nSPARQL: A Navigational Language for RDF [1]

Jorge Pérez [a,c], Marcelo Arenas [a,c], Claudio Gutierrez [b,c]

[a] *Department of Computer Science, Pontificia Universidad Católica de Chile*
[b] *Department of Computer Science, Universidad de Chile*
[c] *Khipu: South Andean Center for Database Research*

Abstract

Navigational features have been largely recognized as fundamental for graph database query languages. This fact has motivated several authors to propose RDF query languages with navigational capabilities. In this paper, we propose the query language nSPARQL that uses *nested regular expressions* to navigate RDF data. We study some of the fundamental properties of nSPARQL and nested regular expressions concerning expressiveness and complexity of evaluation. Regarding expressiveness, we show that nSPARQL is expressive enough to answer queries considering the semantics of the RDFS vocabulary by directly traversing the input graph. We also show that nesting is necessary in nSPARQL to obtain this last result, and we study the expressiveness of the combination of nested regular expressions and SPARQL operators. Regarding complexity of evaluation, we prove that given an RDF graph G and a nested regular expression E, this problem can be solved in time $O(|G| \cdot |E|)$.

1 Introduction

The Resource Description Framework (RDF) [8,18] is the W3C recommendation data model for the representation of information about resources on the Web. The RDF specification includes a set of reserved keywords with its own semantics, the RDFS vocabulary. This vocabulary is designed to describe special relationships between resources like typing and inheritance of classes and properties [8]. As with any data structure designed to model information, a

Email addresses: jperez@ing.puc.cl (Jorge Pérez), marenas@ing.puc.cl (Marcelo Arenas), cgutierr@dcc.uchile.cl (Claudio Gutierrez).

[1] This is an extended and revised version of [7,26].

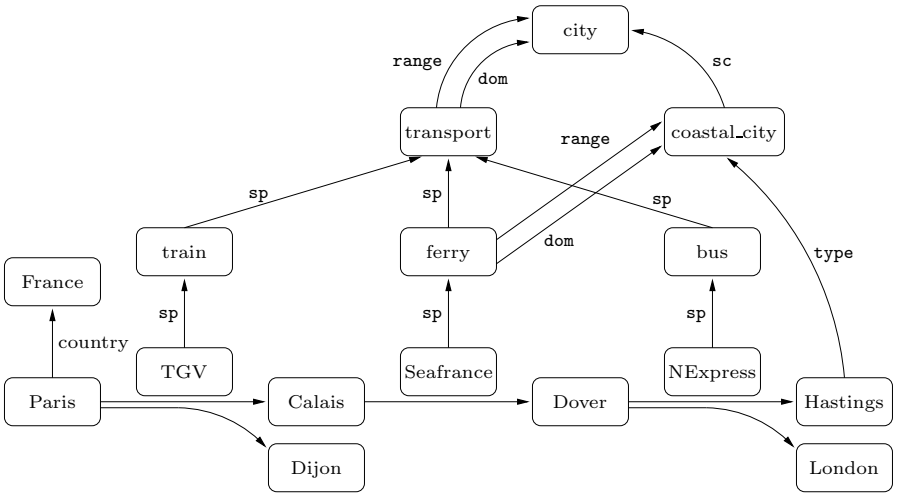


Fig. 1. An RDF graph storing information about transportation services between cities.

natural question that arises is what the desiderata are for an RDF query language. Among the multiple design issues to be considered, it has been largely recognized that navigational capabilities are of fundamental importance for data models with explicit tree or graph structure (like XML and RDF).

Recently, the W3C Working Group issued the specification of a query language for RDF, called SPARQL [27], which is a W3C recommendation since January 2008. SPARQL is designed much in the spirit of classical relational languages such as SQL. It has been noted that, although RDF is a directed labeled graph data format, SPARQL only provides limited navigational functionalities. This is more notorious when one considers the RDFS vocabulary (which current SPARQL specification does not cover), where testing conditions like being a subclass of or a sub-property of naturally requires navigating the RDF data. A good illustration of this is shown by the following query, which cannot be expressed in SPARQL without some navigational capabilities. Consider the RDF graph shown in Fig. 1. This graph stores information about cities, transportation services between cities, and further relationships among those transportation services (in the form of RDFS annotations). For instance, in the graph we have that a "Seafrance" service is a sub-property of a "ferry" service, which in turn is a sub-property of a general "transport" service. Assume that we want to test whether a pair of cities A and B are connected by a sequence of transportation services, but without knowing in advance what services provide those connections. We can answer such a query by testing whether there is a path connecting A and B in the graph, such that every edge in that path is connected with "transport" by following a sequence of sub-property relationships. For instance, for "Paris" and "Calais" the condition holds, since "Paris" is connected with "Calais" by an edge with label "TGV", and "TGV" is a sub-property of "train", which in turn is a sub-property of "transport". Notice that the condition also holds for "Paris" and "Dover".

Driven by these motivations, we introduce a language for navigating RDF data grounded on paths expressed with regular expressions, that takes advantage of the special features of RDF. Besides regular expressions, our proposed language borrows the notion of *branching* from XPath [11], to obtain what we call *nested regular expressions*. We also introduce the language nSPARQL, that incorporates these navigational capabilities to a fragment of SPARQL, and provide formal evidence that the capabilities of nSPARQL can be used to pose many interesting and natural queries over RDF data.

We formally study several fundamental properties of nSPARQL. The first of these fundamental questions is whether the navigational capabilities of nSPARQL can be implemented efficiently. In this paper, we show that this is indeed the case. More precisely, we show that nested regular expressions can be evaluated efficiently; if the appropriate data structure is used to store RDF graphs, then given an RDF graph G, a nested regular expression E and nodes A, B in G, it is possible to check in time $O(|G| \cdot |E|)$ whether B is reachable from A in G by following a path in E.

The second fundamental question about nSPARQL is how expressive the language is. We first show that nSPARQL is expressive enough to capture the deductive rules of RDFS. Evaluating queries which involve the RDFS vocabulary is challenging, and there is not yet consensus in the Semantic Web community on how to define a query language for RDFS. In this respect, we show that the RDFS evaluation of an important fragment of SPARQL can be obtained by posing nSPARQL queries that directly traverse the input RDF data. It should be noticed that nested regular expressions are used in nSPARQL to capture the inference rules of RDFS. Thus, a second natural question about nSPARQL is whether these expressions are necessary to obtain this result. In this paper, we show that nesting is indeed necessary in nSPARQL to deal with the semantics of RDFS. More precisely, we show that regular expressions alone are not enough in nSPARQL to obtain the RDFS evaluation of some queries by simply navigating the RDF data.

Finally, we also consider the question of whether the SPARQL operators add expressive power to nSPARQL. Given that nested regular expressions are a powerful navigational tool, one may wonder whether the SPARQL operators can be somehow represented by using these expressions. Or even if this is not the case, one may wonder whether there exist natural queries that can be expressed in nSPARQL, which cannot be expressed by using only nested regular expressions. In our last result, we show that this is the case. More precisely, we prove that there are simple and natural queries that can be expressed in nSPARQL and cannot be expressed by using only nested regular expressions.

Organization of the paper. In Section 2, we introduce some basic no-

tions about RDF, RDFS, and SPARQL. In Section 3, we define the notion of nested regular expression, and prove that these expressions can be evaluated efficiently. In Section 4, we define the language nSPARQL. In Section 5, we show that nSPARQL can be used to answer RDFS queries. In Section 6, we present some result regarding the expressiveness of nSPARQL. In particular, we show that if nesting is disallowed in nSPARQL, then one obtains an strictly less expressive language that cannot encode the process of inference in RDFS. Related work is given in Section 7, and the concluding remarks are given in Section 8.

2 Preliminaries

RDF is a graph data format for the representation of information in the Web. An RDF statement is a *subject-predicate-object* structure, called RDF *triple*, intended to describe resources and properties of those resources. For the sake of simplicity, we assume that RDF data is composed only by elements from an infinite set U of IRIs. Formally, an RDF triple is a tuple $(s, p, o) \in U \times U \times U$, where s is the *subject*, p the *predicate* and o the *object*. An RDF graph is a finite set of RDF triples. Moreover, we denote by $\mathrm{voc}(G)$ the elements from U that are mentioned in G.

In this paper, we do not consider *anonymous resources* called blank nodes in the RDF data model, that is, our study focuses on *ground* RDF graphs. It should be noticed that SPARQL [27], the W3C standard query language for RDF, considers blank nodes simply as constants values without giving them their existential semantics [18]. Moreover, there is not yet full consensus in the Semantic Web community on how an RDF query language should deal with the semantics of blank nodes. Thus, we decide not to make an assumption about this semantics and focus in this paper on the fragment of RDF consisting of ground graphs, for which the query answering process has a well-understood semantics.

Fig. 1 shows an RDF graph that stores information about transportation services between cities. In this figure, a triple (s, p, o) is depicted as an edge $s \xrightarrow{p} o$, that is, s and o are represented as nodes and p is represented as an edge label. For example, $(\mathrm{Paris}, \mathrm{TGV}, \mathrm{Calais})$ is a triple in the graph that states that TGV provides a transportation service from Paris to Calais. Notice that an RDF graph is not a standard labeled graph as its set of edge labels may have a nonempty intersection with its set of nodes. For instance, in the RDF graph in Fig. 1, TGV is simultaneously acting as a node and as an edge label.

The RDF specification includes a set of reserved words (reserved elements

(1) Sub-property:	*(2) Subclass:*	*(3) Typing:*
(a) $\frac{(\mathcal{A},\texttt{sp},\mathcal{B})\ (\mathcal{B},\texttt{sp},\mathcal{C})}{(\mathcal{A},\texttt{sp},\mathcal{C})}$	*(a)* $\frac{(\mathcal{A},\texttt{sc},\mathcal{B})\ (\mathcal{B},\texttt{sc},\mathcal{C})}{(\mathcal{A},\texttt{sc},\mathcal{C})}$	*(a)* $\frac{(\mathcal{A},\texttt{dom},\mathcal{B})\ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\texttt{type},\mathcal{B})}$
(b) $\frac{(\mathcal{A},\texttt{sp},\mathcal{B})\ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\mathcal{B},\mathcal{Y})}$	*(b)* $\frac{(\mathcal{A},\texttt{sc},\mathcal{B})\ (\mathcal{X},\texttt{type},\mathcal{A})}{(\mathcal{X},\texttt{type},\mathcal{B})}$	*(b)* $\frac{(\mathcal{A},\texttt{range},\mathcal{B})\ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{Y},\texttt{type},\mathcal{B})}$

Table 1
RDFS inference rules.

from U) with predefined semantics, the RDFS vocabulary (RDF Schema [8]). This set of reserved words is designed to deal with inheritance of classes and properties, as well as typing, among other features [8]. In this paper, we consider the subset of the RDFS vocabulary composed by rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain and rdf:type, which are denoted by `sc`, `sp`, `range`, `dom` and `type`, respectively. This fragment of RDFS was considered in [23]. In that paper, the authors provide a formal semantics for it, and also show that this fragment is well-behaved as the remaining RDFS vocabulary does not interfere with the semantics of this fragment. The semantics proposed in [23] was shown to be equivalent to the full RDFS semantics when one focuses on the fragment mentioned above. We further assume in this paper that the reserved words `sc`, `sp`, `range`, `dom` and `type`, can only occur in the predicate position of RDF triples.

We use the system of rules in Tab. 1. This system was proved in [23] to be sound and complete for the inference problem for RDFS in the presence of `sc`, `sp`, `range`, `dom` and `type`, under some mild assumptions (see [23] for further details). In every rule, letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{Y}$, stand for *variables* to be replaced by actual terms. More formally, an *instantiation* of a rule is a replacement of the variables occurring in the triples of the rule by elements of U. An *application* of a rule to a graph G is defined as follows. Given a rule r, if there is an instantiation $\frac{R}{R'}$ of r such that $R \subseteq G$, then the graph $G' = G \cup R'$ is the result of an application of r to G. We say that a triple t is *deduced from* G, if either $t \in G$ or there exists a graph G' such that $t \in G'$ and G' is obtained from G by successively applying the rules in Tab. 1.

Example 2.1. Let G be the RDF graph in Fig. 1. This graph contains RDFS annotations for transportation services. For instance, (Seafrance, `sp`, ferry) states that Seafrance is a sub-property of ferry. Thus, we know that there is a ferry going from Calais to Dover since (Calais, Seafrance, Dover) is in G. This conclusion can be obtained by a single application of rule (1b) to triples (Seafrance, `sp`, ferry) and (Calais, Seafrance, Dover), from which we deduce triple (Calais, ferry, Dover). Moreover, by applying the rule (3b) to this last triple and (ferry, `range`, coastal_city), we deduce triple (Dover, `type`, coastal_city) and, thus, we conclude that Dover is a coastal city. □

Jointly with the release of RDF in 1999 as Recommendation of the W3C, the natural problem of querying RDF data was raised. Since then, several designs and implementations of RDF query languages have been proposed [14]. In 2004, the RDF Data Access Working Group (part of the Semantic Web Activity) released a first public working draft of a query language for RDF, called SPARQL [27]. Currently, SPARQL is a W3C recommendation, and has become the standard language for querying RDF data.

In this section, we present the query language SPARQL by considering the algebraic formalization introduced in [25]. Assume the existence of an infinite set V of variables disjoint from U. A SPARQL graph pattern is recursively defined as follows:

- A tuple from $(U \cup V) \times (U \cup V) \times (U \cup V)$ is a graph pattern (a *triple pattern*).
- If P_1 and P_2 are graph patterns, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, and $(P_1 \text{ UNION } P_2)$ are graph patterns.
- If P is a graph pattern and R is a SPARQL *built-in* condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern.

A SPARQL *built-in* condition is a Boolean combination of terms constructed by using equality ($=$) among elements in $U \cup V$, and the unary predicate bound over variables. Formally,

- if $?X,\ ?Y \in V$ and $c \in U$, then $\text{bound}(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions; and
- 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.

Given a graph pattern P, we denote by $\text{var}(P)$ the set of variables occurring in P. Similarly, for a built-in condition R, $\text{var}(R)$ denotes the set of variables occurring in R. We impose the following safety condition to graph patterns. A graph pattern Q is *safe* if for every sub-pattern $(P \text{ FILTER } R)$ of Q, it holds that $\text{var}(R) \subseteq \text{var}(P)$. This is a natural safety condition which is present in most database query languages. In this paper, we assume that every graph pattern is safe.

To define the semantics of SPARQL graph patterns, we need to introduce some terminology. A *mapping* μ from V to U is a partial function $\mu : V \to U$. For a triple pattern t, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, denoted by $\text{dom}(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* if for every $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. Let Ω_1 and Ω_2 be sets of mappings. We define the

join, the union, the difference, and the left-outer join between Ω_1 and Ω_2 as:

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\},\\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 \setminus \Omega_2 &= \{\mu \in \Omega_1 \mid \text{ for all } \mu' \in \Omega_2,\ \mu \text{ and } \mu' \text{ are not compatible}\},\\
\Omega_1 \leftouterjoin \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2).
\end{aligned}$$

The *evaluation* of a graph pattern over an RDF graph G, denoted by $[\![\cdot]\!]_G$, is defined recursively as follows:

- $[\![t]\!]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G\}$, where $\text{var}(t)$ is the set of variables occurring in t.
- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$, $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$, and $[\![(P_1 \text{ OPT } P_2)]\!]_G = [\![P_1]\!]_G \leftouterjoin [\![P_2]\!]_G$.

