An Extension of SPARQL for RDFS

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

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

Abstract. RDF Schema (RDFS) extends RDF with a schema vocabulary with a predefined semantics. Evaluating queries which involve this 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 paper, we introduce a language for querying RDFS data. This language is obtained by extending SPARQL with *nested regular expressions* that allow to *navigate* through an RDF graph with RDFS vocabulary. This language is expressive enough to answer SPARQL queries involving RDFS vocabulary, by directly traversing the input graph.

1 Introduction

The Resource Description Framework (RDF) [16,6,14] is a data model for representing information about World Wide Web resources. The RDF specification includes a set of reserved IRIs, the RDFS vocabulary (called RDF Schema), that has a predefined semantics. This vocabulary is designed to describe special relationships between resources like typing and inheritance of classes and properties, among others features [6].

Jointly with the RDF release in 1998 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 (see Haase et al. [12] and Furche et al. [9] for detailed comparisons of RDF query languages). 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 [21]. Since then, SPARQL has been rapidly adopted as the standard to query Semantic Web data. In January 2008, SPARQL became a W3C Recommendation.

The specification of SPARQL is targeted to RDF data, not including RDFS vocabulary. The reasons to follow this approach are diverse, including: (1) the lack of a standard definition of a semantics for queries under the presence of vocabulary and, hence, the lack of consensus about it; (2) the computational complexity challenges of querying in the presence of a vocabulary with a predefined semantics; and (3) practical considerations about real-life RDF data spread on the Web. These reasons explain also why most of the groups working on the definition of RDF query languages have focused in querying plain RDF data.

Nevertheless, there are several proposals to address the problem of querying RDFS data. Current practical approaches taking into account the predefined

V. Christophides et al. (Eds.): SWDB-ODBIS 2007, LNCS 5005, pp. 1–20, 2008.

semantics of the RDFS vocabulary (e.g. Harris and Gibbins [11], Broekstra et al. [7] in Sesame), roughly implement the following procedure. Given a query Q over an RDF data source G with RDFS vocabulary, the *closure* of G is computed first, that is, all the implicit information contained in G is made explicit by adding to G all the statements that are logical consequences of G. Then the query Q is evaluated over this extended data source. The theoretical formalization of such an approach was studied by Gutierrez et al. [10].

From a practical point of view, the above approach has several drawbacks. First, it is known that the size of the closure of a graph G is of quadratic order in the worst case, making the computation and storage of the closure too expensive for web-scale applications. Second, once the closure has been computed, all 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 must scan a large part of the input data. Third, the approach is not goal-oriented. Although in practice most queries will use just a small 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 data source G and a query Q that asks whether a resource A is a *sub-class of* a resource B. In its abstract syntax, RDF statements are modeled as a *subject-predicate-object* structure of the form (s, p, o), called an RDF triple. Furthermore, the keyword rdfs:subClassOf is used in RDFS to denote the sub-class relation between resources. Thus, answering Q amounts to check whether the triple (A, rdfs:subClassOf, B) is a logical consequence of G. The predefined semantics of RDFS states that rdfs:subClassOf is a transitive relation among resources. Then to answer Q, a goal-oriented approach should not compute the closure of the entire input graph G (which could be of quadratic order in the size of G), but instead it should just verify whether there exist resources $R_1, R_2, \ldots R_n$ such that $A = R_1$, $B = R_n$, and (R_i, rdfs:subClassOf, R_{i+1}) is a triple in G for $i = 1, \ldots, n-1$. That is, we can answer Q by checking the existence of an rdfs:subClassOf-path from A to B in G, which takes linear time in the size of G [18].

It was shown by Muñoz el al. [18] that testing whether an RDFS triple is implied by an RDFS data source G can be done without computing the closure of G. The idea is that the RDFS deductive rules allow to determine if a triple is implied by G by essentially checking the existence of paths over 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. For example, to check whether (A, rdfs:subClassOf, 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 rdfs:subClassOf. This observation motivates the use of extended triple patterns of the form $(A, \text{rdfs:subClassOf}^{+}, B)$, where $\text{rdfs:subClassOf}^{+}$ is the regular expression denoting paths of length at least 1 and where each edge has label rdfs:subClassOf. Thus, one can readily see that a language for navigating RDFS data would be useful for obtaining the answer of queries considering the predefined semantics of the RDFS vocabulary.

Driven by this motivation, in this paper we introduce a language that extends SPARQL with navigational capabilities. The resulting language turns out to be expressive enough to capture the deductive rules of RDFS. Thus, we can obtain the RDFS evaluation of an important fragment of SPARQL by navigating directly the input RDFS data source, without computing the closure.

This idea can be developed at several levels. We first consider a navigational language that includes regular expressions and takes advantage of the special features of RDF. Paths defined by regular expressions has been widely used in graph databases [17,3], and recently, have been also proposed in the RDF context [1,4,2,15,5]. We show that although paths defined in terms of regular expressions are useful, regular expressions alone are not enough to obtain the RDFS evaluation of some queries by simply navigating RDF data. Thus, we enrich regular expressions by borrowing the notion of *branching* from XPath [8], to obtain what we call *nested regular expressions*. Nested regular expressions are enough for our purposes and, furthermore, they provide an interesting extra expressive power to define complex path queries over RDF data with RDFS vocabulary.

Organization of the paper. In Section 2, we present a summary of the basics of RDF, RDFS, and SPARQL, based on Muñoz et al. [18] and Pérez et al. [20]. Section 3 is the core part of the paper, and introduces our proposal for a navigational language for RDF. We first discuss the related work on navigating RDF in Section 3.1. In Section 3.2, we introduce a first language for navigating RDF graphs based on regular expressions, and we discuss why regular expressions alone are not enough for our purposes. Section 3.3 presents the language of nested regular expressions, and shows how these expressions can be used to obtain the RDFS evaluation of SPARQL patterns. In Section 3.4, we give some examples of the extra expressive power of nested regular expressions, showing the usefulness of the language to extract complex path relations from RDF graphs. Finally, Section 4 presents some conclusions.

