nSPARQL: A Navigational Language for RDF

Jorge Pérez[1], Marcelo Arenas[1], and Claudio Gutierrez[2]

[1] Pontificia Universidad Católica de Chile
[2] Universidad de Chile

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 particular, we have argued in a previous paper that *nested regular expressions* are appropriate to navigate RDF data, and we have proposed the nSPARQL query language for RDF, that uses nested regular expressions as building blocks. In this paper, we study some of the fundamental properties of nSPARQL 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 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 the evaluation of a nested regular expression E over an RDF graph G can be computed in time $O(|G| \cdot |E|)$.

1 Introduction

The Resource Description Framework (RDF) [8,14] 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 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 [20], 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 subproperty 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

A. Sheth et al. (Eds.): ISWC 2008, LNCS 5318, pp. 66–81, 2008.

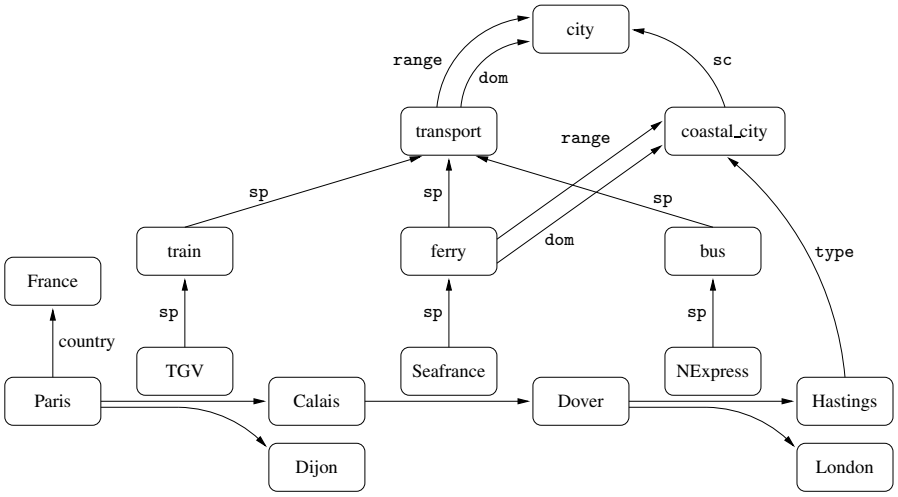

Fig. 1. An RDF graph storing information about transportation services between cities

services (in the form of RDFS annotations). For instance, in the graph we have that a "Seafrance" service is a subproperty of a "ferry" service, which in turn is a subproperty 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 subproperty 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 subproperty of "train", which in turn is a subproperty of "transport". Notice that the condition also holds for "Paris" and "Dover".

Driven by these considerations, we introduced in [7] the language nSPARQL, that incorporates navigational capabilities to a fragment of SPARQL. The main goal of [7] was not to formally study nSPARQL, but instead to provide evidence that the navigational capabilities of nSPARQL can be used to pose many interesting and natural queries over RDF data. Our goal in this paper is to formally study some 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, the building blocks of nSPARQL patterns are *nested regular expressions*, which specify how to navigate RDF data. Thus, we show in this paper that nested regular expressions can be evaluated efficiently; if the appropriate data structure is used to store RDF graphs, the evaluation of a nested regular expression E over an RDF graph G can be computed in time $O(|G| \cdot |E|)$.

The second fundamental question about nSPARQL is how expressive is the language. In this paper, 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 encode 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 to deal with the semantics of RDFS. More precisely, we show that regular expressions alone are not enough to obtain the RDFS evaluation of some queries by simply navigating 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 notions about RDF and RDFS. 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, and study the expressiveness of this language. Concluding remarks and related work are given in Section 5.

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[1]. More 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.

Figure 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 from U) with predefined semantics, the RDFS vocabulary (RDF Schema [8]). This set of

[1] 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. We neither make a special distinction between IRIs and Literals.

Table 1. RDFS inference rules

1. Subproperty:	*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)* $\frac{(\mathcal{A},\texttt{range},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{Y},\texttt{type},\mathcal{B})}$

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 [17]. 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 [17] was shown to be equivalent to the full RDFS semantics when one focuses on the mentioned fragment.