The semantics of FILTER expressions goes as follows. Given a mapping μ and a built-in condition R, we say that μ satisfies R, denoted by $\mu \models R$, if (we omit the usual rules for Boolean operators):

- R is bound($?X$) and $?X \in \text{dom}(\mu)$;
- R is $?X = c$, where $c \in U$, $?X \in \text{dom}(\mu)$ and $\mu(?X) = c$;
- R is $?X = ?Y$, $?X \in \text{dom}(\mu)$, $?Y \in \text{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$.

Then $[\![(P \text{ FILTER } R)]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu \models R\}$.

It was shown in [25], among other algebraic properties, that AND and UNION are associative and commutative, thus permitting us to avoid parenthesis when writing sequences of either AND operators or UNION operators.

2.2 The semantics of SPARQL over RDFS

SPARQL follows a *subgraph-matching* approach, and thus, a SPARQL query treats RDFS vocabulary without considering its predefined semantics. For example, consider the RDF graph G in Fig. 2, which stores information about soccer players, and consider the graph pattern $P = (?X, \text{works\_in}, ?C)$. Note that, although the triples (Ronaldinho, works_in, Barcelona) and (Sorace, works_in, Everton) can be *deduced* from G, we obtain the empty set as the result of evaluating P over G as there is no triple in G with works_in in the predicate position.

We are interested in defining the semantics of SPARQL over RDFS, that is, taking into account not only the explicit RDF triples of a graph G, but also the triples that can be derived from G according to the semantics of RDFS.

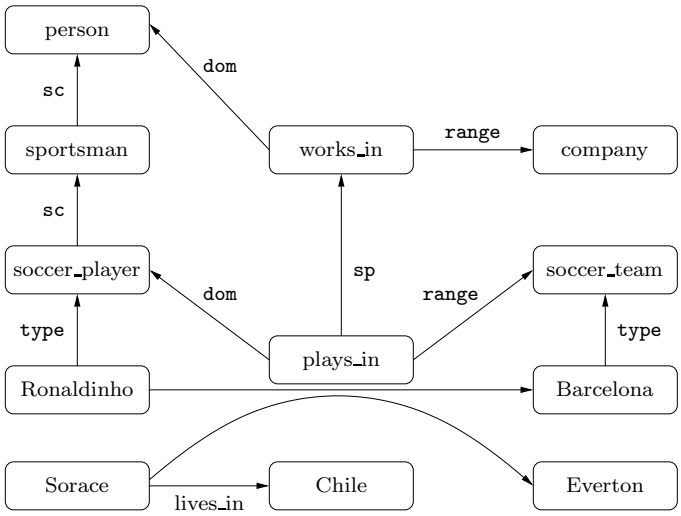


Fig. 2. An RDF graph storing information about soccer players.

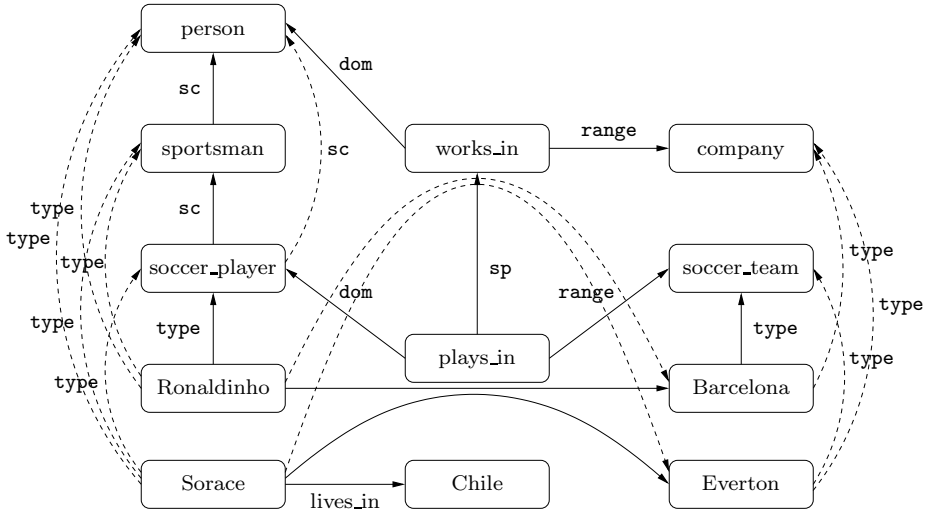


Fig. 3. The closure of the RDF graph of Fig. 2.

Let the *closure* of an RDF graph G, denoted by $\mathrm{cl}(G)$, be the graph obtained from G by successively applying the rules in Tab. 1 until the graph does not change. For instance, Fig. 3 shows the closure of the RDF graph of Fig. 2. The solid lines in Fig. 3 represent the triples in the original graph, and the dashed lines the additional triples in the closure. The most direct way of defining a semantics for the RDFS evaluation of SPARQL patterns is by considering not the original graph but its closure. Thus, if we now evaluate pattern $P = (?X, \text{works\_in}, ?C)$ over the RDF graph in Fig. 3, we obtain the mappings $\{?X \to \text{Ronaldinho}, ?C \to \text{Barcelona}\}$ and $\{?X \to \text{Sorace}, ?C \to \text{Everton}\}$. The theoretical formalization of such an approach was studied in [15]. The following definition formalizes this notion.

Definition 2.2 *Given a* SPARQL *graph pattern* P*, the* RDFS evaluation of P over G*, denoted by* $[\![P]\!]_G^{\mathrm{rdfs}}$*, is defined as the set of mappings* $[\![P]\!]_{\mathrm{cl}(G)}$*, that*

is, as the evaluation of P over the closure of G.

It is important to notice that the previous definition gives a simple algorithm for evaluating SPARQL queries over RDFS data; given a query Q over an RDF graph G with RDFS vocabulary, the closure $\mathrm{cl}(G)$ of G is materialized first, and then Q is evaluated over $\mathrm{cl}(G)$. In particular, this algorithm has the advantage that if several queries are to be posed over a fixed RDF graph G, then $\mathrm{cl}(G)$ has to be computed only once in order to answer these queries.

Unfortunately, the approach mentioned in the previous paragraph has some drawbacks that limit their practical applicability, First, it is known that the closure of a graph G can be of quadratic size in the size of G [23], making the computation and storage of the closure too expensive for web-scale applications [2]. Second, once the closure has been computed, the queries are evaluated over a data source which can be much larger than the original one. This can be particularly inefficient for queries that do not involve the RDFS vocabulary and do not need any RDFS inference in order to be answered. Third, the approach is not goal-oriented. Although in practice most queries use just a fragment of the RDFS vocabulary and would need only to scan a small part of the initial data, all the vocabulary and the data is considered when computing the closure.

Let us present a simple scenario that exemplifies the benefits of a goal-oriented approach. Consider an RDF graph G and a query Q that asks whether a resource A is a sub-class of a resource B. Answering Q amounts to check whether the triple (A, `sc`, B) is implied by G. The predefined semantics of RDFS states that `sc` is a transitive relation among resources. Thus, to answer Q, a goal-oriented approach should not compute the closure of the entire input graph, but instead it should just verify whether there exist resources R_1, R_2, ..., R_n such that $A = R_1$, $B = R_n$, and $(R_i, \mathtt{sc}, R_{i+1})$ is a triple in G for $i = 1, \ldots, n-1$. That is, one can answer Q by checking the existence of an `sc`-path from A to B in G, which can be done in linear time by using a graph reachability algorithm. It should be noticed that the approach that materializes $\mathrm{cl}(G)$ cannot take less that quadratic time in answering this query, as the size of $\mathrm{cl}(G)$ may be quadratic in the size of G.

It was shown in [23] that testing whether a triple is implied by an RDFS graph G can be done without computing the closure of G. The idea is that the RDFS vocabulary is weak enough to warrant that one can determine whether a triple is implied by G just by checking the existence of some paths in G, very much like our simple example above. The good news is that these paths can be specified by using regular expressions plus some additional features.

[2] Currently, one can find RDF databases of one or more gigabytes of data, whose closure may need an exabyte of storage.

For example, to check whether (A, sc, B) belongs to the closure of a graph G, we already saw that it is enough to check whether there is a path from A to B in G where each edge has label sc. This observation motivates the use of extended triple patterns of the form (A, $\texttt{sc}^+$, B), where $\texttt{sc}^+$ is the regular expression denoting paths of length at least 1 and where each edge has label sc. Thus, one can readily see that a language for navigating RDFS data would be useful for answering queries according to the predefined semantics of the RDFS vocabulary.

Driven by this motivation, in this paper we introduce the query language nSPARQL that extends SPARQL with navigational capabilities. This language is expressive enough to capture the deductive rules of RDFS. And not only that, it can be used to pose many interesting and natural queries over RDF data, which otherwise cannot be expressed. In the rest of this paper, we formally introduce nSPARQL, and study some of its fundamental properties. We start this study by introducing the notion of nested regular expression in the following section, which is the building block of nSPARQL.

3 Nested Regular Expressions for RDF Data

One of our main goals in this paper is to define a query language that allows to pose interesting navigational queries over an RDF graph. In this section, we define such a navigational language providing a formal syntax and semantics. Our language uses, as usual for graph query languages [22,5], regular expressions to define paths on graph structures, but taking advantage of the special features of RDF graphs. We extend classical regular expressions by borrowing the notion of *branching* from XPath [11], to obtain the language of *nested regular expressions* to navigate RDF data.

The navigation of a graph is usually done by using an operator *next*, which allows one to move from one node to an adjacent one in a graph. In our setting, we have RDF "graphs", which are sets of triples, not classical graphs. In particular, the sets of nodes and edge labels of an RDF graph can have a nonempty intersection. Hence, a language for navigating RDF graphs should be able to deal with this type of objects. In this section, we introduce the notion of *nested regular expression* to navigate through an RDF graph. This notion takes into account the special features of the RDF data model. In particular, nested regular expressions use three different *navigation axes* next, edge and node, and their inverses $\texttt{next}^{-1}$, $\texttt{edge}^{-1}$ and $\texttt{node}^{-1}$, to move through an RDF triple. These axes are shown in Fig. 4.

A navigation axis allows one to move one step forward (or backward) in an RDF graph. Thus, a sequence of these axes defines a path in an RDF graph.

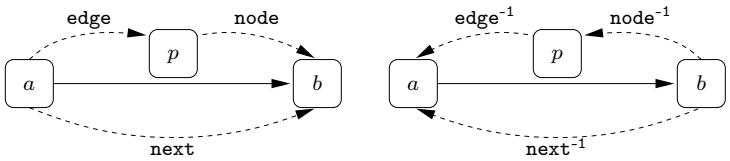


Fig. 4. Forward and backward axes for an RDF triple (a, p, b).

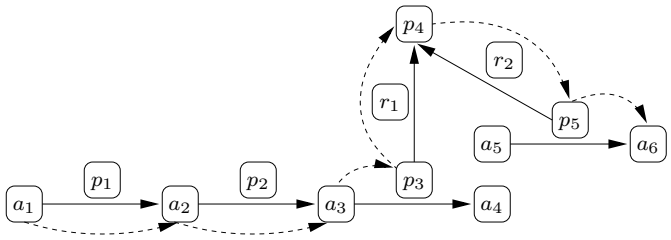


Fig. 5. Nodes a_1 and a_6 are connected by a path that follows the sequence of navigational axes `next/next/edge/next/next-1/node`.

For instance, in the graph of Fig. 5, the sequence of axes

$$\texttt{next/next/edge/next/next}^{-1}\texttt{/node}$$

defines a path between nodes a_1 and a_6 (the path is shown with dashed lines in the figure). Moreover, one can use classical regular expressions over these axes to define a set of paths that can be used in a query. The language considers an additional axis `self` that is used not to actually navigate, but instead to test the label of a specific node in a path. The language also allows *nested expressions* that can be used to test for the existence of certain paths starting at any axis. The following grammar defines the syntax of nested regular expressions:

$$\begin{aligned} exp \;:=\; & \text{axis} \mid \text{axis}::a\ (a \in U) \mid \text{axis}::[exp] \mid \\ & exp/exp \mid exp|exp \mid exp^* \end{aligned} \quad (1)$$

where axis $\in \{\texttt{self}, \texttt{next}, \texttt{next}^{-1}, \texttt{edge}, \texttt{edge}^{-1}, \texttt{node}, \texttt{node}^{-1}\}$.

Before introducing the formal semantics of nested regular expressions, we give some intuition about how these expressions are evaluated in an RDF graph. The most natural navigation axis is `next`$::a$, with a an arbitrary element from U. Given an RDF graph G, the expression `next`$::a$ is interpreted as the *a-neighbor* relation in G, that is, the pairs of nodes (x, y) such that $(x, a, y) \in G$. Given that in the RDF data model a node can also be the label of an edge, the language allows us to navigate from a node to one of its leaving edges by using the `edge` axis. More formally, the interpretation of `edge`$::a$ is the pairs of nodes (x, y) such that $(x, y, a) \in G$. The nesting construction $[exp]$ is used to check for the existence of a path defined by expression exp. For instance, when evaluating nested expression `next`$::[exp]$ in a graph G, we retrieve the pairs of nodes (x, y) such that there exists z with $(x, z, y) \in G$, and such that

$$
\begin{array}{rcl}
[\![\texttt{self}]\!]_G & = & \{(x,x) \mid x \in \mathrm{voc}(G)\} \\
[\![\texttt{self::}a]\!]_G & = & \{(a,a)\} \\
[\![\texttt{next}]\!]_G & = & \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,z,y) \in G\} \\
[\![\texttt{next::}a]\!]_G & = & \{(x,y) \mid (x,a,y) \in G\} \\
[\![\texttt{edge}]\!]_G & = & \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,y,z) \in G\} \\
[\![\texttt{edge::}a]\!]_G & = & \{(x,y) \mid (x,y,a) \in G\} \\
[\![\texttt{node}]\!]_G & = & \{(x,y) \mid \text{there exists } z \text{ s.t. } (z,x,y) \in G\} \\
[\![\texttt{node::}a]\!]_G & = & \{(x,y) \mid (a,x,y) \in G\} \\
[\![\mathrm{axis}^{-1}]\!]_G & = & \{(x,y) \mid (y,x) \in [\![\mathrm{axis}]\!]_G\} \quad \text{with axis} \in \{\texttt{next},\texttt{node},\texttt{edge}\} \\
[\![\mathrm{axis}^{-1}\texttt{::}a]\!]_G & = & \{(x,y) \mid (y,x) \in [\![\mathrm{axis::}a]\!]_G\} \quad \text{with axis} \in \{\texttt{next},\texttt{node},\texttt{edge}\} \\
[\![exp_1/exp_2]\!]_G & = & \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,z) \in [\![exp_1]\!]_G \text{ and } (z,y) \in [\![exp_2]\!]_G\} \\
[\![exp_1|exp_2]\!]_G & = & [\![exp_1]\!]_G \cup [\![exp_2]\!]_G \\
[\![exp^*]\!]_G & = & [\![\texttt{self}]\!]_G \cup [\![exp]\!]_G \cup [\![exp/exp]\!]_G \cup [\![exp/exp/exp]\!]_G \cup \cdots \\
[\![\texttt{self::}[exp]]\!]_G & = & \{(x,x) \mid x \in \mathrm{voc}(G) \text{ and there exists } z \text{ s.t. } (x,z) \in [\![exp]\!]_G\} \\
[\![\texttt{next::}[exp]]\!]_G & = & \{(x,y) \mid \text{there exist } z,w \text{ s.t. } (x,z,y) \in G \text{ and } (z,w) \in [\![exp]\!]_G\} \\
[\![\texttt{edge::}[exp]]\!]_G & = & \{(x,y) \mid \text{there exist } z,w \text{ s.t. } (x,y,z) \in G \text{ and } (z,w) \in [\![exp]\!]_G\} \\
[\![\texttt{node::}[exp]]\!]_G & = & \{(x,y) \mid \text{there exist } z,w \text{ s.t. } (z,x,y) \in G \text{ and } (z,w) \in [\![exp]\!]_G\} \\
[\![\mathrm{axis}^{-1}\texttt{::}[exp]]\!]_G & = & \{(x,y) \mid (y,x) \in [\![\mathrm{axis::}[exp]]\!]_G\} \quad \text{with axis} \in \{\texttt{next},\texttt{node},\texttt{edge}\}
\end{array}
$$

Table 2
Formal semantics of nested regular expressions.

there is a path in G that follows expression exp starting in z.