2 RDFS and SPARQL

In this section, we present the algebraic formalization of the core fragment of SPARQL over RDF graphs introduced in [20], and then we extend this formalization to RDFS graphs. But before doing that, we introduce some notions related to RDF and the core fragment of RDFS.

2.1 The RDF Data Model

RDF is a graph data format for representing information in the Web. An RDF statement is a *subject-predicate-object* structure, called an 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

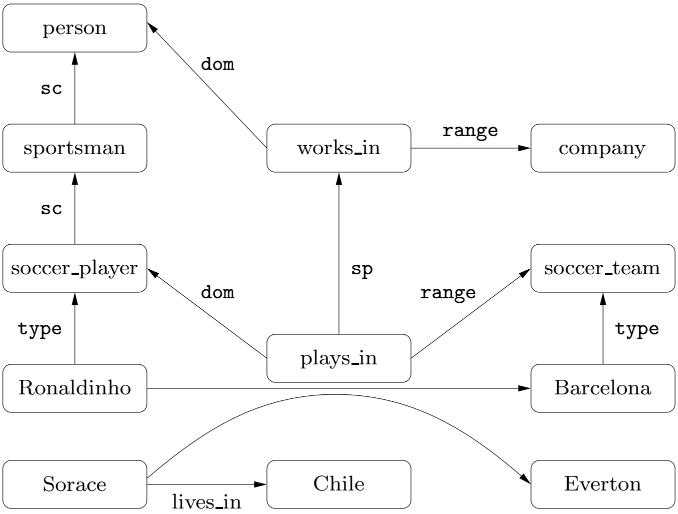

Fig. 1. An RDF graph storing information about soccer players

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 (or RDF data source) is a finite set of RDF triples.

Figure 1 shows an RDF graph that stores information about soccer players. In this figure, a triple (s, p, o) is depicted as an arc $s \xrightarrow{p} o$, that is, s and o are represented as nodes and p is represented as an arc label. For example, (Sorace, lives_in, Chile) is a triple in the RDF graph in Figure 1. Notice that, an RDF graph is not a standard labeled graph as its set of labels may have a nonempty intersection with its set of nodes. For instance, consider triples (Ronaldinho, plays_in, Barcelona) and (plays_in, `sp`, works_in) in the RDF graph in Figure 1. In this example, plays_in is the predicate of the first triple and the subject of the second one, and thus, acts simultaneously as a node and an edge label.

The RDF specification includes a set of reserved IRIs (reserved elements from U) with predefined semantics, the RDFS vocabulary (RDF Schema [6]). This set of reserved words is designed to deal with inheritance of classes and properties, as well as typing, among other features [6]. In this paper, we consider the subset of the RDFS vocabulary composed by the special IRIs rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain and rdf:type, which are denoted by `sc`, `sp`, `range`, `dom` and `type`, respectively. The RDF graph in Figure 1 uses these keywords to relate resources. For instance, the graph contains triple (sportsman, `sc`, person), thus stating that sportsman is a *sub-class of* person.

The fragment of RDFS consisting of the keywords `sc`, `sp`, `range`, `dom` and `type` was considered in [18]. In that paper, the authors provide a formal semantics

[1] In this paper, we do not consider anonymous resources called blank nodes in the RDF data model, that is, our study focus on *ground* RDF graphs. We neither make a special distinction between IRIs and Literals.

for it, and also show it to be well-behaved as the remaining RDFS vocabulary does not interfere with the semantics of this fragment. This together with some other results from [18] provide strong theoretical and practical evidence for the importance of this fragment. In this paper, we consider the keywords `sc`, `sp`, `range`, `dom` and `type`, and we use the semantics for them from [18], instead of using the full RDFS semantics (these two were shown to be equivalent in [18]).

For the sake of simplicity, we do not include here the model theoretical semantics for RDFS from [18], and we only present the system of rules from [18] that was proved to be equivalent to the model theoretical semantics (that is, was proved to be sound and complete for the inference problem for RDFS in the presence of `sc`, `sp`, `range`, `dom` and `type`). Table 1 shows the inference system for the fragment of RDFS considered in this paper. Next we formalize the notion of deduction for this system of inference rules. 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. Finally, the *closure* of an RDF graph G, denoted by $\mathrm{cl}(G)$, is defined as the graph obtained from G by successively applying the rules in Table 1 until the graph does not change.

Example 1. Consider the RDF graph in Figure 1. By applying the rule (1b) to (Ronaldinho, plays_in, Barcelona) and (plays_in, `sp`, works_in), we obtain that (Ronaldinho, works_in, Barcelona) is in the closure of the graph. Moreover, by applying the rule (3b) to this last triple and (works_in, `range`, company), we obtain that (Barcelona, `type`, company) is also in the closure of the graph. Figure 2 shows the complete closure of the RDF graph in Figure 1. The solid lines in Figure 2 represent the triples in the original graph, and the dashed lines the additional triples in the closure. □

In [18], it was shown that if the number of triples in G is n, then the closure $\mathrm{cl}(G)$ could have, in the worst case, $\Omega(n^2)$ triples.

Table 1.