We use the system of rules in Tab. 1. This system was proved in [17] 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 [17] 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 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 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 subproperty 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 (Calais, Seafrance, Dover) and (Seafrance, `sp`, ferry), 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. □

3 Nested Regular Expressions for RDF Data

Navigating graphs is done usually 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, instead of classical edges (pair of nodes), we have directed triples of nodes (*hyperedges*). Hence, a language for navigating RDF graphs should be able to deal with this type of objects. In [7], we

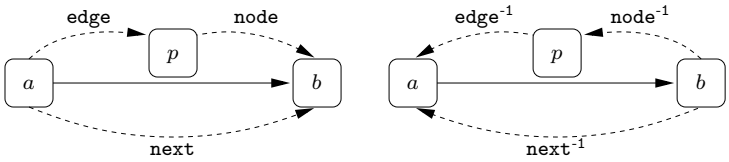

Fig. 2. Forward and backward axes for an RDF triple (a, p, b)

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* to move through an RDF triple. These axes are shown in Fig. 2 (together with their inverses).

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, and one can use classical regular expressions over these axes to define a set of paths that can be used in a query. An additional axis `self` 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:

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

where axis $\in \{$`self`, `next`, `next`$^{-1}$, `edge`, `edge`$^{-1}$, `node`, `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 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, $a \in U$, $\text{voc}(G)$ 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 shortcut 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.

Table 2. Formal semantics of nested regular expressions

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

Example 2. Let G be the graph in Fig. 1, and consider expression $exp_1 = \texttt{next::}[\texttt{next::sp/self::}\text{train}]$. The nested expression $[\texttt{next::sp/self::}\text{train}]$ performs an existential test; it defines the set of nodes z in G such that there exists a path from z 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 $[\![exp_1]\!]_G = \{(\text{Paris}, \text{Calais}), (\text{Paris}, \text{Dijon})\}$. □

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 Complexity of Evaluating Nested Regular Expressions

In this section, we study the complexity of evaluating nested regular expressions over RDF graphs. We present an algorithm for this problem that works in time proportional to the size of the input graph times the size of the expression being evaluated. As is customary when studying the complexity of the evaluation problem for a query language (cf. [21]), we consider its associated decision problem. For nested regular expressions, this problem is defined as:

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

We assume that an RDF graph G is stored as an adjacency list that makes explicit the navigation axes (and their inverses). Thus, every $u \in \mathrm{voc}(G)$ 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 $u \in \mathrm{voc}(G)$. 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 $u \in \mathrm{voc}(G)$, 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.

In this section, we assume some familiarity with automata theory. Recall that given a regular expression r, one can construct in linear time a nondeterministic finite automaton with ε-transitions $\mathcal{A}_r$ that accepts the language generated by r.

A key idea in the algorithm introduced in this section 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::$[exp']$,
$\mathbf{D}_0(exp_1/exp_2) = \mathbf{D}_0(exp_1|exp_2) = \mathbf{D}_0(exp_1) \cup \mathbf{D}_0(exp_2)$,
$\mathbf{D}_0(exp^*) = \mathbf{D}_0(exp)$,

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:

$$exp \;=\; \texttt{next}::a/(\texttt{next}::[\texttt{next}::a/\texttt{self}::b])^*/(\texttt{next}::[\texttt{node}::b] \mid \texttt{next}::a)^+,$$

we have $\mathbf{D}_0(exp) = \{\texttt{next}::a,\ \texttt{next}::[\texttt{next}::a/\texttt{self}::b],\ \texttt{next}::[\texttt{node}::b]\}$. Notice that a nested regular expression exp can be viewed as a classical regular expression over 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 alphabet $\mathbf{D}_0(exp)$.

The algorithm for the evaluation of nested regular expressions is similar to the algorithms for the evaluation of some temporal logics [11] and propositional dynamic logic [1]. Given an RDF graph G and a nested regular expression exp, it proceeds by recursively labeling every node u of G with a set $\mathrm{label}(u)$ of nested expressions. Initially, $\mathrm{label}(u)$ is the empty set. Then at the end of the execution of the algorithm, it holds that $exp \in \mathrm{label}(u)$ if and only if there exists z such that $(u, z) \in [\![exp]\!]_G$. In the algorithm, we use the product automaton $G \times \mathcal{A}_{exp}$, which is constructed as follows. 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 $\mathrm{voc}(G) \times Q$, and its transition function $\delta' : (\mathrm{voc}(G) \times Q) \times (\mathbf{D}_0(exp) \cup \{\varepsilon\}) \to 2^{\mathrm{voc}(G) \times Q}$ is defined as follows. For every $(u, p) \in \mathrm{voc}(G) \times Q$ 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 $(\text{axis}::a, v) \in \alpha(u)$,
- $s =$ axis::a and $(\text{axis}::a, v) \in \alpha(u)$,
- $s =$ axis::$[exp]$ and there exists b such that $(\text{axis}::b, v) \in \alpha(u)$ and $exp \in \mathrm{label}(b)$,

