# Constrained regular expressions for answering RDF-path queries modulo RDFS

Faisal Alkhateeb, Jérôme Euzenat

# Constrained regular expressions for answering RDF-path queries modulo RDFS

Faisal Alkhateeb [a],* and Jérôme Euzenat [b]

[a] *Faculty of IT, Yarmouk University, Irbid, Jordan. E-mail: alkhateebf@yu.edu.jo*

[b] *INRIA and LIG, Montbonnot, France. E-mail: Jerome.Euzenat@inria.fr*

**Abstract.** The standard SPARQL query language is currently defined for querying RDF graphs without RDFS semantics. Several extensions of SPARQL to RDFS semantics have been proposed. In this paper, we discuss extensions of SPARQL that use regular expressions to navigate RDF graphs and may be used to answer queries considering RDFS semantics. In particular, we present and compare nSPARQL and our proposal CPSPARQL. We show that CPSPARQL is expressive enough to answer full SPARQL queries modulo RDFS. Finally, we compare the expressiveness and complexity of both nSPARQL and the corresponding fragment of CPSPARQL, that we call cpSPARQL. We show that both languages have the same complexity through cpSPARQL, being a proper extension of SPARQL graph patterns, is more expressive than nSPARQL.

Keywords: semantic web, query language, RDF, RDFS, SPARQL, regular expression, constrained regular expression, nSPARQL, CPSPARQL, cpSPARQL

## 1. Introduction

RDF (Resource Description Framework [21]) is a knowledge representation language dedicated to the description of documents and more generally of resources within the semantic web.

SPARQL [30] is the standard language for querying RDF data. It has been well-designed for that purpose, but very often, RDF data is expressed in the framework of a schema or an ontology in RDF Schema or OWL. RDF Schema (or RDFS) [10] together with OWL [22] are two ontology languages recommended by the W3C for defining the vocabulary used in RDF graphs. Recently, [15] presented extensions of the SPARQL 1.1 entailment regimes to incorporate RDFS and OWL semantics. Extending SPARQL for dealing with this kind of data is thus a major issue. We consider here the case of RDF Schema (RDFS) or rather a large fragment of RDF Schema [24].

Two main approaches can be developed for answering a SPARQL query $Q$ modulo an RDF schema $S$ against an RDF graph $G$, i.e., evaluating the queries by interpreting the queried graph and the query under the RDFS semantics: the eager approach transforms the data so that the evaluation of the SPARQL query $Q$ against the transformed RDF graph $\tau(G)$ returns the answer, while the lazy approach transforms the query so that the transformed query $\tau(Q)$ against the RDF graph $G$ returns the answers. The approaches are not exclusive, as shown by [26], though no hybrid approach has been developed so far for SPARQL.

There already have been proposals along the second approach. For instance, [28] provides a query language, called nSPARQL, allowing for navigating graphs in the style of XPath. Then queries are rewritten so that query evaluation navigates the data graph for taking the RDF Schema into account. Other attempts, such as SPARQ2L [6] and SPARQLeR [20] are not known to address queries with respect to RDF Schema. SPARQL-DL [31] addresses OWL but is restricted with respect to SPARQL.

An independently developed extension of SPARQL, called PSPARQL is proposed in [5], which adds path expressions to SPARQL. It is shown in [4] that answering SPARQL queries modulo RDF Schema could

*Corresponding author.

be achieved by transforming them into PSPARQL queries. PSPARQL fully preserves SPARQL, i.e., any SPARQL query is a valid PSPARQL query. The time complexity of PSPARQL (i.e., the complexity of evaluating a PSPARQL query against an RDF graph) is the same as that of SPARQL [2]. Nonetheless, the transformation cannot be generally applied to PSPARQL and thus it is not generally sufficient for answering PSPARQL queries modulo RDFS [4].

To overcome this limitation, an extension of PSPARQL called CPSPARQL is proposed in [3,4], that uses constrained regular expressions instead of regular expressions.

In this paper, we show that cpSPARQL, a restriction of CPSPARQL, can express all nSPARQL queries with the same complexity. The advantage of using CPSPARQL is that, contrary to nSPARQL, it is a strict extension of SPARQL and cpSPARQL graph patterns are a strict extension of SPARQL graph patterns as well as a strict extension of PSPARQL graph patterns. Hence, using a proper extension of SPARQL like CPSPARQL is preferable to a restricted path-based languages. In particular, this allows for implementing the SPARQL RDFS entailment regime.

In order to compare cpSPARQL and nSPARQL, we adopt in this paper a notation similar to nSPARQL, i.e., adding XPath axes, which is slightly different from the original CPSPARQL syntax presented in [3,4]. After presenting the syntax and semantics of both nSPARQL and CPSPARQL, we show that:

- CPSPARQL can answer full SPARQL queries modulo RDFS (Section 4.3);
- We offer an efficient algorithm for answering cpSPARQL queries (Section 5);
- cpSPARQL has the same complexity as nSPARQL (Section 5);
- Any nSPARQL triple pattern can be expressed as a cpSPARQL triple pattern, but not vice versa (Section 6).

**Paper Outline**. The remainder of the paper is organized as follows. In Section 2, we introduce RDF and the SPARQL language. Section 3 is dedicated to the presentation of the nSPARQL query language. The CPSPARQL and cpSPARQL languages are presented in detail with their main results in Section 4 and we show how to use them for answering SPARQL and CPSPARQL queries modulo RDF Schemas. The complexity results are presented in Section 5. In Section 6, we compare the expressiveness of cpSPARQL and nSPARQL. We discuss more precisely other related work in Section 8. Finally, we conclude in Section 9.

## 2. Preliminaries

In this section, we present RDF as well as its recommended query language SPARQL.

### 2.1. RDF

We introduce below the syntax and the semantics of (simple semantics as introduced in [18]) of the language.

#### 2.1.1. RDF syntax

RDF graphs are constructed over the set of URI references (or urirefs), blanks, and literals [12]. To simplify notations, and without loss of generality, we do not distinguish here between simple and typed literals.
**Terminology.** An *RDF terminology*, noted $\mathcal{T}$, is the union of 3 pairwise disjoint infinite sets of *terms*: the set $\mathcal{U}$ of *urirefs*, the set $\mathcal{L}$ of *literals* and the set $\mathcal{B}$ of *variables*. The *vocabulary* $\mathcal{V}$ denotes the set of *names*, i.e., $\mathcal{V} = \mathcal{U} \cup \mathcal{L}$. We use the following notations for the elements of these sets: a variable will be prefixed by `?` (like `?x1`), a literal will be expressed between quotation marks (like `"27"`), remaining elements will be urirefs (like `price`).

Basically, an RDF graph is a set of triples of the form $\langle subject, predicate, object\rangle$ whose domain is defined in the following definition. We generalize RDF graphs by allowing variables (blanks) in predicate position.

**Definition 1** (RDF graph, GRDF graph)**.** *An* RDF triple *is an element of* $\mathcal{U} \times \mathcal{U} \times \mathcal{U}$. *An* RDF graph *is a set of RDF triples. A* GRDF graph *(for generalized RDF) is a set of triples of* $(\mathcal{U} \cup \mathcal{B}) \times (\mathcal{U} \cup \mathcal{B}) \times \mathcal{T}$.

In this definition we do not consider blank nodes in RDF because the official specification of SPARQL treats blank nodes in RDF graphs simply as constants (as if they were URIs) without considering their existential semantics. However, if the existential semantics of blank nodes is considered when querying RDF, the results of this paper may indirectly apply by using the graph homomorphism technique [9].

If $G$ is an RDF graph, we use $voc(G)$ to denote the set of terms appearing in at least one triple of $G$.

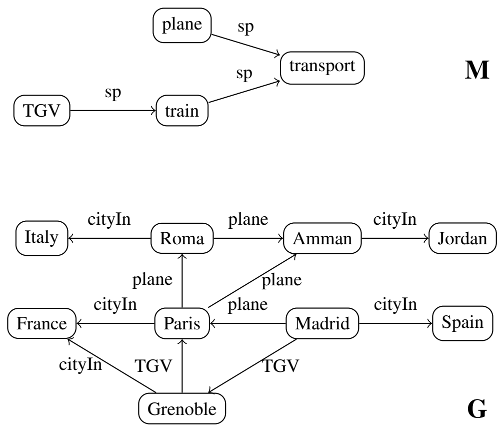

Fig. 1. An RDF graph ($G$) with its schema ($M$) representing information about transportation means between several cities.

**Example 1** (RDF Graph)**.** *RDF can be used for representing information about cities, transportation means between cities, and relationships between the transportation means. The following triples are part of the RDF graph of Figure 1:*

```
Grenoble TGV Paris .
Paris plane Amman .
TGV subPropertyOf transport .
...
```

*For instance, a triple* ⟨`Paris`, `plane`, `Amman`⟩ *means that there exists a transportation mean* `plane` *from* `Paris` *to* `Amman`*.*

*2.1.2. RDF semantics*

The formal semantics of RDF expresses the conditions under which an RDF graph describes a particular world, i.e., an interpretation is a model for the graph [18]. We define in the following simple semantics without considering RDFS semantics.

**Definition 2** (Mapping)**.** *Let $V_1 \subseteq \mathcal{T}$, and $V_2 \subseteq \mathcal{T}$ be two sets of terms. A map from $V_1$ to $V_2$ is a function $\mu : V_1 \to V_2$ such that $\forall x \in (V_1 \cap \mathcal{V})$, $\mu(x) = x$.*

In the following definition, we provide a characterization of the entailment of RDF graphs (respectively, GRDF graphs) in terms of subset for the case of graphs without variables (respectively, in terms of homomorphism when the graphs have variables).

**Definition 3** (RDF, GRDF entailment)**.** *An RDF graph $G$* RDF-entails *an RDF graph $P$ (denoted by $G \models_{rdf} P$) iff $\forall\langle s,p,o\rangle \in P$, then $\langle s,p,o\rangle \in G$. An RDF graph $G$* RDF-entails *a GRDF graph $P$ (denoted by $G \models_{rdf} P$) if there exists a mapping $\mu : \mathcal{T}(P) \to \mathcal{T}(G)$ such that iff $\forall\langle s,p,o\rangle \in P$, then $\langle\mu(s),\mu(p),\mu(o)\rangle \in G$.*

## 2.2. SPARQL

We define in the following subsections the syntax and the semantics of SPARQL.