1. *Subproperty:*

   (a) $\frac{(\mathcal{A},\texttt{sp},\mathcal{B})\ \ (\mathcal{B},\texttt{sp},\mathcal{C})}{(\mathcal{A},\texttt{sp},\mathcal{C})}$ *(b)* $\frac{(\mathcal{A},\texttt{sp},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\mathcal{B},\mathcal{Y})}$

2. *Subclass:*

   (a) $\frac{(\mathcal{A},\texttt{sc},\mathcal{B})\ \ (\mathcal{B},\texttt{sc},\mathcal{C})}{(\mathcal{A},\texttt{sc},\mathcal{C})}$ *(b)* $\frac{(\mathcal{A},\texttt{sc},\mathcal{B})\ \ (\mathcal{X},\texttt{type},\mathcal{A})}{(\mathcal{X},\texttt{type},\mathcal{B})}$

3. *Typing:*

   (a) $\frac{(\mathcal{A},\texttt{dom},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{X},\texttt{type},\mathcal{B})}$ *(b)* $\frac{(\mathcal{A},\texttt{range},\mathcal{B})\ \ (\mathcal{X},\mathcal{A},\mathcal{Y})}{(\mathcal{Y},\texttt{type},\mathcal{B})}$

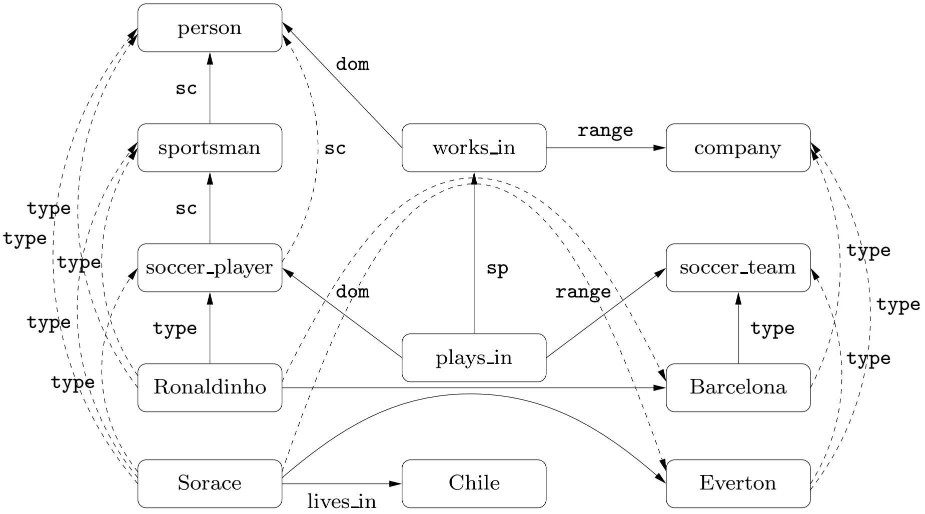

Fig. 2. The closure of the RDF graph in Figure 1

2.2 SPARQL

SPARQL is essentially a graph-matching query language. A SPARQL query is of the form $H \leftarrow B$. The *body* B of the query, is a complex RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts and constraints over the values of the variables. The *head* H of the query, is an expression that indicates how to construct the answer to the query. The evaluation of a query Q against an RDF graph G is done in two steps: the body of Q is matched against G to obtain a set of bindings for the variables in the body, and then using the information on the head of Q, these bindings are processed applying classical relational operators (projection, distinct, etc.) to produce the answer to the query. This answer can have different forms, e.g. a yes/no answer, a table of values, or a new RDF graph. In this paper, we concentrate on the body of SPARQL queries, i.e. in the graph pattern matching facility.

Assume the existence of an infinite set V of variables disjoint from U. A SPARQL graph pattern is defined recursively as follows [20]:

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

A SPARQL *built-in* condition is a Boolean combination of terms constructed by using the equality (=) among elements in $U \cup V$ and constant, and the unary predicate bound($\cdot$) 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 $\mu : V \to U$. Slightly abusing notation, 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. Intuitively, μ_1 and μ_2 are compatibles if μ_1 *can be extended* with μ_2 to obtain a new mapping, and vice versa. Note that two mappings with disjoint domains are always compatible, and that the empty mapping $\mu_\emptyset$ (i.e. the mapping with empty domain) is compatible with any other mapping.

Let Ω_1 and Ω_2 be sets of mappings. We define the join of, the union of and the difference between Ω_1 and Ω_2 as:

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

Based on the previous operators, we define the left outer-join as:

$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).$$

Intuitively, $\Omega_1 \bowtie \Omega_2$ is the set of mappings that result from extending mappings in Ω_1 with their compatible mappings in Ω_2, and $\Omega_1 \smallsetminus \Omega_2$ is the set of mappings in Ω_1 that cannot be extended with any mapping in Ω_2. The operation $\Omega_1 \cup \Omega_2$ is the usual set theoretical union. A mapping μ is in $\Omega_1 ⟕ \Omega_2$ if it is the extension of a mapping of Ω_1 with a compatible mapping of Ω_2, or if it belongs to Ω_1 and cannot be extended with any mapping of Ω_2. These operations resemble relational algebra operations over sets of mappings (partial functions).

We are ready to define the semantics of graph pattern expressions as a function that takes a pattern expression and returns a set of mappings. 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.
- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
- $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$.
- $[\![(P_1 \text{ OPT } P_2)]\!]_G = [\![P_1]\!]_G ⟕ [\![P_2]\!]_G$.

The idea behind the OPT operator is to allow for *optional matching* of patterns. Consider pattern expression $(P_1 \text{ OPT } P_2)$ and let μ_1 be a mapping in $[\![P_1]\!]_G$. If there exists a mapping $\mu_2 \in [\![P_2]\!]_G$ such that μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ belongs to $[\![(P_1 \text{ OPT } P_2)]\!]_G$. But if no such a mapping μ_2 exists, then μ_1 belongs to $[\![(P_1 \text{ OPT } P_2)]\!]_G$. Thus, operator OPT allows information to be added to a mapping μ if the information is available, instead of just rejecting μ whenever some part of the pattern does not match.