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 $u \in \mathrm{voc}(G)$. That is, $G \times \mathcal{A}_{exp}$ is the standard product automaton if G is viewed as an NFA over alphabet $\mathbf{D}_0(exp)$. It is straightforward to prove that $G \times \mathcal{A}_{exp}$ can be constructed in time $O(|G| \cdot |\mathcal{A}_{exp}|)$.

Now we have all the necessary ingredients to present the algorithm for the evaluation problem for nested regular expressions. This algorithm is split in two procedures: LABEL labels G according to nested expression 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 reaches a state (v, q_f) in $G \times \mathcal{A}_{exp}$, with $q_f \in F$ **do**
6. $\mathrm{label}(u) := \mathrm{label}(u) \cup \{exp\}$

EVAL$(G, exp, (a, b))$:
1. **for each** $u \in \mathrm{voc}(G)$ **do**
2. $\mathrm{label}(u) := \emptyset$
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

It is not difficult to see that these procedures work in time $O(|G| \cdot |exp|)$. Just observe that step 5 of procedure LABEL and step 6 of procedure EVAL, can be done in time linear in the size of $G \times \mathcal{A}_{exp}$ by traversing $G \times \mathcal{A}_{exp}$ in a depth first search manner.

Theorem 1. *Procedure* EVAL *solves the evaluation problem for nested regular expressions in time* $O(|G| \cdot |exp|)$.

4 The Navigational Language nSPARQL

In this section, we introduce the language nSPARQL, and we formally study its expressiveness. nSPARQL is essentially obtained by using triple patterns with nested regular expressions in the predicate position, plus SPARQL operators AND, OPT, UNION, and FILTER. Before formally introducing nSPARQL, we recall the necessary definitions about SPARQL.

SPARQL [20] is the standard language for querying RDF data. We use here the algebraic formalization introduced in [19]. Assume the existence of an infinite set V of variables disjoint from U. A SPARQL graph pattern is 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.

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 $\mathrm{dom}(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are *compatible* if for every $x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$, 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:

– $[\![t]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(t) \text{ and } \mu(t) \in G\}$, where $\mathrm{var}(t)$ 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 $\mathrm{bound}(?X)$ and $?X \in \mathrm{dom}(\mu)$;
– R is $?X = c$, where $c \in U$, $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) = c$;
– R is $?X = ?Y$, $?X \in \mathrm{dom}(\mu)$, $?Y \in \mathrm{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$.

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

It was shown in [19], 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.

Now we formally define the language *nested* SPARQL (or just nSPARQL), by considering triples with nested regular expressions in the predicate position. 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 patterns are recursively defined from nre-triples:

– An nre-triple is an nSPARQL pattern.
– If P_1 and P_2 are nSPARQL 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 patterns.

To define the semantics of nSPARQL, we just need to define the semantics of nre-triples. 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 (has the same evaluation of) 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 loosing in nSPARQL is only the possibility of using variables in the three positions of a triple pattern.

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 [7], we introduced nSPARQL as a way to overcome this limitation. The main goal of [7] was not to formally study nSPARQL, but instead to provide evidence that the navigational capabilities of nSPARQL can be used to pose many interesting and natural queries over RDF data. Our goal in this paper is to formally justify nSPARQL. 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 this section, we study some fundamental properties related to the expressiveness of nSPARQL. But before doing that, we provide some additional examples of queries that are likely to occur in the Semantic Web, but cannot be expressed in SPARQL without using nested regular expressions.

Example 3. 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 $[\![P_2]\!]_G = \{\{?X \rightarrow \text{Paris}, ?Y \rightarrow \text{France}\}, \{?X \rightarrow \text{Calais}\}\}$. □

Example 4. 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 5. Let G be the graph in Fig. 1 and P_1 the following pattern:

$$P_1 = (?X,\ \texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}],\ ?Y). \tag{2}$$

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 another, we can use the following nSPARQL pattern:

$$P_2 = (?X,\ (\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+,\ ?Y). \tag{3}$$

In this section, we formally prove that (2) and (3) cannot be expressed without using nested expressions of the form axis::$[exp]$. □

4.1 On RDFS and nSPARQL

We claimed in [7] that the language of nested regular expressions is powerful enough to deal with the predefined semantics of RDFS. In this section, we formally prove this fact. More precisely, we show that if one wants to answer a SPARQL query P according to the semantics of RDFS, then one can rewrite P into an nSPARQL query Q such that Q retrieves the answer to P by directly traversing the input graph. We also show that the nesting operation is crucial for this result.

