# Expressive Languages for Querying the Semantic Web

Marcelo Arenas
Dept. of Computer Science
PUC Chile
Av. Vicuña Mackenna 4860
Santiago, Chile
marenas@ing.puc.cl

Georg Gottlob
Dept. of Computer Science
University of Oxford
Parks Road OX1 3JP
Oxford, United Kingdom
georg.gottlob@cs.ox.ac.uk

Andreas Pieris
Dept. of Computer Science
University of Oxford
Parks Road OX1 3JP
Oxford, United Kingdom
andreas.pieris@cs.ox.ac.uk

## ABSTRACT

The problem of querying RDF data is a central issue for the development of the Semantic Web. The query language SPARQL has become the standard language for querying RDF, since its standardization in 2008. However, the 2008 version of this language missed some important functionalities: reasoning capabilities to deal with RDFS and OWL vocabularies, navigational capabilities to exploit the graph structure of RDF data, and a general form of recursion much needed to express some natural queries. To overcome these limitations, a new version of SPARQL, called SPARQL 1.1, was recently released, which includes entailment regimes for RDFS and OWL vocabularies, and a mechanism to express navigation patterns through regular expressions. Unfortunately, there are still some useful navigation patterns that cannot be expressed in SPARQL 1.1, and the language lacks of a general mechanism to express recursive queries.

To the best of our knowledge, there is no RDF query language that combines the above functionalities, and which can also be evaluated efficiently. It is the aim of this work to fill this gap. Towards this direction, we focus on the OWL 2 QL profile of OWL 2, and we show that every SPARQL query enriched with the above features can be naturally translated into a query expressed in a language which is based on an extension of Datalog which allows for value invention and stratified negation. However, the query evaluation problem for this language is highly intractable, which is not surprising since it is expressive enough to encode some inherently hard queries. We identify a natural fragment of it, and we show it to be tractable and powerful enough to define SPARQL queries enhanced with the desired functionalities.

## Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages–*Data manipulation languages, query languages*

## Keywords

Semantic Web; RDF; SPARQL; Query Answering; Datalog-based Languages

## 1. INTRODUCTION

The Resource Description Framework (RDF) is the W3C recommendation data model to represent information about World Wide Web resources. An atomic piece of data in RDF is a *Uniform Resource Identifier (URI)*. In the RDF data model, URIs are organised as RDF graphs, that is, labeled directed graphs where node labels and edge labels are URIs. As with any data structure designed to model information, the natural problem of querying RDF data has been widely studied. Since its release in 1998, several designs and implementations of RDF query languages have been proposed [16]. In 2004, a first public working draft of a language, called SPARQL, was released by the W3C, which is in fact a graph-matching query language. Since then, SPARQL has been adopted as the standard language for querying the Semantic Web, and in 2008 it became a W3C recommendation [29].

One of the distinctive features of Semantic Web data is the existence of vocabularies with predefined semantics: the *RDF Schema (RDFS)* [7] and the *Ontology Web Language (OWL)* [24], which can be used to derive logical conclusions from RDF graphs. Thus, it would be desirable to have an RDF query language equipped with reasoning capabilities to deal with these vocabularies. Besides, it has also been recognised that navigational capabilities are of fundamental importance for data models with an explicit graph structure such as RDF [2, 4, 5, 15, 27], and, more generally, it is also well-accepted that a general form of recursion is a central feature for a graph query language [5, 23, 30]. Thus, it would also be desirable to have an RDF query language with such functionalities.

Unfortunately, the 2008 version of SPARQL missed the above crucial functionalities. To overcome these limitations, a new version of SPARQL, called SPARQL 1.1 [21], was recently released, which includes entailment regimes for RDFS and OWL vocabularies, and a mechanism to express navigation patterns through regular expressions. However, it has already been proved that there exist some very natural and useful queries that require of a more general form of recursion and cannot be expressed in SPARQL 1.1 [23, 30].

To the best of our knowledge, there is no RDF query language that combines all the functionalities mentioned above. Thus, it is the precise aim of the current work to bridge the gap between the existing RDF query languages and the three desired functionalities. Our ultimate goal is to propose an expressive query language that supports these features, and which can also be evaluated efficiently. Towards this direction, we first need to answer the following key question: what is the right syntax for such a language? Interestingly, Datalog with stratified negation [1, 13] has been shown to be expressive enough to represent every SPARQL query [2, 3, 4, 28, 31], so it has been used as a natural platform for the extensions of SPARQL with richer navigation capabilities and recursion mechanisms [23, 30]. Besides, some extensions of Datalog with existential quantification in the heads of rules have shown to be appropriate to encode some inferencing mechanisms in OWL [8].

Therefore, Datalog and some of its extensions – in particular, the members of the recently introduced Datalog$^{\pm}$ family of knowledge representation and query languages [10] – appear as a natural option for our purposes. However, for the language obtained by extending Datalog with existential quantification, the query evaluation problem is undecidable (this is implicit in [6]). In fact, undecidability holds even in the case of *data complexity* [32], that is, when the input query is fixed, and only the extensional database (or the RDF graph) is considered as part of the input [8]. It is thus a very important and challenging task to single out an expressive RDF query language that: (1) is based on Datalog, and thus enables a modular rule-based style of writing queries; (2) is expressive enough for being useful in real Semantic Web applications, and in particular to support reasoning and navigational capabilities, as well as a general form of recursion; (3) ensures the decidability of the query evaluation problem; and (4) has good complexity properties in the case the input query is fixed. This latter issue is of fundamental importance, as a low data complexity is considered to be a key condition for a query language to be useful in practice.

Our contributions can be summarised as follows:

1. We introduce in Section 4 a *modular* query language where reasoning capabilities, navigation capabilities and recursion mechanisms can be placed in different modules. This language is called *triple query language* (TriQ), and it is based on stratified Datalog$^{\exists,\neg,\bot}$, that is, Datalog extended with existential quantifiers in the heads of rules, stratified negation, and negative constraints expressed by using the symbol $\bot$ (*false*) in the heads of rules. In Section 4, we show this language to be expressive enough for encoding some useful but costly queries. In fact, we show that the data complexity of the query evaluation problem for this language is EXPTIME-complete.

2. We show that the modular structure of TriQ queries is very convenient to deal with SPARQL queries over the OWL vocabulary. More precisely, we focus in Section 5 on the profile of OWL, called OWL 2 QL, that is designed to be used in applications where query answering is the most important reasoning task. Then we prove that every SPARQL query under the entailment regime for OWL 2 QL [17, 22] can be naturally translated into a TriQ query. Moreover, we also show in Section 5 that the use of TriQ allows us to formulate SPARQL queries in a simpler way, as a more natural entailment regime described in that section can be easily defined by using this query language.

3. Given the high data complexity of the query evaluation problem for TriQ, we investigate in Section 6 whether the results proved in Section 5 can also be obtained for a tractable fragment of this query language. More precisely, we identify in Section 6 a natural restriction on TriQ queries that gives rise to a language, called TriQ-Lite, with the desired properties. In particular, we prove that the data complexity of the query evaluation problem for this language is PTIME-complete.

4. A key advantage of the modular nature of TriQ-Lite is the fact that, whenever the user wants to pose a new query over an RDF graph, (s)he does not need to modify the module which encodes the OWL 2 QL ontology. In Section 7, we show that this favorable behaviour cannot be achieved if we consider modular Datalog$^{\neg s,\bot}$. In particular, we introduce a novel notion of expressiveness which allows us to collect the queries that can be answered via a fixed program, and we show that TriQ-Lite is more expressive than modular Datalog$^{\neg s,\bot}$ under this notion. Notice that the crucial feature to establish such a result is the existential quantification.

The organisation of the paper is described in the summary of our contributions. Let us just say that in Section 2 we give an example which motivates our query languages, the notation used in the paper is introduced in Section 3, and that some concluding remarks are given in Section 8.

Due to lack of space, we do not give full proofs of the formal results of this paper, however, we provide proof sketches and/or sufficient evidence for the validity of the main results.

## 2. MOTIVATING EXAMPLE

The goal of this section is to show some of the difficulties encountered when querying RDF data with SPARQL, which motivated us to design an RDF query language based on Datalog and some of its extensions. Assume that $G_1$ is an RDF graph containing the following triples:

$$(\text{dbUllman},\ \text{is\_author\_of},\ \text{“The Complete Book"})$$
$$(\text{dbUllman},\ \text{name},\ \text{“Jeffrey Ullman"}).$$

The first triple indicates that the object with URI dbUllman is one of the authors of the book “The Complete Book", while the second triple indicates that the name of dbUllman is “Jeffrey Ullman".

To retrieve the list of authors mentioned in $G_1$ we can use the following SPARQL query:

$$\text{SELECT } ?X \quad (?Y, \text{is\_author\_of}, ?Z)\ \text{AND}\ (?Y, \text{name}, ?X). \tag{1}$$

We use here the algebraic syntax for SPARQL introduced in [26], which is formally defined in Section 3. In the query above, variables starts with the symbol ?. Thus, the triple $(?Y, \text{is\_author\_of}, ?Z)$ is used to retrieve the pairs $(a, b)$ of elements from $G_1$, which are stored in the variables $?Y$ and $?Z$, such that $a$ is an author of $b$. In the same way, the triple $(?Y, \text{name}, ?X)$ is used to retrieve the pairs $(a, c)$ of elements from $G_1$, which are stored in the variables $?Y$ and $?X$, such that $c$ is the name of $a$. Moreover, the operator AND in used to join the results of the triples, while SELECT $?X$ indicates that we are only interested in the values stored in the variable $?X$.

As mentioned in Section 1, one of the distinctive features of Semantic Web data is the use of the RDFS and OWL vocabularies. As an example of this, assume that $G_2$ is an RDF graph consisting of the following triples:

$$\begin{array}{l}
(\text{dbUllman},\ \text{is\_author\_of},\ \text{“The Complete Book"})\\
(\text{dbUllman},\ \text{name},\ \text{“Jeffrey Ullman"})\\
(\text{dbAho},\ \text{is\_coauthor\_of},\ \text{dbUllman})\\
(\text{dbAho},\ \text{name},\ \text{“Alfred Aho"})\\
(\text{r}_1,\ \text{rdf:type},\ \text{owl:Restriction})\\
(\text{r}_1,\ \text{owl:onProperty},\ \text{is\_coauthor\_of})\\
(\text{r}_1,\ \text{owl:someValuesFrom},\ \text{owl:Thing})\\
(\text{r}_2,\ \text{rdf:type},\ \text{owl:Restriction})\\
(\text{r}_2,\ \text{owl:onProperty},\ \text{is\_author\_of})\\
(\text{r}_2,\ \text{owl:someValuesFrom},\ \text{owl:Thing})\\
(\text{r}_1,\ \text{rdfs:subClassOf},\ \text{r}_2).
\end{array} \tag{2}$$

In $G_2$, the URIs with prefix rdfs: are part of the RDFS vocabulary, while the URIs with prefix owl: are part of the OWL vocabulary. More precisely, the third triple above indicates that the object with URI dbAho is a coauthor of the object with URI dbUllman. The fifth, sixth and seventh triples of $G_2$ define $\mathrm{r}_1$ as the class of URIs $a$ for which there exists a URI $b$ such that $(a, \text{is\_coauthor\_of}, b)$ holds, while the following three triples of this graph define $\mathrm{r}_2$ as the class of URIs $a$ for which there exists a URI $b$ such that the triple $(a, \text{is\_author\_of}, b)$ holds. Finally, the last triple of $G_2$ indicates that $\mathrm{r}_1$ is a subclass of $\mathrm{r}_2$.

The last seven triples of $G_2$ indicate that for every pair $a$, $b$ of elements such that $(a, \text{is\_coauthor\_of}, b)$ holds, it must be the case that $a$ is an author of some publication. Thus, if we want to retrieve the list of authors mentioned in $G_2$, then we expect to find dbAho in this list. However, the answer to the SPARQL query (1) over $G_2$ does not include this URI, and we are forced to encode the semantics of the RDFS and OWL vocabularies in the query. In fact, even if we try to obtain the right answer by using SPARQL 1.1 under the entailment regimes for these vocabularies, we are forced by the restrictions of the language [17] to replace the triple $(?Y, \text{is\_author\_of}, ?Z)$ in (1) by:

$$\begin{aligned}&(?Y, \text{rdf:type}, ?Z) \text{ AND}\\ &\quad (?Z, \text{rdf:type}, \text{owl:Restriction}) \text{ AND}\\ &\quad (?Z, \text{owl:onProperty}, \text{is\_author\_of}) \text{ AND}\\ &\quad (?Z, \text{owl:someValuesFrom}, \text{owl:Thing}),\end{aligned}$$

which indicates that we are looking for the objects that are authors of some publication (that is, the objects of type $\mathrm{r}_2$).

As the reader may have noticed, the resulting query is very complicated. In the query language proposed in this paper, the user can use separate modules to encode reasoning capabilities and actual queries. In particular, the user first needs to utilise a module for the RDFS and OWL vocabularies (or for some fragment of them), that could consist of Datalog$^{\exists,\neg,\perp}$ rules such as the following:

$$\begin{aligned}&\text{triple}(?X, \text{rdf:type}, ?Y),\\ &\text{triple}(?Y, \text{rdf:type}, \text{owl:Restriction}),\\ &\text{triple}(?Y, \text{owl:onProperty}, ?Z),\\ &\text{triple}(?Y, \text{owl:someValuesFrom}, ?U) \rightarrow\\ &\qquad\qquad\qquad \exists ?W\, \text{triple}(?X, ?Z, ?W).\end{aligned}$$

In this module, the predicate triple is used to store the triples of the RDF graphs. Notice that the rules of the module are used to encode the semantics of the respective vocabulary. Besides, these rules are fixed, they do not depend on the query that the user is trying to answer. Thus, to pose the desired query, the user just need to write on top of this module a simple query similar to (1):

$$\begin{aligned}&\text{triple}(?Y, \text{is\_author\_of}, ?Z),\\ &\qquad\qquad \text{triple}(?Y, \text{name}, ?X) \rightarrow \text{query}(?X). \qquad (3)\end{aligned}$$

In particular, (s)he does not need any prior knowledge about the semantics and inference rules for the respective vocabulary. In fact, the module for encoding this vocabulary can be publicly available, thus greatly simplifying the process of writing queries.