*2.2.1. SPARQL syntax*

The basic building blocks of SPARQL queries are *graph patterns* which are shared by all SPARQL query forms. Informally, a *graph pattern* can be a triple pattern, i.e., a GRDF triple, a basic graph pattern, i.e., a set of triple patterns such as a GRDF graph in SPARQL (AND), the union of graph patterns (UNION), an optional graph pattern (OPT), or a constraint (FILTER) (cf. [30] for more details).

**Definition 4** (SPARQL graph pattern)**.** *A* SPARQL graph pattern *is defined inductively in the following way:*

- *every GRDF graph is a SPARQL graph pattern;*
- *if $P$, $P'$ are SPARQL graph patterns and $K$ is a SPARQL constraint, then ($P$* `AND` *$P'$), ($P$* `UNION` *$P'$), ($P$* `OPT` *$P'$), and ($P$* `FILTER` *$K$) are SPARQL graph patterns.*

A SPARQL constraint $K$ is a boolean expression involving terms from $(\mathcal{V} \cup \mathcal{B})$, e.g., a numeric test. We do not specify these expressions further.

A SPARQL SELECT query is of the form `SELECT` $\vec{B}$ `FROM` $u$ `WHERE` $P$ where $u$ is the URI of an RDF graph $G$, $P$ is a SPARQL graph pattern and $\vec{B}$ is a tuple of variables appearing in $P$. Intuitively, such a query asks for the assignments of the variables in $\vec{B}$ such that, under these assignments, $P$ is entailed by the graph identified by $u$.

**Example 2** (Query)**.** *The following query searches in the RDF graph of Figure 1 if there exists a direct plane between a city in France and a city in Jordan:*

```
SELECT ?city1 ?city2
FROM ...
WHERE
  ?city1 plane ?city2 .
  ?city1 cityIn France .
  ?city2 cityIn Jordan .
```

*2.2.2. SPARQL semantics*

In the following, we characterize query answering with SPARQL as done in [27]. The approach relies upon the correspondence between maps from RDF graph of the query graph patterns to the RDF knowledge base and GRDF entailment.

**Operations on mappings.** If $\mu$ is a map, then the domain of $\mu$, denoted by $dom(\mu)$, is the subset of $\mathcal{T}$ on which $\mu$ is defined. The restriction of $\mu$ to a set of terms

$X$ is defined by $\mu|_X = \{\langle x, y\rangle \in \mu| \; x \in X\}$ and the completion of $\mu$ to a set of terms $X$ is defined by $\mu|^X = \mu \cup \{\langle x, \texttt{null}\rangle | \; x \in X$ and $x \notin dom(\mu)\}$[1].

If $P$ is a graph pattern, then we use $\mathcal{B}(P)$ to denote the set of variables occurring in $P$ and $\mu(P)$ to denote the graph pattern obtained by the substitution of $\mu(b)$ to each variable $b \in \mathcal{B}(P)$. Two mappings $\mu_1$ and $\mu_2$ are *compatible* when $\forall x \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(x) = \mu_2(x)$. Otherwise, they are said to be incompatible and this is denoted by $\mu_1 \perp \mu_2$. If $\mu_1$ and $\mu_2$ are two compatible mappings, then we denote by $\mu = \mu_1 \oplus \mu_2 : T_1 \cup T_2 \to \mathcal{T}$ the mapping defined by: $\forall x \in T_1, \mu(x) = \mu_1(x)$ and $\forall x \in T_2, \mu(x) = \mu_2(x)$. The *join* and *difference* of two sets of mappings $\Omega_1$ and $\Omega_2$ are defined as follows [27]:

- *(join)* $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \oplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible$\}$;
- *(difference)* $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1$ and $\mu_2$ are not compatible$\}$.

The answers to a basic graph pattern query are those mappings which warrant the entailment of the graph pattern by the queried graph. In the case of SPARQL, this entailment relation is GRDF entailment. Answers to compound graph patterns are obtained through the operations on mappings.

**Definition 5** (Answers to compound graph patterns)**.** *Let $\models_{rdf}$ be the RDF entailment relation on basic graph patterns, P, P′ be SPARQL graph patterns, K be a SPARQL constraint, and G be an RDF graph. The set $\mathcal{S}(P, G)$ of* answers *to P in G is defined inductively in the following way:*

$$\begin{aligned} \mathcal{S}(P, G) &= \{\mu|_{\mathcal{B}(P)} | \; G \models_{rdf} \mu(P)\} \\ & \qquad \textit{if P is a basic graph pattern} \\ \mathcal{S}((P \texttt{ AND } P'), G) &= \mathcal{S}(P, G) \bowtie \mathcal{S}(P', G) \\ \mathcal{S}(P \texttt{ UNION } P', G) &= \mathcal{S}(P, G) \cup \mathcal{S}(P', G) \\ \mathcal{S}(P \texttt{ OPT } P', G) &= (\mathcal{S}(P, G) \bowtie \mathcal{S}(P', G)) \\ & \quad \cup (\mathcal{S}(P, G) \setminus \mathcal{S}(P', G)) \\ \mathcal{S}(P \texttt{ FILTER } K, G) &= \{\mu \in \mathcal{S}(P, G) \mid \mu(K) = \top\} \end{aligned}$$

Note that the operator $\models_{rdf}$ is used to denote the RDF entailment relation on basic graph patterns and we will use simply $\models$ when it is clear from the context. Moreover, the conditions $K$ are interpreted as boolean functions from the terms they involve. Hence, $\mu(K) = \top$ means that this function is evaluated to true once the variables in $K$ are substituted by $\mu$. If not all variables of $K$ are bound, then $\mu(K) \neq \top$. One particular operator that can be used in SPARQL filter conditions is "$bound(?x)$". This operator returns true if the variable $?x$ is bound and in this case $\mu(K)$ is not true whenever a variable is not bound.

As usual for this kind of query language, an answer to a query is an assignment of the distinguished variables (those variables in the `SELECT` part of the query). Such an assignment is a mapping from variables in the query to nodes of the graph. The defined answers may assign only one part of the variables, those sufficient to prove entailment. The answers are these assignments extended to all distinguished variables.

**Definition 6** (Answers to a SPARQL query)**.** *Let* $\texttt{SELECT } \vec{B} \texttt{ FROM } u \texttt{ WHERE } P$ *be a SPARQL query, G be the RDF graph identified by the URI u, and $\mathcal{S}(P, G)$ be the set of answers to P in G, then the* answers *$\mathcal{A}(\vec{B}, G, P)$ to the query are the restriction and completion to $\vec{B}$ of answers to P in G, i.e.,*

$$\mathcal{A}(\vec{B}, G, P) = \{\mu|^{\vec{B}}_{\vec{B}}; \; \mu \in \mathcal{S}(P, G)\}.$$

## 3. nSPARQL

nSPARQL is a query language that uses nested regular expressions in predicate position of graph patterns for navigating the RDF graph [28].

### 3.1. nSPARQL syntax

**Definition 7** (Regular expression)**.** *A* regular expression *is an expression built from the following grammar:*

*exp ::= axis | axis::a | exp | exp/exp | exp|exp | exp**

*with a $\in \mathcal{U}$ and axis $\in$\{*`self`*,* `next`*,* $\texttt{next}^{-1}$*,* `edge`*,* $\texttt{edge}^{-1}$*,* `node`*,* $\texttt{node}^{-1}$ *\}.*

Regarding the precedence among the regular expression operators, it is as follows: *, /, then |. Parentheses may be used for breaking precedence rules.

The model underlying nSPARQL is that of XPath which navigates within XML structures. Hence, the axis denotes the type of node object which is selected at each step, respectively, the current node (`self` or

[1]The `null` symbol is used for denoting the NULL values introduced by the OPTIONAL clause.

`self`$^{-1}$), the nodes reachable through an outbound triple (`next`), the nodes that can reach the current node through an incident triple (`next`$^{-1}$), the properties of outbound triples (`edge`), the properties of incident triples (`edge`$^{-1}$), the object of a predicate (`node`) and the predicate of an object (`node`$^{-1}$). This is illustrated by Figure 2.

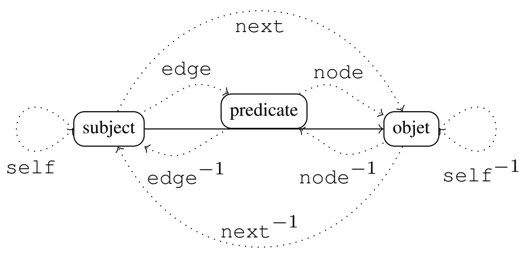

Fig. 2. nSPARQL axes.

**Definition 8** (Nested regular expression)**.** *A* nested regular expression *is an expression built from the following grammar (with $a \in \mathcal{U}$):*

*exp ::= axis | axis::a | axis::[exp] | exp | exp/exp | exp|exp | exp**

Contrary to simple regular expressions, nested regular expressions may constrain nodes to satisfy additional secondary paths.

Nested regular expressions are used in triple patterns in predicate position, to define nSPARQL triple patterns.

**Definition 9** (nSPARQL triple pattern)**.** *An* nSPARQL triple pattern *is a triple $\langle s, p, o\rangle$ such that $s \in \mathcal{T}$, $o \in \mathcal{T}$ and $p$ is a nested regular expression.*

**Example 3** (nSPARQL triple pattern)**.** *Assume that one wants to retrieve the pairs of cities such that there is a way of traveling by any transportation mean. The following nSPARQL pattern expresses this query:*

$P = \langle ?city_1, (next :: [(next :: sp)^*/self :: transport])^+, ?city_2\rangle$

*This pattern expresses a sequence of predicates reaching the "transport" predicate through "subproperty" (sp) predicates.*

nSPARQL is designed as a navigational language, i.e., its main purpose is to find nodes linked by a particular path.

It is also possible to create a query language from nSPARQL triple patterns by simply replacing SPARQL patterns by nSPARQL patterns. Indeed, from nSPARQL triple patterns it is possible to define nSPARQL graph patterns in the usual way.