The evaluation of a nested regular expression exp in a graph G is formally defined as a binary relation $[\![exp]\!]_G$, denoting the pairs of nodes (x, y) such that y is reachable from x in G by following a path that conforms to exp. The formal semantics of the language is shown in Tab. 2. In this table, G is an RDF graph, $a \in U$, $\mathrm{voc}(G)$ is the set of all the elements from U that are mentioned in G, and exp, exp_1, exp_2 are nested regular expressions.

As is customary for regular expressions, given a nested regular expression exp, we use exp^+ as a shorthand for exp^*/exp. The following is a simple example of the evaluation of a nested regular expression. We present more involved examples when introducing the nSPARQL language.

Example 3.1. Let G be the graph in Fig. 1, and consider expression

$$exp_1 = \texttt{next::}[\texttt{next::sp}/\texttt{self::}\mathrm{train}].$$

The expression next::sp/self::train defines the pairs of nodes (z, w) such that from z one can follow an edge labeled sp and reach a node w with label train (expression self::train is used to perform this last test). Thus, the nested expression [next::sp/self::train] performs an existential test; it is satisfied by the nodes in G from which there exists a path that follows an edge labeled sp and reaches a node labeled train. There is a single such node in G, namely TGV. Restricted to graph G, expression exp_1 is equivalent to next::TGV and, thus, it defines the pairs of nodes that are connected by an edge labeled TGV. Hence, the evaluation of exp_1 in G is $[\![exp_1]\!]_G = \{(\text{Paris, Calais}), (\text{Paris, Dijon})\}$. □

In the following section, we introduce the language nSPARQL that combines the operators of SPARQL with the navigational capabilities of nested regular expressions. But before introducing this language, we show that nested regular expressions can be evaluated efficiently, which is an essential requirement if one wants to use nSPARQL for web-scale applications.

3.1 An efficient algorithm for evaluating nested regular expressions

In this section, we present an algorithm that efficiently solves the evaluation problem for nested regular expressions over RDF graphs. More precisely, we consider two problems associated to nested regular expressions. The first one is the decision problem of verifying whether a given pair of nodes is in the evaluation of a nested regular expression over an RDF graph. This decision problem is formally defined as follows:

PROBLEM	:	Evaluation problem for nested regular expressions.
INPUT	:	An RDF graph G, a nested regular expression exp, and a pair (a, b).
QUESTION	:	Is $(a, b) \in [\![exp]\!]_G$?

It is important to notice that this problem considers the pair of nodes (a, b) as part of the input. This is the standard *decision* problem considered when studying the complexity of a query language [28]. The second problem considered in this paper is the following computation problem associated to nested regular expressions:

PROBLEM	:	Computation problem for nested regular expressions.
INPUT	:	An RDF graph G, a nested regular expression exp, and a node a.
OUTPUT	:	List all the elements b such that $(a, b) \in [\![exp]\!]_G$.

Thus, the problem is to give a list with all the nodes that are reachable from a fixed node a by following an expression exp in an RDF graph G.

At this point, the reader may wonder why we do not consider the problem of, giving a nested regular expression exp and an RDF graph G, listing all the pairs in $[\![exp]\!]_G$. Notice that any algorithm that solves this last problem needs at least quadratic time with respect to the RDF graph G, since just writing down the output needs quadratic time in the worst case. On the other hand, this lower bound cannot be directly proved for the two problems considered in this section; the output of the decision problem is of constant size (it is a no/yes answer), and the output of the problem of listing the nodes that are connected to a given node is linear in the size of the input RDF graph. Thus, it is worth studying whether these two problems can be solved efficiently (which is by no means trivial).

In this section, we show that the two problems mentioned above can be solved efficiently (in fact, in linear time with respect to the size of the input RDF graph). More precisely, we provide algorithms that solve these problems in time $O(|G| \cdot |exp|)$, where $|G|$ denotes the size of the input RDF graph and $|exp|$ denotes the size of the nested regular expression being evaluated.

We assume that an RDF graph G is stored as an adjacency list that makes explicit the navigation axes (and their inverses). Thus, every $u \in \text{voc}(G)$ is associated with a list of pairs $\alpha(u)$, where every pair contains a navigation axis and the destination node. For instance, if (s, p, o) is a triple in G, then $(\texttt{next}::p, o) \in \alpha(s)$ and $(\texttt{edge}^{-1}::o, s) \in \alpha(p)$. Moreover, we assume that $(\texttt{self}::u, u) \in \alpha(u)$ for every $u \in \text{voc}(G)$. Notice that if the number of triples in G is N, then the adjacency list representation uses space $O(N)$. Thus, when measuring the size of G, we use $|G|$ to denote the size of its adjacency list representation. We further assume that given an element $u \in \text{voc}(G)$, we can access its associated list $\alpha(u)$ in time $O(1)$. This is a standard assumption for graph data-structures in a RAM model [13]. Fig. 6 shows an example of an adjacency-list representation of an RDF graph. For a nested regular expression exp, we use $|exp|$ to denote the size of the expression.

The idea of the algorithm for the evaluation of nested regular expressions is motivated by the algorithms for the evaluation of some temporal logics [12] and propositional dynamic logic [1,16]. We say that expression exp' is a *nested*

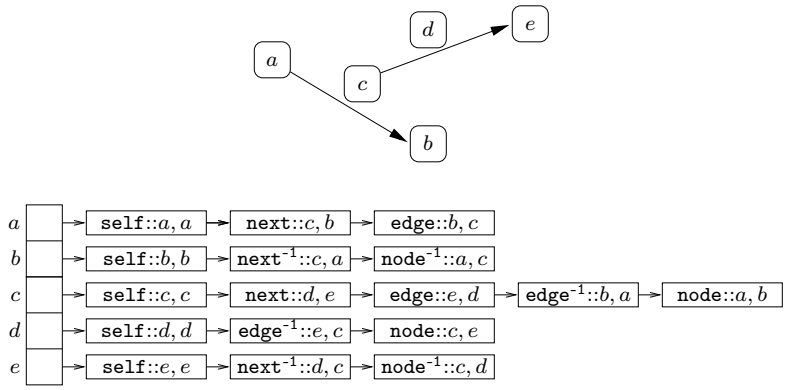


Fig. 6. Adjacency-list representation (below) of an RDF graph (above).

subexpression of *exp*, if the expression axis::[*exp′*] occurs in *exp*, with axis ∈ {self, next, next^{-1}, edge, edge^{-1}, node, node^{-1}}. Given an RDF graph G and a nested regular expression *exp*, the algorithm proceeds by recursively considering the nested subexpressions of *exp*, labeling every node u of G with a set label(u) of nested expressions. Initially, label(u) is the empty set. Then at the end of the execution of the algorithm, it holds that *exp* ∈ label(u) if and only if there exists z such that $(u, z) \in [\![exp]\!]_G$. Before giving any technical details, let us show the general idea of this process with an example. Fig. 7 exemplifies the process for a graph G and the nested expression:

$$\beta \;=\; \texttt{next}::a/(\texttt{next}::[\texttt{next}::b/\texttt{self}::c])^*/(\texttt{edge}::[\texttt{next}::d] \mid \texttt{next}::a)^+. \quad (2)$$

The process first considers the nested subexpressions $\gamma = \texttt{next}::b/\texttt{self}::c$ and $\lambda = \texttt{next}::d$, and marks the nodes in G according to which ones of these subexpressions they satisfy. Thus, after this stage we have that $\gamma \in \text{label}(r_3)$ since $(r_3, c) \in [\![\gamma]\!]_G$, and $\lambda \in \text{label}(r_6)$ since $(r_6, r_7) \in [\![\lambda]\!]_G$ (see Fig. 7). Using this information, we mark the nodes according to whether they satisfy β, but considering the previously computed labels (γ and λ) and the expression $\beta' = \texttt{next}::a/(\texttt{next}::\gamma)^*/(\texttt{edge}::\lambda \mid \texttt{next}::a)^+$. In the example of Fig. 7, we have that $(r_1, r_5) \in [\![\beta]\!]_G$ and, thus, $\beta \in \text{label}(r_1)$.

We now explain how to efficiently carry out the labeling process by using some tools from automata theory (here we assume some familiarity with this theory). A key idea in the algorithm is to associate to each nested regular expression a nondeterministic finite automaton with ε-transitions (ε-NFA). Given a nested regular expression *exp*, we recursively define the set of *depth-0 terms* of *exp*, denoted by $\mathbf{D}_0(exp)$, as follows:

$\mathbf{D}_0(exp) = \{exp\}$ if *exp* is either axis, or axis::a, or axis::[*exp′*],
$\mathbf{D}_0(exp_1/exp_2) = \mathbf{D}_0(exp_1|exp_2) = \mathbf{D}_0(exp_1) \cup \mathbf{D}_0(exp_2)$,
$\mathbf{D}_0(exp^*) = \mathbf{D}_0(exp)$,

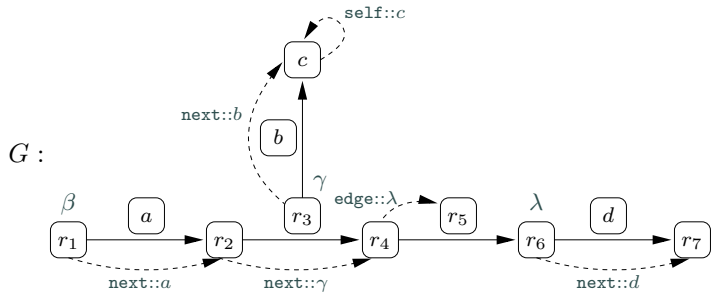


Fig. 7. Example of the labeling process of the RDF graph G according to expression $\beta = \texttt{next}::a/(\texttt{next}::[\texttt{next}::b/\texttt{self}::c])^*/(\texttt{edge}::[\texttt{next}::d] \mid \texttt{next}::a)^+$. First, node r_3 is marked with label $\gamma = \texttt{next}::b/\texttt{self}::c$ (since $(r_3, c) \in [\![\gamma]\!]_G$), and node r_6 with label $\lambda = \texttt{next}::d$ (since $(r_6, r_7) \in [\![\lambda]\!]_G$). Finally, node r_1 is labeled with β (since $(r_1, r_5) \in [\![\beta]\!]_G$). This last label is obtained by considering the expression $\beta' = \texttt{next}::a/(\texttt{next}::\gamma)^*/(\texttt{edge}::\lambda \mid \texttt{next}::a)^+$.

where axis $\in \{\texttt{self}, \texttt{next}, \texttt{next}^{-1}, \texttt{edge}, \texttt{edge}^{-1}, \texttt{node}, \texttt{node}^{-1}\}$. For instance, for the nested expression β in (2), we have that:

$$\mathbf{D}_0(\beta) = \{\, \texttt{next}::a,\ \texttt{next}::[\texttt{next}::b/\texttt{self}::c],\ \texttt{edge}::[\texttt{next}::d] \,\}.$$

Notice that a nested regular expression *exp* can be viewed as a classical regular expression over the alphabet $\mathbf{D}_0(exp)$. We denote by $\mathcal{A}_{exp}$ the ε-NFA that accepts the language generated by the regular expression *exp* over the alphabet $\mathbf{D}_0(exp)$. For example, Fig. 8 shows an ε-NFA $\mathcal{A}_\beta$ that accepts the language generated by expression β in (2) over the alphabet $\mathbf{D}_0(\beta)$. As for the case of RDF graphs, we assume that ε-NFAs are stored using an adjacency-list representation.

In the algorithm, we use the product automaton $G \times \mathcal{A}_{exp}$, which is constructed as follows. Assume that we have the graph G labeled with respect to the nested subexpressions of *exp*, that is, for every node u of G and nested subexpression *exp*$'$ of *exp*, we have that *exp*$' \in \text{label}(u)$ if and only if there exists a node v such that $(u, v) \in [\![exp']\!]_G$. Let Q be the set of states of $\mathcal{A}_{exp}$, and $\delta : Q \times (\mathbf{D}_0(exp) \cup \{\varepsilon\}) \to 2^Q$ the transition function of $\mathcal{A}_{exp}$. The set of states of $G \times \mathcal{A}_{exp}$ is $\text{voc}(G) \times Q$, and its transition function $\delta' : (\text{voc}(G) \times Q) \times (\mathbf{D}_0(exp) \cup \{\varepsilon\}) \to 2^{\text{voc}(G) \times Q}$ is defined as follows. For every $(u, p) \in \text{voc}(G) \times Q$ and $s \in \mathbf{D}_0(exp)$, we have that $(v, q) \in \delta'((u, p), s)$ if and only if $q \in \delta(p, s)$ and one of the following cases hold:

- $s =$ axis and there exists a such that $(\text{axis}::a, v) \in \alpha(u)$,
- $s = \text{axis}::a$ and $(\text{axis}::a, v) \in \alpha(u)$,
- $s = \text{axis}::[exp]$ and there exists b such that $(\text{axis}::b, v) \in \alpha(u)$ and $exp \in \text{label}(b)$,

where axis $\in \{\texttt{self}, \texttt{next}, \texttt{next}^{-1}, \texttt{edge}, \texttt{edge}^{-1}, \texttt{node}, \texttt{node}^{-1}\}$. Additionally, if $q \in \delta(p, \varepsilon)$ we have that $(u, q) \in \delta'((u, p), \varepsilon)$ for every $u \in \text{voc}(G)$. That is,

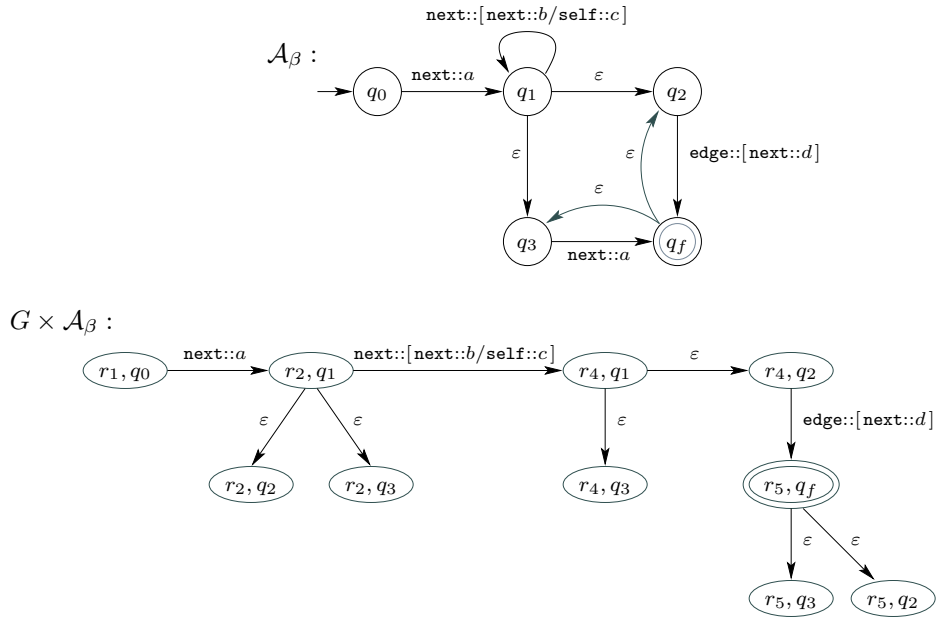


Fig. 8. Automaton $\mathcal{A}_\beta$ for the nested regular expression β in (2), and product automaton $G \times \mathcal{A}_\beta$.