It is a very common practice in the Web to have several URIs for the same object. For example, the following are URIs of Jeffrey Ullman in DBpedia (the RDF version of Wikipedia) and the semantic knowledge base YAGO:

http://dbpedia.org/resource/Jeffrey_Ullman,

http://yago-knowledge.org/resource/Jeffrey_Ullman,

respectively. To alleviate the issue of having pieces of information about the same object that use distinct URIs for this object, the OWL vocabulary includes the keyword owl:sameAs to indicate that two URIs represent the same element. For example, this keyword is used in the following RDF graph $G_3$ to indicate that dbUllman and yagoUllman are URIs for the same object:

(dbUllman, is_author_of, “The Complete Book")

(dbUllman, owl:sameAs, yagoUllman)

(yagoUllman, name, “Jeffrey Ullman").

Assume now that we want to retrieve the list of authors mentioned in $G_3$. If we try to use again the SPARQL query (1), then we obtain the empty answer as the semantics of owl:sameAs is not taken into consideration. To solve this problem, one has to use the following query:

$$\begin{aligned}&\text{SELECT } ?X\\ &\Big( ((?Y, \text{is\_author\_of}, ?Z) \text{ AND } (?Y, \text{name}, ?X)) \qquad (4)\\ &\quad \text{UNION}\\ &\quad ((?Y, \text{is\_author\_of}, ?Z) \text{ AND } (?Y, \text{owl:sameAs}, ?W)\\ &\qquad\qquad \text{AND } (?W, \text{name}, ?X))\Big),\end{aligned}$$

where the operator UNION is used to obtain the union of the results of two queries, and the query after this operator is used to encode the semantics of the owl:sameAs keyword. Thus, as in the previous example, the user is forced to encode the semantics of the OWL vocabulary in the SPARQL query. And, as the reader may have noticed already, the situation gets even worse if we consider the graph $G_2$ in (2) but with the first two triples replaced by the triples in $G_3$. Fortunately, all these problems can be easily solved in our framework by just incorporating a fixed module encoding the semantics of the keyword owl:sameAs, which could consist of Datalog$^{\exists,\neg,\perp}$ rules of the form:

$$\begin{aligned}&\text{triple}(?X_1, \text{owl:sameAs}, ?X_2),\\ &\text{triple}(?Y_1, \text{owl:sameAs}, ?Y_2),\\ &\text{triple}(?X_1, ?U, ?Y_1) \rightarrow\\ &\qquad\qquad \text{triple}(?X_2, ?U, ?Y_2),\end{aligned}$$

and then using the same query (3) on top of the necessary modules.

## 3. DEFINITIONS AND BACKGROUND

Let $\mathbf{U}$, $\mathbf{B}$, $\mathbf{V}$ be pairwise disjoint infinite countable sets. The elements of $\mathbf{U}$ are called URIs, the elements of $\mathbf{B}$ are called blank nodes, and the elements of $\mathbf{V}$ are called variables and are assumed to start with the symbol ?. The sets $\mathbf{U}$ and $\mathbf{B}$ are used when defining both relational databases and RDF graphs, and we also refer to them as constants and (labeled) nulls, respectively.

**RDF and the query language SPARQL.** A triple $(s, p, o) \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$ is called an RDF triple. An RDF graph is a finite set of RDF triples. We use here an algebraic formalisation of SPARQL proposed in [26]. We start by defining the notion of SPARQL *built-in condition*, which is used in filter expressions. Formally, (1) if $?X, ?Y \in \mathbf{V}$ and $c \in \mathbf{U}$, then $?X = c$, $?X = ?Y$ and $\text{bound}(?X)$ are (atomic) built in-conditions; and (2) if $R_1$ and $R_2$ are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions. Then the set of (SPARQL) *graph patterns* is defined recursively as follows: (1) a set $\{\mathbf{t_1}, \ldots, \mathbf{t_n}\}$, where every $\mathbf{t_i} \in (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V})$ $(1 \leq i \leq n)$, is a graph pattern (called a basic graph pattern); (2) if $P_1$ and $P_2$ are graph patterns, then $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$ are graph patterns; (3) if $P$ is a graph pattern and $R$ is a SPARQL built-in condition, then $(P \text{ FILTER } R)$ is a graph pattern; and

(4) if $P$ is a graph pattern and $W$ is a finite set of variables, then (SELECT $W$ $P$) is a graph pattern. From now on, given a graph pattern $P$, we define var$(P)$ as the set of variables occurring in $P$, and likewise for var$(R)$ for a built-in condition $R$. Moreover, we assume that for every graph pattern ($P$ FILTER $R$), it holds that var$(R) \subseteq$ var$(P)$. Finally, we usually omit curly brackets in singleton basic graph patterns, that is, we replace $\{\mathbf{t}\}$ by $\mathbf{t}$, where $\mathbf{t} \in (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{V})$.

To define the semantics of SPARQL, we need to introduce some extra terminology. A *mapping* $\mu$ is a partial function $\mu : \mathbf{V} \to \mathbf{U}$. Abusing notation, for a basic graph pattern $P = \{\mathbf{t_1}, \ldots, \mathbf{t_n}\}$, we denote by $\mu(P)$ the basic graph pattern obtained by replacing the variables occurring in $P$ according to $\mu$. The *domain* of $\mu$, denoted by dom$(\mu)$, is the subset of $\mathbf{V}$ where $\mu$ is defined. Two mappings $\mu_1$ and $\mu_2$ are *compatible*, denoted by $\mu_1 \sim \mu_2$, when for all $?X \in$ dom$(\mu_1) \cap$ dom$(\mu_2)$, it is the case that $\mu_1(?X) = \mu_2(?X)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. Moreover, given a mapping $\mu$ and a set of variables $W$, the *restriction of $\mu$ to $W$*, denoted by $\mu_{|W}$, is a mapping such that dom$(\mu_{|W}) = ($dom$(\mu) \cap W)$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in ($dom$(\mu) \cap W)$. Finally, given a function $h : \mathbf{B} \to \mathbf{U}$, we denote by $h(P)$ the basic graph pattern obtained from $P$ by replacing the blanks nodes occurring in $P$ according to $h$.

To define the semantics of graph patterns, we first need to introduce the notion of satisfaction of a built-in condition by a mapping, and then we need to introduce some operators for mappings. More precisely, given a mapping $\mu$ and a built-in condition $R$, we say that $\mu$ *satisfies* $R$, denoted by $\mu \models R$, if (omitting the usual rules for Boolean connectives): (1) $R$ is bound$(?X)$ and $?X \in$ dom$(\mu)$; (2) $R$ is $?X = c$, $?X \in$ dom$(\mu)$ and $\mu(?X) = c$; and (3) $R$ is $?X = ?Y$, $?X, ?Y \in$ dom$(\mu)$ and $\mu(?X) = \mu(?Y)$. Moreover, given sets $\Omega_1$ and $\Omega_2$ of mappings, the *join* of, the *union* of, the *difference* between and the *left outer-join* between $\Omega_1$ and $\Omega_2$ are defined as follows:

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\},\\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 \smallsetminus \Omega_2 &= \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2 : \mu \not\sim \mu'\},\\
\Omega_1 ⟕ \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).
\end{aligned}$$

Then given an RDF graph $G$ and a graph pattern $P$, the evaluation of $P$ over $G$, denoted by $[\![P]\!]_G$, is recursively defined as follows: (1) if $P$ is a basic graph pattern, then $[\![P]\!]_G = \{\mu \mid$ dom$(\mu) =$ var$(P)$ and there exists $h : \mathbf{B} \to \mathbf{U}$ such that $\mu(h(P)) \subseteq G\}$; (2) if $P$ is ($P_1$ AND $P_2$), then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$; (3) if $P$ is ($P_1$ OPT $P_2$), then $[\![P]\!]_G = [\![P_1]\!]_G ⟕ [\![P_2]\!]_G$; (4) if $P$ is ($P_1$ UNION $P_2$), then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$; (5) if $P$ is ($P_1$ FILTER $R$), then $[\![P]\!]_G = \{\mu \mid \mu \in [\![P_1]\!]_G$ and $\mu \models R\}$; and (6) if $P$ if (SELECT $W$ $P_1$), then $[\![P]\!]_G = \{\mu_{|W} \mid \mu \in [\![P_1]\!]_G\}$.

Notice that according to the semantics of SPARQL [21, 29], the scope of an occurrence of a blank node in a graph pattern is the basic graph pattern containing it. This is reflected in the item (1) of the previous definition, where the existence of function $h : \mathbf{B} \to \mathbf{U}$ is checked at the level of basic graph patterns. In fact, if $P$ is a graph pattern, $P_1$ is a basic graph pattern occurring in $P$, $B$ is a blank node occurring in $P_1$, and $P'$ is the graph pattern obtained from $P$ by replacing every occurrence of $B$ in $P_1$ by a fresh blank node $B'$, then $P$ and $P'$ are equivalent graph patterns according to the previous definition.

**Relational databases and Datalog$^{\exists,\neg s,\perp}$ queries.** A *term* $t$ is a constant ($t \in \mathbf{U}$), labeled null ($t \in \mathbf{B}$), or variable ($t \in \mathbf{V}$). An *atom* has the form $p(t_1, \ldots, t_n)$, where $p$ is an $n$-ary predicate, and $t_1, \ldots, t_n$ are terms. For an atom $\underline{a}$, we denote by dom$(\underline{a})$ and var$(\underline{a})$ the set of its terms and the set of its variables, respectively; these notations extend to sets of atoms. We refer to the predicate of an atom $\underline{a}$ by pred$(\underline{a})$. An *instance* $I$ is a (possibly infinite) set of atoms $p(\mathbf{t})$, where $\mathbf{t}$ is a tuple of constants and labeled nulls. A *database* $D$ is a finite instance where only constants occur; we refer to the constants in $D$ as dom$(D)$.

A Datalog$^{\exists,\neg}$ rule $\rho$ is an expression of the form[1]

$$\underline{a}_1, \ldots, \underline{a}_n, \neg\underline{b}_1, \ldots, \neg\underline{b}_m \to \exists ?Y_1 \ldots \exists ?Y_k\ \underline{c},$$

where: (1) $n \geqslant 1$ and $m, k \geqslant 0$; (2) every $\underline{a}_i$ $(1 \leq i \leq n)$ is an atom with terms from $(\mathbf{U} \cup \mathbf{V})$; (3) every $\underline{b}_i$ $(1 \leq i \leq m)$ is an atom with terms from $(\mathbf{U} \cup \mathbf{V})$; (4) var$(\{\underline{b}_1, \ldots, \underline{b}_m\}) \subseteq$ var$(\{\underline{a}_1, \ldots, \underline{a}_n\})$; (5) var$(\{\underline{a}_1, \ldots, \underline{a}_n, \underline{b}_1, \ldots, \underline{b}_m\}) \cap \{?Y_1, \ldots, ?Y_k\} = \varnothing$; and (6) $\underline{c}$ is an atom with terms from $(\mathbf{U} \cup \{?Y_1, \ldots, ?Y_k\} \cup$ var$(\{\underline{a}_1, \ldots, \underline{a}_n\}))$. The set $\{\underline{a}_1, \ldots, \underline{a}_n\}$ is denoted by body$^+(\rho)$, while $\{\underline{b}_1, \ldots, \underline{b}_m\}$ is denoted by body$^-(\rho)$. The *body* of $\rho$, denoted body$(\rho)$, is defined as (body$^+(\rho) \cup$ body$^-(\rho)$). The atom $\underline{c}$ is the *head* of $\rho$, denoted by head$(\rho)$. A *Datalog$^{\exists,\neg}$ program* $\Pi$ is a finite set of Datalog$^{\exists,\neg}$ rules. Let sch$(X)$, where $X$ is either a program or a set of atoms, be the set of predicates occurring in $X$. A *stratification* of $\Pi$ is a function $\sigma$ : sch$(\Pi) \to [0, \ell]$, where $\ell \geqslant 0$, such that, for each $\rho \in \Pi$ with $p =$ pred(head$(\rho)$): (1) $\sigma(p) \geqslant \sigma(p')$, for each $p' \in$ sch(body$^+(\rho)$); and (2) $\sigma(p) > \sigma(p')$, for each $p' \in$ sch(body$^-(\rho)$). For each $i \in [0, \ell]$, let $\Pi_i = \{\rho \mid \rho \in \Pi \text{ and } \sigma(p) = i\}$. We say that $\Pi$ is *stratified* if there exists a stratification of $\Pi$.

A *constraint* $\nu$ is an assertion of the form $\underline{a}_1, \ldots, \underline{a}_n \to \perp$, where $n \geqslant 1$ and every $\underline{a}_i$ $(1 \leq i \leq n)$ is an atom with terms from $\mathbf{U} \cup \mathbf{V}$. The *body* of $\nu$, denoted body$(\nu)$, is the set $\{\underline{a}_1, \ldots, \underline{a}_n\}$. A *Datalog$^{\exists,\neg,\perp}$ program* $\Pi$ is a finite set of Datalog$^{\exists,\neg}$ rules and constraints. Moreover, we denote by ex$(\Pi)$ the set of Datalog$^{\exists,\neg}$ rules in $\Pi$, and we say that $\Pi$ is *stratified* if ex$(\Pi)$ is stratified. An *answer rule* w.r.t. a program $\Pi$ is a Datalog$^{\exists,\neg}$ rule $\rho$ without existentially quantified variables and negated atoms (i.e., a plain Datalog rule) such that sch(body$(\rho)$) $\subseteq$ sch$(\Pi)$ and pred(head$(\rho)$) $\notin$ sch$(\Pi)$. A *stratified Datalog$^{\exists,\neg,\perp}$ query* $Q$ is a pair $(\Pi, \Lambda)$, where $\Pi$ is a stratified Datalog$^{\exists,\neg,\perp}$ program, and $\Lambda$ is a set of answer rules w.r.t. $\Pi$. Henceforth, for brevity, we write Datalog$^{\exists,\neg s,\perp}$ for stratified Datalog$^{\exists,\neg,\perp}$ programs and queries. Moreover, a supra-index can be removed from Datalog$^{\exists,\neg s,\perp}$ to indicate that the corresponding feature is disallowed. For example, in a Datalog$^{\neg s}$ program neither existential variables in the heads of rules nor constraints are allowed.

