ND CONF. ON AI (AAAI-07)*, pages 364–369. AAAI Press, 2007.

[10] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The dl-lite family and relations. *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH (JAIR)*, 36:1–69, 2009.

[11] Franz Baader, Sebastian Brand, and Carsten Lutz. Pushing the el envelope. In *In Proc. of IJCAI 2005*, pages 364–369. Morgan-Kaufmann Publishers, 2005.

[12] Franz Baader, Ralf Kústers, and Ralf Molitor. Computing least common subsumers in description logics with existential restrictions. pages 96–101. Morgan Kaufmann, 1999.

[13] J. . Baget, M. Leclére, and M. . Mugnier. Walking the decidability line for rules with existential variables. *KR 2010*, pages 466–476, 2010. Cited By (since 1996): 1.

[14] Jean-François Baget, Michel Leclère, and Marie-Laure Mugnier. Walking the Decidability Line for Rules with Existential Variables. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference*, KR '10. AAAI Press, 2010.

[15] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Êric Salvat. Extending decidable cases for rules with existential variables. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 677–682, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[16] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. Extending Decidable Cases for Rules with Existential Variables. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI '09, pages 677–682, 2009.

[17] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9 - 10):1620 – 1654, 2011.

[18] Jean-François Baget and Marie-Laure Mugnier. Extensions of simple conceptual graphs: the complexity of rules and constraints. *JOUR. OF ARTIF. INTELL. RES*, 16:2002, 2002.

[19] Jean-François Baget, Marie-Laure Mugnier, Sebastian Rudolph, and Michaël Thomazo. Walking the complexity lines for generalized guarded existential rules. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 712–717. AAAI Press, 2011.

[20] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Int. Symposium on Principles of Database Systems*, pages 1–16, 1986.

[21] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *In The VLDB Journal*, pages 119–128, 2001.

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

[23] Catriel Beeri and Moshe Y. Vardi. A Proof Procedure for Data Dependencies. *J. ACM*, 31(4):718–741, September 1984.

[24] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, September 1984.

[25] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semant. web*, 2:33–42, January 2011.

[26] Luca Cabibbo. The expressive power of stratified logic programs with value invention. In *IN ICDT95 (FIFTH INTERNATIONAL CONFERENCE ON DATA BASE THEORY), PRAGUE, LECTURE NOTES IN COMPUTER SCIENCE 893*, pages 208–221, 1996.

[27] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In *Proc. of the 11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 70–80. AAAI Press, 2008. Revised version: `http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf`.

[28] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on*

Principles of database systems, PODS '09, pages 77–86, New York, NY, USA, 2009. ACM.

[29] Andrea Calì, Georg Gottlob, and Andreas Pieris. Advanced Processing for Ontological Queries. *PVLDB*, 3(1):554–565, 2010.

[30] Andrea Calì, Georg Gottlob, and Andreas Pieris. Query Answering under Non-guarded Rules in Datalog$^{\pm}$. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Proceedings of the 4th International Conference on Web Reasoning and Rule Systems*, volume 6333 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

[31] Andrea Calì, Domenico Lembo, and Riccardo Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *In Proc. of PODS 2003*, pages 260–271, 2003.

[32] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 71–86. Springer Verlag, September 2009.

[33] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Enhancing ASP by Functions: Decidable Classes and Implementation Techniques. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI '10. AAAI Press, 2010.

[34] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated Reasoning*, 39:385–429, 2007. 10.1007/s10817-007-9078-x.

[35] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.

[36] A. Chandra and M. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[37] Ashok K. Chandra, Harry R. Lewis, and Johann A. Makowsky. Embedded implicational dependencies and their inference problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 342–354, New York, NY, USA, 1981. ACM.

[38] Keith L. Clark. Negation as Failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[39] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12 – 75, 1990.

[40] Chiara Cumbo, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Enhancing the magic-set method for disjunctive datalog programs. In *Proceedings of the the 20th International Conference on Logic Programming - ICLP '04*, volume 3132 of *LNCS*, pages 371–385, 2004.

[41] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33:374–425, September 2001.

[42] Jos de Bruijn and Stijn Heymans. Logical foundations of (e)rdf(s): Complexity and reasoning. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 86–99. Springer Berlin / Heidelberg, 2007.

[43] Roberto De Virgilio, Giorgio Orsi, Letizia Tanca, and Riccardo Torlone. Semantic Data Markets: A Flexible Environment for Knowledge Management. In *Proc. of the 20th ACM international Conference on Information and Knowledge Management*, CIKM '11, New York, NY, USA, 2011. ACM. to appear.

[44] Alin Deutsch, Alan Nash, and Jeff Remmel. The chase revisited-deutschnashremmelpods2008. In *Proc. of the 27th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.*, PODS '08, pages 149–158, New York, NY, USA, 2008. ACM.

[45] Alin Deutsch and Val Tannen. Reformulation of xml queries and constraints. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 225–241, London, UK, UK, 2002. Springer-Verlag.

[46] Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences*, 73(4):584–609, 2007.

[47] Ronald Fagin. A normal form for relational databases that is based on domains and keys. *ACM Transactions on Database Systems*, 6:387–415, 1981.

[48] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, May 2005.