**Definition 10** (nSPARQL graph pattern)**.** *An* nSPARQL graph pattern *is defined inductively by:*

- *every nSPARQL triple pattern is an nSPARQL graph pattern;*
- *if $P_1$ and $P_2$ are two nSPARQL graph patterns and $K$ is a SPARQL constraint, then ($P_1$* `AND` *$P_2$), ($P_1$* `UNION` *$P_2$), ($P_1$* `OPT` *$P_2$), and ($P_1$* `FILTER` *$K$) are nSPARQL graph patterns.*

However, for time complexity reasons the designers of the nSPARQL language choose to define a more restricted language than SPARQL [29]. Contrary to SPARQL queries, nSPARQL queries are reduced to nSPARQL graph patterns, constructed from nSPARQL triple patterns, plus SPARQL operators AND, UNION, FILTER, and OPT. They do not allow for the projection operator SELECT. This prevents, when checking answers, that uncontrolled variables have to be evaluated.

### 3.2. nSPARQL semantics

In order to define the semantics of nSPARQL, we need to know the semantics of nested regular expressions [28].

**Definition 11** (Nested path interpretation [28])**.** *Given a nested path $p$ and an RDF graph $G$, the interpreta-*

*tion of p in G (denoted $[\![p]\!]_G$) is defined by:*

$$
\begin{aligned}
[\![self]\!]_G &= \{\langle x, x\rangle | x \in voc(G)\}\\
[\![self :: a]\!]_G &= \{\langle a, a\rangle\}\\
[\![next]\!]_G &= \{\langle x, y\rangle | \exists z; \langle x, z, y\rangle \in G\}\\
[\![next :: a]\!]_G &= \{\langle x, y\rangle | \langle x, a, y\rangle \in G\}\\
[\![edge]\!]_G &= \{\langle x, y\rangle | \exists z; \langle x, y, z\rangle \in G\}\\
[\![edge :: a]\!]_G &= \{\langle x, y\rangle | \langle x, y, a\rangle \in G\}\\
[\![node]\!]_G &= \{\langle x, y\rangle | \exists z; \langle z, x, y\rangle \in G\}\\
[\![node :: a]\!]_G &= \{\langle x, y\rangle | \langle a, x, y\rangle \in G\}\\
\left[\!\left[axis^{-1}\right]\!\right]_G &= \{\langle x, y\rangle | \langle y, x\rangle \in [\![axis]\!]_G\}\\
\left[\!\left[axis^{-1} :: a\right]\!\right]_G &= \{\langle y, x\rangle | \langle x, y\rangle \in [\![axis :: a]\!]_G\}\\
[\![exp_1 \,/\, exp_1]\!]_G &= [\![exp_1]\!]_G \cup [\![exp_2]\!]_G\\
[\![exp_1 / exp_2]\!]_G &= \{\langle x, y\rangle | \exists z; \langle x, z\rangle \in [\![exp_1]\!]_G\\
&\qquad \wedge \langle z, y\rangle \in [\![exp_2]\!]_G\}\\
[\![exp^*]\!]_G &= [\![self]\!]_G \cup [\![exp]\!]_G \cup [\![exp/exp]\!]_G\\
&\qquad \cup [\![exp/exp/exp]\!]_G \cup \dots\\
[\![self :: [exp]]\!]_G &= \{\langle x, x\rangle | x \in voc(G)\\
&\qquad \wedge \exists z; \langle x, z\rangle \in [\![exp]\!]_G\}
\end{aligned}
$$

$$
\begin{aligned}
[\![next :: [exp]]\!]_G &= \{\langle x, y\rangle | \exists z, w; \langle x, z, y\rangle \in G\\
&\qquad \wedge \langle z, w\rangle \in [\![exp]\!]_G\}\\
[\![edge :: [exp]]\!]_G &= \{\langle x, y\rangle | \exists z, w; \langle x, y, z\rangle \in G\\
&\qquad \wedge \langle z, w\rangle \in [\![exp]\!]_G\}\\
[\![node :: [exp]]\!]_G &= \{\langle x, y\rangle | \exists z, w; \langle z, x, y\rangle \in G\\
&\qquad \wedge \langle z, w\rangle \in [\![exp]\!]_G\}\\
\left[\!\left[axis^{-1} :: [exp]\right]\!\right]_G &= \{\langle x, y\rangle | \langle y, x\rangle \in [\![axis :: [exp]]\!]_G\}
\end{aligned}
$$

The evaluation of a nested regular expression $R$ over an RDF graph $G$ is defined as a binary relation $[\![R]\!]_G$, by a pair of nodes $\langle a, b\rangle$ such that $a$ is reachable from $b$ in $G$ by following a path that conforms to $R$. In the following, we use the positive closure of a path expression $R$ denoted by $R^+$ and defined as $R^+ = R/R^*$.

**Definition 12.** *The evaluation of a nSPARQL triple pattern $t = \langle \mathcal{X}, R, \mathcal{Y}\rangle$ over an RDF graph G is:*

$$
\begin{aligned}
[\![t]\!]_G = \{\mu |\ & dom(\mu) = \{\mathcal{X}, \mathcal{Y}\} \cap \mathcal{B}\\
& \textit{and } \langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![R]\!]_G\}
\end{aligned}
$$

Answers to nSPARQL queries follow the same definition as for SPARQL but this time the answers are constructed from mappings satisfying nSPARQL triple patterns.

**Definition 13** (Answers to an nSPARQL basic graph pattern)**.** *Let P be a basic nSPARQL graph pattern and G be an RDF graph, then the set of* answers *to P over G is:*

$$
\mathcal{A}(G, P) = \{\mu |\ \langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![R]\!]_G, \forall \langle \mathcal{X}, R, \mathcal{Y}\rangle \in P\}
$$

The evaluation of such basic graph patterns is measured with the usual evaluation problem:
**Problem:** Regular expression evaluation
**Input:** An RDF graph $G$, a regular expression $R$, and a pair $\langle a, b\rangle$
**Question:** Does $\langle a, b\rangle \in [\![R]\!]_G$?
We will use this same problem with different type of regular expressions. This problem is solved efficiently through an effective procedure provided in [29].

**Theorem 1** (Complexity of nSPARQL evaluation [29])**.** *The evaluation problem for a nested regular expression R over an RDF graph G can be solved in time $O(|G|.|R|)$.*

### *3.3. Querying RDFS with nSPARQL*

[24] has introduced the reflexive relaxed semantics for RDFS in which `rdfs:subPropertyOf` and `rdfs:subClassOf` of do not have to be reflexive. The reflexive relaxed semantics does not change much RDFS. Indeed, from the standard (reflexive) semantics, we can deduce that any class (respectively, property) is a subclass (respectively, subproperty) of itself. The reflexivity requirement only entails reflectivity assertions which do not interact with other triples unless constraints are added to the `rdfs:subPropertyOf` or `rdfs:subClassOf` properties. Therefore, it is assumed that elements of RDFS vocabulary appear only as predicate.

However, when issuing queries involving these relations, e.g., with a graph pattern like $\langle$`?x sp ?y`$\rangle$, all properties in the graph will be answers. Since this

| Subproperty (sp) | Subclass (sc) | Typing (dom, range |
|---|---|---|
| $\frac{\langle \mathcal{A}, \texttt{sp}, \mathcal{B}\rangle \ \langle \mathcal{B}, \texttt{sp}, \mathcal{C}\rangle}{\langle \mathcal{A}, \texttt{sp}, \mathcal{C}\rangle}$ | $\frac{\langle \mathcal{A}, \texttt{sc}, \mathcal{B}\rangle \ \langle \mathcal{B}, \texttt{sc}, \mathcal{C}\rangle}{\langle \mathcal{A}, \texttt{sc}, \mathcal{C}\rangle}$ | $\frac{\langle \mathcal{A}, \texttt{dom}, \mathcal{B}\rangle \ \langle \mathcal{X}, \mathcal{A}, \mathcal{Y}\rangle}{\langle \mathcal{X}, \texttt{type}, \mathcal{B}\rangle}$ |
| $\frac{\langle \mathcal{A}, \texttt{sp}, \mathcal{B}\rangle \ \langle \mathcal{X}, \mathcal{A}, \mathcal{Y}\rangle}{\langle \mathcal{X}, \mathcal{B}, \mathcal{Y}\rangle}$ | $\frac{\langle \mathcal{A}, \texttt{sc}, \mathcal{B}\rangle \ \langle \mathcal{X}, \texttt{type}, \mathcal{A}\rangle}{\langle \mathcal{X}, \texttt{type}, \mathcal{B}\rangle}$ | $\frac{\langle \mathcal{A}, \texttt{range}, \mathcal{B}\rangle \ \langle \mathcal{X}, \mathcal{A}, \mathcal{Y}\rangle}{\langle \mathcal{Y}, \texttt{type}, \mathcal{B}\rangle}$ |

Table 1

RDFS inference rules

would clutter results, we assume, as done in [24], that queries use the reflexive relaxed semantics. It is easy to recover the standard semantics by providing the additional triples when sp or sc are queried.

In the following, we use the closure graph of an RDF graph $G$, denoted by $closure(G)$, which is defined by the graph obtained by saturating $G$ with all triples that can be deduced using rules of Table 1 [24].

**Definition 14.** *The evaluation of an nSPARQL triple pattern $t = \langle \mathcal{X}, R, \mathcal{Y}\rangle$ over an RDF graph G modulo RDFS is defined as the following set of mappings:*

$$[\![t]\!]_G^{rdfs} = \{\mu | dom(\mu) = \{\mathcal{X}, \mathcal{Y}\} \cap \mathcal{B} \\ \wedge \langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![R]\!]_{closure(G)}\}$$

**Definition 15** (Answers to an nSPARQL basic graph pattern modulo RDFS)**.** *Let $P$ be a basic nSPARQL graph pattern and G be an RDF graph, then the set of* answers *to P over G modulo RDFS is:*

$$\mathcal{A}^o(G, P) = \{\mu \mid \mu \in [\![t]\!]_G^{rdfs}, \forall t \in P\}$$

As presented in [28], nSPARQL can evaluate queries with regard to RDFS by transforming queries using the transformation function $\phi$ defined in the following rules [24]:

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

**Example 4** (nSPARQL evaluation modulo RDFS)**.** *The following nSPARQL graph pattern could be used as a query to retrieve the set of pairs of cities connected by a sequence of transportation means such that one city is from France and the other city is from Jordan:*

$$\{\langle ?city_1, (next :: transport)^+, ?city_2\rangle, \\ \langle ?city_1, next :: cityIn, France\rangle, \\ \langle ?city_2, next :: cityIn, Jordan\rangle\}$$

*When evaluating this graph pattern against the RDF graph of Figure 1 and considering the RDFS semantics, it returns the empty set since there is no explicit property "transport" between the two queried cities. However, it should return the following set of pairs:* $\{\langle ?city_1 \leftarrow Paris, ?city_2 \leftarrow Amman\rangle, \langle ?city_1 \leftarrow Grenoble, ?city_2 \leftarrow Amman\rangle\}$

*To answer the above graph pattern considering RDFS semantics, it could be transformed to the following nSPARQL graph pattern:*

$$\{\langle ?city_1, (next :: [(next :: sp)^*/self :: transport])^+ \\ , ?city_2\rangle, \\ \langle ?city_1, next :: cityIn, France\rangle, \\ \langle ?city_2, next :: cityIn, Jordan\rangle\}$$

This encoding is correct and complete with regard to RDFS entailment.