Due to the lack of space, we do not formally define the semantics of a Datalog$^{\exists,\neg s,\perp}$ query. Instead, we explain how the chase of a database with a Datalog$^{\exists,\neg s}$ program is computed, and then how the semantics of a Datalog$^{\exists,\neg s,\perp}$ query can be defined in terms of this chase. More precisely, a *homomorphism* from a set of atoms $X$ to a set of atoms $X'$ is a partial function $h : \mathbf{U} \cup \mathbf{B} \cup \mathbf{V} \to \mathbf{U} \cup \mathbf{B} \cup \mathbf{V}$ such that: (1) $t \in \mathbf{U}$ implies $h(t) = t$, and (2) $p(t_1, \ldots, t_n) \in X$ implies $p(h(t_1), \ldots, h(t_n)) \in X'$. Then a Datalog$^{\exists,\neg s}$ rule $\rho$ is said to be *applicable* to an instance $I$ if there exists a homomorphism $h$ such that $h$(body$^+(\rho)$) $\subseteq I$ and $h$(body$^-(\rho)$) $\cap I = \varnothing$. Moreover, the result of applying $\rho$ to $I$ in this case is an instance $I' = I \cup h'$(head$(\rho)$), where $h'$ is a homomorphism such that $h'(?X) = h(?X)$ if $?X \in$ var(body$(\rho)$) $\cap$ var(head$(\rho)$) and $h'(?Y)$ is a fresh null if $?Y \in$ var(head$(\rho)$) $\setminus$ var(body$(\rho)$). Finally, the chase of a database $D$ with a Datalog$^{\exists,\neg s}$ program $\Pi$ is

[1]For the sake of brevity, in the rest of the paper we may write rules with more than one atom in the head. This is not a problem as such rules can be transformed into an equivalent set of rules with just one head-atom; see, e.g., [11].

an instance chase$(D, \Pi)$ constructed as follows. Let $\sigma : \text{sch}(\Pi) \rightarrow [0, \ell]$ be a stratification of $\Pi$, and let $\Pi_0, \ldots, \Pi_\ell$ be the partition of $\Pi$ induced by $\sigma$. To construct chase$(D, \Pi)$, we first construct $I_0 = \text{chase}(D, \Pi_0)$ by exhaustively applying the rules of $\Pi_0$ starting from $D$. Then we construct $I_1 = \text{chase}(I_0, \Pi_1)$, but this time by exhaustively applying the rules of $\Pi_1$ starting from $I_0$. In general, we define $I_{i+1} = \text{chase}(I_i, \Pi_{i+1})$ for $i \in \{0, \ldots, \ell - 1\}$, and we define chase$(D, \Pi)$ as $I_\ell$.

Let $Q = (\Pi, \Lambda)$ be a Datalog$^{\exists,\neg s,\perp}$ query and $D$ a database. The evaluation of $Q$ over $D$, denoted by ans$(Q, D)$, is defined as follows. If there is a constraint $\nu$ in $\Pi$ for which there exists a homomorphism $h$ such that $h(\text{body}(\nu)) \subseteq \text{chase}(\text{ex}(\Pi), D)$, then $D$ is *inconsistent* w.r.t. $Q$; otherwise, $D$ is *consistent* w.r.t. $Q$. If $D$ is inconsistent w.r.t. $Q$, then we define ans$(Q, D)$ as $\top$, where $\top$ is a special symbol used to indicate such inconsistency. If $D$ is consistent w.r.t. $Q$, which is the case we are really interested on, then ans$(Q, D)$ is defined as:

$$\left\{ p(h(t_1), \ldots, h(t_n)) \;\middle|\; \begin{array}{l} \exists \rho \in \Lambda\ \exists \text{ homomorphism } h\text{:} \\ h(\text{body}(\rho)) \subseteq \text{chase}(\text{ex}(\Pi), D), \\ \text{head}(\rho) = p(t_1, \ldots, t_n), \text{ and} \\ h(t_i) \in \mathbf{U} \text{ for every } i \in [1, n] \end{array} \right\}$$

The decision problem associated to query evaluation is as follows: given an atom $p(\mathbf{t})$, a Datalog$^{\exists,\neg s,\perp}$ query $Q$, and a database $D$, decide whether $p(\mathbf{t}) \in \text{ans}(Q, D)$. In this work we are interested on the *data complexity* of this problem, i.e., the complexity calculated by considering $Q$ as fixed.

**Guardedness.** The problem of evaluating Datalog$^{\exists,\neg s,\perp}$ queries is undecidable; this is implicit in [6, 8]. Several decidability restrictions have been proposed in the literature. The restriction which is relevant for the present work is based on the notion of guardedness. Before we proceed further, let us recall the notion of the affected position [8]. A *position* $p[i]$ identifies the $i$-th attribute of a predicate $p$. We refer to the *arity* of $p$ by arity$(p)$. Given a set of predicates $X$, the set of positions of $X$, denoted pos$(X)$, is the set $\{p[i] \mid p \in \text{sch}(X) \text{ and } i \in [1, \text{arity}(p)]\}$. Given a Datalog$^{\exists,\neg s}$ program $\Pi$, the set of *affected positions* of sch$(\Pi)$, denoted by affected$(\Pi)$, is defined as follows: (1) if there exists $\rho \in \Pi$ such that at position $\pi$ an existentially quantified variable occurs, then $\pi \in \text{affected}(\Pi)$; and (2) if there exists $\rho \in \Pi$ and a variable $?V$ that occurs in body$^+(\rho)$ only at positions of affected$(\Pi)$, and $?V$ appears in head$(\rho)$ at position $\pi$, then $\pi \in \text{affected}(\Pi)$. Let nonaffected$(\Pi)$ be the set $(\text{pos}(\Pi) \setminus \text{affected}(\Pi))$.

*Example 1.* Consider the Datalog$^{\exists,\neg s}$ program $\Pi$:

$$\begin{aligned} p(?X, ?Y), s(?Y, ?Z) &\rightarrow \exists ?W\, t(?Y, ?X, ?W) \\ t(?X, ?Y, ?Z) &\rightarrow \exists ?W\, p(?W, ?Z) \\ p(?X, ?Y), \neg r(?X) &\rightarrow \exists ?Z\, q(?X, ?Z). \end{aligned}$$

Due to the existentially quantified variables, we get that $t[3]$, $p[1]$ and $q[2]$ belong to affected$(\Pi)$. Since the variable $?Z$ occurs in the body-atom of the second rule at position $t[3]$ which is affected, and also at position $p[2]$ in the head of the same rule, $p[2] \in \text{affected}(\Pi)$. Similarly, $t[2]$ and $q[1]$ are affected positions of sch$(\Pi)$. Notice that, although $?Y$ occurs in the body of the first rule at the affected position $p[2]$, and also at position $t[1]$ in the head of the same rule, $t[1]$ is not affected since $?Y$ occurs also at position $s[1] \notin \text{affected}(\Pi)$. Finally, observe that $q[1]$ is affected, even if $?X$ occurs in the body of the third rule at the non-affected position $r[1]$, since we consider only positive atoms. $\square$

A Datalog$^{\exists,\neg s}$ program $\Pi$ is *weakly-guarded* if for each $\rho \in \Pi$, there exists $\underline{a} \in \text{body}^+(\rho)$, called *weak-guard*, which contains all the variables of var(body$(\rho)$) that appear in body$^+(\rho)$ only at positions of affected$(\Pi)$ [8]. Observe that the program $\Pi$ in Example 1 is weakly-guarded. A *weakly-guarded Datalog*$^{\exists,\neg s,\perp}$ *query* is a Datalog$^{\exists,\neg s,\perp}$ query $(\Pi, \Lambda)$ where ex$(\Pi)$ is weakly-guarded.

## 4. MODULAR QUERIES

The main goal of this paper is to construct a query language with reasoning capabilities to deal with the OWL vocabulary, navigational capabilities to exploit the graph structure of RDF data, and a general form of recursion much needed to express some natural and useful queries. To this end, we introduce in this section a query language where these functionalities can be placed in different modules. In this paper, we exploit this modular structure and show it to be very convenient to deal with SPARQL queries over the OWL vocabulary (Section 5) and to find a tractable query language with the desired features (Section 6). Besides, this modularity allows us, for example, to easily replace a translation of a SPARQL query by a more efficient encoding if needed.

We now introduce our main language called *triple query language* (TriQ). A modular Datalog$^{\exists,\neg s,\perp}$ query $M$ is a pair $[Q_1, Q_2]$, where both $Q_1$ and $Q_2$ are Datalog$^{\exists,\neg s,\perp}$ queries. A TriQ query is a modular Datalog$^{\exists,\neg s,\perp}$ query $[Q_1, Q_2]$, where both $Q_1$ and $Q_2$ are weakly-guarded. In order to define the semantics of modular Datalog$^{\exists,\neg s,\perp}$, we first need to define when a database $D$ is consistent w.r.t. a modular Datalog$^{\exists,\neg s,\perp}$ query $M = [Q_1, Q_2]$. Formally, if $D$ is inconsistent w.r.t. $Q_1$ or ans$(Q_1, D)$ is inconsistent w.r.t $Q_2$ in the sense defined in Section 3, then we say that ans$(M, D) = \top$. Otherwise, the evaluation of $M$ over $D$ is performed in a modular way, that is, ans$(M, D)$ is defined as ans$(Q_2, \text{ans}(Q_1, D))$.

In this paper, we consider modular queries with *two* modules, as these are enough for our purpose. It should be noted that, however, these queries can be easily extended to handle an arbitrary number of modules. In particular, the semantics of a modular query of the form $M = [Q_1, Q_2, \ldots, Q_k]$ is defined by considering a sequence $D_1, \ldots, D_k$ of databases such that $D_1 = \text{ans}(Q_1, D)$ and $D_{i+1} = \text{ans}(Q_{i+1}, D_i)$ $(1 \leq i < k)$, and then letting ans$(M, D)$ be $D_k$.

A natural question at this point is how expressive TriQ is. Interestingly, we show in the next example that this language is expressive enough for encoding some very useful but costly queries; e.g., whether a graph contains a clique of size $k$.

*Example 2.* Let $G = (V, E)$ be an undirected graph with $n > 0$ vertices, and let $k > 0$. We will construct a database $D$ and a TriQ query $M = [(\Pi_{aux}, \Lambda_{copy}), (\Pi_{clique}, \Lambda_{clique})]$ such that $G$ contains a $k$-clique iff yes() $\in$ ans$(M, D)$. The database $D$ encodes the graph $G$ and the value $k$. More precisely, for each node $c \in V$, $D$ contains atom node$_0(c)$, and for each edge $(v, w) \in E$, $D$ contains edge$_0(v, w)$. Moreover, number $k$ is encoded in $D$ by using atoms succ$_0(0, 1), \ldots$, succ$_0(k-1, k)$. The set of rules $\Pi_{aux}$ is used to compute some auxiliary relations that are needed when checking whether $G$ contains a $k$-clique. More precisely, $\Pi_{aux}$ contains two rules to define the usual linear order on $[0, k]$:

$$\begin{aligned} \text{succ}_0(?X, ?Y) &\rightarrow \text{less}_0(?X, ?Y) \\ \text{succ}_0(?X, ?Y), \text{less}_0(?Y, ?Z) &\rightarrow \text{less}_0(?X, ?Z), \end{aligned}$$

and contains the following rules to define the minimum and maximum elements of this linear order:

$$\begin{aligned} \text{less}_0(?X, ?Y) &\rightarrow \text{not\_max}(?X) \\ \text{less}_0(?X, ?Y) &\rightarrow \text{not\_min}(?Y) \\ \text{less}_0(?X, ?Y), \neg\text{not\_min}(?X) &\rightarrow \text{zero}_0(?X) \\ \text{less}_0(?Y, ?X), \neg\text{not\_max}(?X) &\rightarrow \text{max}_0(?X). \end{aligned}$$

The set of answer rules $\Lambda_{copy}$ is just used to copy the atoms of $D$ and the results of $\Pi_{aux}$ into a new schema that can be queried by $(\Pi_{clique}, \Lambda_{clique})$. In fact, for each predicate $p_0$ occurring in $D$ or $\Pi_{aux}$, we have a rule $p_0(\mathbf{X}) \rightarrow p(\mathbf{X})$.

Let us now give the key idea underlying $\Pi_{clique}$. Intuitively, $\Pi_{clique}$ constructs a tree of mappings (rooted at some dummy mapping), where a mapping at level $i \in [1, k]$ actually maps the set of integers $[1, i]$ to the vertices of $G$. Each mapping $\mu$ at level $i < k$ has $n$ child-mappings, one for each node of $G$. The child-mapping $\mu'$ of $\mu$ (for a node $v$) simply extends $\mu$ by mapping $(i + 1)$ to $v$. The $k$-th level of the tree contains all the possible $n^k$ mappings $\mu : [1, k] \rightarrow V$. It is then easy to check whether there exists a mapping that maps $[1, k]$ to a clique of $G$.