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:

- R is bound($?X$) and $?X \in \mathrm{dom}(\mu)$;
- R is $?X = c$, $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) = c$;
- R is $?X =?Y$, $?X \in \mathrm{dom}(\mu)$, $?Y \in \mathrm{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$;
- R is $(\neg R_1)$, R_1 is a built-in condition, and it is not the case that $\mu \models R_1$;
- R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$;
- R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions, $\mu \models R_1$ and $\mu \models R_2$.

Then $[\![(P \text{ FILTER } R)]\!]_G = \{\mu \in [\![P]\!]_G \mid \mu \models R\}$, that is, $[\![(P \text{ FILTER } R)]\!]_G$ is the set of mappings in $[\![P]\!]_G$ that satisfy R.

It was shown in [20], 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.

In the rest of the paper, we usually represent sets of mappings as tables where each row represents a mapping in the set. We label every row with the name of a mapping, and every column with the name of a variable. If a mapping is not defined for some variable, then we simply leave empty the corresponding position. For instance, the table:

	$?X$	$?Y$	$?Z$	$?V$	$?W$
μ_1 :	a	b			
μ_2 :		c			d
μ_3 :			e		

represents the set $\Omega = \{\mu_1, \mu_2, \mu_3\}$, where

- $\mathrm{dom}(\mu_1) = \{?X, ?Y\}$, $\mu_1(?X) = a$ and $\mu_1(?Y) = b$;
- $\mathrm{dom}(\mu_2) = \{?Y, ?W\}$, $\mu_2(?Y) = c$ and $\mu_2(?W) = d$;
- $\mathrm{dom}(\mu_3) = \{?Z\}$ and $\mu_3(?Z) = e$.

We sometimes write $\{\{?X \rightarrow a, ?Y \rightarrow b\}, \{?Y \rightarrow c, ?W \rightarrow d\}, \{?Z \rightarrow e\}\}$ for the above set of mappings.

Example 2. Let G be the RDF graph shown in Figure 1, and consider SPARQL graph pattern $P_1 = ((?X, \text{plays\_in}, ?T) \text{ AND } (?X, \text{lives\_in}, ?C))$. Intuitively, P_1 retrieves the list of soccer players in G, including the teams where they play in and the countries where they live in. Thus, we have:

$[\![P_1]\!]_G =$

$?X$	$?T$	$?C$
Sorace	Everton	Chile

Notice that in this case we have not obtained any information about Ronaldinho, since in the graph there is not data about the country where Ronaldinho lives in. Consider now the pattern $P_2 = ((?X, \text{plays\_in}, ?T) \text{ OPT } (?X, \text{lives\_in}, ?C))$. Intuitively, P_2 retrieves the list of soccer players in G, including the teams where they play in and the countries where they live in. But, as opposed to P_1, pattern

P_2 does not fail if the information about the country where a soccer player lives in is missing. In this case, we have:

$[\![P_2]\!]_G =$

$?X$	$?T$	$?C$
Sorace	Everton	Chile
Ronaldinho	Barcelona	

□

2.3 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 instance, let G be the RDF graph shown in Figure 1, 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 (that is, $[\![P]\!]_G = \emptyset$) 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. The most direct way of defining such a semantics is by considering not the original graph but its closure. The following definition formalizes this notion.

Definition 1 (RDFS evaluation). *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.*

Example 3. Let G be the RDF graph shown in Figure 1, and consider the graph pattern expression:

$$P = ((?X, \texttt{type}, \text{person}) \text{ AND } (?X, \text{lives\_in}, \text{Chile}) \text{ AND } (?X, \text{works\_in}, ?C)),$$

intended to retrieve the list of people in G (resources of type person) that lives in Chile, and the companies where they work in. The evaluation of P over G results in the empty set, since both $[\![(?X, \texttt{type}, \text{person})]\!]_G$ and $[\![(?X, \text{works\_in}, ?C)]\!]_G$ are empty. On the other hand, the RDFS evaluation of P over G contains the following tuples:

$[\![P]\!]_G^{\text{rdfs}} = [\![P]\!]_{\text{cl}(G)} =$

$?X$	$?C$
Sorace	Everton

□

It should be noticed that in Definition 1, we do not provide a procedure for evaluating SPARQL over RDFS. In fact, as we have mentioned before, a direct implementation of this definition leads to an inefficient procedure for evaluating SPARQL queries, as it requires a pre-calculation of the closure of the input graph.

3 Navigational RDF Languages

Our main goal is to define a query language that allows to obtain the RDFS evaluation of a pattern directly from an RDF graph, without computing the entire closure of the graph. We have provided some evidence that a language for navigating RDF graphs could be useful in achieving our goal. In this section, we define such a language for *navigating* RDF graphs, providing a formal syntax and semantics. Our language uses, as usual for graph query languages [17,3], regular expressions to define paths on graph structures, but taking advantage of the special features of RDF graphs. More precisely, we start by introducing in Section 3.2 a language that extends SPARQL with regular expressions. Although regular expressions capture in some cases the semantics of RDFS, we show in Section 3.2 that regular expressions alone are not enough to obtain the RDFS evaluation of some queries. Thus, we show in Section 3.3 how to extend regular expressions by borrowing the notion of *branching* from XPath [8], and we explain why this enriched language is enough for our purposes. Finally, we show in Section 3.4 that the enriched language provides some other interesting features that give extra expressiveness to the language, and that deserve further investigation. But before doing all this, we briefly review in Section 3.1 some of the related work on navigating RDF.