**Theorem 2** (Completeness of $\phi$ [28])**.** *Let $\langle \mathcal{X}, p, \mathcal{Y}\rangle$ be a SPARQL triple pattern with $\mathcal{X}, \mathcal{Y} \in (\mathcal{U} \cup \mathcal{B})$ and $p \in \mathcal{U}$, then for any RDF graph G:*

$$[\![\langle \mathcal{X}, p, \mathcal{Y}\rangle]\!]_G^{rdfs} = [\![\langle \mathcal{X}, \phi(p), \mathcal{Y}\rangle]\!]_G$$

## 4. CPSPARQL and cpSPARQL: syntax and semantics

CPSPARQL has been defined for addressing two main issues. The first one comes from the need to extend PSPARQL and thus to allow for expressing constraints on nodes of traversed paths; while the other comes from the need to answer PSPARQL queries modulo RDFS so that the transformation rules could be applied to PSPARQL queries [2].

In addition to CPSPARQL, we present cpSPARQL, a language using CPSPARQL graph patterns in the same way as nSPARQL does.

### *4.1. CPSPARQL syntax*

The notation that we use in this paper for the syntax of CPSPARQL is slightly different from the one defined in the original proposal [2]. The original one uses `edge` and `node` constraints to express constraints on predicates (or edges) and nodes of RDF graphs, respectively. In this paper, as done for nSPARQL, we adopt the `axes` borrowed from XPath, with which the reader may be more familiar. This will also allow us to better compare cpSPARQL and nSPARQL. Additionally, in the original proposal, the `ALL` and `EXISTS` keywords were used to allow expressing constraints on all traversed nodes or to check the existence of a node in the traversed path that satisfies the given constraint. We do not use these keywords in the fragment presented below since they do not add expressiveness with respect to the RDFS semantics, i.e., the fragment still captures the RDFS semantics.

Constraints act as filters for paths that must be traversed by constrained regular expressions and select those whose nodes satisfy encountered constraint.

**Definition 16** (Constrained regular expression)**.** *A* constrained regular expression *is an expression built from the following grammar:*

$$exp ::= axis \mid axis::a \mid axis::[x : \psi] \mid axis::]x : \psi[ \\ \mid exp \mid exp/exp \mid exp|exp \mid exp^*$$

*with $\psi$ being a set of triples belonging to $(\mathcal{U} \cup \mathcal{B} \cup \{x\} \times exp \times \mathcal{T} \cup \{x\})$ $UNION$ $\{FILTER\!-\!expressions\}$ over $\mathcal{B} \cup \{x\}$. $\psi$ is called a CPRDF-constraint and $x$ its head variable.*

Constrained regular expressions require the item in one axis to satisfy a particular constraint, i.e., to satisfy a particular graph pattern (here an RDF graph) or filter. We introduce the closed square brackets and open square brackets notation for distinguishing between constraints which export their variable (it may be assigned by the mapping) and constraints which do not export it (the variable is only notational). This is equivalent to the initial CPSPARQL formulation, in which the variable was always exported, since CPSPARQL can ignore such variables through projection.

In the following, we have used $\mathcal{B}(R)$ to denote the set of variables occurring as the head variable of an open bracket constraint in $R$.

Constraint nesting is allowed because constrained regular expressions may be used in the graph pattern of another constrained regular expression as in the following example.

**Example 5** (Constrained regular expression)**.** *The following constrained regular expression could be used to find nodes connected by transportation means that are not buses:*

$$(next :: [?p : \{\langle ?p, (next :: sp)^*, transport\rangle \\ FILTER(?p! = bus)\}])^+$$

In contrast to nested regular expressions, constrained regular expressions can apply constrains (such as SPARQL constraints) in addition to simple nested path constraints.

Constrained regular expressions are used in triple patterns, in predicate position, to define CPSPARQL.

**Definition 17** (CPSPARQL triple pattern)**.** *A* CPSPARQL triple pattern *is a triple $\langle s, p, o\rangle$ such that $s \in \mathcal{T}$, $o \in \mathcal{T}$ and $p$ is a constrained regular expression.*

**Definition 18** (CPSPARQL graph pattern)**.** *A* CPSPARQL graph pattern *is defined inductively by:*

- *every CPSPARQL triple pattern is a CPSPARQL graph pattern;*
- *if $P_1$ and $P_2$ are two CPSPARQL graph patterns and $K$ is a SPARQL constraint, then ($P_1$* `AND` *$P_2$), ($P_1$* `UNION` *$P_2$), ($P_1$* `OPT` *$P_2$), and ($P_1$* `FILTER` *$K$) are CPSPARQL graph patterns.*

**Example 6** (CPSPARQL graph pattern)**.** *The following CPSPARQL graph pattern could be used to retrieve the set of pairs of cities connected by a sequence of transportation means (which are not buses) such that one city in France and the other one in Jordan:*

$$\{\langle ?city_1, (next :: [?p\!:\!\{\langle ?p, (next :: sp)^*, transport\rangle \\ FILTER(?p! = bus)\}])^+, ?city_2\rangle \\ \langle ?city_1, next :: cityIn, France\rangle \\ \langle ?city_2, next :: cityIn, Jordan\rangle\}$$

*If open square brackets were used, this graph pattern would, in addition, bind the $?p$ variable to a matching value, i.e., the transportation means used.*

By restricting CPRDF constraints, it is possible to define a far less expressive language. cpSPARQL is such a language: instead of general GRDF graphs as constraints, it only allows at most one triple (with a cpSPARQL regular expression as predicate)

**Definition 19** (cpSPARQL regular expression)**.** *A* cpSPARQL regular expression *is an expression built from the following grammar:*

$$\begin{aligned} exp ::= &\ axis \mid axis{::}a \mid axis{::}]?x : TRUE[ \\ &\mid axis{::}[?x : \{\langle ?x, exp, v\rangle\}\{FILTER(?x)\}] \\ &\mid exp \mid exp/exp \mid exp|exp \mid exp^* \end{aligned}$$

*such that $v$ is either a distinct variable $?y$ or a constant (an element of $U \cup L$) and $FILTER(?x)$ is the usual SPARQL filter condition containing at most the variable $?x$ and $v$ if $v$ is a variable.*

The first specific form, with open square brackets, has been preserved so that cpSPARQL triples cover SPARQL basic graph patterns, i.e., allow for variables in predicate position. In the other specific forms, a cpSPARQL constraint is either a cpSPARQL regular expression containing $?x$ as the only variable and/or a SPARQL FILTER constraint. Hence, such a regular expression may have several constraints, but each constraint can only expose one variable and it cannot refer to variables defined elsewhere.

Deciding if a CPSPARQL triple is a cpSPARQL triple can be performed in linear time in the size of the regular expression used.

**Example 7** (cpSPARQL triple patterns)**.** *The query of Example 3 could be expressed by the following cpSPARQL pattern:*

$$\begin{aligned} &\langle ?city_1, \\ &\quad (next :: [?p : \{\langle ?p, (next :: sp)^*, transport\rangle\}])^+, \\ &\quad ?city_2\rangle \end{aligned}$$

*The constraint $\psi = ?p : \{\langle ?p, (next :: sp)^*, transport\rangle\}$ is used to restrict the properties (in this pattern the constraint is applied to properties since the axis* `next` *is used) to be only a transportation mean.*

*Example 5 provides another cpSPARQL regular expression. By contrast, CPSPARQL graph patterns allow for queries like:*

$$\begin{aligned} next :: [?p; \{&\langle ?p, (next :: sp)*, ?z\rangle, \\ &\langle ?q, (next :: sp)*, ?z\rangle, \\ &\langle ?p, owl : inverseOf, ?q\rangle, \\ &FILTER(regex(?z, iata.org))\}] \end{aligned}$$

*which is not a cpSPARQL regular expression since it uses more than two variables.*

It is possible to develop languages based on cpSPARQL regular expressions following what is done with constrained regular expressions.

## *4.2. CPSPARQL semantics*

Intuitively, a constrained regular expression $next{::}[\psi]$ (where $\psi = ?p : \{\langle ?p, sp^*, transport\rangle\}$) is equivalent to $next{::}p$ if $p$ satisfies the constraint $\psi$, i.e., $p$ should be a sub-property of $transport$ (when $p$ is substituted to the variable $?p$).

**Definition 20** (Satisfied constraint in an RDF graph)**.** *Let $G$ be an RDF graph, $s$ and $o$ be two nodes of $G$ and $\psi = x : C$ be a constraint, then $s$ and $o$ satisfies $\psi$ in $G$ (denoted $\langle s, o\rangle \in [\![\psi]\!]_G$) if one of the following conditions is satisfied:*

1. *$C$ is a triple pattern $C = \langle \mathcal{X}, R, \mathcal{Y}\rangle$, and $\langle \mathcal{X}^x_s, \mathcal{Y}^y_o\rangle \in [\![R^x_s]\!]_G$, where $K^z_r$ means that $r$ is substituted to the variable $z$ if $K = z$ or $K$ contains the variable $z$. If $z$ is a constant then $z = r$.*
2. *$C$ is a SPARQL filter constraint and $C^{x,y}_{s,o} = \top$, where $C^{x,y}_{s,o} = \top$ means that the constraint obtained by the substitution of $s$ to each occurrence of the variable $x$ and $o$ to each occurrence of the variable $y$ in $C$ is evaluated to true*[2].
3. *$C = P\ FILTER\ K$, then 1 and 2 should be satisfied*

As for nested regular expressions, the evaluation of a constrained regular expression $R$ over an RDF graph $G$ is defined as a binary relation $[\![R]\!]_G$, by a pair of nodes $\langle a, b\rangle$ such that $a$ is reachable from $b$ in $G$ by following a path that conforms to $R$. The following definition extends Definition 11 to take into account the semantics of terms with constraints.