Now we define $\Pi_{clique}$. In this program, apart from the predicates occurring in the head of the rules of $\Lambda_{copy}$, we also have the following predicates: (1) ism; the atom $\text{ism}(\mu, i)$ says that $\mu$ is a mapping at level $i$ of the tree; (2) map; the atom $\text{map}(\mu, i, v)$ says that $\mu(i) = v$; (3) next; the atom $\text{next}(\mu, v, \mu')$ encodes the fact that $\mu'$ is obtained from $\mu$ by mapping $(i + 1)$ to $v$ (assuming that $\mu$ is a mapping at level $i$); (4) noclique; the atom $\text{noclique}(\mu)$ says that $\mu$ does not map to a clique; and (5) $\text{clique}_k$; which is a propositional predicate used to indicate that some $k$-clique has been found. Moreover, the program $\Pi_{clique}$ consists of the following rules:

$$\text{zero}(?X) \rightarrow \exists ?Y\, \text{ism}(?Y, ?X)$$

$$\text{ism}(?X, ?Y), \text{succ}(?Y, ?Z), \text{node}(?W) \rightarrow$$
$$\exists ?U\, \text{next}(?X, ?W, ?U), \text{ism}(?U, ?Z), \text{map}(?U, ?Z, ?W)$$

$$\text{next}(?X, ?Y, ?Z), \text{map}(?X, ?U, ?V) \rightarrow \text{map}(?Z, ?U, ?V)$$

$$\text{less}(?X, ?Y), \text{map}(?Z, ?X, ?W),$$
$$\text{map}(?Z, ?Y, ?U), \neg\text{edge}(?W, ?U) \rightarrow \text{noclique}(?Z)$$

$$\text{less}(?X, ?Y), \text{map}(?Z, ?X, ?W),$$
$$\text{map}(?Z, ?Y, ?W) \rightarrow \text{noclique}(?Z)$$

$$\text{ism}(?X, ?Y), \text{max}(?Y), \neg\text{noclique}(?X) \rightarrow \text{clique}_k()$$

Notice that the purpose of the fifth rule is to avoid that the same node is used more than once in a clique (which can happen if $G$ contains self-loops). Finally, $\Lambda_{clique}$ contains the single answer rule $\text{clique}_k() \rightarrow \text{yes}()$, which returns "yes" if a $k$-clique has been found. $\square$

The previous example gives evidence that the data complexity of the query evaluation problem for TriQ is intractable. In what follows, we show that this problem is indeed EXPTIME-complete. The lower bound follows immediately from the fact that the same problem for weakly-guarded $\text{Datalog}^{\exists}$ queries is already EXPTIME-hard in data complexity [8]. The problem of deciding whether a database $D$ is inconsistent w.r.t. a weakly-guarded $\text{Datalog}^{\exists,\neg s,\perp}$ query can be reduced to the problem of query evaluation for weakly-guarded $\text{Datalog}^{\exists,\neg s}$ queries. Thus, to establish the desired upper bound, it suffices to show that the evaluation problem for weakly-guarded $\text{Datalog}^{\exists,\neg s}$ queries is feasible in EXPTIME. This can be established by a careful extension of the existing alternating algorithm for weakly-guarded $\text{Datalog}^{\exists}$ queries.

THEOREM 1. *The query evaluation problem for* TriQ *is* EXPTIME-*complete in data complexity.*

It is important to mention that recently, independently from our work, it has been shown that weakly-guarded $\text{Datalog}^{\exists,\neg s}$ can express all the queries that can be evaluated in exponential time in data complexity [19]. Combining this result with the upper bound in Theorem 1, we obtain that weakly-guarded $\text{Datalog}^{\exists,\neg s}$ captures EXPTIME. This result implies that TriQ and weakly-guarded $\text{Datalog}^{\exists,\neg s}$ are equally expressive query languages. However, the modular nature of TriQ allows us to write more intuitive and succinct queries than weakly-guarded $\text{Datalog}^{\exists,\neg s}$.

## 5. FROM SPARQL OVER OWL 2 QL TO TRIQ

The first version of the Web ontology language OWL was released in 2004 [24]. The second version of this language, which is called OWL 2, was released in 2012 [20]. This new version includes three profiles that can be implemented more efficiently [25]. One of these profiles, called OWL 2 QL, is based on the description logic $\text{DL-Lite}_{\mathcal{R}}$ [12] and is designed to be used in applications where query answering is the most important reasoning task. As the main goal of our paper is to design a query language that naturally embeds the fundamental features for querying RDF, we focus on OWL 2 QL in this section, and show that every SPARQL query under the OWL 2 direct semantics entailment regime [17, 22] can be naturally translated into a TriQ query. But not only that, a second goal of this section is to show that the use of TriQ allows us to formulate SPARQL queries in a simpler way, as a more natural notion of entailment can be easily encoded by using this query language.

For the sake of presentation, we first disregard the direct semantics entailment regime and show in Section 5.1 that each SPARQL query can be translated into a modular $\text{Datalog}^{\neg s}$ query. Then we extend this translation in Section 5.2 to prove that every SPARQL query under the direct semantics entailment regime can be transformed into a TriQ query. Moreover, we show in Section 5.3 that a more natural notion of entailment, which is obtained by removing a restriction from the regime proposed in [17], can also be encoded in TriQ. It should be noticed that it is known that SPARQL can be translated into $\text{Datalog}^{\neg s}$ [2, 3, 4, 28, 31], if one focuses on RDF graphs with RDFS vocabulary extended with a special symbol to represent the null value (and with a built-in predicate to check for this symbol). Thus, the goal of Section 5.1 is not to prove that SPARQL can be embedded into modular $\text{Datalog}^{\neg s}$, but instead to propose a translation that does not need to use a special symbol for the null value, and which can be easily extended to deal not only with the RDFS vocabulary but also with the vocabulary used in OWL 2 QL ontologies (as shown in Section 5.2).

### 5.1 Translating SPARQL into modular $\text{Datalog}^{\neg s}$

In this section, we show that every SPARQL query can be easily translated into a modular $\text{Datalog}^{\neg s}$ query. The main ingredients of this translation are shown in the following example. From now on, given an RDF graph $G$, assume that $\tau_{\text{db}}(G)$ is the instance of the relational schema $\{\text{triple}(\cdot, \cdot, \cdot)\}$ naturally associated with $G$, that is, a tuple $(a, b, c)$ is in the relation triple in $\tau_{\text{db}}(G)$ if and only if $(a, b, c) \in G$.

*Example 3.* Let $P_1$ be the graph pattern $(?X, \text{name}, ?Y)$, where name is a constant, which is asking for the list of pairs $(a, b)$ of elements from an RDF graph $G$ such that $b$ is the name of $a$ in $G$. This graph pattern can be easily represented as a Datalog program over $\tau_{\text{db}}(G)$:

$$\text{triple}(?X, \text{name}, ?Y) \quad \rightarrow \quad \text{query}_{P_1}(?X, ?Y).$$

The predicate $\text{query}_{P_1}$ is used in this program to store the answer to the graph pattern $P_1$. Now assume that $P_2$ is the graph pattern

$(?X, \text{name}, B)$, where $B$ is a blank node. This time we are asking for the list of elements in an RDF graph $G$ that have a name (the blank node $B$ is used in $P_2$ to indicate that $?X$ has a name, but that we are not interested in retrieving it). As in the previous case, this graph pattern can be easily represented as a Datalog program over $\tau_{\text{db}}(G)$:

$$\text{triple}(?X, \text{name}, ?Y) \quad \rightarrow \quad \text{query}_{P_2}(?X). \tag{5}$$

Given that blank nodes are used as existential variables in basic graph patterns, variable $?Y$ is used in the previous rule to represent blank node $B$. However, this time we do not include variable $?Y$ in the head of the rule as we are not interested in retrieving names. As a third example, consider the following graph pattern $P_3$:

$$(?X, \text{name}, ?Y) \quad \text{OPT} \quad (?X, \text{phone}, ?Z),$$

where phone is a constant. For every constant $a$ in an RDF graph $G$, this graph pattern is asking for the name and phone number of $a$, if the information about the phone number of $a$ is available in $G$, and otherwise it is only asking for the name of $a$. To represent this graph pattern in Datalog$^{\neg s}$, we first assume that $Q_1$, $Q_2$ are basic graph patterns $(?X, \text{name}, ?Y)$ and $(?X, \text{phone}, ?Z)$, respectively, and we construct, as before, the following rules to represent them:

$$\text{triple}(?X, \text{name}, ?Y) \quad \rightarrow \quad \text{query}_{Q_1}(?X, ?Y) \tag{6}$$
$$\text{triple}(?X, \text{phone}, ?Z) \quad \rightarrow \quad \text{query}_{Q_2}(?X, ?Z). \tag{7}$$

Predicates $\text{query}_{Q_1}$ and $\text{query}_{Q_2}$ are used in the representation of graph pattern $P_3$ in Datalog$^{\neg s}$. More precisely, we first construct a set of rules for the cases where the information about phone numbers is available:

$$\text{query}_{Q_1}(?X, ?Y), \text{query}_{Q_2}(?X, ?Z) \rightarrow \text{query}_{P_3}(?X, ?Y, ?Z) \tag{8}$$
$$\text{query}_{Q_1}(?X, ?Y), \text{query}_{Q_2}(?X, ?Z) \rightarrow \text{compatible}_{P_3}(?X). \tag{9}$$

As for the previous graph patterns, we use a predicate $\text{query}_{P_3}$ to store the answers to the query. But in this case, we also include a predicate $\text{compatible}_{P_3}$, which is used to store the list of individuals with phone numbers. This predicate is used in the definition of the third rule utilised to represent graph pattern $P_3$, which takes care of the individuals without phone numbers:

$$\text{query}_{Q_1}(?X, ?Y), \neg\text{compatible}_{P_3}(?X) \rightarrow \text{query}^{\{3\}}_{P_3}(?X, ?Y). \tag{10}$$

Notice that in this case a binary predicate $\text{query}^{\{3\}}_{P_3}$ is used to store the answer, which has a supra-index $\{3\}$ to indicate that the third argument in the answer to $P_3$ is missing (which is the phone number). □

The approach shown in Example 3 can be generalised to represent any graph pattern. To formally prove this claim, we need to introduce some terminology. Assume first that $P = \{\mathbf{t}_1, \ldots, \mathbf{t}_n\}$ is a basic graph pattern such that $\mathbf{t}_i = (u_i, v_i, w_i)$ for every $i \in \{1, \ldots, n\}$, and $\text{var}(P) = \{?X_1, \ldots, ?X_k\}$. Then define as follows a Datalog program $\tau_{\text{bgp}}(P)$ encoding $P$. Assume that $\varsigma$ is a substitution such that for every symbol $u$ occurring in $P$, it holds that $\varsigma(u) = u$ if $u \in (\mathbf{U} \cup \mathbf{V})$, and $\varsigma(u)$ is a fresh variable if $u$ is a blank node. Then $\tau_{\text{bgp}}(P)$ is defined as:

$$\text{triple}(\varsigma(u_1), \varsigma(v_1), \varsigma(w_1)), \ldots, \text{triple}(\varsigma(u_n), \varsigma(v_n), \varsigma(w_n)) \rightarrow \text{query}_P(?X_1, \ldots, ?X_k)$$

For example, if $P_2$ is the basic graph pattern $(?X, \text{name}, B)$ mentioned in Example 3, then $\tau_{\text{bgp}}(P_2)$ consists of the rule (5). Now assume that $P$ is a graph pattern. Then define $\tau_{\text{bgp}}(P)$ as the Datalog program consisting of the rules $\tau_{\text{bgp}}(Q)$ for every basic graph pattern $Q$ occurring in $P$. For example, if $P_3$ is the pattern

$$(?X, \text{name}, ?Y) \ \text{OPT} \ (?X, \text{phone}, ?Z)$$

mentioned in Example 3, then $\tau_{\text{bgp}}(P_3)$ consists of the rules (6) and (7). Moreover, define $\tau_{\text{opr}}(P)$ as a Datalog$^{\neg s}$ program representing the non-basic graph patterns occurring in $P$. Due to the lack of space, we do not provide these rules here, we just point out that these rules are used to encode the semantics of the SPARQL operators occurring in $P$ as shown in Example 3. In fact, if $P_3$ is again the graph pattern $(?X, \text{name}, ?Y)$ OPT $(?X, \text{phone}, ?Z)$ mentioned in this example, then $\tau_{\text{opr}}(P_3)$ consists of the rules (8), (9) and (10).

Let $P$ be a graph pattern. The union of $\tau_{\text{bgp}}(P)$ and $\tau_{\text{opr}}(P)$ forms the translation of $P$. Notice that $\tau_{\text{bgp}}(P)$ and $\tau_{\text{opr}}(P)$ are kept separately, as $\tau_{\text{bgp}}(P)$ is used as a bridge from the data stored in an RDF graph and the Datalog$^{\neg s}$ program $\tau_{\text{opr}}(P)$ used to compute the answers to $P$. Thus, the modular Datalog$^{\neg s}$ query representing $P$ is defined as:

$$\tau_{\text{dat}}(P) \quad = \quad [(\varnothing, \tau_{\text{bgp}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))],$$

where $\tau_{\text{out}}(P)$ is a set of answer rules for the Datalog$^{\neg s}$ program $\tau_{\text{opr}}(P)$, which is defined as follows. Recall that some atoms of the form $\text{query}^J_P(?X_1, \ldots, ?X_k)$ occur in $\tau_{\text{opr}}(P)$, where $J$ is a set of indexes. For example, if $P_3$ is defined as in the previous paragraph, then $\text{query}_{P_3}(?X, ?Y, ?Z)$ and $\text{query}^{\{3\}}_{P_3}(?X, ?Y)$ occur in $\tau_{\text{opr}}(P_3)$ (if $J = \varnothing$, then we use $\text{query}_P(?X_1, \ldots, ?X_k)$ instead of $\text{query}^{\varnothing}_P(?X_1, \ldots, ?X_k)$). Then for every atom $\text{query}^J_P(?X_1, \ldots, ?X_k)$ occurring in $P$, the following copying rule is included in $\tau_{\text{out}}(P)$:

$$\text{query}^J_P(?X_1, \ldots, ?X_k) \rightarrow \text{answer}^J_P(?X_1, \ldots, ?X_k).$$

In order to prove the correctness of our translation, we need to define one last term. Let $P$ be a graph pattern, $G$ an RDF graph and $\underline{a} = \text{answer}^J_P(t_1, \ldots, t_k)$ an atom in $\text{ans}(\tau_{\text{dat}}(P), \tau_{\text{db}}(G))$. By construction, in the set of answer rules $\tau_{\text{out}}(P)$ there is a single atom $\text{answer}^J_P(?X_1, \ldots, ?X_k)$ having $\text{answer}^J_P$ as its predicate (which may occur in several rules). Then define as follows a mapping $\mu_{\underline{a},P}$ corresponding to $\underline{a}$ given $P$: $\text{dom}(\mu_{\underline{a},P}) = \{?X_1, \ldots, ?X_k\}$ and $\mu_{\underline{a},P}(?X_i) = t_i$ for every $i \in \{1, \ldots, k\}$. Moreover, define as follows a set of mappings corresponding to the answers of $\tau_{\text{dat}}(P)$ given $\tau_{\text{db}}(G)$:

$$[\![(\tau_{\text{dat}}(P), \tau_{\text{db}}(G))]\!] \ = \ \{\mu_{\underline{a},P} \mid \underline{a} \in \text{ans}(\tau_{\text{dat}}(P), \tau_{\text{db}}(G))\}.$$

With this notation, we are ready to prove that our translation is correct.