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

[50] Birte Glimm, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. Conjunctive query answering for the description logic SHIQ. *J. Artif. Int. Res.*, 31(1):157–204, January 2008.

[51] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51:17–28, 2002.

[52] Sergio Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):368–385, March/April 2003.

[53] Sergio Greco, Francesca Spezzano, and Irina Trubitsyna. Stratification criteria and rewriting techniques for checking chase termination. *PVLDB*, 4(11):1158–1168, 2011.

[54] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 48–57, New York, NY, USA, 2003. ACM.

[55] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.*, 3:158–182, October 2005. See `URL:http://swat.cse.lehigh.edu/projects/lubm/`.

[56] V. Haarslev and R. Möller. Racer system description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR'2001*, pages 701–705, Siena, Italy, 2001. Springer-Verlag.

[57] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *In ECOOP Proceedings*, pages 2–27. Springer, 2006.

[58] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible sroiq. In *In KR*, pages 57–67. AAAI Press, 2006.

[59] U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ- Descrption Logic to Disjunctive Datalog Programs. In *Proc. of the 9th International Conference on Knowledge Representation and Reasoning*, KR '04, pages 152–162, Whistler, Canada, 2004.

[60] D.S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167–189, February 1984.

[61] D.S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167 – 189, 1984.

[62] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[63] Markus Krotzsch. Proceedings of the twenty-second international joint conference on artificial intelligence efficient rule-based inferencing for owl el.

[64] Markus Krotzsch, Frederick Maier, Adila A. Krisnadhi, and Pascal Hitzler. A better uncle for owl – nominal schemas for integrating rules and ontologies, 2011.

[65] Markus Krötzsch and Sebastian Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 963–968. AAAI Press, 2011.

[66] Markus Krotzsch, Sebastian Rudolph, and Pascal Hitzler. Elp: Tractable rules for owl 2. Technical report, 2008.

[67] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Efficiently computable *Datalog*$^{\exists}$ programs. In *KR*, 2012.

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

[69] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[70] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, December 1979.

[71] Daniel Mailharro. A classification and constraint-based framework for configuration. *Artif. Intell. for Engineering Design, Analysis and Manufacturing*, 12:383–397, 1998.

[72] Bruno Marnette. Generalized schema-mappings: from termination to tractability. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 13–22, New York, NY, USA, 2009. ACM.

[73] John McCarthy. Circumscription — a Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13(1–2):27–39, 1980.

[74] Michael Meier, Michael Schmidt, and Georg Lausen. On Chase Termination Beyond Stratification. *PVLDB*, 2(1):970–981, 2009.

[75] Jack Minker. On Indefinite Data Bases and the Closed World Assumption. In Donald W. Loveland, editor, *Proceedings 6^{th} Conference on Automated Deduction (CADE '82)*, volume 138 of *Lecture Notes in Computer Science*, pages 292–308, New York, 1982. Springer.

[76] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.

[77] Marie-Laure Mugnier. Ontological query answering with existential rules. In *Proceedings of the 5th international conference on Web reasoning and rule systems*, RR'11, pages 2–23, Berlin, Heidelberg, 2011. Springer-Verlag.

[78] Peter F. Patel-Schneider and Ian Horrocks. A comparison of two modelling paradigms in the semantic web, 2007.

[79] H. Pérez-Urbina, B. Motik, and I. Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of the 22st International Workshop on Description Logics*, volume 477 of *DL '09*. CEUR-WS.org, 2009.

[80] Héctor Pérez-urbina, Boris Motik, and Ian Horrocks. Rewriting conjunctive queries under description logic constraints. Technical report, 2008.

[81] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on Data Semantics X*, volume 4900 of *Lecture Notes in Computer Science*, pages 133–173. Springer Berlin / Heidelberg, 2008.

[82] Raymond Reiter. On Closed World Data Bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[83] Mariano Rodriguez-Muro and Diego Calvanese. Dependencies: Making ontology based data access work in practice. In *Proc. of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 477, Santiago, Chile., 2011.

[84] Mariano Rodriguez-Muro and Diego Calvanese. Dependencies to optimize ontology based data access. In Riccardo Rosati, Sebastian Rudolph, and Michael Zakharyaschev, editors, *Description Logics*, volume 745 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[85] R. Rosati and A. Almatelli. Improving Query Answering over DL-Lite Ontologies. In *Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR 2010)*, KR '10, pages 290–300, Toronto, Ontario, Canada, 2010. AAAI Press.

[86] Sebastian Rudolph, Markus Krötzsch, and Pascal Hitzler. All elephants are bigger than all mice. In *Proceedings of the 21st International Workshop on Description Logics*, volume 353 of *DL '08*. CEUR-WS.org, 2008.

[87] Sebastian Rudolph, Markus KrÃ¶tzsch, and Pascal Hitzler. Cheap boolean role constructors for description logics.

[88] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5(2):51–53, June 2007.

[89] D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proc. of the 3rd Int. Joint Conf. on Automated Reasoning*, volume 4130 of *IJCAR '06*, pages 292–297, Seattle, WA, USA., 2006.

[90] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proceedings of the Seventh Symposium on Principles of Database Systems (PODS'88)*, pages 221–230, 1988.