$G \times \mathcal{A}_{exp}$ is the standard product automaton of G and $\mathcal{A}_{exp}$ if G is viewed as an ε-NFA over the alphabet $\mathbf{D}_0(exp)$. Fig. 8 shows the product automaton $G \times \mathcal{A}_\beta$ for the nested expression β in (2) and the graph G of Fig. 7 (labeled with respect to the nested subexpressions of β). For space reasons, we have only depicted the states of $G \times \mathcal{A}_\beta$ that are reachable from the initial state. For instance, we have that there is a transition from (r_2, q_1) to (r_4, q_1) with symbol $\texttt{next}::[\texttt{next}::b/\texttt{self}::c]$ since: (i) there is a transition from q_1 to q_1 with $\texttt{next}::[\texttt{next}::b/\texttt{self}::c]$ in $\mathcal{A}_\beta$, and (ii) $(\texttt{next}::r_3, r_4) \in \alpha(r_2)$ and $\gamma = \texttt{next}::b/\texttt{self}::c \in \text{label}(r_3)$.

There are two key observations about the product automaton defined above that should be made. Let G be a graph labeled with respect to the nested subexpressions of exp, and $\mathcal{A}_{exp}$ an ε-NFA for exp. Assume that q_0 is the initial state of $\mathcal{A}_{exp}$ and q_f is one of its final states. The first observation is that if there exists two elements $u, v \in \text{voc}(G)$ such that from (u, q_0) one can reach state (v, q_f) in $G \times \mathcal{A}_{exp}$, then $(u, v) \in [\![exp]\!]_G$. In the example of Fig. 8, we have that $(r_1, r_5) \in [\![\beta]\!]_G$ since we can reach state (r_5, q_f) from state (r_1, q_0) in $G \times \mathcal{A}_\beta$. The second observation is that given a nested regular expression exp, one can construct in linear time an ε-NFA for exp by using standard techniques. Thus, given a nested regular expression exp and an RDF graph G that has been labeled with respect to the nested subexpressions of exp, it is easy to see that automaton $G \times \mathcal{A}_{exp}$ can be constructed in time $O(|G| \cdot |\mathcal{A}_{exp}|)$.

Now we have all the necessary ingredients to formally present the algorithm for the evaluation problem for nested regular expressions. This algorithm is split in two procedures: Label labels G according to the nested subexpressions

of *exp* as explained above, and EVAL returns YES if $(a, b) \in [\![exp]\!]_G$ and NO otherwise.

LABEL(G, exp):
1. **for each** axis::$[exp']$ $\in \mathbf{D}_0(exp)$ **do**
2. call LABEL(G, exp')
3. construct $\mathcal{A}_{exp}$, and assume that q_0 is its initial state and F is its set of final states
4. construct $G \times \mathcal{A}_{exp}$
5. **for each** state (u, q_0) that is connected to a state (v, q_f) in $G \times \mathcal{A}_{exp}$, with $q_f \in F$ **do**
6. label(u) := label$(u) \cup \{exp\}$

EVAL$(G, exp, (a, b))$:
1. **for each** $u \in$ voc(G) **do**
2. label$(u) := \emptyset$
3. call LABEL(G, exp)
4. construct $\mathcal{A}_{exp}$, and assume that q_0 is its initial state and F is its set of final states
5. construct $G \times \mathcal{A}_{exp}$
6. **if** a state (b, q_f), with $q_f \in F$, is reachable from (a, q_0) in $G \times \mathcal{A}_{exp}$
7. **then return** YES
8. **else return** NO

Theorem 3.2 *Procedure* EVAL *solves the evaluation problem for nested regular expressions in time* $O(|G| \cdot |exp|)$.

Proof. Assume that for every u, it holds that label$(u) = \emptyset$. We argue that after the execution of procedure LABEL(G, exp), it holds that $exp \in$ label(u) if and only if there exists v such that $(u, v) \in [\![exp]\!]_G$. We proceed by induction on the depth in the tree of recursive calls to LABEL. The base case is when $\mathbf{D}_0(exp)$ have no expressions of the form axis::$[exp']$. The property in this case follows by the definition of the product automaton $G \times \mathcal{A}_{exp}$. It is easy to see that there exists a state (u, q_0) that is connected to a state (v, q_f) in $G \times \mathcal{A}_{exp}$, with q_0 the initial state of $\mathcal{A}_{exp}$ and q_f a final state of $\mathcal{A}_{exp}$, if and only if $(u, v) \in [\![exp]\!]_G$. Just recall that $\mathbf{D}_0(exp)$ has no expressions of the form axis::$[exp']$ and, thus, *exp* is a standard regular expression with no nested subexpressions. Now, assume that $\mathbf{D}_0(exp)$ contains some expression of the form axis::$[exp']$. At the beginning, procedure LABEL is executed for every expression exp' such that axis::$[exp'] \in \mathbf{D}_0(exp)$. By induction hypothesis, after these calls we have that if axis::$[exp'] \in \mathbf{D}_0(exp)$, then $exp' \in$ label(u) if and only if there exists v such that $(u, v) \in [\![exp']\!]_G$. Again by the definition of the product automaton $G \times \mathcal{A}_{exp}$, it is easy to see that there exists a state (u, q_0) that is connected to a state (v, q_f) in $G \times \mathcal{A}_{exp}$, with q_0 the initial state of $\mathcal{A}_{exp}$ and q_f a final state of $\mathcal{A}_{exp}$, if and only if $(u, v) \in [\![exp]\!]_G$. The correctness of procedure EVAL follows directly from this property.

It only remains to show that procedure EVAL runs in time $O(|G| \cdot |exp|)$. To prove this, we need some terminology. Define the set of *depth-i terms* $(i \geq 1)$

of a nested expression *exp* by

$$\mathbf{D}_i(exp) = \bigcup_{\text{axis::}[exp']\in\mathbf{D}_{i-1}(exp)} \mathbf{D}_0(exp').$$

For instance, for the nested expression

$$\beta = \texttt{next::}a/(\texttt{next::}[\texttt{next::}[\texttt{edge::}b]])^+/\texttt{next::}[(\texttt{next::}b)^*/\texttt{self::}d],$$

we have that

$$\begin{aligned}
&\mathbf{D}_0(\beta) = \{\texttt{next::}a,\ \texttt{next::}[\texttt{next::}[\texttt{edge::}b]],\ \texttt{next::}[(\texttt{next::}b)^*/\texttt{self::}d]\},\\
&\mathbf{D}_1(\beta) = \{\texttt{next::}[\texttt{edge::}b],\ \texttt{next::}b,\ \texttt{self::}d\},\\
&\mathbf{D}_2(\beta) = \{\texttt{edge::}b\},\\
&\mathbf{D}_3(\beta) = \emptyset.
\end{aligned}$$

Define the *depth* of a nested expression *exp* as the minimum integer d such that $\mathbf{D}_{d+1}(exp) = \emptyset$. For instance, the depth of expression β above is 2.

The first important observation is that the total number of recursive calls to Label in the execution of the algorithm is at most:

$$k = \sum_{i=1}^{d} |\,\mathbf{D}_i(exp)|,$$

where d is the depth of *exp*. Notice that k is bounded by $|exp|$. The second important observation is that for every nested regular expression exp', one can construct an automaton $\mathcal{A}_{exp'}$ whose size is linear in the number of occurrences of symbols of $\mathbf{D}_0(exp')$ in exp'. With these observations, we conclude that the total time spent constructing all the product automata in the entire execution of Label is $O(|G| \cdot |exp|)$. The final observation is how to efficiently execute step 5 in each recursive call to Label. Assume that Label has been called with expression exp'. Notice that in step 5 we do not need to make an iteration over the states of $G \times \mathcal{A}_{exp'}$, but just to perform a depth-first search to find the nodes of the form (u, q_0) of $G \times \mathcal{A}_{exp'}$, with q_0 the initial state of $\mathcal{A}_{exp'}$, that reach a node (v, q_f) with q_f a final state of $\mathcal{A}_{exp'}$. This search can be done in time $O(|G| \cdot |\mathcal{A}_{exp'}|)$. With these observations it is easy to conclude that with input *exp*, procedure Label runs in time $O(|G| \cdot |exp|)$. Similarly, step 6 of procedure Eval can be carried out in time $O(|G| \cdot |exp|)$ and, thus, Eval runs in time $O(|G| \cdot |exp|)$. □

The above algorithms can be directly used to efficiently solve the computation problem of listing all the nodes that are reachable from a fixed node by following a nested regular expression. As we have shown, after the call to Label(G, *exp*), we have that $(u, v) \in [\![exp]\!]_G$ if and only if there exists a path

from a state (u, q_0) to a state (v, q_f) in the automaton $G \times \mathcal{A}_{exp}$, where q_0 is the initial state of $\mathcal{A}_{exp}$ and q_f is one of its final states. Thus, given an element $a \in \text{voc}(G)$, one can list all the elements $b \in \text{voc}(G)$ such that $(a, b) \in [\![exp]\!]_G$ by performing a depth-first search over $G \times \mathcal{A}_{exp}$ starting at (a, q_0). The whole process takes time $O(|G| \cdot |exp|)$. Thus, we have the following result.

Theorem 3.3 *Given an RDF graph G, a nested regular expression exp and an element a, listing all the elements b such that $(a, b) \in [\![exp]\!]_G$ can be done in time $O(|G| \cdot |exp|)$.*

4 nSPARQL: An RDF Query Language with Navigational Capabilities

In this section, we introduce the language *nested SPARQL* (or just nSPARQL), which is an RDF query language with navigational capabilities.

The query language nSPARQL is essentially obtained by using triple patterns with nested regular expressions in the predicate position, plus SPARQL operators AND, OPT, UNION, and FILTER as we defined them in Section 2.1. More precisely, a *nested-regular-expression triple* (or just nre-triple) is a tuple t of the form (x, exp, y), where $x, y \in U \cup V$ and exp is a nested regular expression. nSPARQL graph patterns are then recursively defined from nre-triples:

- An nre-triple is an nSPARQL graph pattern.
- If P_1 and P_2 are nSPARQL graph patterns and R is a built-in condition, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, $(P_1 \text{ UNION } P_2)$, and $(P_1 \text{ FILTER } R)$ are nSPARQL graph patterns.

It should be noticed that in nSPARQL, we are using the algebraic formalization for SPARQL graph patterns proposed in [25]. In particular, we are not considering *projection* (i.e. the SELECT operator) in our language. Clearly, projection would add expressive power to the language, but at the cost of increasing the complexity of the evaluation problem for some fragments of the language (see Section 7 for a detailed discussion on this topic). Thus, we have decided to use the core algebraic fragment of SPARQL [25] to construct our nSPARQL language. As we show in the next sections, this decision allows us to obtain a language with good properties regarding expressiveness and complexity of evaluation.

To define the semantics of nSPARQL, we just need to define the semantics of nre-triples, as the semantics of the operators AND, OPT, UNION, and FILTER is defined exactly as for the case of SPARQL (see Section 2.1). The evaluation of an nre-triple $t = (?X, exp, ?Y)$ over an RDF graph G is defined as the following set of mappings:

$$[\![t]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \{?X, ?Y\} \text{ and } (\mu(?X), \mu(?Y)) \in [\![exp]\!]_G\}.$$

Similarly, the evaluation of an nre-triple $t = (?X, exp, a)$ over an RDF graph G, where $a \in U$, is defined as $\{\mu \mid \mathrm{dom}(\mu) = \{?X\} \text{ and } (\mu(?X), a) \in [\![exp]\!]_G\}$, and likewise for $(a, exp, ?X)$ and (a, exp, b) with $b \in U$.

Notice that every SPARQL triple $(?X, p, ?Y)$ with $p \in U$ is equivalent to nSPARQL triple $(?X, \texttt{next}::p, ?Y)$. Also notice that, since variables are not allowed in nested regular expressions, the occurrence of variables in the predicate position of triple patterns is forbidden in nSPARQL. Nevertheless, every SPARQL triple of the form $(?X, ?Y, a)$, with $a \in U$, is equivalent to nSPARQL pattern $(?X, \texttt{edge}::a, ?Y)$. Similarly, the triple $(a, ?X, ?Y)$ is equivalent to $(?X, \texttt{node}::a, ?Y)$. Thus, what we are losing in nSPARQL is only the possibility of using variables in the three positions of a triple pattern. We decided not to include this type of triples in nSPARQL for two reasons: they do not clearly represent the idea of navigation, and they are not needed in nSPARQL to evaluate queries according to the semantics of RDFS (see Section 5). In fact, a triple of the form $(?X, exp, ?Y)$ indicates that one has to navigate from $?X$ to $?Y$ by following pattern exp, while a navigation criteria like this cannot be used to evaluate a triple pattern of the form $(?X, ?Y, ?Z)$.

As pointed out in the introduction, it has been largely recognized that navigational capabilities are fundamental for graph databases query languages. However, although RDF is a directed labeled graph data format, SPARQL only provides limited navigational functionalities. In this paper, we propose nSPARQL as a way to overcome this limitation. Moreover, we study some of its fundamental properties in order to provide formal evidence of its usefulness as a query language for RDF. In particular, we have already shown that nested regular expressions can be evaluated efficiently, which is an essential requirement if one wants to use nSPARQL for web-scale applications. In the following sections, we study some fundamental properties related to the expressiveness of nSPARQL. But before doing that, we show through some examples how the navigational capabilities of nSPARQL can be used to express queries that are likely to occur in the Semantic Web, but cannot be expressed in SPARQL without using nested regular expressions.

Example 4.1. Let G be the RDF graph of Fig. 1 and P_1 the following pattern:

$$P_1 = (?X, (\texttt{next}::\text{TGV} \mid \texttt{next}::\text{Seafrance})^+, \text{Dover}) \text{ AND } (?X, \texttt{next}::\text{country}, ?Y)$$

Pattern P_1 retrieves cities, and the country where they are located, such that there is a way to travel from those cities to Dover using either TGV or Seafrance in every direct trip. The evaluation of P_1 over G is $\{\{?X \rightarrow \text{Paris}, ?Y \rightarrow \text{France}\}\}$. Notice that although there is a direct way to travel from Calais to Dover using Seafrance, Calais does not appear in the result since there is no information in G about the country where Calais is located.