THEOREM 2. *For every graph pattern $P$ and RDF graph $G$, it holds that $[\![P]\!]_G = [\![(\tau_{dat}(P), \tau_{db}(G))]\!]$.*

## 5.2 SPARQL entailment regime and TriQ

As pointed out before, several functionalities were included in SPARQL 1.1 [21] to overcome some of the limitations of the first version of this language. In particular, SPARQL 1.1 includes an entailment regime to deal with RDFS and OWL vocabularies [17, 22]. In this section, we show that this functionality can be encoded by using TriQ.

We start by indicating how OWL 2 QL ontologies are stored as RDF graphs in our setting. In the specification of OWL 2 [20], it is defined a standard syntax to represent OWL 2 ontologies as RDF triples. For the sake of readability, we use here a simplified version of this syntax, having in mind that the results of this section can be readily adapted to the standard syntax. More precisely, define the vocabulary $\Sigma$ of an OWL 2 QL ontology as a finite set of unary and binary predicates, which are called classes and properties, respectively. Moreover, define a basic property over $\Sigma$ as either $p$ or $p^-$, where $p$ is a property in $\Sigma$, and define a basic class over $\Sigma$ as either $a$ or $\exists r$, where $a$ is a class in $\Sigma$ and $r$ is a basic property over $\Sigma$. Then to represent an OWL 2 QL ontology over a vocabulary $\Sigma$, we first include the following triples to indicate what the classes and properties in $\Sigma$ are. For every class $a$ in $\Sigma$, we include the triple $(a, \text{rdf:type}, \text{owl:Class})$. Notice that this triple uses the reserved URIs rdf:type and owl:Class, and indicates that $a$ is of type (rdf:type) class (owl:Class). Moreover, for every property $p$ in $\Sigma$, we include the following triples:

$$\begin{array}{ll}
(p, \text{rdf:type}, \text{owl:Prop}) & (p, \text{owl:inv}, p^-) \\
(p^-, \text{rdf:type}, \text{owl:Prop}) & (p^-, \text{owl:inv}, p) \\
(\exists p, \text{owl:rest}, p) & (\exists p, \text{rdf:type}, \text{owl:Class}) \\
(\exists p^-, \text{owl:rest}, p^-) & (\exists p^-, \text{rdf:type}, \text{owl:Class})
\end{array}$$

The triples $(p, \text{rdf:type}, \text{owl:Prop})$, $(p^-$, rdf:type, owl:Prop$)$ indicate that $p$ and $p^-$ are properties (owl:Prop). It is important to notice that $p$ and $p^-$ are assumed to be (distinct) URIs. The triple $(p^-, \text{owl:inv}, p)$ indicates that $p^-$ is the inverse property of $p$, while triples $(\exists p, \text{owl:rest}, p)$, $(\exists p^-, \text{owl:rest}, p^-)$ indicate that $\exists p$ and $\exists p^-$ are restrictions of $p$ and $p^-$, respectively, where $\exists p$ and $\exists p^-$ are also assumed to be URIs. Finally, $(\exists p, \text{rdf:type}, \text{owl:Class})$ and $(\exists p^-, \text{rdf:type}, \text{owl:Class})$ indicate that $\exists p$ and $\exists p^-$ are classes. It is important to notice that the notation used above is a simplification of the notation used in OWL 2, as owl:Prop, owl:inv and owl:rest correspond to the OWL 2 keywords owl:ObjectProperty, owl:inverseOf and owl:Restriction, respectively. Besides, a triple such as $(\exists p, \text{owl:rest}, p)$ is represented by means of several triples in OWL 2, as shown in Section 2.

In order to represent the axioms in an OWL 2 QL ontology, we include the following triples. To indicate that a basic class $b_1$ is a sub-class of a basic class $b_2$, we include the triple $(b_1, \text{rdfs:sc}, b_2)$. Similarly, to indicate that a basic property $r_1$ is a sub-property of a basic property $r_2$, we include the triple $(r_1, \text{rdfs:sp}, r_2)$. Finally, to indicate that basic classes $b_1$ and $b_2$ are disjoint, we include the triple $(b_1, \text{owl:disj}, b_2)$. It should be noticed that, as before, rdfs:sc, rdfs:sp and owl:disj are shorthands for the keywords rdfs:subClassOf, rdfs:subPropertyOf and owl:disjointWith, respectively, which can be used in OWL 2.

Finally, in order to represent the membership assertions in an OWL 2 QL ontology, we include the following triples. To indicate that a constant $a$ belong to a basic class $b$, we include the triple $(a, \text{rdf:type}, b)$. Similarly, to indicate that a constant $a_1$ is related to a constant $a_2$ through a property $p$, we include the triple $(a_1, p, a_2)$.

From now on, we say that an RDF graph $G$ represents an OWL 2 QL ontology if there exists an OWL 2 QL ontology $\mathcal{O}$ such that, the translation into RDF of $\mathcal{O}$ according to the previous rules generates $G$.

As a second step in our construction, we show how a graph pattern is evaluated under the OWL 2 direct semantics entailment regime defined in [17]. To compute the answer to a graph pattern, this regime is first applied at the level of basic graph patterns, and then the results of this step are combined using the standard semantics for the SPARQL operators [22]. Thus, we only need to define the OWL 2 direct semantics entailment regime for basic graph patterns.

Assume that $P$ is a basic graph pattern. Under the OWL 2 direct semantics entailment regime, the evaluation of $P$ over an RDF graph $G$ adopts an active domain semantics, that is, it uses the notion of entailment in OWL 2 QL (which corresponds to the notion of entailment in DL-Lite$_\mathcal{R}$) but allowing the variables and blank nodes in $P$ to take only values from $G$. For example, assume that we are given an RDF graph $G$ consisting of the following triples:

$$(\text{dog}, \text{rdf:type}, \text{animal}) \;\; (\text{animal}, \text{rdfs:sc}, \exists\text{eats}), \qquad (11)$$

which indicate that dog is an animal, and every animal eats something. Moreover, assume that we want to retrieve the list of elements of $G$ that eat something. The natural way to formulate this query is by using a graph pattern of the form $(?X, \text{eats}, B)$, where $B$ is a blank node. However, the answer to this query is empty under the OWL 2 direct semantics entailment regime, as there are no elements $a$, $b$ in $G$ that can be assigned to $?X$ and $B$ in such a way that the triple $(a, \text{eats}, b)$ is implied by the axioms in $G$. In other words, the answer to $(?X, \text{eats}, B)$ is empty under the active domain semantics adopted in SPARQL 1.1. To obtain a correct answer in this case, we can consider the graph pattern $(?X, \text{rdf:type}, \exists\text{eats})$, as the triples in $G$ can be used to infer the triple $(\text{dog}, \text{rdf:type}, \exists\text{eats})$, from which the correct answer dog is obtained.

Let $G$ be an RDF graph representing an OWL 2 QL ontology. Given a triple $\mathbf{t} \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$, we use notation $G \models \mathbf{t}$ to indicate that $\mathbf{t}$ is implied by $G$ as defined in [25, 17], which in turn is based on the notion of entailment for DL-Lite$_\mathcal{R}$ [12]. Moreover, given a basic graph pattern $P$, the evaluation of $P$ over $G$ under the OWL 2 direct semantics entailment regime, denoted by $[\![P]\!]^{\mathbf{U}}_G$, is defined as:

$$\{\mu \mid \text{dom}(\mu) = \text{var}(P) \text{ and there exists } h : \mathbf{B} \to \mathbf{U} \text{ such that for every } \mathbf{t} \in \mu(h(P))\text{: } G \models \mathbf{t}\}. \qquad (12)$$

Notice that the supra-index $\mathbf{U}$ in $[\![P]\!]^{\mathbf{U}}_G$ is used to indicate that every variable and blank node in $P$ has to be assigned a constant, as $\mathbf{U}$ is the range of functions $h$ and $\mu$ in the previous definition. Moreover, the evaluation of a graph pattern $P$ over an RDF graph $G$ under the OWL 2 direct semantics entailment regime, denoted by $[\![P]\!]^{\mathbf{U}}_G$, is recursively defined as the usual semantics for graph patterns (which is given in Section 3) but replacing the rule for evaluating basic graph patterns by rule (12). In what follows, we define a fixed Datalog$^{\exists,\neg s,\bot}$ program $\tau_{\text{owl2ql}}$ that is used to encode the semantics $[\![\cdot]\!]^{\mathbf{U}}_G$. In this program, we first include a Datalog rule to store in a unary predicate C all the URIs from the graph (recall that we assume that an RDF graph does not contain any blank nodes):

$$\text{triple}(?X, ?Y, ?Z) \;\; \to \;\; \text{C}(?X), \text{C}(?Y), \text{C}(?Z). \qquad (13)$$

Then we define some Datalog rules that store the different elements in the ontology:

$$\begin{array}{rcl}
\text{triple}(?X, \text{rdf:type}, ?Y) & \to & \text{type}(?X, ?Y) \\
\text{triple}(?X, \text{rdfs:sp}, ?Y) & \to & \text{sp}(?X, ?Y) \\
\text{triple}(?X, \text{owl:inv}, ?Y) & \to & \text{inv}(?X, ?Y) \\
\text{triple}(?X, \text{owl:rest}, ?Y) & \to & \text{rest}(?X, ?Y) \\
\text{triple}(?X, \text{rdfs:sc}, ?Y) & \to & \text{sc}(?X, ?Y) \\
\text{triple}(?X, \text{owl:disj}, ?Y) & \to & \text{disj}(?X, ?Y) \\
\text{triple}(?X, ?Y, ?Z) & \to & \text{triple}_1(?X, ?Y, ?Z)
\end{array}$$

If we have triples $(a, \text{rdf:type}, b)$, $(b, \text{rdfs:sc}, \exists r)$ in an OWL 2 QL ontology, then the Datalog$^{\exists,\neg s,\bot}$ program $\tau_{\text{owl2ql}}$ will create a triple of the form $(a, r, z)$, where $z$ is a null value. If $(a, r, z)$ is stored in the relation triple, then by using rule (13) we will conclude that $\text{C}(z)$ holds, violating the intended interpretation of predicate C. To solve this problem, we include the last Datalog rule above to produce a copy of the predicate triple in the predicate $\text{triple}_1$. In this way, the new values are added to $\text{triple}_1$, that is, we do not modify the predicate triple but instead we have that both $\text{triple}_1(a, \text{rdf:type}, b)$ and $\text{triple}_1(b, \text{rdfs:sc}, \exists r)$ hold, from which we conclude that $\text{triple}_1(a, r, z)$ also holds. Moreover, we include the following rules to reason about properties:

$$\begin{aligned}\text{sp}(?X_1, ?X_2), \text{inv}(?Y_1, ?X_1),&\\ \text{inv}(?Y_2, ?X_2) &\rightarrow \text{sp}(?Y_1, ?Y_2)\\ \text{type}(?X, \text{owl:Prop}) &\rightarrow \text{sp}(?X, ?X)\\ \text{sp}(?X, ?Y), \text{sp}(?Y, ?Z) &\rightarrow \text{sp}(?X, ?Z)\end{aligned}$$

we include the following rules to reason about classes:

$$\begin{aligned}\text{sp}(?X_1, ?X_2), \text{rest}(?Y_1, ?X_1),&\\ \text{rest}(?Y_2, ?X_2) &\rightarrow \text{sc}(?Y_1, ?Y_2)\\ \text{type}(?X, \text{owl:Class}) &\rightarrow \text{sc}(?X, ?X)\\ \text{sc}(?X, ?Y), \text{sc}(?Y, ?Z) &\rightarrow \text{sc}(?X, ?Z)\end{aligned}$$

and we include the following rule to reason about disjointness constraints:

$$\begin{aligned}\text{disj}(?X_1, ?X_2), \text{sc}(?Y_1, ?X_1),&\\ \text{sc}(?Y_2, ?X_2) &\rightarrow \text{disj}(?Y_1, ?Y_2)\end{aligned}$$

Finally, we include the following rules to reason about membership assertions:

$$\begin{aligned}&\text{triple}_1(?X, ?U, ?Y), \text{sp}(?U, ?V) \rightarrow \text{triple}_1(?X, ?V, ?Y)\\ &\text{triple}_1(?X, ?U, ?Y), \text{inv}(?U, ?V) \rightarrow \text{triple}_1(?Y, ?V, ?X)\\ &\text{type}(?X, ?Y), \text{rest}(?Y, ?U) \rightarrow \exists ?Z\, \text{triple}_1(?X, ?U, ?Z)\\ &\text{type}(?X, ?Y), \text{sc}(?Y, ?Z) \rightarrow \text{type}(?X, ?Z)\\ &\text{triple}_1(?X, ?U, ?Y), \text{rest}(?Z, ?U) \rightarrow \text{type}(?X, ?Z)\\ &\text{type}(?X, ?Y), \text{type}(?X, ?Z), \text{disj}(?Y, ?Z) \rightarrow \bot\end{aligned}$$

