Comparison of constrained regular expressions for answering RDF-path queries modulo RDFS

Faisal Alkhateeb\textsuperscript{a,*}, and Jérôme Euzenat\textsuperscript{b}
\textsuperscript{a} Faculty of IT, Yarmouk University, Irbid, Jordan. E-mail: alkhateebf@yu.edu.jo
\textsuperscript{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 to SPARQL have been proposed to query RDF graphs considering RDFS semantics. In this paper, we discuss extensions of SPARQL that uses 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 \cite{20}) is a knowledge representation language dedicated to the description of documents and more generally of resources within the semantic web.

SPARQL 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) \cite{9} together with OWL \cite{21} are two ontology languages recommended by the W3C for defining the vocabulary used in RDF graphs. Recently, \cite{14} 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 \cite{23}.

Two main approaches can be developed for answering a SPARQL query $Q$ modulo a schema $S$ against an RDF graph $G$: 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 \cite{25}, though no hybrid approach has been developed so far for SPARQL.

There already have been proposals along the second approach \cite{27}. It consists of providing a query language, called nSPARQL, allowing for navigating graphs in the style of XPath. Then queries are rewritten so that the query evaluation navigates the data graph for taking the RDF Schema into account. Other attempts, such as SPARQL2L \cite{19} and SPARQLeR \cite{6} are not known to address queries with respect to RDF Schema. SPARQL-DL \cite{30} addresses OWL but is restricted with respect to SPARQL.
On our side, we have independently developed an extension of SPARQL, called PSPARQL [5], which adds path expressions to SPARQL. We have shown in [4] that answering SPARQL queries modulo RDF Schema could be achieved by transforming them into PSPARQL queries. PSPARQL fully preserves SPARQL, i.e., any SPARQL query is a valid PSPARQL query. The complexity of PSPARQL 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, we use an extension of PSPARQL, called CPSPARQL [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. Moreover, they are a strict extension of PSPARQL graph patterns, which have the same expressiveness as the path language retained by W3C for SPARQL 1.1 [16]. Hence, we think that the use of a proper extension of SPARQL like CPSPARQL is preferable to strict path based languages. In particular, this allows for implementing the SPARQL RDFS entailment regime.

In order to allow the comparison between 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 linear 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 it 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

The Resource Description Framework (RDF [20]) is a W3C recommended language for expressing data on the semantic web. We introduce below the syntax and the semantics (simple semantics [17]) of the language.

2.1.1. RDF syntax

RDF graphs are constructed over the set of URI references (or urirefs), blanks, and literals [11]. To simplify notations, and without loss of generality, we do not distinguish here between simple and typed literals.

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

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

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:

Paris plane Amman .
TGV subPropertyOf transport .

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

For instance, a triple \((\text{Paris}, \text{plane}, \text{Amman})\) 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 [17]. The usual notions of validity, satisfiability and consequence are entirely determined by these conditions.

**Definition 2** (RDF Interpretation). Let \(V \subseteq (U \cup \mathcal{L})\) be a vocabulary. An RDF interpretation of \(V\) is a tuple \(I = (I_R, I_P, I_{\text{EXT}}, \iota)\) such that:

- \(I_R\) is a set of resources that contains \(V \cap \mathcal{L}\);
- \(I_P \subseteq I_R\) is a set of properties;
- \(I_{\text{EXT}} : I_P \rightarrow 2^{I_R \times I_R}\) associates to each property a set of pairs of resources called the extension of the property;
- the interpretation function \(\iota : V \rightarrow I_R\) associates to each name in \(V\) a resource of \(I_R\), such that if \(v \in \mathcal{L}\), then \(\iota(v) = v\).