[2]Except for the case of bound (see Definition 5 and the discussion after it.)

**Definition 21** (Constrained path interpretation)**.** *Given a constrained regular expression $P$ and an RDF graph $G$. If $P$ is unconstrained then the interpretation of $P$ in $G$ (denoted $[\![P]\!]_G$) is as in Definition 11; otherwise the interpretation of $P$ in $G$ is defined as:*

$$
\begin{aligned}
[\![self :: [\psi]]\!]_G &= \{\langle x, x\rangle \mid \exists z; x \in voc(G) \wedge \\
&\qquad \langle x, z\rangle \in [\![\psi]\!]_G\} \\
[\![next :: [\psi]]\!]_G &= \{\langle x, y\rangle \mid \exists z, w; \langle x, z, y\rangle \in G \wedge \\
&\qquad \langle z, w\rangle \in [\![\psi]\!]_G\} \\
[\![edge :: [\psi]]\!]_G &= \{\langle x, y\rangle \mid \exists z, w; \langle x, y, z\rangle \in G \wedge \\
&\qquad \langle z, w\rangle \in [\![\psi]\!]_G\} \\
[\![node :: [\psi]]\!]_G &= \{\langle x, y\rangle \mid \exists z, w; \langle z, x, y\rangle \in G \wedge \\
&\qquad \langle z, w\rangle \in [\![\psi]\!]_G\} \\
[\![axis^{-1} :: [\psi]]\!]_G &= \{\langle x, y\rangle \mid \langle y, x\rangle \in [\![axis :: [\psi]]\!]_G\}
\end{aligned}
$$

**Definition 22** (Answer to a CPSPARQL triple pattern)**.** *The evaluation of a CPSPARQL triple pattern $t = \langle \mathcal{X}, R, \mathcal{Y}\rangle$ over an RDF graph $G$ is defined as the following set of mappings:*

$[\![t]\!]_G = \{\mu \mid dom(\mu) = \{\mathcal{X}, \mathcal{Y}\} \cap \mathcal{B} \cup \mathcal{B}(R)$ *and* $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\mu(R)]\!]_G\}$ *such that $\mu(R)$ is the constrained regular expression obtained by substituting the variable $?x$ appearing in a constraint with open brackets in $R$ by $\mu(?x)$.*

This semantics also applies to cpSPARQL graph patterns.

### *4.3. Querying RDFS with CPSPARQL*

Like for nSPARQL, constraints allow for encoding RDF Schemas within queries.

**Definition 23** (RDFS triple pattern expansion)**.** *Given an RDF triple $t$, the* RDFS expansion *of $t$, denoted by $\tau(t)$, is defined as:*

$$
\begin{aligned}
\tau(\langle s, \mathtt{sc}, o\rangle) &= \langle s, \mathtt{next::sc}^+, o\rangle \\
\tau(\langle s, \mathtt{sp}, o\rangle) &= \langle s, \mathtt{next::sp}^+, o\rangle \\
\tau(\langle s, \mathtt{dom}, o\rangle) &= \langle s, \mathtt{next::dom}, o\rangle \\
\tau(\langle s, \mathtt{range}, o\rangle) &= \langle s, \mathtt{next::range}, o\rangle \\
\tau(\langle s, \mathtt{type}, o\rangle) &= \langle s, \mathtt{next::type/next::sc}^*| \\
&\quad \mathtt{edge/(next::sp)^*/next::dom/(next::sc)^*}| \\
&\quad \mathtt{node^{-1}/(next::sp)^*/next::range/(next::sc)^*}, o\rangle \\
\tau(\langle s, p, o\rangle) &= \langle s, (\mathtt{next::}[?x : \{\langle ?x, (\mathtt{next::sp})^*, \\
&\quad p\rangle\}]), o\rangle, \\
&\quad p \notin \{\mathtt{sp}, \mathtt{sc}, \mathtt{type}, \mathtt{dom}, \mathtt{range}\}
\end{aligned}
$$

The RDFS expansion of an RDF triple is a cpSPARQL triple.

The extra variable "$?x$" introduced in the last item of the transformation, is only used inside the constraint of the constrained regular expression and so it is not considered to be in $dom(\mu)$, i.e., only variables occurring as a subject or an object in a CPSPARQL triple pattern are considered in mappings (see Definition 22). Therefore, the SELECT operator (projection) is not needed in cpSPARQL to restrict the results of the transformed triple as in the case of PSPARQL [5], as illustrated in the following example.

**Example 8** (SPARQL query transformation)**.** *Consider the following SPARQL query that searches pairs of nodes connected with a property p*

```
SELECT ?X ?Y
WHERE ?X p ?Y .
```

*It is possible to answer this query modulo RDFS by transforming this query into the following PSPARQL query:*

```
SELECT ?X ?Y
WHERE ?X ?P ?Y . ?P sp* p .
```

The evaluation of the above PSPARQL query is the mapping $\{?X \leftarrow a, ?P \leftarrow b, ?Y \leftarrow c\}$. So, to actually obtain the desired result, a projection (SELECT) operator must be performed since the extra variable $?P$ is used in the transformation. It is argued in [29] that including the SELECT (projection) operator to the conjunctive fragment of PSPARQL makes the evaluation problem NP-hard.

On the other hand, the query could be answered by transforming it, with the $\tau$ function of Definition 26, to the following cpSPARQL query (in which there is no need for the projection operator):

```
(?X, (next::[?x:  ?x, (next::sp)*, p ]), ?Y)
```

Since the variable $?x$ is used inside the constraint, the answer to this query will be $\{?X \leftarrow a, ?Y \leftarrow b\}$ (see Definition 24).

This has the important consequence that any nSPARQL graph pattern can be translated in a cpSPARQL graph pattern with similar structure and no additional variable. Hence, no additional projection operation (SELECT) is required for answering nSPARQL queries in cpSPARQL.

The proof of the following Theorem follows from the results in [28] except the last step since all other transformation steps are the same as the ones presented in [28].

**Theorem 3.** *Let $\langle \mathcal{X}, p, \mathcal{Y}\rangle$ be a SPARQL triple pattern with $\mathcal{X}, \mathcal{Y} \in (\mathcal{U} \cup \mathcal{B})$ and $p \in \mathcal{U}$, then $[\![\langle \mathcal{X}, p, \mathcal{Y}\rangle]\!]_G^{rdfs}$ = $[\![\langle \mathcal{X}, \tau(p), \mathcal{Y}\rangle]\!]_G$ for any RDF graph G.*

*Proof.* We need to prove only the last step since all other transformation steps are the same as the ones in [28]. That is $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, p, \mathcal{Y}\rangle]\!]_G^{rdfs}$ iff $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, \tau(p), \mathcal{Y}\rangle]\!]_G$.

- ($\Rightarrow$) Suppose that $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, p, \mathcal{Y}\rangle]\!]_G^{rdfs}$. In this case, there exists $p_1$ such that ($p_1$ $sp$ $p_2$ $sp$ … $sp$ $p_n = p$) and $\langle \mu(\mathcal{X}), p_1, \mu(\mathcal{Y})\rangle \in G$ as well as $\langle \mu(\mathcal{X})$, `next::`$p_1, \mu(\mathcal{Y})\rangle \in G$. Let us consider now the transformed triple $\tau(t) = \langle \mathcal{X}, (\texttt{next::}\psi), \mathcal{Y}\rangle$ (where $\psi = [?p : \{\langle ?p, (\texttt{next::sp})^*, p\rangle\}]$). The mappings for the variable $?p$ will be $\{\langle ?p, p_i\rangle \mid i = 1, \dots, n\}$ (since $[\![\psi]\!]_G = \{\langle p_i, p\rangle \mid i = 1, \dots, n\}$). Now according to Definitions 21 and 22, $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, (\texttt{next::}\psi), \mathcal{Y}\rangle]\!]_G$ iff $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in G$ and $p_1 \in [\![\psi]\!]_G$, and this condition holds.
- ($\Leftarrow$) We have to prove that if $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, (\texttt{next::}[\psi]), \mathcal{Y}\rangle]\!]_G$ (with $\psi = ?p : \{\langle ?p, (\texttt{next::sp})^*, p\rangle\}$), then $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, p, \mathcal{Y}\rangle]\!]_G^{rdfs}$. Suppose that $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, (\texttt{next::}\psi), \mathcal{Y}\rangle]\!]_G$. In this case, there exists $p_1$ such that $\langle \mu(\mathcal{X})$, `next::`$p_1, \mu(\mathcal{Y})\rangle \in G$ and $p_1 \in [\![\psi]\!]_G$, that is, $\langle p_1$,`next::sp`$, p_2\rangle$, …, $\langle p_{n-1}$,`next::sp`$, p_n = p\rangle \in G$. Therefore, $\langle \mu(\mathcal{X}), \mu(\mathcal{Y})\rangle \in [\![\langle \mathcal{X}, p, \mathcal{Y}\rangle]\!]_G^{rdfs}$ since $\langle p_1, (\texttt{next::sp})^*, p\rangle$ and $\langle \mu(\mathcal{X})$, `next::`$p_1$, $\mu(\mathcal{Y})\rangle \in G$.

□

## 5. Complexity of evaluating cpSPARQL

The complexity of cpSPARQL is given with respect to the following problem:
**Problem:** cpSPARQL evaluation
**Input:** An RDF graph $G$, a cpSPARQL regular expression $R$, and a pair $\langle a, b\rangle$
**Question:** Does $\langle a, b\rangle \in [\![R]\!]_G$?