Given a graph pattern $P$ and an RDF graph $G$, to compute $[\![P]\!]^{\mathbf{U}}_G$ we need to include $\tau_{\text{owl2ql}}$ in the modular $\text{Datalog}^{\neg s}$ query $\tau_{\text{dat}}(P)$ defined in Section 5.1. More precisely, assuming that $\tau_{\text{dat}}(P) = [(\varnothing, \tau_{\text{bgp}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))]$, we need to replace $\varnothing$ by $\tau_{\text{owl2ql}}$ in the first component of $\tau_{\text{dat}}(P)$, but taking into consideration the active domain semantics in the entailment regime just defined. For example, assume that $P$ is the basic graph pattern $(?X, \text{eats}, B)$ and $G$ is the RDF graph in (11) storing information about animals. Then we have that $\tau_{\text{bgp}}(P)$ is the following answer rule:

$$\text{triple}(?X, \text{eats}, ?Y) \quad \rightarrow \quad \text{query}_P(?X). \tag{14}$$

In order to combine this rule with $\tau_{\text{owl2ql}}$, we first need to consider the fact that all the triples inferred by using the axioms in $G$ are stored in the predicate $\text{triple}_1$. Thus, we need to replace triple by $\text{triple}_1$ in the rule (14):

$$\text{triple}_1(?X, \text{eats}, ?Y) \quad \rightarrow \quad \text{query}_P(?X).$$

Moreover, we need to enforce the constraint that every variable and blank node in $P$ can only take a value from $G$ (the active domain semantics restriction), which is done by including the predicate C:

$$\text{triple}_1(?X, \text{eats}, ?Y), \text{C}(?X), \text{C}(?Y) \quad \rightarrow \quad \text{query}_P(?X). \tag{15}$$

Thus, given a graph pattern $P$, let $\tau^{\mathbf{U}}_{\text{bgp}}(P)$ be the set of answer rules obtained from $\tau_{\text{bgp}}(P)$ by first replacing triple by $\text{triple}_1$ in every rule of $\tau_{\text{bgp}}(P)$, and then adding $\text{C}(?X)$ in the body of every resulting rule $\rho$ if $?X$ occurs in $\rho$. Moreover, let $\tau^{\mathbf{U}}_{\text{dat}}(P) = [(\tau_{\text{owl2ql}}, \tau^{\mathbf{U}}_{\text{bgp}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))]$. Then it is possible to prove that:

THEOREM 3. *For every graph pattern $P$ and RDF graph $G$, it holds that $[\![P]\!]^{\mathbf{U}}_G = [\![(\tau^{\mathbf{U}}_{dat}(P), \tau_{db}(G))]\!]$.*

Moreover, we have that:

PROPOSITION 4. *For every graph pattern $P$, $\tau^{\mathbf{U}}_{dat}(P)$ is a* TriQ *query.*

## 5.3 Removing the active domain restriction

Consider the basic graph pattern:

$$Q_0 \quad = \quad \{(?X, \text{eats}, B), (B, \text{rdf:type}, \text{plant\_material})\},$$

which is asking for the lists of animals that eat some plant material, and assume that $G$ is an RDF graph. Under the active domain semantics, $a$ is an answer to $Q_0$ over $G$ if we can replace blank node $B$ by a specific plant material $b$ such that $G$ implies $(?X, \text{eats}, b)$. But what happens if such a concrete witness cannot be found in $G$, and we can only infer that $a$ is an answer to $Q_0$ by using the axioms in the ontology. For example, this could happens if $G$ stores information only about herbivores, so it includes the axiom $(\exists \text{eats}^-, \text{rdfs:sc}, \text{plant\_material})$. In this case, $Q_0$ has to be replaced by a basic graph pattern of the form:

$$\{(?X, \text{rdf:type}, \exists \text{eats}), (\exists \text{eats}^-, \text{rdfs:sc}, \text{plant\_material})\}$$

in order to obtain the correct answers. And even worse, what happens if the query has to be distributed over several RDF graphs, which is a very common scenario in the Web. Then the user is forced to use a graph pattern of the form:

$$\begin{aligned}&\{(?X, \text{eats}, B), (B, \text{rdf:type}, \text{plant\_material})\} \text{ UNION}\\ &\{(?X, \text{rdf:type}, \exists \text{eats}), (\exists \text{eats}^-, \text{rdfs:sc}, \text{plant\_material})\},\end{aligned}$$

in which some inferences have to be encoded. All these issues can be solved if we do not force $B$ to take values only in $G$, as this allows us to use the initial basic graph pattern $Q_0$. This gives rise to the semantics $[\![P]\!]^{\text{ALL}}_G$ that is defined exactly as $[\![P]\!]^{\mathbf{U}}_G$, but considering every basic graph pattern as a conjunctive query, and treating blank nodes as existential variables that are not forced to take only values in $G$ (they can take values in the interpretations of $G$).

At this point, one may be tempted to think that the semantics $[\![\cdot]\!]^{\text{ALL}}$ can be directly defined by transforming every basic graph pattern into a conjunctive query, which has to be evaluated over a DL ontology. In fact, this approach works well with our initial query $Q_0$, which can be transformed into the conjunctive query $\exists Y\, \text{eats}(X, Y) \wedge \text{plant\_material}(Y)$. However, there are simple queries for which this approach does not work. For instance, consider the basic graph pattern $(?X, \text{rdfs:sc}, \exists \text{eats})$. Given that $?X$ is used to store class names, this pattern cannot be transformed into a conjunctive query in order to define its semantics; instead we need to replace $?X$ by every class name $C$, and then verify whether the inclusion $C \sqsubseteq \exists \text{eats}$ is implied by the DL ontology in order to define its semantics. Thus, the goal of this section is to show that the more natural semantics $[\![\cdot]\!]^{\text{ALL}}$ can be easily defined by using modular $\text{Datalog}^{\exists, \neg s, \bot}$, without needing to differentiate between variables that are used to store individuals, classes or properties.

Given a basic graph pattern $Q$, let $\tau^{\text{ALL}}_{\text{bgp}}(Q)$ be the answer rule obtained from $\tau^{\mathbf{U}}_{\text{bgp}}(Q)$ by removing every atom of the form $\text{C}(?X)$ such that $?X \notin \text{var}(Q)$ (that is, every atom $\text{C}(?X)$ such that $?X$ is a variable associated to a blank node occurring in $Q$). For example, assume that $P$ is the basic graph pattern $(?X, \text{eats}, B)$. Then we have that $\tau^{\mathbf{U}}_{\text{bgp}}(P)$ is the rule (15), from which we conclude that $\tau^{\text{ALL}}_{\text{bgp}}(P)$ is the following rule:

$$\text{triple}_1(?X, \text{eats}, ?Y), \text{C}(?X) \quad \rightarrow \quad \text{query}_Q(?X).$$

Moreover, given a graph pattern $P$, define $\tau_{\text{bgp}}^{\text{ALL}}(P)$ as the Datalog program consisting of the rules $\tau_{\text{bgp}}^{\text{ALL}}(Q)$ for every basic graph pattern $Q$ occurring in $P$. Finally, define $\tau_{\text{dat}}^{\text{ALL}}(P)$ as $[(\tau_{\text{owl2ql}}, \tau_{\text{bgp}}^{\text{ALL}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))]$. With this very simple modification of $\tau_{\text{dat}}^{\mathbf{U}}(P)$, we can formally define the semantics $[\![\cdot]\!]^{\text{ALL}}$:

*Definition 1.* Given a graph pattern $P$ and an RDF graph $G$, define $[\![P]\!]_G^{\text{ALL}}$ as $[\![(\tau_{\text{dat}}^{\text{ALL}}(P), \tau_{\text{db}}(G))]\!]$.

We conclude by pointing out that $\tau_{\text{dat}}^{\text{ALL}}(P)$ is a TriQ query, for every graph pattern $P$. Thus, this query language is expressive enough to represent the OWL 2 direct semantics entailment regime for the case of OWL 2 QL ontologies, even if the active domain restriction is not imposed.

## 6. A TRACTABLE QUERY LANGUAGE

TriQ forms a natural language which embeds the fundamental features for querying RDF, as shown in Section 5. Unfortunately, Theorem 1 shows that this language is highly intractable in data complexity. Then the question that comes up is whether a fragment of this language exists which is powerful enough for expressing every SPARQL query under the entailment regime for OWL 2 QL, and at the same time ensures the tractability of query evaluation. Towards the identification of such a sublanguage, we single out a fragment of TriQ for which the query evaluation problem can be reduced in polynomial time to the query evaluation problem for linear Datalog$^{\exists}$. Recall that a rule $\rho$ is called linear if body$(\rho)$ contains only one atom. Interestingly, the query evaluation problem for linear Datalog$^{\exists}$ can be reduced to first-order query evaluation [9]. Thus, our reduction allows us not only to find the desired tractable fragment, but also to exploit the mature and efficient relational database technology to answer queries in this fragment.

### 6.1 The query language TriQ-Lite

After a careful analysis of ex$(\tau_{\text{owl2ql}})$, that is, the program obtained after eliminating the constraint occurring in $\tau_{\text{owl2ql}}$, we observed that it enjoys the following interesting property. Let $D$ be a database and $\rho \in$ ex$(\tau_{\text{owl2ql}})$. If $\rho$ is triggered with a homomorphism $h$ during the construction of chase(ex$(\tau_{\text{owl2ql}}), D)$, then for each body-variable $?V$ of $\rho$ that participates in a join operation (i.e., appears more than once in body$(\rho)$) we have that $h(?V) \in$ dom$(D)$. Inspired by this observation, we introduce a syntactic condition that is sufficient to ensure the above semantic property.

Let $\Pi$ be a Datalog$^{\exists,\neg s,\perp}$ program. $\Pi$ is called *constant-join* if for every $\rho \in$ ex$(\Pi)$ and every variable $?V \in$ var(body$(\rho)$) that occurs more than once in body$(\rho)$, it holds that $?V$ appears in body$^+(\rho)$ at a position of nonaffected(ex$(\Pi)$).

*Example 4.* Consider the Datalog$^{\exists,\neg s}$ program $\Pi$:

$$\begin{array}{rcl} p(?X,?Y), s(?Y,?Z) & \rightarrow & \exists ?W\, t(?Y,?X,?W) \\ t(?X,?Y,?Z) & \rightarrow & \exists ?W\, p(?W,?Z) \\ s(?X,?Y), \neg r(?X), \neg r(?Y) & \rightarrow & \exists ?Z\, q(?X,?Z). \end{array}$$

Clearly, affected$(\Pi) = \{t[3], p[1], q[2], p[2], t[2]\}$. The first rule is constant-join since at least one occurrence of the variable $?Y$ appears at the non-affected position $s[1]$. The second rule is trivially constant-join since each variable occurs in its body only once. Finally, the third rule is constant-join since both $?X$ and $?Y$ occur in a positive atom at a non-affected position. Therefore, $\Pi$ is a constant-join program. □

A Datalog$^{\exists,\neg s,\perp}$ query $(\Pi, \Lambda)$ is constant-join if the program $\Pi$ is constant-join. A modular Datalog$^{\exists,\neg s,\perp}$ query $[Q_1, Q_2]$ is called constant-join if both $Q_1$ and $Q_2$ are constant-join. Finally, a TriQ-Lite query is defined as a constant-join TriQ query. Then we have that:

PROPOSITION 5. *For every graph pattern $P$, both $\tau_{dat}^{\mathbf{U}}(P)$ and $\tau_{dat}^{\text{ALL}}(P)$ are* TriQ-Lite *queries.*

By combining Theorem 3 and Proposition 5, we immediately get that:

COROLLARY 6. *Every SPARQL query under the entailment regime for OWL 2 QL can be expressed as a* TriQ-Lite *query.*

After posing the constant-join condition on TriQ, we obtain a language for which the query evaluation problem is tractable in data complexity:

THEOREM 7. *Query evaluation for* TriQ-Lite *is* PTIME-*complete in data complexity.*

Since every Datalog program is a weakly-guarded constant-join Datalog$^{\exists,\neg s,\perp}$ program, every Datalog query can be rewritten as an equivalent TriQ-Lite query. This allows us to deduce the lower bound in Theorem 7, as the query evaluation problem for Datalog is PTIME-hard in data complexity (see, e.g., [14]). Thus, the rest of this section is devoted to establish the membership of our problem in PTIME.

The problem of checking whether a database $D$ is inconsistent w.r.t. a TriQ-Lite query $M = [Q_1, Q_2]$ can be reduced to the query evaluation problem for weakly-guarded constant-join Datalog$^{\exists,\neg s}$. To check whether $D$ is inconsistent w.r.t. $Q_1 = (\Pi_1, \Lambda_1)$, we first construct the weakly-guarded constant-join Datalog$^{\exists,\neg s}$ query $Q_1' = (\text{ex}(\Pi), \Lambda \cup \Gamma)$, where $\Gamma$ contains an answer rule body$(\nu) \rightarrow p_\nu()$ for each constraint $\nu$ in $\Pi$ (notice that these constraints are the only source of inconsistency). Then we check whether an atom $p_\nu()$ belongs to ans$(Q_1', D)$, which implies that $\nu$ is violated and, thus, $D$ is inconsistent w.r.t. $Q_1$. In the same way, we check whether the instance ans$(Q_1, D)$ is inconsistent w.r.t. $Q_2$.

Whenever the input database is inconsistent with the given query $M$, we return the symbol $\top$. Otherwise, we continue by focusing on the query obtained from $M$ by eliminating the constraints. By definition of the semantics of modular queries, to show that the query evaluation problem for TriQ-Lite is in PTIME in data complexity, it suffices to show that the same problem is in PTIME for weakly-guarded constant-join Datalog$^{\exists,\neg s}$. Thus, Theorem 7 follows from the following result.