3.1 Related Work

The idea of having a language to navigate through an RDF graph is not new. In fact, several languages have been proposed in the literature [1,4,2,15,5]. Nevertheless, none of these languages is motivated by the necessity to evaluate queries over RDFS, and none of them is comparable in expressiveness with the language proposed in this paper. Kochut et al. [15] propose a language called SPARQLeR as an extension of SPARQL. This language allows to extract semantic associations between RDF resources by considering paths in the input graph. SPARQLeR works with path variables intended to represent a sequence of resources in a path between two nodes in the input graph. A SPARQLeR query can also put restrictions over those paths by checking whether they conform to a regular expression. With the same motivation of extracting semantic associations from RDF graphs, Anyanwu et al. [5] propose a language called SPARQ2L. SPARQ2L extends SPARQL by allowing path variables and path constraints. For example, some SPARQ2L constraints are based on the presence (or absence) of some nodes or edges, the length of the retrieved paths, and on some structural properties of these paths. In [5], the authors also 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 [2]. PSPARQL is an extension of SPARQL obtained by allowing regular expressions in the predicate position of triple patterns. Thus, this language can be used to obtain pair of nodes that are connected by a path whose labeling conforms to a regular expression. PSPARQL also allows variables inside regular expressions, thus permitting to retrieve data *along* the traversed paths. In [2], the authors propose a formal semantics for PSPARQL, and also study some theoretical aspects of this language such as the complexity of query evaluation. VERSA [19] and RxPath [22] are proposals motivated by XPath with emphasis on some implementation issues.

3.2 Navigating RDF through Regular Expressions

Navigating graphs is done usually by using an operator *next*, which allows 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 [13]. 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. The language introduced in this paper deals with this problem by using three different navigation axes, which are shown in Figure 3 (together with their inverses).

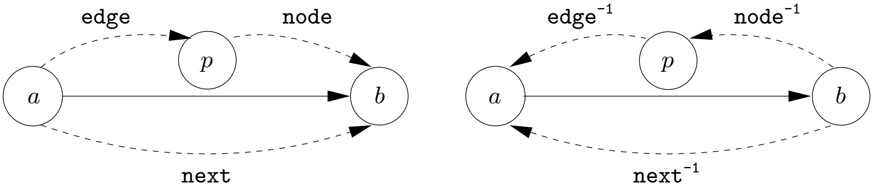

Fig. 3. Forward and backward axes for an RDF triple (a, p, b)

A navigation axis allows moving 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. More precisely, the following grammar defines the regular expressions in our language:

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

where axis $\in$ {`self`, `next`, $\texttt{next}^{-1}$, `edge`, $\texttt{edge}^{-1}$, `node`, $\texttt{node}^{-1}$}. The additional axis `self` is not used to navigate, but instead to test the label of a specific node in a path. We call *regular path expressions* to expressions generated by (1).

Before introducing the formal semantics of regular path 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 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 $(x, y, a) \in G$. We formally define the evaluation of a regular path expression p in a graph G as a binary relation $[\![p]\!]_G$, denoting the pairs of nodes (x, y) such that

Table 2. Formal semantics of regular path expressions

$$\begin{aligned}
\llbracket \texttt{self} \rrbracket_G &= \{(x,x) \mid x \in \mathrm{voc}(G)\} \\
\llbracket \texttt{self}::a \rrbracket_G &= \{(a,a)\}
\end{aligned}$$

$$\begin{aligned}
\llbracket \texttt{next} \rrbracket_G &= \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,z,y) \in G\} \\
\llbracket \texttt{next}::a \rrbracket_G &= \{(x,y) \mid (x,a,y) \in G\} \\
\llbracket \texttt{edge} \rrbracket_G &= \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,y,z) \in G\} \\
\llbracket \texttt{edge}::a \rrbracket_G &= \{(x,y) \mid (x,y,a) \in G\} \\
\llbracket \texttt{node} \rrbracket_G &= \{(x,y) \mid \text{there exists } z \text{ s.t. } (z,x,y) \in G\} \\
\llbracket \texttt{node}::a \rrbracket_G &= \{(x,y) \mid (a,x,y) \in G\}
\end{aligned}$$

$$\begin{aligned}
\llbracket \mathrm{axis}^{\text{-1}} \rrbracket_G &= \{(x,y) \mid (y,x) \in \llbracket \mathrm{axis} \rrbracket_G\} \quad \text{with axis} \in \{\texttt{next}, \texttt{node}, \texttt{edge}\} \\
\llbracket \mathrm{axis}^{\text{-1}}::a \rrbracket_G &= \{(x,y) \mid (y,x) \in \llbracket \mathrm{axis}::a \rrbracket_G\} \quad \text{with axis} \in \{\texttt{next}, \texttt{node}, \texttt{edge}\}
\end{aligned}$$

$$\begin{aligned}
\llbracket exp_1 / exp_2 \rrbracket_G &= \{(x,y) \mid \text{there exists } z \text{ s.t. } (x,z) \in \llbracket exp_1 \rrbracket_G \text{ and } (z,y) \in \llbracket exp_2 \rrbracket_G\} \\
\llbracket exp_1 | exp_2 \rrbracket_G &= \llbracket exp_1 \rrbracket_G \cup \llbracket exp_2 \rrbracket_G \\
\llbracket exp^* \rrbracket_G &= \llbracket \texttt{self} \rrbracket_G \cup \llbracket exp \rrbracket_G \cup \llbracket exp/exp \rrbracket_G \cup \llbracket exp/exp/exp \rrbracket_G \cup \cdots
\end{aligned}$$

y is reachable from x in G by following a path whose labels are in the language defined by p. The formal semantics of the language is shown in Table 2. In this table, G is an RDF graph, $a \in U$, voc(G) is the set of all the elements from U that are mentioned in G, and exp, exp_1, exp_2 are regular path expressions.

Example 4. Consider an RDF graph G storing information about transportation services between cities. A triple (C_1, tc, C_2) in the graph indicates that there is a direct way of traveling from C_1 to C_2 by using the transportation company tc.