We can relax this last restriction by using the OPT operator:

$P_2 = (?X, (\texttt{next}::\text{TGV} \mid \texttt{next}::\text{Seafrance})^+, \text{Dover})\ \text{OPT}\ (?X, \texttt{next}::\text{country}, ?Y)$

Then we have that $[\![P_2]\!]_G = \{\{?X \to \text{Paris}, ?Y \to \text{France}\}, \{?X \to \text{Calais}\}\}$. □

Example 4.2. Assume that we want to obtain the pairs of cities $(?X, ?Y)$ such that there is a way to travel from $?X$ to $?Y$ by using either Seafrance or NExpress, with an intermediate stop in a city that has a direct NExpress trip to London. Consider nested expression:

$$exp_1 = (\texttt{next}::\text{Seafrance} \mid \texttt{next}::\text{NExpress})^+/ \\ \texttt{self}::[\texttt{next}::\text{NExpress}/\texttt{self}::\text{London}]/(\texttt{next}::\text{Seafrance} \mid \texttt{next}::\text{NExpress})^+$$

Then pattern $P = (?X, exp_1, ?Y)$ answers our initial query. Notice that expression $\texttt{self}::[\texttt{next}::\text{NExpress}/\texttt{self}::\text{London}]$ is used to perform the intermediate existential test of having a direct NExpress trip to London. □

Example 4.3. Let G be the graph in Fig. 1 and P_1 the following pattern:

$$P_1 = (?X,\ \texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}],\ ?Y). \tag{3}$$

Pattern P_1 defines the pairs of cities $(?X, ?Y)$ such that, there exists a triple $(?X, p, ?Y)$ in the graph and a path from p to transport where every edge has label sp. Thus, nested expression $[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}]$ is used to emulate the process of inference in RDFS; it retrieves all the nodes that are *sub-properties* of transport (rule (1a) in Tab. 1). Therefore, pattern P_1 retrieves the pairs of cities that are connected by a direct transportation service, which could be a train, ferry, bus, etc. In general, if we want to obtain the pairs of cities such that there is a way to travel from one city to the other, we can use the following nSPARQL pattern:

$$P_2 = (?X,\ (\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+,\ ?Y). \tag{4}$$

□

In the following section, we prove that the navigational capabilities of nSPARQL can be used to answer queries according to the semantics of RDFS. In particular, we show that the use of $[\ \cdot\ ]$ is essential for this result, as patterns of the form (3) are used in this proof, and they cannot be expressed in nSPARQL without using nesting.

5 On RDFS and nSPARQL

In this section, we formally prove that the language of nested regular expressions is powerful enough to deal with the predefined semantics of RDFS. More precisely, we show that if one wants to compute the answer of a SPARQL graph pattern P according to the semantics of RDFS, then one can rewrite P into an nSPARQL graph pattern Q such that Q retrieves the answer to P by directly traversing the input graph.

Let us show with an example how nSPARQL can be used to obtain the RDFS evaluation of some patterns by directly traversing the input graph.

Example 5.1. Let G be the RDF graph in Fig. 2, and assume that we want to obtain the *type* information of Ronaldinho. This information can be obtained by computing the RDFS evaluation of the pattern $(\text{Ronaldinho}, \texttt{type}, ?C)$. By simply inspecting the closure of G in Fig. 3, we obtain that the RDFS evaluation of $(\text{Ronaldinho}, \texttt{type}, ?C)$ is the set of mappings:

$$\{\{?C \to \text{soccer\_player}\}, \{?C \to \text{sportsman}\}, \{?C \to \text{person}\}\}$$

However, if we directly evaluate this pattern over G, we obtain a single mapping $\{?C \to \text{soccer\_player}\}$. Consider now the nSPARQL pattern:

$$P = (\text{Ronaldinho},\ \texttt{next::type}/(\texttt{next::sc})^*,\ ?C).$$

The expression $\texttt{next::type}/(\texttt{next::sc})^*$ is intended to obtain the pairs of nodes such that there is a path between them that starts with label `type` followed by zero or more labels `sc`. When evaluating this expression in G, we obtain the set of pairs {(Ronaldinho, soccer_player), (Ronaldinho, sportsman), (Ronaldinho, person), (Barcelona, soccer_team)}. Thus, the evaluation of P results in the set of mappings:

$$\{\{?C \to \text{soccer\_player}\}, \{?C \to \text{sportsman}\}, \{?C \to \text{person}\}\}$$

In this case, pattern P is enough to obtain the type information of Ronaldinho in G according to the RDFS semantics, that is,

$$[\![(\text{Ronaldinho}, \texttt{type}, ?C)]\!]_G^{\text{rdfs}} = [\![(\text{Ronaldinho},\ \texttt{next::type}/(\texttt{next::sc})^*,\ ?C)]\!]_G.$$

Although the expression $\texttt{next::type}/(\texttt{next::sc})^*$ is enough to obtain the type information for Ronaldinho in G, it cannot be used in general to obtain the type information of a resource. For instance, in the same graph, assume that we want to obtain the type information of Everton. In this case, if we evaluate the pattern $(\text{Everton}, \texttt{next::type}/(\texttt{next::sc})^*, ?C)$ over G, we obtain the empty

set. Consider now the nSPARQL pattern

$$Q = (\text{Everton},\ \texttt{node}^{-1}/(\texttt{next::sp})^*/\texttt{next::range},\ ?C).$$

With the expression $\texttt{node}^{-1}/(\texttt{next::sp})^*/\texttt{next::range}$, we follow a path that first navigates from a node to one of its incoming edges by using $\texttt{node}^{-1}$, and then continues with zero or more `sp` edges and a final `range` edge. The evaluation of this expression over G results in the set {(Everton, soccer_team), (Everton, company), (Barcelona, soccer_team), (Barcelona, company)}. Thus, the evaluation of Q in G is the set of mappings:

$$\{\{?C \to \text{soccer\_team}\}, \{?C \to \text{company}\}\}$$

By looking at the closure of G in Fig. 3, we see that pattern Q obtains exactly the type information of Everton in G, that is, $[\![(\text{Everton}, \texttt{type}, ?C)]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$. □

Next we show that the ideas of the previous example can be generalized to the entire RDFS vocabulary. More precisely, we show that if a SPARQL pattern P is constructed by using triple patterns having at least one position with a non-variable element, then the RDFS evaluation of P can be obtained by directly traversing the input graph with an nSPARQL pattern. More precisely, consider the following *translation* function from elements in U to nested regular expressions:

$$\begin{array}{lcl}
\mathit{trans}(\texttt{sc}) & = & (\texttt{next::sc})^+ \\
\mathit{trans}(\texttt{sp}) & = & (\texttt{next::sp})^+ \\
\mathit{trans}(\texttt{dom}) & = & \texttt{next::dom} \\
\mathit{trans}(\texttt{range}) & = & \texttt{next::range} \\
\mathit{trans}(\texttt{type}) & = & (\ \texttt{next::type}/(\texttt{next::sc})^* \mid \\
 & & \quad \texttt{edge}/(\texttt{next::sp})^*/\texttt{next::dom}/(\texttt{next::sc})^* \mid \\
 & & \quad \texttt{node}^{-1}/(\texttt{next::sp})^*/\texttt{next::range}/(\texttt{next::sc})^*\ ) \\
\mathit{trans}(p) & = & \texttt{next::}[(\texttt{next::sp})^*/\texttt{self::}p] \quad \text{for } p \notin \{\texttt{sc}, \texttt{sp}, \texttt{range}, \texttt{dom}, \texttt{type}\}.
\end{array}$$

Notice that we have implicitly used this translation function in Examples 4.3 and 5.1. In the following lemma, we show that given an RDF graph G and a triple pattern t not containing a variable in the predicate position, the above translation function can be used to obtain the RDFS evaluation of t over G by navigating G through a nested regular expression.

Lemma 5.2 *Let (x, p, y) be a* SPARQL *triple pattern with $x, y \in U \cup V$ and $p \in U$. Then $[\![(x, p, y)]\!]_G^{\text{rdfs}} = [\![(x, \mathit{trans}(p), y)]\!]_G$ for every RDF graph G.*

Proof. The proof follows by a case by case analysis of the rules in Tab. 1. For the cases where p is either $\mathtt{sp}, \mathtt{sc}, \mathtt{dom}$, or $\mathtt{range}$, the proof is straightforward. The most complicated case is for the $\mathtt{type}$ keyword. To prove the lemma, it is enough to show that a triple of the form $(a, \mathtt{type}, b)$ can be deduced from G if and only if $(a, b) \in [\![\mathit{trans}(\mathtt{type})]\!]_G$. We show the "only if" part of this property by using an inductive argument. The other direction is similar. The induction is on the number of rules used in a deduction of a triple of the form $(a, \mathtt{type}, b)$. If no rule is used, then $(a, \mathtt{type}, b) \in G$. In this case it is clear that $(a, b) \in [\![\mathit{trans}(\mathtt{type})]\!]_G$ since $(a, b) \in [\![\mathtt{next}{::}\mathtt{type}/(\mathtt{next}{::}\mathtt{sc})^*]\!]_G$. If $(a, \mathtt{type}, b) \notin G$ then we have to consider three cases, depending on whether rule (2b), (3a), or (3b) is the last rule used in a deduction of $(a, \mathtt{type}, b)$. Assume first that rule (2b) is the last rule used in a deduction of $(a, \mathtt{type}, b)$. Then we know that there exist triples $(a, \mathtt{type}, c)$ and $(c, \mathtt{sc}, b)$ that can be deduced from G. By induction hypothesis we know that $(a, c) \in [\![\mathit{trans}(\mathtt{type})]\!]_G$. Moreover, $(c, \mathtt{sc}, b)$ can only be generated by using a sequence of (zero or more) applications of rule (2a), which implies the existence of a path that follows only $\mathtt{sc}$ edges between c and b. Thus, $(c, b) \in [\![(\mathtt{next}{::}\mathtt{sc})^*]\!]_G$, and then $(a, b) \in [\![\mathit{trans}(\mathtt{type})/(\mathtt{next}{::}\mathtt{sc})^*]\!]_G$. By the definition of $\mathit{trans}(\mathtt{type})$, it is easy to see that $[\![\mathit{trans}(\mathtt{type})/(\mathtt{next}{::}\mathtt{sc})^*]\!]_G = [\![\mathit{trans}(\mathtt{type})]\!]_G$ and, thus, $(a, b) \in [\![\mathit{trans}(\mathtt{type})]\!]_G$. Assume now that rule (3a) is the last rule used in a deduction of $(a, \mathtt{type}, b)$. Then there must exist triples (a, q, c) and $(q, \mathtt{dom}, b)$ that can be deduced from G. Notice that we are assuming that RDFS vocabulary only occurs in predicate position of triples. Thus, no rules can be used to deduce a triple with $\mathtt{dom}$ in predicate position. Then we have that $(q, \mathtt{dom}, b)$ belongs to G, and then $(q, b) \in [\![\mathtt{next}{::}\mathtt{dom}]\!]_G$. Also notice that $q \notin \{\mathtt{sp}, \mathtt{sc}, \mathtt{dom}, \mathtt{range}, \mathtt{type}\}$. Thus, given that (a, q, c) is deduced from G, there exists a triple (a, q', c) in G and a path from q' to q that follows only $\mathtt{sp}$ edges. We have that $(a, q) \in [\![\mathtt{edge}/(\mathtt{next}{::}\mathtt{sp})^*]\!]_G$, and then $(a, b) \in [\![\mathtt{edge}/(\mathtt{next}{::}\mathtt{sp})^*/\mathtt{next}{::}\mathtt{dom}]\!]_G$. Finally, since $[\![\mathtt{edge}/(\mathtt{next}{::}\mathtt{sp})^*/\mathtt{next}{::}\mathtt{dom}]\!]_G \subseteq [\![\mathit{trans}(\mathtt{type})]\!]_G$, we have that $(a, b) \in [\![\mathit{trans}(\mathtt{type})]\!]_G$. The analysis for the case of rule (3b) is analogous.

The only remaining case is when $p \notin \{\mathtt{sp}, \mathtt{sc}, \mathtt{dom}, \mathtt{range}, \mathtt{type}\}$. We have to show that (a, p, b) can be deduced from G if and only if $(a, b) \in [\![\mathit{trans}(p)]\!]_G$. We show the "only if" direction, as the other direction is similar. If (a, p, b) belongs to G, then we have that $(a, b) \in [\![\mathtt{next}{::}p]\!]_G = [\![\mathtt{next}{::}[\mathtt{self}{::}p]]\!]_G$. Thus, since $[\![\mathtt{next}{::}[\mathtt{self}{::}p]]\!]_G \subseteq [\![\mathit{trans}(p)]\!]_G$, we have that $(a, b) \in [\![\mathit{trans}(p)]\!]_G$. Assume now that $(a, p, b) \notin G$. It is straightforward to see that (a, p, b) is deduced from G if and only if there exists a triple (a, p', b) in G and a path from p' to p that follows only $\mathtt{sp}$ edges. Then we have that $(p', p) \in [\![(\mathtt{next}{::}\mathtt{sp})^+/\mathtt{self}{::}p]\!]_G$ and, thus, $(a, b) \in [\![\mathtt{next}{::}[(\mathtt{next}{::}\mathtt{sp})^+/\mathtt{self}{::}p]]\!]_G \subseteq [\![\mathit{trans}(p)]\!]_G$. □

Suppose now that we have a SPARQL triple pattern t with a variable in the predicate position, but such that the subject and object of t are not both variables. We show how to construct an nSPARQL pattern P_t such

that $[\![t]\!]^{\text{rdfs}}_G = [\![P_t]\!]_G$. Assume that $t = (x, ?Y, a)$ with $x \in U \cup V$, $?Y \in V$, and $a \in U$, that is, t does not contain a variable in the object position. Consider for every $p \in \{\texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}, \texttt{type}\}$, the pattern $P_{t,p}$ defined as $((x, \mathit{trans}(p), a)$ AND $(?Y, \texttt{self}::p, ?Y))$. Then define pattern P_t as follows:

$$P_t = ((x, \texttt{edge}::a/(\texttt{next}::\texttt{sp})^*, ?Y) \text{ UNION } P_{t,\texttt{sc}} \text{ UNION } P_{t,\texttt{sp}} \text{ UNION } P_{t,\texttt{dom}} \text{ UNION } P_{t,\texttt{range}} \text{ UNION } P_{t,\texttt{type}}).$$

We can similarly define pattern P_t for a triple pattern $t = (a, ?Y, x)$, where $a \in U$, $?Y \in V$ and $x \in U \cup V$. Thus, we have the following result.

Lemma 5.3 *Let $t = (x, ?Y, z)$ be a triple pattern such that $?Y \in V$ and $x \notin V$ or $z \notin V$. Then $[\![t]\!]^{\text{rdfs}}_G = [\![P_t]\!]_G$ for every RDF graph G.*

Proof. The proof follows from Lemma 5.2, and the fact that the evaluation of nre-triple $(?Y, \texttt{self}::p, ?Y)$ is always a single mapping μ such that $\text{dom}(\mu) = \{?Y\}$ and $\mu(?Y) = p$. □

Let $\mathcal{T}$ be the set of triple patterns of the form (x, y, z) such that $x \notin V$ or $y \notin V$ or $z \notin V$. We have translated every triple pattern $t \in \mathcal{T}$ into an nSPARQL pattern P_t such that $[\![t]\!]^{\text{rdfs}}_G = [\![P_t]\!]_G$. Moreover, for every triple pattern t, its translation is of size linear in the size of t. Given that the semantics of SPARQL is defined from the evaluation of triple patterns, we can state the following result. Its proof follows directly from Lemmas 5.2 and 5.3.