PROPOSITION 8. *The query evaluation problem for weakly-guarded constant-join Datalog$^{\exists,\neg s}$ is in* PTIME *in data complexity.*

The key idea underlying the proof of this proposition is to construct a polynomial-time reduction, from the query evaluation problem for weakly-guarded constant-join Datalog$^{\exists,\neg s}$ to the query evaluation problem for linear Datalog$^{\exists}$. This reduction is computed in two steps. In the first step, we eliminate the negation from the given query $(\Pi, \Lambda)$ to produce $(\Pi^+, \Lambda)$. More specifically, assume that $\rho \in \Pi$ and $?V$ is a variable occurring in an atom of body$^-(\rho)$. Then $?V$ must occur in an atom of body$^+(\rho)$ and, therefore, $?V$ must appear at a non-affected position due to the constant-join condition. Hence, in the computation of the chase of a database $D$ with $\Pi$, every predicate in body$^-(\rho)$ can store only constants occurring in $D$. Thus, given that $\Pi$ is stratified, $\Pi^+$ can be computed from $\Pi$ in a standard way, by replacing each negated atom $\neg p(\mathbf{t})$ with a

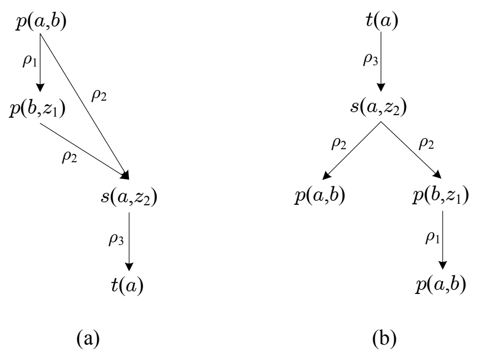

**Figure 1: (a) The part of chase$(D,\Pi)$ which entails $\underline{a} = t(a)$; (b) The proof-tree for $\underline{a} = t(a)$ w.r.t. $D$ and $\Pi$.**

positive atom $\bar{p}(\mathbf{t})$, where the relation $\bar{p}$ stores the complement of $p$ with respect to the ground chase (that is, the atoms of the chase with only constants).

The second step is to convert each non-linear rule in $\Pi^+$ into a linear one. Assume that $\rho \in \Pi$ and $\underline{a}$ is the weak-guard of $\rho$. Since $\Pi^+$ is both weakly-guarded and constant-join, if $\rho$ is triggered during the construction of the chase, then every atom in body$(\rho) \setminus \{\underline{a}\}$ is mapped only to the ground chase. Thus, to convert $\rho$ into a linear rule, we just need to keep the rule $h(\underline{a}) \rightarrow h(\text{head}(\rho))$, for every homomorphism $h$ that maps the atoms in body$(\rho) \setminus \{\underline{a}\}$ to the ground chase. Let us now give some more details regarding the two steps just described.

### 6.1.1 *From Weakly-guarded constant-join Datalog$^{\exists,\neg s}$ to linear Datalog$^{\exists}$*

It is clear that the ground part of the chase plays a central role in our reduction. In what follows, we first explain how this part of the chase is computed. Then we explain more formally how negation can be eliminated, and how non-linear rules can be transformed into linear ones.

**Computing the ground chase.** The *ground chase* of a database $D$ with a Datalog$^{\exists}$ program $\Pi$, denoted by $chase_{\downarrow}(D,\Pi)$, is defined as $\{\underline{a} \in \text{chase}(D,\Pi) \mid \text{dom}(\underline{a}) \subset \mathbf{U}\}$. This instance is constructible in polynomial time w.r.t. $D$ when $\Pi$ is constant-join (recall that $\Pi$ is assumed to be fixed). To prove this, it suffices to show that the problem of deciding whether a ground atom $\underline{a}$ belongs to chase$(D,\Pi)$ is feasible in polynomial time w.r.t. $D$. We propose a recursive alternating algorithm, called Proof, which solves the above problem by constructing a proof-tree of $\underline{a}$ w.r.t. $D$ and $\Pi$ (if it exists). Such a proof-tree is a tree-like representation of the part of the chase on which $\underline{a}$ depends on. For example, if $\Pi$ is

$$\begin{array}{lcl}
\rho_1 & : & p(?X,?Y) \rightarrow \exists ?Z\, p(?Y,?Z)\\
\rho_2 & : & p(?X,?Y), p(?Y,?Z) \rightarrow \exists ?W\, s(?X,?W)\\
\rho_3 & : & s(?X,?Y) \rightarrow t(?X),
\end{array}$$

$D = \{p(a,b)\}$ and $\underline{a} = t(a)$, then the part of the chase on which $\underline{a}$ depends on is depicted in Figure 1(a), while a proof-tree for $\underline{a}$ w.r.t. $D$ and $\Pi$ is shown in Figure 1(b).

Proof$(\underline{a}, D, \Pi)$ constructs a proof-tree (if it exists) by starting from $\underline{a}$ and applying resolution steps until the database $D$ is reached. Whenever a rule $\rho$ is used in a resolution step, the variables of body$(\rho)$ which are not in a join and do not appear in head$(\rho)$ are replaced by the special symbol $\star$, which plays the role of a witness for them. It is clear that for such non-join variables, what is important is not their actual value but the existence of a witness. Then each atom of body$(\rho)$ is considered as a new ground atom which is proved recursively in a universal branch (recall that Proof is an alternating algorithm). At each step of the computation of Proof at most polynomially many constants of dom$(D)$ must be remembered. This can be achieved using logarithmically many bits on $|\text{dom}(D)|$, and thus Proof needs logarithmic space w.r.t. $D$ at each step of its computation. Since ALOGSPACE = PTIME, we immediately get that our algorithm describes a polynomial time procedure w.r.t. $D$, and the next crucial technical result follows:

LEMMA 9. *Consider a database $D$ and a constant-join Datalog$^{\exists}$ program $\Pi$. Then, $chase_{\downarrow}(D,\Pi)$ is constructible in polynomial time w.r.t. $D$.*

Consider a weakly-guarded constant-join Datalog$^{\exists,\neg s}$ query $Q = (\Pi, \Lambda)$ and a database $D$. Having Lemma 9 in place, we are now ready to show how negation can be eliminated, and how non-linear rules can be transformed into linear ones, in order to obtain a linear Datalog$^{\exists}$ query $Q_L = (\Pi_L, \Lambda)$ and a database $D_L \supseteq D$ such that ans$(Q,D)$ = ans$(Q_L, D_L)$.

**Eliminating negation.** Since $\Pi$ is stratified, there is a stratification $\sigma$ : sch$(\Pi) \rightarrow \{0,\ldots,k\}$. Let $\Pi_0, \ldots, \Pi_k$ be the partition of $\Pi$ induced by $\sigma$. Then we denote by $\Pi_i^+$ ($i \in [1,k]$) the program obtained from $\Pi_i$ by replacing each negative atom $\neg p(\mathbf{t})$ with the positive atom $\bar{p}(\mathbf{t})$. Let $sch^-(\Pi_i)$ be the set of predicates occurring in $\Pi_i$ in at least one negative atom. We inductively define $D_k^\star$ and $\Pi_k^\star$ as: (1) $D_0^\star = D$ and $\Pi_0^\star = \Pi_0$; and (2) for each $i \in [1,k]$, $D_i^\star = (D_{i-1}^\star \cup C_{i-1})$, where $C_i$ is the set:

$$\left\{ \bar{p}(\mathbf{u}) \;\middle|\; \begin{array}{l} p \in sch^-(\Pi_i), \mathbf{u} \in (\text{dom}(D))^{\text{arity}(p)} \\ \text{and } p(\mathbf{u}) \notin chase_{\downarrow}(D_{i-1}^\star, \Pi_{i-1}^\star) \end{array} \right\},$$

and $\Pi_i^\star = \Pi_{i-1}^\star \cup \Pi_i^+$. For each $i \in [1,k]$, $\Pi_{i-1}^\star$ is constant-join. Thus, we have by Lemma 9 that $C_{i-1}$ can be constructed in polynomial time. Hence, $D_k^\star$ and $\Pi_k^\star$ are both constructible in polynomial time w.r.t. $D$. Let $D_L = D_k^\star$. It is easy to verify that ans$(Q,D)$ = ans$((\Pi_k^\star, \Lambda), D_L)$.

**Constructing linear Datalog$^{\exists}$ program $\Pi_L$.** Observe that $\Pi_k^\star$ can be partitioned into $\{\Pi_1, \Pi_2\}$, where $\Pi_1$ consists of the linear rules of $\Pi_k^\star$ and $\Pi_2 = \Pi_k^\star \setminus \Pi_1$. Clearly, $\Pi_2$ is a weakly-guarded constant-join Datalog$^{\exists}$ program. For a rule $\rho \in \Pi_2$, let guard$(\rho)$ be the weak-guard of $\rho$, and side$(\rho)$ be the set of atoms (body$(\rho) \setminus \{\text{guard}(\rho)\}$). For each $\rho \in \Pi_2$, let $H_\rho$ be the set of substitutions $\{h \mid h : \text{var}(\text{side}(\rho)) \rightarrow \text{dom}(D)\}$. Moreover, let $\Pi_2'$ be the program $\{h(\text{guard}(\rho)) \rightarrow h(\text{head}(\rho)) \mid \rho \in \Pi_2, h \in H_\rho$ and $h(\text{side}(\rho)) \subseteq chase_{\downarrow}(D_k^\star, \Pi_k^\star)\}$. We have that $\Pi_L = \Pi_1 \cup \Pi_2'$ is a linear Datalog$^{\exists}$ program such that ans$((\Pi_k^\star, \Lambda), D_L)$ and ans$((\Pi_L, \Lambda), D_L)$ coincide. Clearly, for each $\rho \in \Pi_2$, $|H_\rho| = n^k$, where $n = |\text{dom}(D)|$ and $k = |\text{var}(\text{side}(\rho))|$, which implies that $H_\rho$ can be constructed in polynomial time w.r.t. $D$. Hence, $\Pi_L$ is of polynomial size w.r.t. $D$, and it can be constructed in polynomial time w.r.t. $D$ by Lemma 9.

Having the above reduction in place, it is easy to see that query evaluation for weakly-guarded constant-join Datalog$^{\exists,\neg s}$ is feasible in polynomial time in data complexity. At this point, it is important to clarify that the linear Datalog$^{\exists}$ program $\Pi_L$ is not fixed since it depends on the database $D$. However, the problem of computing ans$((\Pi_L, \Lambda), D_L)$ is feasible in polynomial time as $\Lambda$ is fixed (this is implicit in [18]), from which the desired upper bound follows.

## 7. PROGRAM EXPRESSIVE POWER

We have shown that TriQ-Lite is expressive enough to represent the OWL 2 entailment regime for the case of OWL 2 QL. In fact, as shown in Section 5, given a graph pattern $P$ and an RDF graph $G$, the natural semantics $[\![P]\!]_G^{\text{ALL}}$ can be computed via the TriQ-Lite query $\tau^{\text{ALL}}_{\text{dat}}(P) = [(\tau_{\text{owl2ql}}, \tau^{\text{ALL}}_{\text{bgp}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))]$. Importantly, the program $\tau_{\text{owl2ql}}$ does not depend on $P$. In other words, given a new graph pattern $P'$, we just need to construct the programs $\tau^{\text{ALL}}_{\text{bgp}}(P')$, $\tau_{\text{opr}}(P')$ and $\tau_{\text{out}}(P')$ without altering $\tau_{\text{owl2ql}}$ to compute $[\![P']\!]^{\text{ALL}}_G$. This is quite beneficial in practice since, whenever the user wants to pose a new query over an RDF graph, (s)he does not need to change the part of the modular query which encodes the OWL 2 QL ontology. A natural question at this point is whether this favorable behavior can be achieved if $\tau_{\text{owl2ql}}$ is replaced by a Datalog$^{\neg s,\bot}$ program, i.e., without allowing existentially quantified variables in the heads of rules in this program. In this section, we give a negative answer to this question. Given a Datalog$^{\neg s,\bot}$ program $\Pi$, define $\tau^{\text{ALL}}_{\text{dat},\Pi}(P)$ as the query $[(\Pi, \tau^{\text{ALL}}_{\text{bgp}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))]$, i.e., the query obtained by replacing $\tau_{\text{owl2ql}}$ in $\tau^{\text{ALL}}_{\text{dat}}(P)$ with $\Pi$. Then we can show that:

THEOREM 10. *There exist an RDF graph $G$ and a graph pattern $P$ such that, for every Datalog$^{\neg s,\bot}$ program $\Pi$, $[\![P]\!]^{\text{ALL}}_G \neq [\![(\tau^{\text{ALL}}_{dat,\Pi}(P), \tau_{db}(G))]\!]$.*

Let us construct an RDF graph $G$ and a graph pattern $P$ satisfying the statement of Theorem 10. Consider the basic classes $b_1$ and $b_2$, and the basic property $p$. Let $\mathcal{O}$ be the OWL 2 QL ontology which encodes the following:

1. $b_1$ is a sub-class of $b_2$;
2. $b_2$ is a sub-class of $\exists p$;
3. $\exists p^-$ is a sub-class of $b_2$; and
4. the constant $a$ belongs to $b_1$.