If we assume that G does not mention any of the RDFS keywords, then the expression:

$$(\texttt{next}::\mathrm{KoreanAir})^+ \mid (\texttt{next}::\mathrm{AirFrance})^+$$

defines the pairs of cities (C_1, C_2) in G such that there is a way of flying from C_1 to C_2 in either KoreanAir or AirFrance. Moreover, by using axis `self`, we can test for a stop in a specific city. For example, the expression:

$$(\texttt{next}::\mathrm{KoreanAir})^+/\texttt{self}::\mathrm{Paris}/(\texttt{next}::\mathrm{KoreanAir})^+$$

defines the pairs of cities (C_1, C_2) such that there is a way of flying from C_1 to C_2 with KoreanAir with a stop in Paris. □

Once regular path expressions have been defined, the natural next step is to extend the syntax of SPARQL to allow them in triple patterns. A *regular path triple* is a tuple of the form $t = (x, exp, y)$, where $x, y \in U \cup V$ and exp is a regular path expression. Then the evaluation of a regular path triple $t = (?X, exp, ?Y)$ over an RDF graph G is defined as the following set of mappings:

$$\llbracket t \rrbracket_G = \{\mu \mid \mathrm{dom}(\mu) = \{?X, ?Y\} \text{ and } (\mu(?X), \mu(?Y)) \in \llbracket exp \rrbracket_G\}.$$

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

We call *regular* SPARQL (or just rSPARQL) to SPARQL extended with regular path triples. The semantics of rSPARQL patterns is defined recursively as in Section 2, but considering the special semantics of regular path triples. The following example shows that rSPARQL is useful to represent RDFS deductions.

Example 5. Let G be the RDF graph in Figure 1, 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 Figure 2, we obtain that:

$$[\![(\text{Ronaldinho}, \texttt{type}, ?C)]\!]_G^{\text{rdfs}} =$$

$?C$
soccer_player
sportsman
person

However, if we directly evaluate this pattern over G we obtain a single mapping:

$$[\![(\text{Ronaldinho}, \texttt{type}, ?C)]\!]_G =$$

$?C$
soccer_player

Consider now the rSPARQL pattern:

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

The regular path expression $\texttt{next::type}/(\texttt{next::sc})^*$ is intended to obtain the pairs of nodes such that, there is a path between them that has `type` as its first label 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:

$$[\![P]\!]_G =$$

$?C$
soccer_player
sportsman
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 $(\text{Everton}, \texttt{next}::\texttt{type}/(\texttt{next}::\texttt{sc})^*, ?C)$ over G, we obtain the empty set. Consider now the rSPARQL pattern

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

With the expression $\texttt{node}^{\texttt{-1}}/(\texttt{next}::\texttt{sp})^*/\texttt{next}::\texttt{range}$, we follow a path that first navigates from a node to one of its incoming edges by using $\texttt{node}^{\texttt{-1}}$, and then continues with zero or more $\texttt{sp}$ edges and a final $\texttt{range}$ edge. The evaluation of this expression in 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:

$[\![Q]\!]_G =$

$?C$
soccer_team
company

By looking at the closure of G in Figure 2, we see that pattern Q obtains exactly the type information of Everton in G, that is, $[\![(\text{Everton}, \texttt{type}, ?C)]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$.
□

The previous example shows the benefits of having regular path expressions to obtain the RDFS evaluation of a pattern P over an RDF graph G just by navigating G. We are interested in whether this can be done in general for every SPARQL pattern. More formally, we are interested in the following problem:

> Given a SPARQL pattern P, is there an rSPARQL pattern Q such that for every RDF graph G, it holds that
>
> $$[\![P]\!]_G^{\text{rdfs}} = [\![Q]\!]_G?$$

Unfortunately, the answer to this question is negative for some SPARQL patterns. Let us show this failure with an example. Assume that we want to obtain the RDFS evaluation of pattern $P = (?\text{X}, \text{works\_in}, ?Y)$ in 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 $\texttt{sp}$ 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 path expression:

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

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. Thus, there is no way to know what was the initial edge. In fact, if we evaluate the pattern $Q = (?X, \texttt{edge}/(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{works\_in}/(\texttt{next}^{\texttt{-1}}::\texttt{sp})^*/\texttt{node}, ?Y)$ over the graph G in Figure 1, we obtain the set of mappings:

$[\![Q]\!]_G =$

$?X$	$?Y$
Ronaldinho	Barcelona
Ronaldinho	Everton
Sorace	Barcelona
Sorace	Everton

By simply inspecting the closure of G in Figure 2, we obtain that:

$[\![P]\!]_G^{\text{rdfs}} =$

$?X$	$?Y$
Ronaldinho	Barcelona
Sorace	Everton

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

In general, it can be shown that there is no rSPARQL triple pattern Q such that for every RDF graph G, it holds that $[\![(?X, \text{works\_in}, ?Y)]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$. It is worth mentioning that this failure persists for a general rSPARQL pattern Q, that is, if Q is allowed to use all the expressive power of SPARQL patterns (it can use operators AND, UNION, OPT and FILTER) plus regular path expressions in triple patterns.

3.3 Navigating RDF through Nested Regular Expressions

We have seen that regular path expressions are not enough to obtain the RDFS evaluation of a graph pattern. In this section, we introduce a language that extends regular path expressions with a *nesting operator*. Nested expressions can be used to test for the existence of certain paths starting at any axis of a regular path expression. We will see that this feature is crucial in obtaining the RDFS evaluation of SPARQL patterns by directly traversing RDF graphs.