Theorem 5.4 *Let P be a* SPARQL *pattern constructed from triple patterns in $\mathcal{T}$. Then there exists an* nSPARQL *pattern Q such that $[\![P]\!]^{\text{rdfs}}_G = [\![Q]\!]_G$ for every RDF graph G. Moreover, pattern Q can be automatically constructed from P in linear time.*

We conclude this section by pointing out that the combination of the translation function presented in this section and nested regular expressions can be very useful in practice, as it allows one to write more expressive queries that need to take into consideration the semantics of the RDFS vocabulary. In fact, the following example shows a query that needs of this combination in order to be expressed.

Example 5.5. Let G be the RDF graph shown in Fig. 1. Assume that one wants to retrieve the pairs of cities such that there is a way of traveling (by using any transportation service) between those cities, and such that every stop in the trip is a coastal city. The following nSPARQL graph pattern expresses this query:

$$P = (?X, (\mathit{trans}(\text{transport})/\texttt{self}::[\mathit{trans}(\texttt{type})/\texttt{self}::\text{coastal\_city}])^+, ?Y).$$

□

6 On the Expressiveness of nSPARQL

A fundamental question about any query language is what are the relationship between its different elements, and whether some of these elements are redundant. In this section, we raise this question for the case of nSPARQL. In particular, we consider in Section 6.1 the question of whether the nesting construct $[\,\cdot\,]$ is necessary in order to encode the inference process of RDFS, and then we consider in Section 6.2 the question of whether the SPARQL operators add expressive power to nSPARQL. In both cases, we obtain a positive answer.

6.1 Regular expressions alone are not enough

Regular expressions are the most common way of giving navigational capabilities to query languages over graph databases [5], and recently to query languages over RDF graphs [3,20,6]. Our language not only allows regular expressions over navigational axes but also nesting of those regular expressions. In our setting, regular expressions are obtained by forbidding the nesting operator and, thus, they are generated by the following grammar:

$$\mathit{exp} \;:=\; \text{axis} \;\mid\; \text{axis}::a\ (a \in U) \;\mid\; \mathit{exp}/\mathit{exp} \;\mid\; \mathit{exp}|\mathit{exp} \;\mid\; \mathit{exp}^* \qquad (5)$$

where axis $\in \{\texttt{self}, \texttt{next}, \texttt{next}^{-1}, \texttt{edge}, \texttt{edge}^{-1}, \texttt{node}, \texttt{node}^{-1}\}$. Let *regular* SPARQL (or just rSPARQL) be the language obtained from nSPARQL by restricting nre-triples to contain in the predicate position only regular expressions (generated by grammar (5)). Notice that rSPARQL is a fragment of nSPARQL and, thus, the semantics for rSPARQL is inherited from nSPARQL.

Next we prove that regular expressions are not enough to obtain the RDFS evaluation of some simple SPARQL patterns by directly traversing the RDF graphs. In fact, we actually show that even for the case of a SPARQL triple pattern, it could be the case that its RDFS evaluation cannot be obtained by any rSPARQL pattern. But before formally proving this, let us give some intuition about this failure.

Example 6.1. Assume that we want to obtain the RDFS evaluation of pattern $P = (?\text{X}, \text{works\_in}, ?Y)$ over an RDF graph G. This can be done by first finding

all the properties p that are sub-properties of works_in, and then finding all the resources a and b such that (a, p, b) is a triple in G. A way to answer P by navigating the graph would be to find the pairs of nodes (a, b) such that there is a path from a to b that: (1) goes from a to one of its leaving edges, then (2) follows a sequence of zero or more `sp` edges until it reaches a works_in edge, and finally (3) returns to the initial edge and moves forward to b. If such a path exists, then it is clear that (a, works_in, b) can be deduced from the graph. The following is a natural attempt to obtain the described path with a regular expression:

$$\texttt{edge}/(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{works\_in}/(\texttt{next}^{-1}::\texttt{sp})^*/\texttt{node}.$$

The problem with the above expression is that, when the path returns from works_in, no information about the path used to reach works_in has been stored. In fact, if we evaluate the pattern

$$Q = (?X, \texttt{edge}/(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{works\_in}/(\texttt{next}^{-1}::\texttt{sp})^*/\texttt{node}, ?Y)$$

over the graph G in Fig. 2, we obtain the set of mappings:

$$\{\{?X \to \text{Ronaldinho}, ?Y \to \text{Barcelona}\}, \{?X \to \text{Ronaldinho}, ?Y \to \text{Everton}\},$$
$$\{?X \to \text{Sorace}, ?Y \to \text{Barcelona}\}, \{?X \to \text{Sorace}, ?Y \to \text{Everton}\}\}$$

By simply inspecting the closure of G in Fig. 3, we obtain that:

$$[\![P]\!]_G^{\text{rdfs}} = \{\{?X \to \text{Ronaldinho}, ?Y \to \text{Barcelona}\}, \{?X \to \text{Sorace}, ?Y \to \text{Everton}\}\}$$

and, thus, we have that Q is not the right representation of P according to the RDFS semantics (since $[\![P]\!]_G^{\text{rdfs}} \neq [\![Q]\!]_G$). $\square$

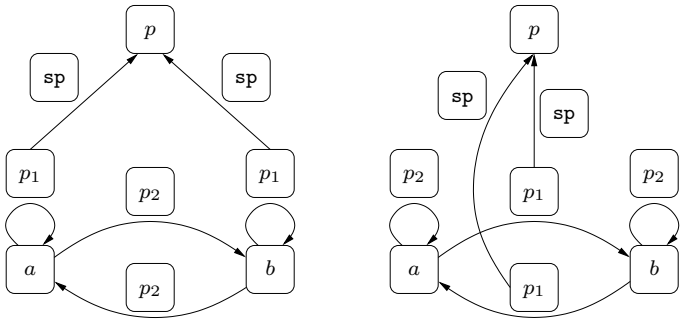
The following theorem shows that the failure in Example 6.1 is not a coincidence, as there exist SPARQL patterns (in fact, an infinite number of patterns) that cannot be rewritten into rSPARQL in order to obtain their RDFS evaluation. Recall that $\mathcal{T}$ is defined as the set of triple patterns (x, y, z) such that $x \notin V$ or $y \notin V$ or $z \notin V$.

Theorem 6.2 *There exists a* SPARQL *pattern P constructed from triple patterns in $\mathcal{T}$ such that for no* rSPARQL *pattern Q, it holds that $[\![P]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$ for every RDF graph G.*

In fact, we prove a stronger result, namely that there even exists a triple pattern t such that, there is no rSPARQL pattern Q satisfying that $[\![t]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$ for every RDF graph G. It should be pointed out that the rSPARQL

pattern Q in the above theorem is allowed to use all the SPARQL operators AND, FILTER, UNION and OPT, besides regular expressions.

Proof. Let $P = (?X, p, ?Y)$, where $p \in U \setminus \{\texttt{sp}, \texttt{sc}, \texttt{type}, \texttt{dom}, \texttt{range}\}$. Next we show that there is no rSPARQL pattern Q such that $[\![P]\!]^{\text{rdfs}}_G = [\![Q]\!]_G$ for every RDF graph G. On the contrary, assume that Q_0 is an rSPARQL pattern such that $[\![P]\!]^{\text{rdfs}}_G = [\![Q_0]\!]_G$ for every RDF graph G. Furthermore, assume that a, b, p_1 and p_2 are elements from U that are not mentioned in Q_0, and let G_1 and G_2 be the RDF graphs shown in the following figure:



G_1 G_2

That is, $G_1 = \{(a, p_1, a),\ (b, p_1, b),\ (a, p_2, b),\ (b, p_2, a),\ (p_1, \texttt{sp}, p)\}$ and $G_2 = \{(a, p_2, a),\ (b, p_2, b),\ (a, p_1, b),\ (b, p_1, a),\ (p_1, \texttt{sp}, p)\}$. We note that

$$[\![P]\!]^{\text{rdfs}}_{G_1} = \{\{?X \to a, ?Y \to a\},\ \{?X \to b, ?Y \to b\}\}, \tag{6}$$

$$[\![P]\!]^{\text{rdfs}}_{G_2} = \{\{?X \to a, ?Y \to b\},\ \{?X \to b, ?Y \to a\}\}. \tag{7}$$

In what follows, we show that $[\![Q_0]\!]_{G_1} = [\![Q_0]\!]_{G_2}$ and, thus, we obtain a contradiction since we assume that $[\![P]\!]^{\text{rdfs}}_{G_1} = [\![Q_0]\!]_{G_1}$ and $[\![P]\!]^{\text{rdfs}}_{G_2} = [\![Q_0]\!]_{G_2}$, and by (6) and (7) we have that $[\![P]\!]^{\text{rdfs}}_{G_1} \neq [\![P]\!]^{\text{rdfs}}_{G_2}$.

To prove that $[\![Q_0]\!]_{G_1} = [\![Q_0]\!]_{G_2}$, it is enough to show that $[\![t]\!]_{G_1} = [\![t]\!]_{G_2}$ for every triple mentioned in Q_0. Given that Q_0 is an rSPARQL pattern, to prove the previous condition, we need to show that $[\![t]\!]_{G_1} = [\![t]\!]_{G_2}$ for every triple t of the form (z, exp, w) with $z, w \in U \cup V$ and exp a regular expression generated by grammar (5). We show first that for a pattern t of the form $(?Z, exp, ?W)$ with $?Z, ?W \in V$, it holds that $[\![t]\!]_{G_1} = [\![t]\!]_{G_2}$. In order to prove the latter condition, it is enough to show that $[\![(?Z, exp, ?W)]\!]_{G_1} = [\![(?Z, exp, ?W)]\!]_{G_2}$ with exp being any of the atomic cases in the definition of regular expressions (see grammar (5)).

- If $t = (?Z, \texttt{next}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to a, ?W \to a\},\ \{?Z \to a, ?W \to b\},\ \{?Z \to b, ?W \to a\},\ \{?Z \to b, ?W \to b\},\ \{?Z \to p_1, ?W \to p\}\}$.
- If $t = (?Z, \texttt{next}^{-1}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to a, ?W \to a\},\ \{?Z \to$

$a, ?W \to b\}$, $\{?Z \to b, ?W \to a\}$, $\{?Z \to b, ?W \to b\}$, $\{?Z \to p, ?W \to p_1\}\}$.

- If $t = (?Z, \texttt{edge}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to a, ?W \to p_1\}, \{?Z \to a, ?W \to p_2\}, \{?Z \to b, ?W \to p_1\}, \{?Z \to b, ?W \to p_2\}, \{?Z \to p_1, ?W \to \texttt{sp}\}\}$.
- If $t = (?Z, \texttt{edge}^{-1}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to p_1, ?W \to a\}, \{?Z \to p_2, ?W \to a\}, \{?Z \to p_1, ?W \to b\}, \{?Z \to p_2, ?W \to b\}, \{?Z \to \texttt{sp}, ?W \to p_1\}\}$.
- If $t = (?Z, \texttt{node}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to p_1, ?W \to a\}, \{?Z \to p_1, ?W \to b\}, \{?Z \to p_2, ?W \to a\}, \{?Z \to p_2, ?W \to b\}, \{?Z \to \texttt{sp}, ?W \to p\}\}$.
- If $t = (?Z, \texttt{node}^{-1}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to a, ?W \to p_1\}, \{?Z \to a, ?W \to p_2\}, \{?Z \to b, ?W \to p_1\}, \{?Z \to b, ?W \to p_2\}, \{?Z \to p, ?W \to \texttt{sp}\}\}$.
- If $t = (?Z, \texttt{self}, ?W)$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to a, ?W \to a\}, \{?Z \to b, ?W \to b\}, \{?Z \to p_1, ?W \to p_1\}, \{?Z \to p_2, ?W \to p_2\}, \{?Z \to \texttt{sp}, ?W \to \texttt{sp}\}, \{?Z \to p, ?W \to p\}\}$.
- If $t = (?Z, \texttt{next}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$ (recall that a, b, p_1, and p_2 are not mentioned in Q_0), then we have to consider three sub-cases:
  - If $c = p$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
  - If $c = \texttt{sp}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to p_1, ?W \to p\}\}$.
  - If $c \in U \setminus \{a, b, p_1, p_2, p, \texttt{sp}\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
- If $t = (?Z, \texttt{next}^{-1}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$, then we have to consider three sub-cases:
  - If $c = p$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
  - If $c = \texttt{sp}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to p, ?W \to p_1\}\}$.
  - If $c \in U \setminus \{a, b, p_1, p_2, p, \texttt{sp}\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
- If $t = (?Z, \texttt{edge}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$, then we have to consider three sub-cases:
  - If $c = p$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to p_1, ?W \to \texttt{sp}\}\}$.
  - If $c = \texttt{sp}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
  - If $c \in U \setminus \{a, b, p_1, p_2, p, \texttt{sp}\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
- If $t = (?Z, \texttt{edge}^{-1}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$, then we have to consider three sub-cases:
  - If $c = p$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to \texttt{sp}, ?W \to p_1\}\}$.
  - If $c = \texttt{sp}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
  - If $c \in U \setminus \{a, b, p_1, p_2, p, \texttt{sp}\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
- If $t = (?Z, \texttt{node}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
- If $t = (?Z, \texttt{node}^{-1}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \emptyset$.
- If $t = (?Z, \texttt{self}::c, ?W)$, where $c \in U \setminus \{a, b, p_1, p_2\}$, then $[\![t]\!]_{G_1} = [\![t]\!]_{G_2} = \{\{?Z \to c,\ ?W \to c\}\}$.

Assume now that t is of the form (z, exp, w) with $z \in U$ or $w \in U$. We have that $z, w \in U \setminus \{a, b, p_1, p_2\}$ and, thus, the only interesting case is when

$z \in \{p, \mathtt{sp}\}$ or $u \in \{p, \mathtt{sp}\}$. We can use a similar argument to the one shown above to prove that $[\![t]\!]_{G_1} = [\![t]\!]_{G_2}$. This concludes the proof of the theorem. □

It should be noticed that Theorems 6.2 and 5.4 imply that nSPARQL is strictly more expressive than rSPARQL.

Corollary 6.3 *There exists an* nSPARQL *pattern that is not equivalent to any* rSPARQL *pattern.*

6.2 On the expressiveness of the SPARQL operators in nSPARQL

Clearly, nested regular expressions add expressive power to SPARQL. The opposite question is whether using SPARQL operators in nSPARQL patterns add expressive power to the language. Next we show that this is indeed the case. In particular, we show that there are simple and natural queries that can be expressed by using nSPARQL features and that cannot be simulated by using only nested regular expressions. Let us present the intuition of this result with an example.