Assume that $G$ is the RDF graph which encodes the ontology $\mathcal{O}$ according to the rules given in Section 5.2. Moreover, consider the graph pattern $P$:

$$\{(?X, p, B_1), (B_1, p, B_2)\} \text{ UNION } \{(?X, p, B_3), (B_3, p, ?Y)\}$$

where each $B_i$ ($1 \leq i \leq 3$) is a blank node. Recall that $\tau^{\text{ALL}}_{\text{dat}}(P)$ is the TriQ-Lite query $[(\tau_{\text{owl2ql}}, \tau^{\text{ALL}}_{\text{bgp}}(P)), (\tau_{\text{opr}}(P), \tau_{\text{out}}(P))]$, while $\tau_{\text{db}}(G)$ is the instance associated to $G$. It is not difficult to verify that the atom $\underline{a} = \text{answer}^{\{2\}}_P(a)$ belongs to $\text{ans}(\tau^{\text{ALL}}_{\text{dat}}(P), \tau_{\text{db}}(G))$, which in turn implies that $\mu_{\underline{a},P} \in [\![(\tau^{\text{ALL}}_{\text{dat}}(P), \tau_{\text{db}}(G))]\!]$. However, for every $\underline{a}_b = \text{answer}_P(a, b)$, where $b \in \mathbf{U}$, it holds that $\underline{a}_b \notin \text{ans}(\tau^{\text{ALL}}_{\text{dat}}(P), \tau_{\text{db}}(G))$, and thus $\mu_{\underline{a}_b,P} \notin [\![(\tau^{\text{ALL}}_{\text{dat}}(P), \tau_{\text{db}}(G))]\!]$. But, we can show that for every Datalog$^{\neg s,\bot}$ program $\Pi$, $\underline{a}$ belongs to $\text{ans}(\tau^{\text{ALL}}_{\text{dat},\Pi}(P), \tau_{\text{db}}(G))$ if and only if there exists $b \in \mathbf{U}$ such that $\underline{a}_b \in \text{ans}(\tau^{\text{ALL}}_{\text{dat},\Pi}(P), \tau_{\text{db}}(G))$. From the previous discussion, we conclude that for every Datalog$^{\neg s,\bot}$ program $\Pi$, it must be the case that $[\![(\tau^{\text{ALL}}_{\text{dat}}(P), \tau_{\text{db}}(G))]\!] \neq [\![(\tau^{\text{ALL}}_{\text{dat},\Pi}(P), \tau_{\text{db}}(G))]\!]$. Theorem 10 follows since $[\![P]\!]^{\text{ALL}}_G = [\![(\tau^{\text{ALL}}_{\text{dat}}(P), \tau_{\text{db}}(G))]\!]$.

This result is a strong sign that the existential quantification in rule-heads allows us to obtain a modular query language which is more powerful than Datalog$^{\neg s,\bot}$. In what follows, we give a formal proof of this fact. However, before showing this for modular query languages, we would like to concentrate first on non-modular query languages, and show that weakly-guarded constant-join Datalog$^{\exists,\neg s,\bot}$ is more expressive than Datalog$^{\neg s,\bot}$ – towards this direction, we introduce the notion of *program expressive power*. This is an interesting result on its own, stressing out the importance of the existentially quantified variables in rule-heads even for non-modular query languages. Henceforth, given a (modular) query language $\mathcal{L}$, a program that can appear in a (modular) query which falls in $\mathcal{L}$ is called $\mathcal{L}$-program.

Consider a query language $\mathcal{L}$, and a program $\Pi$. The program expressive power of $\Pi$ relative to $\mathcal{L}$, denoted by $\mathsf{Pep}_{\mathcal{L}}[\Pi]$, is defined as the set of triples $(D, \Lambda, \underline{a})$ such that $(\Pi, \Lambda)$ is a query in $\mathcal{L}$, and $\underline{a} \in \text{ans}((\Pi, \Lambda), D)$. Notice that if $\Pi$ is not an $\mathcal{L}$-program, then $\mathsf{Pep}_{\mathcal{L}}[\Pi] = \varnothing$. In fact, $\mathsf{Pep}_{\mathcal{L}}[\Pi]$ encodes the set of atoms that can be inferred from a database $D$ via a query $Q$ in $\mathcal{L}$, where $\Pi$ is the program of $Q$. It is now natural to define the program expressive power of $\mathcal{L}$ as $\mathsf{Pep}[\mathcal{L}] = \{\mathsf{Pep}_{\mathcal{L}}[\Pi] \mid \Pi \text{ is an } \mathcal{L}\text{-program}\}$. In other words, $\mathsf{Pep}[\mathcal{L}]$ is a family of sets of triples, where each of its members encodes the program expressive power of an $\mathcal{L}$-program relative to $\mathcal{L}$. Given two languages $\mathcal{L}$ and $\mathcal{L}'$, we write $\mathcal{L}' \preceq_{\mathsf{Pep}} \mathcal{L}$ if $\mathsf{Pep}[\mathcal{L}'] \subseteq \mathsf{Pep}[\mathcal{L}]$. Finally, we say that $\mathcal{L}$ is more expressive than $\mathcal{L}'$ w.r.t. the program expressive power, written as $\mathcal{L}' \prec_{\mathsf{Pep}} \mathcal{L}$, if $\mathcal{L}' \preceq_{\mathsf{Pep}} \mathcal{L}$ and $\mathcal{L} \not\preceq_{\mathsf{Pep}} \mathcal{L}'$. By exploiting the construction given in the proof of Theorem 10, we can show the following result:

THEOREM 11. *It holds that,*

*Datalog$^{\neg s,\bot}$ $\prec_{\mathsf{Pep}}$ Weakly-guarded constant-join Datalog$^{\exists,\neg s,\bot}$.*

Notice that the same result can be shown for other Datalog-based languages such as guarded Datalog$^{\exists,\neg s,\bot}$ [9]. Let us now focus on modular query languages. Before we formally state the desired result, we first need to adapt the notion of program expressive power for modular query languages; this will give rise to the notion of *modular program expressive power*.

Consider a modular query language $\mathcal{L}$, a query language $\mathcal{L}'$ and a program $\Pi$. The modular program expressive power of $\Pi$ relative to $\mathcal{L}$ and $\mathcal{L}'$, denoted by $\mathsf{MPep}_{\mathcal{L},\mathcal{L}'}[\Pi]$, is defined as the set of 4-tuples $(D, \Lambda, Q, \underline{a})$ such that $M = [(\Pi, \Lambda), Q]$ is a modular query in $\mathcal{L}$, $Q$ is in $\mathcal{L}'$, and $\underline{a} \in \text{ans}(M, D)$. Notice that if $\Pi$ is not an $\mathcal{L}$-program, then $\mathsf{MPep}_{\mathcal{L},\mathcal{L}'}[\Pi] = \varnothing$. Intuitively, $\mathsf{MPep}_{\mathcal{L},\mathcal{L}'}[\Pi]$ encodes the set of atoms that can be inferred from a database $D$ via a modular query $[Q_1, Q_2]$ in $\mathcal{L}$, where $\Pi$ is the program of $Q_1$ and $Q_2$ is a query in $\mathcal{L}'$. We now define the modular program expressive power of $\mathcal{L}$ relative to $\mathcal{L}'$ as $\mathsf{MPep}_{\mathcal{L}'}[\mathcal{L}] = \{\mathsf{MPep}_{\mathcal{L},\mathcal{L}'}[\Pi] \mid \Pi \text{ is an } \mathcal{L}\text{-program}\}$. In other words, $\mathsf{MPep}_{\mathcal{L}'}[\mathcal{L}]$ is a family of sets of 4-tuples, where each of its members encodes the modular program expressive power of an $\mathcal{L}$-program relative to $\mathcal{L}$ and $\mathcal{L}'$. Given two modular query languages $\mathcal{L}$ and $\mathcal{L}'$, we write $\mathcal{L}' \preceq_{\mathsf{MPep}} \mathcal{L}$ if $\mathsf{MPep}_{\mathcal{L}\cap\mathcal{L}'}[\mathcal{L}'] \subseteq \mathsf{MPep}_{\mathcal{L}\cap\mathcal{L}'}[\mathcal{L}]$[2]. Finally, we say that $\mathcal{L}$ is more expressive than $\mathcal{L}'$ w.r.t. the modular program expressive power, written as $\mathcal{L}' \prec_{\mathsf{MPep}} \mathcal{L}$, if $\mathcal{L}' \preceq_{\mathsf{MPep}} \mathcal{L}$ and $\mathcal{L} \not\preceq_{\mathsf{MPep}} \mathcal{L}'$. As for Theorem 11, by exploiting the construction given in the proof of Theorem 10, we obtain that:

THEOREM 12. *Modular Datalog$^{\neg s,\bot}$ $\prec_{\mathsf{MPep}}$ TriQ-Lite.*

Notice that modular Datalog$^{\neg s,\bot}$ and Datalog$^{\neg s,\bot}$ are equally expressive in the classical sense. However, we need to use modular Datalog$^{\neg s,\bot}$ in the statement of Theorem 12 as we are comparing modular program expressive powers in this theorem.

Several query languages that enhance SPARQL with navigation capabilities and/or recursion mechanisms have been proposed, most notably nSPARQL [27], PSPARQL [2], recursive triple algebra [23], and NEMODEQ [30]. Each one of these languages $\mathcal{L}$ is contained in Datalog$^{\neg s,\bot}$, in the sense that every $\mathcal{L}$-query can be expressed as a Datalog$^{\neg s,\bot}$ query. Therefore, we can consider the Datalog version $\mathcal{L}^{\text{dat}}$ of $\mathcal{L}$, and then we can use $\mathcal{L}^{\text{dat}}$ (resp., modular $\mathcal{L}^{\text{dat}}$) in order to compare the program expressive power (resp., modular program expressive power) of $\mathcal{L}$ and weakly-guarded constant-join Datalog$^{\exists,\neg s,\bot}$ (resp., TriQ-Lite); notice that modular $\mathcal{L}^{\text{dat}}$ is as expressive as $\mathcal{L}^{\text{dat}}$ in the usual sense. Then, from Theorems 11 and 12 we obtain the following result:

[2]Notice that we also use $\mathcal{L}\cap\mathcal{L}'$ as a (non-modular) query language.

COROLLARY 13. *Assume that $\mathcal{L}$ is one of the query languages nSPARQL, PSPARQL, recursive triple algebra and NEMODEQ. Then, the following hold:*

1. $\mathcal{L}^{\text{dat}} \prec_{\mathsf{Pep}}$ *Weakly-guarded constant-join Datalog*$^{\exists,\neg s,\bot}$;
2. *Modular* $\mathcal{L}^{\text{dat}} \prec_{\mathsf{MPep}}$ TriQ-Lite.

## 8. CONCLUSIONS

We considered the problem of bridging the gap between the existing RDF query languages and key features for querying RDF data such as reasoning capabilities, navigational capabilities, and a general form of recursion. A modular query language has been proposed which is expressive enough to encode every SPARQL query under the entailment regime for OWL 2 QL. Moreover, this language allows us to formulate SPARQL queries in a simpler way, as it can easily encode a more natural notion of entailment. Interestingly, the proposed language incorporates the main RDF query languages that can be found in the literature.

## 9. ACKNOWLEDGMENTS

Part of the work of M. Arenas was performed while visiting Oxford, UK, while part of the work of A. Pieris was performed while visiting Santiago, Chile. M. Arenas was supported by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004 and Fondecyt grant 1131049, G. Gottlob was supported by ERC grant 246858 (DIADEM), and A. Pieris was supported by EPSRC grant EP/J008346/1 (PrOQAW). We thank the anonymous referees for many helpful comments.

## 10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.

[3] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *ISWC*, pages 114–129, 2008.

[4] M. Arenas, C. Gutierrez, and J. Pérez. Foundations of RDF databases. In *RW*, pages 158–204, 2009.

[5] P. Barceló. Querying graph databases. In *PODS*, pages 175–188, 2013.

[6] C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In *ICALP*, pages 73–85, 1981.

[7] D. Brickley and R. Guha. RDF vocabulary description language 1.0: RDF schema. W3C Recommendation 10 February 2004, `http://www.w3.org/TR/rdf-schema`.

[8] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.*, 48:115–174, 2013.

[9] A. Calì, G. Gottlob, and T. Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.

[10] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *LICS*, pages 228–242, 2010.

[11] A. Calì, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.

[12] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. Autom. Reasoning*, 39(3):385–429, 2007.

[13] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

[14] E. Dantsin, T. Eiter, G. Georg, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[15] V. Fionda, C. Gutierrez, and G. Pirrò. Semantic navigation on the web of data: specification of routes, web fragments and actions. In *WWW*, pages 281–290, 2012.

[16] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF querying: Language constructs and evaluation methods compared. In *Reasoning Web*, pages 1–52, 2006.

[17] B. Glimm and C. Ogbuji. SPARQL 1.1 entailment regimes. W3C Recommendation 21 March 2013, `http://www.w3.org/TR/sparql11-entailment/`.

[18] G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, pages 1–13, 2011.

[19] G. Gottlob, S. Rudolph, and M. Simkus. Expressiveness of guarded existential rule languages. In *PODS*, 2014. To appear.

[20] W. O. W. Group. OWL 2 web ontology language document overview (second edition). W3C Recommendation 11 December 2012, `http://www.w3.org/TR/owl2-overview/`.

[21] S. Harris and A. Seaborne. SPARQL 1.1 query language. W3C Recommendation 21 March 2013, `http://www.w3.org/TR/sparql11-query/`.

[22] I. Kollia, B. Glimm, and I. Horrocks. SPARQL query answering over owl ontologies. In *ESWC (1)*, pages 382–396, 2011.

[23] L. Libkin, J. L. Reutter, and D. Vrgoc. Trial for RDF: adapting graph query languages for RDF data. In *PODS*, pages 201–212, 2013.

[24] D. L. McGuinness and F. van Harmelen. OWL web ontology language overview. W3C Recommendation 10 February 2004, `http://www.w3.org/TR/owl-features/`.

[25] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 web ontology language profiles (second edition). W3C Recommendation 11 December 2012, `http://www.w3.org/TR/owl2-profiles/`.

[26] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3), 2009.

[27] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: a navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[28] A. Polleres. From sparql to rules (and back). In *WWW*, pages 787–796, 2007.

[29] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation 15 January 2008, `http://www.w3.org/TR/rdf-sparql-query`.

[30] S. Rudolph and M. Krötzsch. Flag & check: data access with monadically defined queries. In *PODS*, pages 151–162, 2013.

[31] S. Schenk. A SPARQL semantics based on Datalog. In *KI*, pages 160–174, 2007.

[32] M. Y. Vardi. The complexity of relational query languages. In *STOC*, pages 137–146, 1982.