The syntax of nested regular expressions is defined by the following grammar:

$$exp \;:=\; \text{axis} \;\mid\; \text{axis}::a \; (a \in U) \;\mid\; \text{axis}::[exp] \;\mid\; exp/exp \;\mid\; exp|exp \;\mid\; exp^* \qquad (2)$$

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

The nesting construction $[exp]$ is used to check for the existence of a path defined by expression exp. For instance, when evaluating nested expression $\texttt{next}::[exp]$ in a graph G, we retrieve the pair 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 formal semantics of nested regular path expressions is shown in Table 3. The semantics for the navigation axes of the form 'axis' and 'axis::a', as well as the concatenation, disjunction, and star closure of expressions, is defined as for the case of regular path expressions (see Table 2).

Table 3. Formal semantics of nested regular path expressions

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

Example 6. Consider an RDF graph G storing information about transportation services between cities. As in Example 4, a triple (C_1, tc, C_2) in the graph indicates that there is a direct way of traveling from C_1 to C_2 by using the transportation company tc. Then the nested expression:

$$(\texttt{next}::\text{KoreanAir})^+/\texttt{self}::[(\texttt{next}::\text{AirFrance})^*/\texttt{self}::\text{Paris}]/(\texttt{next}::\text{KoreanAir})^+,$$

defines the pairs of cities (C_1, C_2) such that, there is a way of flying from C_1 to C_2 with KoreanAir with a stop in a city C_3 from which one can fly to Paris with AirFrance. Notice that $\texttt{self}::[(\texttt{next}::\text{AirFrance})^*/\texttt{self}::\text{Paris}]$ is used to test for the existence of a flight (that can have some stops) from C_3 to Paris with AirFrance. □

Recall that rSPARQL was defined as the extension of SPARQL with regular path expressions in the predicate position of triple patterns. Similarly, *nested* SPARQL (or just nSPARQL) is defined as the extension of SPARQL with nested regular expressions in the predicate position of triple patterns. The following example shows the benefits of using nSPARQL when trying to obtain the RDFS evaluation of a pattern by directly traversing an RDF graph.

Example 7. Consider the SPARQL pattern $P = (?X, \text{works\_in}, ?Y)$. We have seen that it is not possible to obtain the RDFS evaluation of P with an rSPARQL pattern. Consider now the nested regular expression:

$$\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{works\_in}]. \tag{3}$$

It defines the pairs (a, b) of resources in an RDF graph G such that, there exist a triple (a, x, b) and a path from x to works_in in G where every edge has label sp. The expression $(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{works\_in}$ is used to simulate the inference process in RDFS; it retrieves all the nodes that are *sub-properties* of works_in. Thus, expression (3) is exactly what we need to obtain the RDFS evaluation of pattern P. In fact, if G is the RDF graph in Figure 1 and Q the nSPARQL pattern:

$$Q = (?X,\ \texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{works\_in}],\ ?Y),$$

then we obtain

$[\![Q]\!]_G =$

$?X$	$?Y$
Ronaldinho	Barcelona
Sorace	Everton

This is exactly the RDFS evaluation of P in G, that is, $[\![P]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$. □

It turns out that nested expressions are the necessary ingredient to obtain the RDFS evaluation of SPARQL patterns by navigating RDF graphs. To show that this holds, consider the following *translation* function from elements in U to nested expressions:

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

By using the results of [18], it can be shown that for every SPARQL triple pattern of the form (x, a, y), where $x, y \in U \cup V$ and $a \in U$, it holds that:

$$[\![(x, a, y)]\!]_G^{\text{rdfs}} = [\![(x, \text{trans}(a), y)]\!]_G$$

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 (and without explicitly computing the closure of G).

Given that the syntax and semantics of SPARQL patterns are defined from triple patterns, the previous property also holds for SPARQL patterns including operators AND, OPT, UNION and FILTER. That is, if P is a SPARQL pattern constructed by using triple patterns from the set $(U \cup V) \times U \times (U \cup V)$, then there is an nSPARQL pattern Q such that for every RDF graph G, it holds that $[\![P]\!]_G^{\text{rdfs}} = [\![Q]\!]_G$.

It should be noticed that, if variables are allowed in the predicate position of triple patterns, in general there is no hope to obtain the RDFS evaluation without computing the closure, since a triple pattern like $(?X, ?Y, ?Z)$ can be used to retrieve the entire closure of an RDF graph.

3.4 The Extra Expressive Power of Nested Regular Expressions

Nested regular expressions were designed to be expressive enough to capture the semantics of RDFS. Beside this feature, nested regular expressions also provide some other interesting features that give extra expressiveness to the language. With nested regular expressions, one is allowed to define complex paths by using concatenation, disjunction and star closure, over nested expressions. It is also allowed to use various levels of nesting in expressions. Note that these features are not needed in the translations presented in the previous section.

The following example shows that the extra expressiveness of nested regular expressions can be used to formulate interesting and natural queries, which cannot be expressed by using regular path expressions.

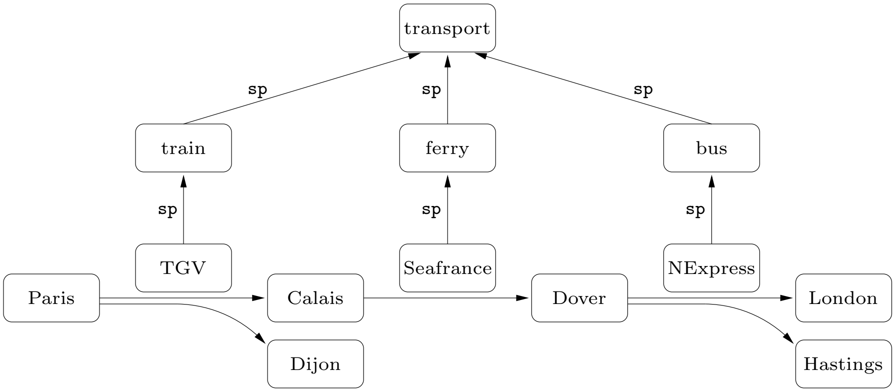

Fig. 4. An RDF graph storing information about transportation services between cities

Example 8. Consider the RDF graph with transportation information in Figure 4. As in the previous examples, if C_1 and C_2 are cities and (C_1, tc, C_2) is a triple in the graph, then there is a direct way of traveling from C_1 to C_2 by using the transportation company tc. For instance, (Paris, TGV, Calais) indicates that TGV provides a transportation service from Paris to Calais. In the figure, we also have extra information about the travel services. For example, TGV is a sub-property of train and then, if (Paris, TGV, Calais) is in the graph, we can infer that there is a train going from Paris to Calais.