We follow [28] to store an RDF graph as an adjacency list: every $u \in voc(G)$ is associated with a list of pairs $\alpha(u)$. For instance, if $\langle s, p, o\rangle \in G$, then $\langle \texttt{next}::p, o\rangle \in \alpha(s)$ and $\langle \texttt{edge}^{-1}::o, s\rangle \in \alpha(p)$. Also, $\langle \texttt{self}::u, u\rangle \in \alpha(u)$, for $u \in voc(G)$. The set of terms of a constrained regular expression $R$, denoted by $\mathcal{T}(R)$, is constructed as follows:

$$\mathcal{T}(R) = \{R\} \text{if } R \text{ is either } \texttt{axis}, \texttt{axis}::\text{a, or } \texttt{axis}::\psi$$

$$\mathcal{T}(R_1/R_2) = \mathcal{T}(R_1|R_2) = \mathcal{T}(R_1) \cup \mathcal{T}(R_2)$$

$$\mathcal{T}(R_1^*) = \mathcal{T}(R_1)$$

Let $\mathcal{A}_R = (Q, \mathcal{T}(R), s_0, F, \delta)$ be the $\epsilon - NFA$ of $R$ constructed in the usual way using the terms $\mathcal{T}(R)$, where $\delta : Q \times (\mathcal{T}(R) \cup \{epsilon\}) \rightarrow 2^Q$ be its transition function. In the evaluation algorithm, we use the product automaton $G \times \mathcal{A}_R$ (in which $\delta' : \langle voc(G) \times Q\rangle \times (\mathcal{T}(R) \cup \{epsilon\}) \rightarrow 2^{voc(G) \times Q}$ is its transition function). We construct $G \times \mathcal{A}_R$ as follows:

- $\langle u, q\rangle \in voc(G) \times Q$, for every $u \in voc(G)$ and $q \in Q$;
- $\langle v, q\rangle \in \delta'(\langle u, p\rangle, s)$ iff $q \in \delta(p, s)$; and one of the following conditions satisfied:
  * $s = \texttt{axis}$ and there exists $a$ s.t. $\langle \texttt{axis}::a, v\rangle \in \alpha(u)$
  * $s = \texttt{axis}::a$ and $\langle \texttt{axis}::a, v\rangle \in \alpha(u)$
  * $s = \texttt{axis}::\psi$ and there exists $b$ s.t. $\langle \texttt{axis}::b, v\rangle \in \alpha(u)$ and $b \in [\![\psi]\!]_G$

Algorithm 2 (Eval) solves the evaluation problem for a constrained regular expression $R$ over an RDF graph $G$. This algorithm is almost the same as the one in [29] which solves the evaluation problem for nested regular expressions $R$ over an RDF graph $G$. The Eval algorithm calls the Algorithm 1 (LABEL), which is an adaptation of the LABEL algorithm of [29] in which we modify only the first two steps. These two steps are based on the transformation rules from nSPARQL expressions to cpSPARQL expressions (i.e., the equivalences between them).

The algorithm as the same $O(|G|.|R|)$ time complexity as usual regular expressions [33,23] and nested regular expressions [29] evaluation.

**Algorithm 1 LABEL**(G, exp):

```
1. for each $axis :: [\psi] \in D_0(exp)$ do
2.     call Label(G, exp') //where $exp' = exp1/self :: p$ if $\psi = ?x :< ?x, exp1, p >$; $exp' = exp1/self :: p$ if $\psi = ?x :< ?x, exp1, ?y >$
3. construct $A_{exp}$, and assume that $q_0$ is its initial state and $F$ is its set of final states
4. construct $G \times A_{exp}$
5. for each state $(u, q_0)$ that is connected to a state $(v, q_f)$ in $G \times A_{exp}$, with $q_f \in F$ do
6.     $label(u) := label(u) \cup exp$
```

**Algorithm 2 Eval**$(G, R, \langle a, b\rangle)$

```
Data: An RDF graph $G$, a constrained regular expression $R$, and a pair $\langle a, b\rangle$.
Result: YES if $\langle a, b\rangle \in [\![R]\!]_G$; otherwise NO.

for each $u \in voc(G)$ do
    $label(u) := \emptyset$
LABEL $(G, R)$
construct $\mathcal{A}_R$ (assume $q_0$ : initial state and $F$ : set of final states)
construct the product automaton $G \times \mathcal{A}_R$
if a state $\langle b, q_f\rangle$ with $q_f \in F$, is reachable from $\langle a, q_0\rangle$ in $G \times \mathcal{A}_R$ then
    return YES;
else
    return NO;
end if
```

**Theorem 4** (Complexity of cpSPARQL evaluation). *Eval solves the evaluation problem for constrained regular expression in time $O(|G|.|R|)$.*

*Proof.* Let $R$ be the a constrained regular expression, $G$ be an RDF graph, $\langle a, b\rangle$ be a pair of nodes, $A_R$ be the automaton recognizing the language of $R$.

Constructing the automaton of $R$ can be done in NLOGSPACE as in the usual automata [33,23] (with the alphabet described in the text above). For simplicity and without loss of generality, we use the `next` axis to illustrate the construction of the product automaton. This is because the axis determines the node to be checked (subject, predicate or object) and thus does not affect the construction. The construction of the product automaton is done as follows:

– If $R = axis ::]?x : TRUE[$ then checking whether the pair $\langle a, b\rangle$ is in $[[R]]G$ can be done in $O(|G|)$ since it is sufficient to substitute each node $n$ to $?x$ and check whether $\langle a, n, b\rangle$ is in $G$ (according to the axis).
– Otherwise, call the Eval algorithm (where $D_0$ is defined as done in [29]). Note that if $\langle s_i, next :: [FILTER(?x)], s_j\rangle \in A_R$ and $\langle n_i, next :: p, n_j\rangle \in G$, then add $\langle s_j, n_j\rangle$ to the product automaton if $p$ satisfies the SPARQL filter constraint by substituting only the node $p$ to the variable $?x$. Checking if a node $n$ satisfies a SPARQL filter constraint can be done in $O(1)$.
Additionally, if $\langle ?x, next::p, ?y\rangle.FILTER(?x, ?y) \in A_R$ and $\langle n_i, next :: p, n_j\rangle \in G$, then add $\langle s_j, n_j\rangle$ to the product automaton if $\langle n_i, next :: p, n_j\rangle \in G$ and the SPARQL filter constraint is satisfied by substituting the node $n_i$ to the variable $?x$ and $n_j$ to the variable $?y$.

So, constructing the product automaton ($G \times \mathcal{A}_R$) can be done in time $O(|G|.|R|)$. Hence, checking if the pair $\langle a, b\rangle \in [\![R]\!]_G$ is equivalent to checking if the language accepted by ($G \times \mathcal{A}_R$) is not empty, which can be done in $O(|G|.|R|)$. □

## 6. On the expressiveness of cpSPARQL and nSPARQL

In this section, we compare the expressiveness of cpSPARQL with nSPARQL. We identify several assertions which together show that cpSPARQL is strictly more expressive than nSPARQL and that even if nSPARQL were added projection, it would remain strictly less expressive than CPSPARQL.

**Nested regular expressions (nSPARQL) cannot express all (SPARQL) triple patterns**

Although it is explained in [28,29] that SPARQL triple patterns can be encoded by nested regular expressions, triple patterns with three variables (subject, predicate, object) could not be expressed by nested regular expressions since variables are not allowed in nested regular expressions. The reader may wonder whether this is useful or not. The following query is a useful example:

```
SELECT *
WHERE ?s foaf:name "Faisal". ?s ?p ?o .
```

That could be used to retrieve all RDF data about a person named "Faisal". However, cpSPARQL triple patterns are proper extension of SPARQL triple patterns and thus the above query could be expressed by the following query:

```
SELECT *
WHERE ?s next::foaf:name "Faisal".
       ?s next::]?p:TRUE[ ?o .
```

## nSPARQL without SELECT cannot express all CPSPARQL

We show in the following that some queries, which can be expressed by CPSPARQL, can only be expressed in nSPARQL with the projection (the SELECT operator):

Assume that one wants to retrieve pairs of distinct nodes having a common ancestor. Then the following nSPARQL pattern can express this query:

$$\{\langle ?person1, (\texttt{next::ascendant})^+/ (\texttt{next}^{-1}\texttt{::ascendant})^+, ?person2\rangle, FILTER(!(?person1 = ?person2))\}$$

The same query with the restriction that the name of the common ancestor should contain a given family name, for instance "alkhateeb", requires the use of extra variable to pose the constraint:

$$\{\langle ?person1, (\texttt{next::ascendant})^+, ?ancestor\rangle, \langle ?person2, (\texttt{next::ascendant})^+, ?ancestor\rangle, FILTER(!(?person1 = ?person2) \&\&(regex(?ancestor, \text{"}alkhateeb\text{"})))\}$$

Notice that the evaluation of this graph pattern is the mapping $\{?person1 \leftarrow p1, ?ancestor \leftarrow p3, ?person2 \leftarrow p2\}$. Therefore, to obtain the desired result, the projection operator must be performed:

$$\mu_{?person1,?person2}( \{\langle ?person1, (\texttt{next::ascendant})^+, ?ancestor\rangle, \langle ?person2, (\texttt{next::ascendant})^+, ?ancestor\rangle, FILTER(!(?person1 = ?person2) \&\&(regex(?ancestor, \text{"}alkhateeb\text{"})))\})$$

So, the above query cannot be expressed in nSPARQL without the use of SELECT, which is not allowed in nSPARQL [29]. Besides, any SPARQL query that uses SELECT over a set of variables such that there exists at least one existential variable, i.e., a variable not in the SELECT, used in a FILTER constraint cannot be expressed by nSPARQL graph patterns.

However, the following CPSPARQL graph pattern could be used to express the above query:

$$\{\langle ?person1, (\texttt{next::ascendant})^+ /\texttt{self::}[?ancestor : FILTER(regex(?ancestor, \text{"}alkhateeb\text{"}))] /(\texttt{next}^{-1}\texttt{::ascendant})^+, ?person2\rangle, FILTER(!(?person1 = ?person2))\}$$

## nSPARQL cannot express all cpSPARQL, even with SELECT

In the following discussion, we show that there exists a cpSPARQL regular expression that cannot be expressed in a nested regular expression as well as some natural and useful queries that can be expressed in CPSPARQL patterns cannot be expressed in nSPARQL patterns even with the SELECT operator.