SPARQL follows a *subgraph-matching* approach, and thus, a SPARQL query treats RDFS vocabulary without considering its predefined semantics. 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. Let the *closure* of an RDF graph G, denoted by $\text{cl}(G)$, be the graph obtained from G by successively applying the rules in Tab. 1 until the graph does not change. 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. The theoretical formalization of such an approach was studied in [12]. The following definition formalizes this notion.

Definition 1. *Given a* SPARQL *graph pattern P, the* RDFS evaluation of P over G, *denoted by $[\![P]\!]_G^{\text{rdfs}}$, is defined as the set of mappings $[\![P]\!]_{\text{cl}(G)}$, that is, as the evaluation of P over the closure of G.*

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 [3,16,6]. 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:

$$exp \;:=\; \text{axis} \mid \text{axis::}a\ (a \in U) \mid exp/exp \mid exp|exp \mid exp^* \qquad (4)$$

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 (4)). Notice that rSPARQL is a fragment of nSPARQL and, thus, the semantics for rSPARQL is inherited from nSPARQL.

Our next result shows that regular expressions are not enough to obtain the RDFS evaluation of some simple SPARQL patterns by directly traversing RDF graphs. In fact, the following theorem shows that there is a SPARQL triple pattern whose RDFS evaluation cannot be obtained by any rSPARQL pattern.

Theorem 2. *Let $p \in U \setminus \{\texttt{sp}, \texttt{sc}, \texttt{type}, \texttt{dom}, \texttt{range}\}$ and consider triple pattern $(?X, p, ?Y)$. There is no* rSPARQL *pattern Q such that $[\![(?X, p, ?Y)]\!]^{\text{rdfs}}_G = [\![Q]\!]_G$ for every RDF graph G.*

nSPARQL and RDFS evaluation. In this section, 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:

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

Notice that we have implicitly used this translation function in Example 5.

Lemma 1. *Let (x, a, y) be a* SPARQL *triple pattern with $x, y \in U \cup V$ and $a \in U$, then $[\![(x, a, y)]\!]^{\text{rdfs}}_G = [\![(x, trans(a), y)]\!]_G$ for every RDF graph G.*

That is, given an RDF graph G and a triple pattern t not containing a variable in the predicate position, it is possible to obtain the RDFS evaluation of t over G by navigating G through a nested regular expression.

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 $[\![t]\!]_G^{\text{rdfs}} = [\![P_t]\!]_G$. 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 then 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 2. *Let $t = (x, ?Y, z)$ be a triple pattern such that $?Y \in V$, and $x \notin V$ or $z \notin V$. Then $[\![t]\!]_G^{\text{rdfs}} = [\![P_t]\!]_G$ for every RDF graph G.*

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 $[\![t]\!]_G^{\text{rdfs}} = [\![P_t]\!]_G$. 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.

Theorem 3. *Let P be a* SPARQL *pattern constructed from triple patterns in $\mathcal{T}$. Then there exists an* nSPARQL *pattern Q such that $[\![P]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$ for every RDF graph G. Moreover, the size of Q is linear in the size of P.*

The following example shows that one can combine the translation function presented in this section with nested regular expression patterns to obtain more expressive queries that take into account the RDFS semantics.

Example 6. 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 pattern answers this query:

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

Notice that Theorems 2 and 3 imply that nSPARQL is strictly more expressive than rSPARQL. We state this result in the following corollary.

Corollary 1. *There exists an* nSPARQL *pattern that is not equivalent to any* rSPARQL *pattern.*

4.2 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 7. 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:

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

The first nre-triple $(?X, \texttt{next}{::}\text{Seafrance}/\texttt{next}^{-1}, ?X)$ 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 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, UNION and FILTER. The query used in the proof is similar to that of Example 7. It retrieves the nodes $?X$ for which there exists at least two different nodes connected with $?X$.