Example 6.4. Let G be the RDF graph shown in Fig. 1. Assume that one wants to retrieve from G the cities $?X$ such that there exists exactly one city that can be reached from $?X$ by using a direct Seafrance service. The following nSPARQL pattern answers this query:

$$\begin{array}{l}[\ (?X, \mathtt{next}{::}\text{Seafrance}/\mathtt{next}^{-1}, ?X) \\ \quad \text{OPT}\ (\ (\ (?X, \mathtt{next}{::}\text{Seafrance}, ?Y)\ \text{AND}\ (?X, \mathtt{next}{::}\text{Seafrance}, ?Z)\ ) \\ \qquad\qquad \text{FILTER}\ \neg ?Y = ?Z\ )\ ]\ \ \text{FILTER}\ \neg\, \text{bound}(?Y)\end{array}$$

The first nre-triple $(?X, \mathtt{next}{::}\text{Seafrance}/\mathtt{next}^{-1}, ?X)$ retrieves the cities $?X$ that are connected with some other city by a Seafrance service. The optional part obtains additional information for those cities $?X$ that are connected with at least two different cities by a Seafrance service. Finally, the pattern filters out those cities for which no optional information was added (by using $\neg\, \text{bound}(?Y)$). That is, only the cities $?X$ that are connected with exactly one city by a Seafrance service remains in the evaluation. If we evaluate the above pattern over G, we obtain a single mapping μ such that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = \text{Calais}$. □

The nSPARQL graph pattern in the above example is essentially counting (up to a fixed threshold) the cities that are connected with $?X$ by a Seafrance service. In the next result, we show that some counting capabilities cannot be obtained by using nSPARQL patterns without considering the OPT operator, even if we combine nested regular expressions by using the operators AND,

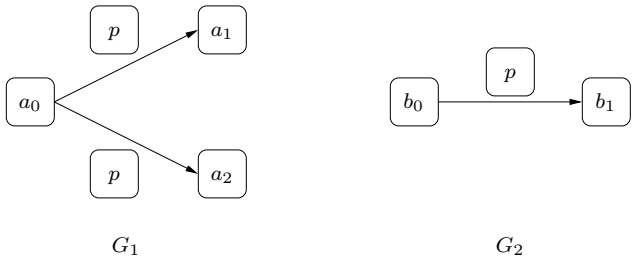


Fig. 9. Two RDF graphs.

UNION and FILTER. The graph pattern used in the proof is similar to that of Example 6.4. It retrieves the nodes $?X$ for which there exists at least two different nodes connected with $?X$. In particular, we prove a stronger result stating that the fragment of nSPARQL that do not use the OPT operator is strictly less expressive than the whole nSPARQL language.

Theorem 6.5 *There is an* nSPARQL *graph pattern that is not equivalent to any* nSPARQL *graph pattern that uses only* AND*,* UNION*, and* FILTER *operators.*

Proof. Let P be a query such that, for every RDF graph G, it holds that $\mu \in [\![P]\!]_G$ if and only if $\mathrm{dom}(\mu) = \{?X\}$ and G contains tuples $(\mu(?X), p, a)$, $(\mu(?X), p, b)$, where a and b are distinct elements. For example, if G_1 is the RDF graph shown in Fig. 9, and μ_0, μ_1 are mappings such that $\mathrm{dom}(\mu_0) = \mathrm{dom}(\mu_1) = \{?X\}$, $\mu_0(?X) = a_0$ and $\mu_1(?X) = a_1$, then $\mu_0 \in [\![P]\!]_{G_1}$ and $\mu_1 \notin [\![P]\!]_{G_1}$. In fact, in this case we have that $[\![P]\!]_{G_1} = \{\mu_0\}$. We start by showing that P is not equivalent to any nSPARQL pattern constructed by using only the operators AND, UNION and FILTER.

On the contrary, assume that Q is an nSPARQL pattern such that Q does not mention the OPT operator and $[\![Q]\!]_G = [\![P]\!]_G$, for every RDF graph G. It is easy to extend the results of [25] to show that the operator UNION is associative in nSPARQL, and there exist nSPARQL patterns P_1, ..., P_n such that each P_i is constructed by using only the operators AND and FILTER, and $[\![Q]\!]_G = [\![P_1 \,\mathrm{UNION}\, P_2 \,\mathrm{UNION} \cdots \mathrm{UNION}\, P_n]\!]_G$, for every RDF graph G. Without loss of generality, we can assume that each P_i is satisfiable in the sense that there exists an RDF graph G and a mapping μ such that $\mu \in [\![P_i]\!]_G$. Next we show that for every pattern P_i, it is the case that $\mathrm{var}(P_i) = \{?X\}$, where $\mathrm{var}(P_i)$ is the set of variables mentioned in P_i

Claim 6.6: For every $i \in \{1, \ldots, n\}$*, it holds that* $\mathrm{var}(P_i) = \{?X\}$.

Proof of Claim 6.6. We prove the claim by contradiction. Assume that for $i \in \{1, \ldots, n\}$, it holds that $\mathrm{var}(P_i) \neq \{?X\}$. Then either $?X \notin \mathrm{dom}(P_i)$ or there exists $?Y \in \mathrm{dom}(P_i)$ such that $?X$ and $?Y$ are distinct variables. Given that P_i is satisfiable, we know that there exists an RDF graph G and a

mapping μ such that $\mu \in [\![P_i]\!]_G$. Given that $[\![P]\!]_G = [\![Q]\!]_G$, we conclude that $\mu \in [\![P]\!]_G$. Thus, if we assume that $?X \notin \mathrm{dom}(P_i)$, we have that $?X \notin \mathrm{dom}(\mu)$, which contradicts the definition of P. Moreover, if we assume that $?Y \in \mathrm{dom}(P_i)$, with $?X$ and $?Y$ distinct variables, then given that P_i is an nSPARQL expression constructed by using only the operators AND and FILTER, we conclude that $?Y \in \mathrm{dom}(\mu)$, which again contradicts the definition of P. This concludes the proof of the claim. □

For the rest of the proof, let G_1 and G_2 be the RDF graphs shown in Fig. 9, and assume that a_0, a_1, a_2, b_0, b_1 are elements of U that are not mentioned in

$$P_1 \text{ UNION } P_2 \text{ UNION } \cdots \text{ UNION } P_n.$$

Let μ_0 be a mapping such that $\mathrm{dom}(\mu_0) = \{?X\}$ and $\mu_0(?X) = a_0$. As we pointed out above, we have that $\mu_0 \in [\![P]\!]_{G_1}$. Thus, we have that $\mu_0 \in [\![Q]\!]_{G_1}$, which implies that $\mu_0 \in [\![P_k]\!]_{G_1}$, for some $k \in \{1, \ldots, n\}$.

Claim 6.7: Let $f : U \to U$ be a function defined as $f(a_0) = b_0$, $f(a_1) = f(a_2) = b_1$ and $f(d) = d$ for every $d \in U \setminus \{a_0, a_1, a_2\}$, axis $\in \{$`self`, `next`, `next`$^{-1}$, `edge`, `edge`$^{-1}$, `node`, `node`$^{-1}\}$ and c an element of U that is mentioned in P_k.

(1) If $(x, y) \in [\![axis]\!]_{G_1}$, then $(f(x), f(y)) \in [\![axis]\!]_{G_2}$.
(2) If $(x, y) \in [\![axis::c]\!]_{G_1}$, then $(f(x), f(y)) \in [\![axis::c]\!]_{G_2}$.

Proof of Claim 6.7. It is enough to prove the claim for axis $\in \{$`self`, `next`, `edge`, `node`$\}$.

(1) If axis = `self`, then we have that $[\![\text{axis}]\!]_{G_1} = \{(a_0, a_0), (a_1, a_1), (a_2, a_2), (p, p)\}$. Thus, by definition of f and given that (b_0, b_0), (b_1, b_1) and (p, p) belong to $[\![\text{axis}]\!]_{G_2}$, we conclude that the claim holds. If axis = `next`, then we have that $[\![\text{axis}]\!]_{G_1} = \{(a_0, a_1), (a_0, a_2)\}$. Thus, by definition of f and given that $(b_0, b_1) \in [\![\text{axis}]\!]_{G_2}$, we conclude that the claim holds. If axis = `edge`, then we have that $[\![\text{axis}]\!]_{G_1} = \{(a_0, p)\}$. Thus, by definition of f and given that $(b_0, p) \in [\![\text{axis}]\!]_{G_2}$, we have that the claim holds. Finally, if axis = `node`, then we have that $[\![\text{axis}]\!]_{G_1} = \{(p, a_1), (p, a_2)\}$. Thus, by definition of f and given that $(p, b_1) \in [\![\text{axis}]\!]_{G_2}$, we conclude that the claim holds.

(2) If axis = `self`, then we have that the claim trivially holds since $[\![\text{axis::}c]\!]_{G_1} = [\![\text{axis::}c]\!]_{G_2} = \{(c, c)\}$ and $f(c) = c$ (given that c is mentioned in P_k and a_0, a_1 and a_2 are not mentioned in this expression). Thus, assume that axis $\in \{$`next`, `edge`, `node`$\}$. If $c \neq p$, then the claim trivially holds since $[\![\text{axis::}c]\!]_{G_1} = \emptyset$ (given that a_0, a_1 and a_2 are not mentioned in P_k). Thus, we assume that $c = p$. If axis = `edge` or axis = `node`, then the claim also holds since $[\![\text{axis::}p]\!]_{G_1} = \emptyset$. Thus, we only need to consider the case axis = `next`. But $[\![\texttt{next::}p]\!]_{G_1} = \{(a_0, a_1), (a_0, a_2)\}$ and, therefore, the claim holds by definition of f and given that $[\![\texttt{next::}p]\!]_{G_2} = \{(b_0, b_1)\}$. This concludes the proof of the

claim. □

From Claim 6.7, we obtain the following corollary.

Corollary 6.8 *Let $f : U \to U$ be a function defined as $f(a_0) = b_0$, $f(a_1) = f(a_2) = b_1$ and $f(d) = d$ for every $d \in U \setminus \{a_0, a_1, a_2\}$, and exp a nested regular expression mentioned in P_k. If $(x, y) \in [\![exp]\!]_{G_1}$, then $(f(x), f(y)) \in [\![exp]\!]_{G_2}$.*

Let ξ_0 be a mapping such $\mathrm{dom}(\xi_0) = \{?X\}$ and $\xi_0(?X) = b_0$. We use Corollary 6.8 to prove that $\xi_0 \in [\![P_k]\!]_{G_2}$. Given that $\mathrm{var}(P_k) = \{?X\}$, $\mu_0 \in [\![P_k]\!]_{G_1}$ and P_k is an nSPARQL pattern constructed by using only the operators AND and FILTER, to prove that $\xi_0 \in [\![P_k]\!]_{G_2}$, it is enough to show that for every triple pattern t in P_k: (a) if $\mathrm{var}(t) = \emptyset$, then t holds in G_2, and (b) if $\mathrm{var}(t) = \{?X\}$, then $\xi_0 \in [\![t]\!]_{G_2}$. Next we show that (a) and (b) hold.

- If $t = (c_1, exp, c_2)$, where $c_1, c_2 \in U$, then given that $\mu_0 \in [\![P_k]\!]_{G_1}$ and P_k is an nSPARQL pattern constructed by using only the operators AND and FILTER, we conclude that $(c_1, c_2) \in [\![exp]\!]_{G_1}$. Thus, we have by Corollary 6.8 that $(f(c_1), f(c_2)) \in [\![exp]\!]_{G_2}$. But we know that a_0, a_1 and a_2 are not mentioned in P_k and, hence, we have that $f(c_1) = c_1$ and $f(c_2) = c_2$. We conclude that $(c_1, c_2) \in [\![exp]\!]_{G_2}$ and, therefore, t holds in G_2.
- If $t = (c, exp, ?X)$, where $c \in U$, then given that $\mu_0 \in [\![P_k]\!]_{G_1}$ and P_k is an nSPARQL pattern constructed by using only the operators AND and FILTER, we conclude that $(c, a_0) \in [\![exp]\!]_{G_1}$. Thus, we have by Corollary 6.8 that $(f(c), b_0) \in [\![exp]\!]_{G_2}$. But we know that a_0, a_1 and a_2 are not mentioned in P_k and, hence, we have that $f(c) = c$. We conclude that $(c, b_0) \in [\![exp]\!]_{G_2}$ and, therefore, $\xi_0 \in [\![t]\!]_{G_2}$.
- If $t = (?X, exp, c)$, where $c \in U$, then we conclude that $\xi_0 \in [\![t]\!]_{G_2}$ as in the previous case.
- If $t = (?X, exp, ?X)$, then given that $\mu_0 \in [\![P_k]\!]_{G_1}$ and P_k is an nSPARQL pattern constructed by using only the operators AND and FILTER, we conclude that $(a_0, a_0) \in [\![exp]\!]_{G_1}$. Thus, we have by Corollary 6.8 that $(b_0, b_0) \in [\![exp]\!]_{G_2}$ and, hence, $\xi_0 \in [\![t]\!]_{G_2}$.

Thus, we conclude that $\xi_0 \in [\![P_k]\!]_{G_2}$. But this implies that $\xi_0 \in [\![Q]\!]_{G_2}$, which leads to a contradiction since $[\![Q]\!]_{G_2} = [\![P]\!]_{G_2}$ and $\xi_0 \notin [\![P]\!]_{G_2}$ (by definition of P).

To conclude the proof of the theorem, it only remains to show that P can be expressed in nSPARQL. Consider first the following nSPARQL pattern:

$$
\begin{aligned}
Q \;=\; [\; & (?Y, \texttt{next::}p/\texttt{next}^{-1}, ?Y) \\
& \text{OPT } (\; (\; (?Y, \texttt{next::}p, ?Z) \text{ AND } (?Y, \texttt{next::}p, ?W)\;) \\
& \qquad \text{FILTER } \neg ?Z = ?W\;)\;] \;\; \text{FILTER } \neg\, \mathrm{bound}(?Z)
\end{aligned}
$$

As we explained in Example 6.4, given an RDF graph G, a mapping μ is in $[\![Q]\!]_G$ if and only if $\mathrm{dom}(\mu) = \{?Y\}$ and there exists exactly one element b

such that $(\mu(?Y), p, b)$ is in G. In fact, if G_1 and G_2 are the graphs shown in Fig. 9, we have that $[\![Q]\!]_{G_1} = \emptyset$, while $[\![Q]\!]_{G_2} = \{\mu\}$ where μ is such that $\mathrm{dom}(\mu) = \{?Y\}$ and $\mu(?Y) = b_0$. Consider now pattern:

$$\Big[\ (?X, \texttt{next}{::}p/\texttt{next}^{\texttt{-1}}, ?X) \quad \text{OPT}\ \Big(\ \big(\ (?X, \texttt{next}{::}p/\texttt{next}^{\texttt{-1}}, ?X)\ \text{AND}\ Q\ \big)\ \ \text{FILTER}\ ?X = ?Y\ \Big)\ \Big]\ \ \text{FILTER}\ \neg\,\mathrm{bound}(?Y)$$

It is not difficult to see that the above graph pattern is equivalent to our initial query P. This concludes the proof of the theorem. □

7 Related work.

The language of nested regular expressions has been motivated by some features of query languages for graphs and trees, namely, XPath [11], temporal logics [12] and propositional dynamic logic [1,16]. In fact, nested regular expressions are constructed by borrowing the notions of *branching* and navigation axes from XPath [11], and adding them to regular expressions over RDF graphs. The algorithm that we present in Section 3.1 is motivated by standard algorithms for some temporal logics [12] and propositional dynamic logic [1,16].