If one wants to restrict the query of Example 3 such that every stop is a city in the same country (for example, France), then the following nested regular expression expresses this query:

$\langle ?city_1, (next :: [(next :: sp)^* /self :: transport] /self :: [next :: cityIn/self :: France])^+, ?city_2\rangle$

This query also could be expressed in the following constrained regular expressions:

$\langle ?city_1, (next :: [\psi_1]/self :: [\psi_2])^+, ?city_2\rangle$, where:
$\psi_1 = ?x : \{\langle ?x, (next :: sp)^*, transport\rangle\}$, and
$\psi_2 = ?x : \{\langle ?x, next :: cityIn, France\rangle\}$

If one wants that each stop satisfies a specific constraint, e.g., cities with a population size larger than $20,000$ inhabitants, and each transportation mean belongs to Air France, i.e., its URI is in the airfrance domain name. Then this query is expressed by the following constrained regular expression:

$P = \langle ?city_1, (next :: [\psi_1]/self :: [\psi_2])^+, ?city_2\rangle$, where:
$\psi_1 = ?x : \{\langle ?x, (next :: sp)^*, transport\rangle. FILTER (regex(?x, \text{"}www.AirFrance.fr/\text{"}))\}$, and
$\psi_2 = ?x : \{\langle ?x, next :: size, ?size\rangle. FILTER (?size > 20,000)\}$

However, this query cannot be expressed by a nested regular expression, since it is not possible to apply constraints, such as SPARQL constraints, to the nodes traversed or expressed by nested regular expressions. Only navigational constraints can be expressed.

In this case, the variables $?x$ and $?size$ are not exported. Hence, the above query can be expressed by a cpSPARQL regular expression without requiring the SELECT operation. This cannot be expressed by a nested regular expression.

**Theorem 5.** *Not all constrained regular expression $R$ can be expressed as a nested regular expression $R'$ such that $[\![R]\!]_G = [\![R']\!]_G$, for every RDF graph $G$.*

*Proof.* Consider, without loss of generality, RDF graphs containing a predicate $s$ whose range is the set of integers. If one wants to select nodes which have a $s$-transition whose value is over 3, this could be expressed by the following constrained regular expression:

$$R = self :: [?s : \{\langle ?n, next :: s, ?s\rangle.FILTER(?s > 3)\}]$$

Consider a graph $G$ with two triples $\langle u, s, 2\rangle$ and $\langle v, s, 4\rangle$. The evaluation of $R$ will return $[\![R]\!]_G = \{\langle v, v\rangle\}$.

A nSPARQL nested regular expression $R'$ corresponding to $R$, should be able to select the pair $\langle v, v\rangle$ as an answer. However, the two subgraphs made of the triples in $G$ are isomorphic with respect to their structure. Hence, any nSPARQL nested regular expression retrieving one of them (a node which is the source of a $s$-edge) will retrieve both of them.

Even assuming that literals are followed and may be constrained by value, which is not the case in the current definition of nSPARQL, it would be necessary to enumerate the $s$-values larger than 3 (say $4, 5 \ldots$) to design an expression such as:

$$R' = self :: [next :: s/self :: (4|5|\ldots)]$$

However, there is an infinite number of such values and for the queries to be strictly equivalent, i.e., to provide the same answers for any graph, it is necessary to cover them all. Indeed, if one value is missing, then it is possible to create a graph $G$ for which the answers to $R$ and $R'$ do not coincide.

It is thus not possible to express a query equivalent to $R$ in nSPARQL. □

The type of counter-examples exhibited by the proof of Theorem 5 may seem caricatural. However, they illustrate the capability to apply (non navigational) constraints to values which nSPARQL lacks. Beside such a minimal example set forth for proving the theorem the same capability is used in more elaborate path queries seen in examples of previous sections (selecting path with intermediate nodes or intermediate predicates satisfying some constraints).

This capability to express constraints on values in path expressions, available in XPath as well, is invaluable for selecting exactly those paths that are useful instead of being constrained to resort to a posteriori selection. This provides interesting computational properties discussed in Section 7.

The following is another counter-example that could not be expressed as a nested regular expression.

**Example 9.** *Consider the following RDF graph representing flights belonging to different airline companies and other transportation means between cities:*

$$\{\langle city_1, airfrance : flight_1, city_2\rangle$$
$$\{\langle city_2, airfrance : flight_2, city_3\rangle$$
$$\ldots$$
$$\{\langle city_i, anothercomapny : flight_1, city_j\rangle$$

*Suppose that one wants to search pairs of cities connected by a sequence of flights belonging to airfrance company. Since there is no way to select (constrain) the transportation means in nested regular expressions, the only way the user can express such query is to list all flights belonging to airfrance company as follows:*

$$(airfrance : flight_1|...|airfrance : flight_n)^+$$

*However, this requires the user to know in advance these flights. Hence, independent of the RDF graph, the exact meaning of the above query cannot be expressed by nested regular expressions.*

**cpSPARQL can express all nSPARQL**

On the other hand, any nested regular expression $R$ could be translated to a constrained regular expression $R_1 = trans(R)$ as follows:

1. if $R$ is either `axis` or `axis`::a, then $trans(R) = R$;
2. if $R = R_1/R_2$, then $trans(R) = trans(R_1)/trans(R_2)$;
3. if $R = R_1|R_2$, then $trans(R) = trans(R_1)|trans(R_2)$;
4. if $R = (R_1)^*$, then $trans(R) = (trans(R_1))^*$;
5. if $R = exp_1 :: [exp_2]$, then $trans(R) = exp_1 :: [\psi]$, where:
   - $\psi = ?x : \{\langle ?x, trans(exp_3), p\rangle\}$, if $exp_2 = exp_3/self :: p$
   - $\psi = ?x : \{\langle ?x, trans(exp_2), ?y\rangle\}$, otherwise.

In the last clause of this transformation, when the nested regular expression $R = exp_1 :: [exp_2]$, it is required to check the existence of two pairs of nodes that satisfies the sub-expression $exp_2$ (see Definition 21). Similarly, in cpSPARQL it is necessary to express this nested regular expression as a triple in which the constraint is satisfied by the existence of a pair of nodes that replaces the variables $?x$ and $?y$.

This transformation process is illustrated by the following example.

**Example 10** (From nSPARQL to cpSPARQL). *Consider the following nested regular expression:*

$$R_1 = (next :: [(next :: sp)^*/self :: transport])^+$$

*according to the transformation rules above, the constrained regular expression equivalent to this expression $R_2$*

$$\begin{aligned}
&= trans(R_1)\\
&= trans((next :: [(next :: sp)^*/self :: transport])^+)\\
&= (trans(next :: [(next :: sp)^*/self :: transport]))^+\\
&= next :: [?x : \{\langle ?x, trans((next :: sp)^*), transport\rangle\}]\\
&= next :: [?x : \{\langle ?x, (trans(next :: sp))^*, transport\rangle\}]\\
&= next :: [?x : \{\langle ?x, (next :: sp)^*, transport\rangle\}]
\end{aligned}$$

*by successively using rules 4, 5, 4, 1, and 5.*

**Theorem 6.** *Any nested regular expression $R$ can be transformed into a constrained regular expression $trans(R)$ such that $[\![R]\!]_G = [\![trans(R)]\!]_G$, for every RDF graph G.*

*Proof.* We give in the following the induction proof outline based on the structure of nSPARQL.

- if $R$ is either `axis` or `axis`::a, then $trans(R) = R$ and thus $[[R]]_G = [[trans(R)]]_G$.
- Now assume that $[[R_1]]_G = [[trans(R_1)]]_G$ and $[[R_2]]_G = [[trans(R_2)]]_G$, then $[[R_1|R_2]]_G = [[trans(R_1)]]_G \cup [[trans(R_2)]]_G = [[trans(R_1)|trans(R_2)]]_G = [[trans(R_1|R_2)]]_G$ (based the definition of regular languages). The same case holds for the concatenation $[[R_1/R_2]]_G$ and the closure $(R_1)^*$.
- If $R = R_1 :: [R_2]$, then $trans(R) = R_1 :: [\psi]$, where $\psi = ?x : \{\langle ?x, trans(R_2), ?y\rangle\}$. Based on Definition 11, $[\![R_1 :: [R_2]]\!]_G = \{\langle x, y\rangle \mid \exists z, w \wedge \langle z, w\rangle \in [\![R_2]\!]_G\}$. If $\langle z, w\rangle \in [\![R_2]\!]_G$, then $\langle z, w\rangle \in [\![trans(R_2)]\!]_G$ by substituting $z$ and $w$ to the variables $?x$ and $?y$, respectively (Definitions 20 and 21).

□

## 7. Implementation

CPSPARQL has been implemented in order to evaluate its feasibility[3]. cpSPARQL does not exist as an independent language but is covered by CPSPARQL. This implementation has not been particularly optimized. It passes the W3C compliance tests for SPARQL 1.0 (but 5 tests involving the non implemented `DESCRIBE` clause).

Experiments have been carried out for evaluating the behavior of the system and test its ability to correctly answer SPARQL, PSPARQL, and CPSPARQL queries in reasonable time (against different RDF graph sizes from 5, 10, . . . , up to 100,000 triples in memory graphs). In particular, it showed the capability at stake here: answering SPARQL queries with the RDFS semantics.

The implementation also has been tested thoroughly in [7] and the results show that PSPARQL had better performances than other implementations of SPARQL with paths[4].

It has not been possible to us to compare the performance of our CPSPARQL implementation with other proposals. However, the experimentation has allowed to make interesting observations. Contrary to

[3]The prototype is available at `http://exmo.inria.fr/software/psparql/`.

[4]The queries and the RDF data that are used for the experimental results can be found in
`http://www.dcc.uchile.cl/~jperez/papers/www2012/`

CPSPARQL, nSPARQL is not implemented at the moment, so we must leave the experimental comparison for future work.

In particular, the CPSPARQL prototype shows that queries with constraints are answered faster than the same queries without constraints. Indeed, CPRDF constraints allow for selecting path expressions with nodes satisfying constraints while matching (on the fly instead of filtering them a posteriori). The implemented prototype follows this natural strategy, thus reducing the search space. This strategy promises to be always more efficient than a strategy which applies constraints a posteriori. More details are available in [2].

## 8. Related work

The closest work to ours, nSPARQL, has been presented and compared in detail in Section 3 [28]. However, there are other work which may be considered relevant.