Theorem 4. *There is an* nSPARQL *pattern that is not equivalent to any* nSPARQL *pattern that uses only* AND*,* UNION*, and* FILTER *operators.*

5 Related Work and Concluding Remarks

Related work. The language of nested regular expressions has been motivated by some features of query languages for graphs and trees, namely, XPath [10], temporal logics [11] and propositional dynamic logic [1]. In fact, nested regular expressions are constructed by borrowing the notions of *branching* and navigation axes from XPath [10], 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 [11] and propositional dynamic logic [1].

Regarding languages with navigational capabilities for querying RDF graphs, several proposals can be found in the literature [18,3,16,6,4,2]. 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 [18], whose motivation was to use XPath over the XML serialization of RDF

graphs. Kochut et al. [16] 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.

Alkhateeb has recently shown [2] that PSPARQL, that is, the full SPARQL language extended with regular expressions, can be used to encode RDFS inference. Although PSPARQL [2] and the language rSPARQL that we present in Section 4.1 are similar, when defining rSPARQL we use a fragment of SPARQL, namely, the graph pattern matching facility without solution modifiers like projection. Alkhateeb's encoding [2] needs the projection operator, and in particular, extra variables (not needed in the output solution) appearing in the predicate position of triple patterns. This feature is not allowed in the fragment that we use to construct languages rSPARQL and nSPARQL. Although PSPARQL could be used to answer some RDFS queries, the additional abilities needed in PSPARQL come with an associated complexity impact in the evaluation problem for the conjunctive fragment, namely, NP-completeness [2]. By using the results in [19] and the complexity of the evaluation problem for nested regular expressions, it is easy to show that the complexity of the evaluation problem for the conjunctive fragment of nSPARQL is polynomial.

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. [15,9,13,12]. It would be interesting to compare these approaches with the process of answering a SPARQL query under the RDFS semantics by first compiling it into an nSPARQL query.

Concluding Remarks. In this paper, we have started the formal study of nested regular expressions and the language nSPARQL, that we proposed in [7]. 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.

Acknowledgments. We are grateful to the anonymous referees for their helpful comments. The authors were supported by: Arenas - Fondecyt grant 1070732; 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. Alechina, N., Immerman, N.: Reachability Logic: An Efficient Fragment of Transitive Closure Logic. Logic Journal of the IGPL 8(3), 325–338 (2000)
2. Alkhateeb, F.: Querying RDF(S) with Regular Expressions. PhD Thesis, Université Joseph Fourier, Grenoble (FR) (2008)
3. Alkhateeb, F., Baget, J., Euzenat, J.: RDF with regular expressions. Research Report 6191, INRIA (2007)
4. Alkhateeb, F., Baget, J., Euzenat, J.: Constrained regular expressions in SPARQL. In: SWWS 2008, pp. 91–99 (2008)
5. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1), 1–39 (2008)
6. Anyanwu, K., Maduko, A., Sheth, A.: SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In: WWW 2007, pp. 797–806 (2007)
7. Arenas, M., Gutierrez, C., Pérez, J.: An Extension of SPARQL for RDFS. In: SWDB-ODBIS 2007, pp. 1–20 (2007)
8. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (February 2004), `http://www.w3.org/TR/rdf-schema/`
9. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
10. Clark, J., DeRose, S.: XML Path Language (XPath). W3C Recommendation (November 1999), `http://www.w3.org/TR/xpath`
11. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (2000)
12. Gutierrez, C., Hurtado, C., Mendelzon, A.: Foundations of Semantic Web Databases. In: PODS 2004, pp. 95–106 (2004)
13. Harris, S., Gibbins, N.: 3store: Efficient bulk RDF storage. In: PSSS 2003, pp. 1–15 (2003)
14. Hayes, P.: RDF Semantics. W3C Recommendation (February 2004), `http://www.w3.org/TR/rdf-mt/`
15. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: a declarative query language for RDF. In: WWW 2002, pp. 592–603 (2002)
16. Kochut, K., Janik, M.: SPARQLeR: Extended SPARQL for Semantic Association Discovery. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 145–159. Springer, Heidelberg (2007)
17. Muñoz, S., Pérez, J., Gutierrez, C.: Minimal Deductive Systems for RDF. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 53–67. Springer, Heidelberg (2007)
18. Olson, M., Ogbuji, U.: The Versa Specification, `http://uche.ogbuji.net/tech/rdf/versa/etc/versa-1.0.xml`
19. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
20. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (January 2008), `http://www.w3.org/TR/rdf-sparql-query/`
21. Vardi, M.Y.: The Complexity of Relational Query Languages (Extended Abstract). In: STOC 1982, pp. 137–146 (1982)