Regarding languages with navigational capabilities for querying RDF graphs, several proposals can be found in the literature [24,3,20,6,4,2]. Nevertheless, none of these languages is motivated by the necessity to evaluate queries over RDFS, and none of them is comparable in expressiveness and complexity of evaluation with the language that we study in this paper. Probably the first language for RDF with navigational capabilities was Versa [24], whose motivation was to use XPath over the XML serialization of RDF graphs. Kochut et al. [20] propose SPARQLeR, an extension of SPARQL that works with *path variables* that represent paths between nodes in a graph. This language also allows to check whether a path conforms to a regular expression. Anyanwu et al. [6] propose a language called SPARQ2L. The authors further investigate the implementation of a query evaluation mechanism for SPARQ2L with emphasis in some secondary memory issues. The language PSPARQL was proposed by Alkhateeb et al. in [3]. PSPARQL extends SPARQL by allowing regular expressions in triple patterns. The same authors propose a further extension of PSPARQL called CPSPARQL [4] that allows constraints over regular expressions. CPSPARQL also allows variables inside regular expressions, thus permitting to retrieve data *along* the traversed paths. In [3,4], the authors study some theoretical aspects of (C)PSPARQL. Given the similarities between nSPARQL and PSPARQL, in the next section we include a detailed comparison between these two languages.

In this section, we compare our language with the closest proposal in the literature, namely the PSPARQL language proposed by Alkhateeb et al. in [3]. PSPARQL is a language that extends SPARQL by allowing regular expressions in triple patterns. Alkhateeb has recently shown [2] that PSPARQL can be used to answer queries by considering the special semantics of RDFS. As explained in [2], this answering process needs the projection operator over PSPARQL graph patterns, that is, it needs SELECT in order to accurately capture the RDFS inference rules, and in particular, it needs extra variables (not needed in the output solution) appearing in the predicate position of triple patterns. In this section, we show that these features have an impact on the complexity of the evaluation problem for PSPARQL, as this problem becomes NP-complete for the conjunctive fragment of this language [2]. On the contrary, it is shown in this section that this problem can be solved in polynomial time for the conjunctive fragment of nSPARQL, since this fragment does not include the aforementioned features (as they are not needed in nSPARQL to answer queries with RDFS vocabulary). It is important to notice that not only the computational complexity is a parameter to be taken into account when comparing query languages, but also the expressiveness. In particular, although we show that the conjunctive fragment of nSPARQL can be evaluated more efficiently than the conjunctive fragment of PSPARQL, it is possible to show that PSPARQL is strictly more expressive than nSPARQL. We also discuss this topic in this section.

As mentioned above, PSPARQL [3,2] is a language that extends SPARQL by allowing regular expressions in triple patterns. More precisely, a PSPARQL query Q consists of a SPARQL graph pattern P that may include some regular expressions over the vocabulary U in some triples, and possibly of a SELECT operator that performs a *projection* over a subset of the variables appearing in P.

Example 7.1. Let G be the following RDF graph storing a genealogy tree:

$$\{(\text{Joan, mother, John}),\ (\text{Joan, mother, Peter}),\ (\text{Joan, mother, Mary}),\\ (\text{John, father, Alan}),\ (\text{Mary, mother, Martin}),\ (\text{Martin, father, Mark})\}.$$

Then PSPARQL query $(?X, (\text{mother} + \text{father})^+, ?Y)$ returns the pairs (a, b) of nodes such that b is a descendant of a. Furthermore, the following PSPARQL

query returns the pairs (c, d) of distinct nodes that have a common ancestor:

$$\text{SELECT } ?Y, ?Z \left[\left((?X, (\text{mother} + \text{father})^+, ?Y) \text{ AND } (?X, (\text{mother} + \text{father})^+, ?Z) \right) \text{ FILTER } \neg(?Y = ?Z) \right].$$

Notice that the SELECT operator is used to indicate that only the values of $?Y$ and $?Z$ should be displayed. Thus, if (c, d) is in the answer of this query, then the common ancestors of these nodes are not displayed. □

As we have shown in Section 6.1, regular expressions can be used to obtain the answer to some RDFS queries. For example, consider the RDF graph:

$$G = \{(a, p, b), (p, \texttt{sp}, c), (c, \texttt{sp}, d), (d, \texttt{sp}, e)\}.$$

To retrieve all the properties $?X$ that are *sub-properties* of e, one can use the PSPARQL query $(?X, \texttt{sp}^+, e)$. Suppose now that one wants to retrieve "the pair of nodes $?X$, $?Y$ that are connected by property e". To answer this query, we need to consider the semantics of the RDFS vocabulary. In particular, given that p is a sub-property of c, c is a sub-property of d, and d is a sub-property of e, we have that p is a sub-property of e in G. Thus, given that a is connected with b by property p, we conclude that a is connected with b by property e. Thus, considering the RDFS semantics, the answer to the previous query over G should be the mapping $\{?X \to a, ?Y \to b\}$. In [2], Alkhateeb proposes to encode the above query by using the following PSPARQL pattern:

$$(?X, ?Z, ?Y) \text{ AND } (?Z, \texttt{sp}^*, e). \tag{8}$$

Notice that the evaluation of (8) is the mapping $\{?X \to a, ?Z \to p, ?Y \to b\}$. Thus, to actually obtain the desired result, a projection must be performed over the variables $?X$ and $?Y$, which gives rise to the following PSPARQL query that retrieves the pair of nodes that are connected by property e:

$$\text{SELECT } ?X, ?Y \left[(?X, ?Z, ?Y) \text{ AND } (?Z, \texttt{sp}^*, e) \right]. \tag{9}$$

Hence, one needs the SELECT operator in PSPARQL to accurately answer queries with RDFS vocabulary. On the other hand, the previous query can be answered as follows in nSPARQL:

$$(?X, \texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::e], ?Y).$$

In this graph pattern, the use of projection has been replaced by the nesting construct in nested regular expressions. We observe that from the results in Section 5, we know that nSPARQL does not need the SELECT operator in order to answer queries that consider the special semantics of the RDFS vocabulary.

Next we show that the use of the SELECT operator makes the complexity of the evaluation problem substantially harder. In order to prove this, we need to introduce some terminology. Define the *conjunctive fragment* of PSPARQL as the set of PSPARQL queries constructed by using only the AND and SELECT operators. Similarly, define the conjunctive fragment of nSPARQL as the set of nSPARQL patterns constructed by using only the AND operator. It is important to notice that we have included the SELECT operator in the conjunctive fragment of PSPARQL as it is needed in this language to answer queries with RDFS vocabulary. On the other hand, SELECT is not considered in the conjunctive fragment of nSPARQL as it is not needed in this language for the encoding of RDFS (in fact, the entire language nSPARQL is defined without considering the SELECT operator). Moreover, the evaluation problems for nSPARQL and PSPARQL are defined as follows. Given a mapping μ, an RDF graph G and an nSPARQL graph pattern P (PSPARQL query Q), the problem is to verify whether μ is in the evaluation of P (Q) over G. Notice that for both PSPARQL and nSPARQL, the evaluation problem has been defined as a decision problem [28].

Theorem 7.2

(1) The evaluation problem for the conjunctive fragment of PSPARQL *is NP-complete [2].*
(2) The evaluation problem for the conjunctive fragment of nSPARQL *can be solved in polynomial time.*

The NP-hardness in the first part of the above theorem can be proved by a reduction from the evaluation problem for relational conjunctive queries [10]. The second part of the above theorem follows from the existence of a polynomial-time algorithm for the evaluation problem for nested regular expressions (provided in Section 3.1), and a result in [25] stating that the complexity of the evaluation problem for SPARQL graph patterns constructed by using only the AND operator is polynomial. It should be noticed that if one adds the SELECT operator to the conjunctive fragment of the nSPARQL language, then the evaluation problem becomes NP-complete. Thus, the difference in complexity between the two fragments mentioned in Theorem 7.2 essentially comes from the use of the SELECT operator.

It is important to notice that not only the computational complexity is a parameter to be taken into account when comparing query languages, but also the expressiveness. In this respect, nested regular expressions can be encoded by using regular expressions and the SELECT operator and, thus, the functionalities of nSPARQL can be encoded by using the functionalities of PSPARQL. On the other hand, nSPARQL does not include the SELECT operator and does not allow triples of the form $(?X, ?Y, ?Z)$. Thus, as these elements are included in PSPARQL, it is possible to conclude that PSPARQL

is strictly more expressive than nSPARQL.

We conclude this section by pointing out that Theorem 7.2 tells that the use of projection and extra variables (not mentioned in the output) makes the evaluation problem considerably harder. In fact, nSPARQL has been carefully designed not to use these features, as shown in the following example.

Example 7.3. Let G be an RDF graph storing genealogical information, and assume that we want to retrieve from G the pairs of distinct people that have a common Italian ancestor. This query can be expressed in PSPARQL as follows:

$$\text{SELECT } ?X, ?Y \left[\left((?A, \text{nationality}, \text{Italian}) \text{ AND } (?A, (\text{mother} \mid \text{father})^+, ?X) \text{ AND } (?A, (\text{mother} \mid \text{father})^+, ?Y)\right) \text{ FILTER } \neg(?X = ?Y)\right].$$

We note that the SELECT operator is used in this query to filter out the common ancestor $?A$. Interestingly, this query can be expressed as an nSPARQL graph pattern, without explicitly mentioning the common ancestor. In fact, let *exp* be the following nested regular expression:

$$(\texttt{next}^{-1}::\text{mother} \mid \texttt{next}^{-1}::\text{father})^+/ \texttt{self}::[\texttt{next}::\text{nationality}/\texttt{self}::\text{Italian}]/(\texttt{next}::\text{mother} \mid \texttt{next}::\text{father})^+.$$

Then we have that the above query is equivalent to the following nSPARQL graph pattern expression:

$$(?X, \mathit{exp}, ?Y) \text{ FILTER } \neg(?X = ?Y).$$

□

Example 7.3 shows that some form of projection can be obtained by using nested regular expressions, without using extra variables. Nevertheless, it should be noticed that this encoding of projection in nSPARQL is not general, as one would need extra variables and operator SELECT to fully obtain projection capabilities. Other query languages have followed before the same approach as nSPARQL, and in particular, they have avoided the use of extra variables. For instance, the XML query language Conditional XPath has the same expressive power over trees as first-order logic [21], and includes the language XPath. The main difference between Conditional XPath and first-order logic is the use of extra variables and quantifiers in the latter. Exactly as for the case of PSPARQL and nSPARQL, these extra features come with a severe impact in the complexity of the evaluation problem for these languages,

as this problem can be solved in polynomial time for the case of Conditional XPath [21] (and also for the case of XPath), while it is PSPACE-complete for the case of first-order logic over trees (and NP-complete for the existential fragment of this language).

8 Concluding Remarks

In this paper, we have proposed nested regular expressions to navigate through an RDF graph, and the nSPARQL query language for RDF that uses nested regular expressions as building blocks. We also study some of the fundamental properties of nested regular expressions and nSPARQL. We have shown that nested regular expressions admit a very efficient evaluation method, that justifies its use in practice. We further showed that the language nSPARQL is expressive enough to be used for querying and navigating RDF data. In particular, we proved that besides capturing the semantics of RDFS, nSPARQL provides some other interesting features that allows users to pose natural and interesting queries over RDF data.

Evaluating queries which involve RDFS vocabulary is challenging, and there is not yet consensus in the Semantic Web community on how to define a query language for RDFS. Nevertheless, there have been several proposals and implementations of query languages for RDF data with RDFS vocabulary (e.g. [19,9,17,15]). As future work, it would be interesting to implement nSPARQL, and to compare it with other implementations of RDFS query languages. In particular, it would be interesting to see whether in practice the process of answering a SPARQL query Q under the RDFS semantics can be efficiently done by first transforming Q into an nSPARQL graph pattern Q', and then answering Q' by using the algorithms developed in this paper.

Acknowledgments

The authors would like to thank the anonymous referees for their careful reading of the paper, and for providing many useful comments. The authors were supported by: Arenas - Fondecyt grant 1090565; Gutierrez - Fondecyt grant 1070348; Pérez - Conicyt Ph.D. Scholarship; Arenas, Gutierrez and Pérez - grant P04-067-F from the Millennium Nucleus Center for Web Research.

References

[1] N. Alechina, N. Immerman. *Reachability Logic: An Efficient Fragment of Transitive Closure Logic*. Logic Journal of the IGPL 8(3) (2000), 325-338.

[2] F. Alkhateeb. *Querying RDF(S) with Regular Expressions*. PhD Thesis, Université Joseph Fourier, Grenoble (FR), 2008.

[3] F. Alkhateeb, J.-F. Baget and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). In Web Semantics: Science, Services and Agents on the World Wide Web 7(2), pages 57–73, 2009.

[4] F. Alkhateeb , J.-F. Baget, J. Euzenat. *Constrained regular expressions in SPARQL*, In *SWWS 2008*, pages 91–99.

[5] R. Angles, C. Gutierrez. *Survey of graph database models*. ACM Comput. Surv., 40(1): 1–39 (2008).

[6] K. Anyanwu, A. Maduko, A. Sheth. *SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases*. In *WWW 2007*, pages 797–806.

[7] M. Arenas, C. Gutierrez, J. Pérez. *An Extension of SPARQL for RDFS*. In *SWDB-ODBIS 2007*, pages 1–20.

[8] D. Brickley, R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, February 2004.
`http://www.w3.org/TR/rdf-schema/`

[9] J. Broekstra, A. Kampman, and F. van Harmelen. *Sesame: A generic architecture for storing and querying RDF and RDF schema*. In *ISWC 2002*, pages 54–68.

[10] A. K. Chandra, P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In STOC 1977, pages 77–90.

[11] J. Clark, S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath`

[12] E. Clarke, O. Grumberg, D. Peled. *Model Checking*. The MIT Press 2000.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *Introduction to Algorithms*. The MIT Press, 2003.

[14] T. Furche, B. Linse, F. Bry, D. Plexousakis, G. Gottlob. *RDF Querying: Language Constructs and Evaluation Methods Compared*. In *Reasoning Web 2006*, pages 1-52.

[15] C. Gutierrez, C. Hurtado, A. Mendelzon. *Foundations of Semantic Web Databases*. In *PODS 2004*, pages 95–106.

[16] D. Harel, D. Kozen and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA (2000).

[17] S. Harris and N. Gibbins. *3store: Efficient bulk RDF storage.* In *PSSS 2003*, pages 1–15.

[18] P. Hayes. *RDF Semantics.* W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-mt/`

[19] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. *RQL: a declarative query language for RDF.* In *WWW 2002*, pages 592–603.

[20] K. Kochut, M. Janik. *SPARQLeR: Extended SPARQL for Semantic Association Discovery.* In *ESWC 2007*, pages 145–159.

[21] M. Marx. *Conditional XPath.* ACM Trans. Database Syst. 30(4): 929-959 (2005)

[22] A. Mendelzon, P. Wood. *Finding Regular Simple Paths in Graph Databases.* In *SIAM J. Comput.* 24(6): 1235–1258 (1995).

[23] S. Muñoz, J. Pérez, C. Gutierrez. *Minimal Deductive Systems for RDF.* In *ESWC 2007*, pages 53–67.

[24] M. Olson, U. Ogbuji. *The Versa Specification.* `http://uche.ogbuji.net/tech/rdf/versa/etc/versa-1.0.xml`.

[25] J. Pérez, M. Arenas, C. Gutierrez. *Semantics and Complexity of* SPARQL. ACM Trans. Database Syst. 34(3): Article No. 16 (2009)

[26] J. Pérez, M. Arenas, C. Gutierrez. *nSPARQL: A Navigational Language for RDF.* In *ISWC 2008*, pages 66–81.

[27] E. Prud'hommeaux, A. Seaborne. *SPARQL Query Language for RDF.* W3C Recommendation, January 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[28] M. Y. Vardi. *The Complexity of Relational Query Languages (Extended Abstract).* In *STOC 1982*, pages 137–146.