If we want to know whether there is a way to travel from one city to another (without taking into consideration the kind of transportation), we can use the following expression:

$$(\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+.$$

Assume now that we want to obtain the pairs (C_1, C_2) of cities such that there is a way to travel from C_1 to C_2 with a stop in a city which is either London or is connected by a bus service with London. First, notice that the following nested expression checks whether there is a way to travel from C_1 to C_2 with a stop in London:

$$(\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+/\texttt{self}::\text{London}/ \\ (\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+. \quad (4)$$

Thus, to obtain an expression for our initial query, we only need to replace $\texttt{self}::\text{London}$ in (4) by an expression that checks whether a city is either London or is connected by a bus service with London. The following expression can be used to test the latter condition:

$$(\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{bus}])^*/\texttt{self}::\text{London}. \quad (5)$$

Hence, by replacing $\texttt{self}::\text{London}$ by (5) in nested regular expression (4), we obtain a nested regular expression for our initial query:

$$\begin{multline}(\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+/\\ \texttt{self}::[(\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{bus}])^*/\texttt{self}::\text{London}]\ /\\ (\texttt{next}::[(\texttt{next}::\texttt{sp})^*/\texttt{self}::\text{transport}])^+. \quad (6)\end{multline}$$

Notice that the level of nesting of (6) is 2. If we evaluate (6) over the RDF graph in Figure 4, we obtain the pair (Calais, Hastings) as a possible answer since there is a way to travel from Calais to Hastings with a stop in Dover, from which there is a bus service to London. □

4 Concluding Remarks

The problem of answering queries over RDFS is challenging, due to the existence of a vocabulary with a predefined semantics. Current approaches for this problem pre-compute the closure of RDF graphs. From a practical point of view, these approaches have several drawbacks, among others that they are not goal-oriented: although a query may need to scan a small part of the data, all the data is considered when computing the closure of an RDF graph.

In this paper, we propose an alternative approach to the problem of answering RDFS queries. We present a navigational language constructed from *nested regular expressions*, that can be used to obtain the answer to RDFS queries by navigating the input graph (without pre-computing the closure). Besides capturing the semantics of RDFS, nested regular expressions also provide some other interesting features that give extra expressiveness to the language. We think these features deserve further and deeper investigation.

Acknowledgments. 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. Alkhateeb, F., Baget, J., Euzenat, J.: Complex path queries for RDF. Poster paper in ISWC 2005 (2005)
2. Alkhateeb, F., Baget, J., Euzenat, J.: RDF with regular expressions. Research Report 6191, INRIA (2007)
3. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1), 1–39 (2008)
4. Anyanwu, K., Maduko, A., Sheth, A.: SemRank: ranking complex relationship search results on the semantic web. In: WWW 2005, pp. 117–127 (2005)
5. Anyanwu, K., Maduko, A., Sheth, A.: SPARQ2L: Towards Support for Subgraph Extraction Queries in RDF Databases. In: WWW 2007, pp. 797–806 (2007)

6. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (Feburary 2004), `http://www.w3.org/TR/rdf-schema/`
7. 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)
8. Clark, J., DeRose, S.: XML Path Language (XPath). W3C Recommendation (November 1999), `http://www.w3.org/TR/xpath`
9. Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: RDF Querying: Language Constructs and Evaluation Methods Compared. In: Barahona, P., Bry, F., Franconi, E., Henze, N., Sattler, U. (eds.) Reasoning Web 2006. LNCS, vol. 4126, pp. 1–52. Springer, Heidelberg (2006)
10. Gutierrez, C., Hurtado, C., Mendelzon, A.: Foundations of Semantic Web Databases. In: PODS 2004 (2004)
11. Harris, S., Gibbins, N.: 3store: Efficient bulk RDF storage. In: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS 2003), Sanibel Island, Florida, pp. 1–15 (2003)
12. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A Comparison of RDF Query Languages. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 502–517. Springer, Heidelberg (2004)
13. Hayes, J., Gutierrez, C.: Bipartite Graphs as Intermediate Model for RDF. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 47–61. Springer, Heidelberg (2004)
14. Hayes, P.: RDF Semantics. W3C Recommendation (February 2004), `http://www.w3.org/TR/rdf-mt/`
15. 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)
16. Manola, F., Miller, E., McBride, B.: RDF Primer, W3C Recommendation (February 10, 2004), `http://www.w3.org/TR/REC-rdf-syntax/`
17. Mendelzon, A., Wood, P.: Finding Regular Simple Paths in Graph Databases. SIAM J. Comput. 24(6), 1235–1258 (1995)
18. 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)
19. Olson, M., Ogbuji, U.: The Versa Specification, `http://uche.ogbuji.net/tech/rdf/versa/etc/versa-1.0.xml`
20. 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)
21. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Working Draft (March 2007), `http://www.w3.org/TR/rdf-sparql-query/`
22. Souzis, A.: RxPath Specification Proposal, `http://rx4rdf.liminalzone.org/RxPathSpec`