RQL [19] attempts to combine the relational algebra with some special class hierarchies. It supports a form of transitive expressions over RDFS transitive properties, i.e., subPropertyOf and subClassOf, for navigating through class and property hierarchies. Versa [25], RxPath [32] are all path-based query languages for RDF that are well suited for graph traversal. SPARQLeR [20] extends SPARQL by allowing query graph patterns involving path variables. Each path variable is used to capture simple, i.e., acyclic, paths in RDF graphs, and is matched against any arbitrary composition of RDF triples between given two nodes. This extension offers functionalities like testing the length of paths and testing if a given node is in the found paths. SPARQ2L [6] also allows using path variables in graph patterns. However, these languages have not been shown to evaluate queries with respect to RDF Schema and their evaluation procedure has not been proved complete to our knowledge. Moreover, answering path queries to capture acyclic (simple) paths is NP-complete [23] (see also [7]).

Path queries (queries with regular expressions) can be translated into recursive Datalog programs over a ternary relation triple $\langle$node, predicate, node$\rangle$, which encodes the graph [1]. This could provide a way to evaluate path queries with Datalog. However, such translations may yield to a Datalog program whose evaluation does not terminate. On the other hand, several techniques can be used to optimize path queries and provide good results in comparison with optimized Datalog programs as shown in [14]. Recently, [11] extended Datalog in order to cope with querying modulo ontologies. Ontologies are in DL-Lite and, in particular DL-Lite$_{\mathcal{R}}$ which contains the fragment of RDFS considered here. However, this work only considers conjunctive queries which is not sufficient for evaluating SPARQL queries which contains constructs such as UNION, OPT and constraints (FILTER) which are not found in Datalog. [8] studied from a computational complexity the same fragments with queries containing UNION in addition. However, given that this fragment is larger than the simple path queries considered in nSPARQL and cpSPARQL, the complexity is far higher (coNP).

Standardization efforts have defined the notion of inference regime under definition by the W3C SPARQL working group [16,15]. This notion is relevant to query evaluation modulo RDFS that is exhibited by CPSPARQL and is obviously less relevant to cpSPARQL and nSPARQL. One main difference is that we have departed from the strict definition of "matching graph patterns" with the use of path for exploring the graph, and specifically the graph entailed by RDFS. This avoids the use of RDF graph closure on which strict matching is applied. CPSPARQL and nSPARQL use query rewriting for answering queries modulo RDFS, but, unlike DL-Lite rewriting strategies, the query is rewritten by preserving their structure instead of producing unions of conjunctive queries.

[13] studied the static analysis of PSPARQL query containment: determining whether, for any graph, the answers to a query are contained in those of another query. This is achieved by encoding RDF graphs as transition systems and PSPARQL queries as $\mu$-calculus formulas and then reducing the containment problem to testing satisfiability in the logic.

The language RPL extends nested regular expressions [34] to allow boolean node tests. However, using variables in nested regular expressions of nSPARQL requires extending its syntax and semantics. Hence, comparison between variables and values as well as triple patterns with variables in subject, predicate and object are not allowed (see examples in Section 6).

## 9. Conclusion

The SPARQL query language has proved to be very successful in offering access to triple stores over SPARQL endpoints all over the web. It is a critical element of the semantic web infrastructure. However, by

$$\begin{tikzcd}
\text{SPARQL} \arrow[r] \arrow[d, dotted] & \text{PSPARQL} \arrow[rr] \arrow[d, dotted] & & \text{CPSPARQL} \arrow[d, dotted] & \\
\text{SPARQL gp} \arrow[r] & \text{PSPARQL gp} \arrow[r] & \text{cpSPARQL} \arrow[r] & \text{CPSPARQL gp} & \\
& \text{nSPARQL gp} = \text{nSPARQL} \arrow[ru] & & {} \arrow[r, "\text{is extended by}"] & {} \\
& & & {} \arrow[r, dotted, "\text{uses}"] & {}
\end{tikzcd}$$

Fig. 3. Query languages and their graph patterns

limiting it to querying RDF graphs, little consideration has been made of the semantic aspect of RDF. In particular, querying RDF graphs modulo RDF Schemas or OWL ontologies is a most needed feature.

One possible approach for querying an RDFS graph $G$ in a sound and complete way is by computing the closure graph of $G$, i.e., the graph obtained by saturating $G$ with all informations that can be deduced using a set of predefined rules called RDFS rules, then evaluating the query $Q$ over the closure graph. However, this approach takes time proportional to $|Q| \times |G|^2$ in the worst case [24].

The query language nSPARQL [28] used nested regular expressions for querying RDF graphs considering RDFS semantics without the need to compute the closure graph. In this paper, we have shown that CPSPARQL [3,4] can also be used for evaluating SPARQL queries modulo RDF Schema [2].

More precisely, we showed that cpSPARQL, the fragment of CPSPARQL which is sufficient for capturing RDFS semantics, admits an efficient evaluation algorithm while the whole CPSPARQL language is in theory as efficient as SPARQL is. Moreover, we compared cpSPARQL with nSPARQL and showed that cpSPARQL is strictly more expressive than nSPARQL. PSPARQL defined in [5] and its extension CPSPARQL adopts a semantics based on checking the existence of paths (without counting them). As shown in [7], the semantics of SPARQL 1.1 specification (as of November 2011) [17], and in particular property paths, leads to intractability of the specification.

Figure 3 shows the position of the various languages. nSPARQL and cpSPARQL are good navigational languages for RDF(S). However, cpSPARQL is an extension of SPARQL graph patterns, while nSPARQL does not contain all SPARQL graph patterns. Moreover, using such a path language within the SPARQL structure allows for properly extending SPARQL. Some features (such as filtering nodes inside expressions) are very simple to add to the syntax and semantics of nested regular expressions.

In order to ease the comparison, we defined cpSPARQL as very close to nSPARQL. However, it is likely that more expressive fragments of CPSPARQL graph patterns keeping the same complexity may be found. In particular, we did not kept the capability to express the constraints existentially or universally. This may be useful, for instance, to filter families all children of which are over 18 or families one children of which is over 18.

## References

[1] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS)*, pages 122–133, New York, NY, USA, 1997. ACM.

[2] F. Alkhateeb. *Querying RDF(S) with regular expressions*. Thèse d'informatique, Université Joseph Fourier, Grenoble (FR), 2008. `ftp://ftp.inrialpes.fr/pub/exmo/thesis/thesis-alkhateeb.pdf`.

[3] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Constrained regular expressions in SPARQL. Research Report 6360, INRIA, Montbonnot (FR), 2007.

[4] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Constrained regular expressions in SPARQL. In H. Arabnia and A. Solo, editors, *Proceedings of the international conference on semantic web and web services (SWWS), Las Vegas (NV US)*, pages 91–99, 2008.

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

[6] K. Anyanwu, A. Maduko, and A. Sheth. SPARQ2L: towards support for subgraph extraction queries in RDF databases. In *Proc. 16th international conference on World Wide Web (WWW)*, pages 797–806, 2007.

[7] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 629–638, 2012.

[8] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The DL-Lite family and relations. *Journal of artificial intelligence research*, 36:1–69, 2009.

[9] J.-F. Baget. RDF entailment as a graph homomorphism. In *Proceedings of the 4th International Semantic Web Conference (ISWC'05), Galway (IE)*, pages 82–96, 2005.

[10] D. Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF schema. Recommendation, W3C, 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[11] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proc. 28th ACM Principle of Database Systems conference (PODS), Providence (RI US)*, pages 77–86, 2009.

[12] J. J. Carroll and G. Klyne. RDF concepts and abstract syntax. Recommendation, W3C, February 2004.

[13] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. Psparql query containment. In *Proceedings of the 13th International symposium on database programming languages (DBPL), Seattle (WA US)*, 2011.

[14] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 14–23, 1998.

[15] B. Glimm and M. Kroetzsch. SPARQL beyond subgraph matching. In *Proc? 9th International Semantic Web Conference (ISWC), Shanghai (CN)*, pages 59–66, 2010.

[16] B. Glimm and C. Ogbuji. SPARQL 1.1 entailment regimes. Working draft, W3C, June 2010. `http://www.w3.org/TR/sparql11-entailment`.

[17] S. Harris and A. Seaborne. SPARQL 1.1 query language. workking draft, W3C, 2010.

[18] P. Hayes. RDF semantics. Recommendation, W3C, February 2004.

[19] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *Proc. 11th International Conference on the World Wide Web (WWW), Honolulu (HA US)*, 2002.

[20] K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proceedings of 4th European Semantic Web Conferenc (ESWC'07)*, pages 145–159, 2007.

[21] F. Manola and E. Miller. RDF primer. Recommendation, W3C, 2004. `http://www.w3.org/TR/REC-rdf-syntax/`.

[22] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Recommendation, W3C, 2004. `http://www.w3.org/TR/owl-features/`.

[23] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[24] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In *Proceedings of 4th European Semantic Web Conference (ESWC), Innsbruck (AT)*, pages 53–67, 2007.

[25] M. Olson and U. Ogbuji. Versa: Path-based RDF query language, 2002. `http://copia.ogbuji.net/files/Versa.html`.

[26] J. Pan, E. Thomas, and Y. Zhao. Completeness guaranteed approximation for OWL DL query answering. In *Proc. of the 22nd International Workshop on Description Logics (DL), Oxford (UK)*, September 2009.

[27] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proc. 5th International Semantic Web Conference (ISWC), Athens (GA US)*, pages 30–43, 2006.

[28] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. In *Proceedings of the 7th International Semantic Web Conference, Karlsruhe (DE)*, pages 66–81, 2008.

[29] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010. `http://www.sciencedirect.com/science/article/B758F-4Y95V3X-1/2/9e5098d690fbe4d05a099f4c90a29a10`.

[30] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Recommendation, W3C, January 2008.

[31] E. Sirin and B. Parsia. SPARQL-DL: SPARQL query for OWL-DL. In *Proc. 3rd OWL Experiences and Directions Workshop (OWLED), Innsbruck (AT)*, 2007.

[32] A. Souzis. RxPath specification proposal, 2004. `http://rx4rdf.liminalzone.org/RxPathSpec`.

[33] M. Yannakakis. Graph-theoretic methods in database theory. In *Proc. 9th ACM Symposium on Principles of Database Systems (PODS)*, pages 230–242, 1990.

[34] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: Expressive navigation in RDF graphs. In *Proc. 4th International Conference on Web reasoning and rule systems (RR), Bressanone/Brixen (IT)*, volume 6333 of *Lecture Notes in Computer Science*, pages 251–257, 2010.