**Definition 3** (RDF model). Let \(V \subseteq V\) be a vocabulary, and \(G\) be an RDF graph such that \(\text{voc}(G) \subseteq V\). An RDF interpretation \(I = (I_R, I_P, I_{\text{EXT}}, \iota)\) of \(V\) is an RDF model of \(G\) if there exists a mapping \(\iota' : \mathcal{T}(G) \rightarrow I_R\) that extends \(\iota\), i.e., \(t \in V \cap \mathcal{T}(G) \Rightarrow \iota'(t) = \iota(t)\), such that for each triple \((s, p, o) \in G\), \(\iota'(p) \in I_P\) and \((\iota'(s), \iota'(o)) \in I_{\text{EXT}}(\iota'(p))\). The mapping \(\iota'\) is called a proof of \(G\) in \(I\).

Consequence (or entailment) is defined in the standard way:

**Definition 4** (RDF entailment). A graph \(G\) RDF-entails a graph \(P\) (denoted by \(G \models_{\text{RDF}} P\)) if and only if each RDF model of \(G\) is also an RDF model of \(P\).

2.2. SPARQL

SPARQL is the RDF query language developed by the W3C [29]. SPARQL query answering is characterized by defining a mapping from the query to the RDF graph to be queried.

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 GRDF graph, the union of graph patterns, an optional graph pattern, or a constraint (cf. [29] for more details).

**Definition 5** (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 (\(P \text{ AND } P'\)), (\(P \text{ UNION } P'\)), \(P \text{ OPT } P'\), and (\(P \text{ FILTER } K\)) are SPARQL graph patterns.

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

A SPARQL SELECT query is of the form SELECT \(u\) FROM \(P\) where \(u\) is the URI of an RDF graph \(G\), \(P\) is a SPARQL graph pattern and \(\mathcal{B}\) is a tuple of variables appearing in \(P\). Intuitively, such a query asks for the assignments of the variables in \(\mathcal{B}\) 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:

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

2.2.2. SPARQL semantics

In the following, we characterize query answering with SPARQL as done in [26]. The approach relies upon the correspondence between maps from RDF graph of the query graph patterns to the RDF knowledge base and GRDF entailment.
Definition 6 (Map). Let \( V_1 \subseteq T \), and \( V_2 \subseteq T \) be two sets of terms. A map from \( V_1 \) to \( V_2 \) is a mapping \( \mu : V_1 \rightarrow V_2 \) such that for all \( x \in (V_1 \cap V) \), \( \mu(x) = x \).

Operations on maps. If \( \mu \) is a map, then the domain of \( \mu \), denoted by \( \text{dom}(\mu) \), is the subset of \( T \) on which \( \mu \) is defined. The restriction of \( \mu \) to a set of terms \( X \) is defined by \( \mu|_X = \{(x, y) \in \mu | x \in X\} \) and the completion of \( \mu \) to a set of terms \( X \) is defined by \( \mu^X = \mu \cup \{\langle x, \text{null} \rangle \mid x \in X \text{ and } x \notin \text{dom}(\mu)\} \).

If \( P \) is a graph pattern, then we use \( B(P) \) 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 \( b \in B(P) \). Two maps \( \mu_1 \) and \( \mu_2 \) are compatible when \( \forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2), \mu_1(x) = \mu_2(x) \). 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 maps, then we denote by \( \mu = \mu_1 \oplus \mu_2 : T_1 \cup T_2 \rightarrow T \) the map 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 maps \( \Omega_1 \) and \( \Omega_2 \) are defined as follows [26]:

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

The answers to a basic graph pattern query are those maps 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 maps.

Definition 7 (Answers to compound graph patterns). Let \( \models \) be an 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 \( S(P, G) \) of answers to \( P \) in \( G \) is defined inductively in the following way:

\[
S(P, G) = \{\mu|_{B(P)} | G \models \mu(P)\}
\]

if \( P \) is a basic graph pattern

\[
S((P \land P'), G) = S(P, G) \bowtie S(P', G)
\]

\[
S(P \lor P', G) = S(P, G) \cup S(P', G)
\]

\[
S(P \rightarrow P', G) = (S(P, G) \bowtie S(P', G)) \cup (S(P, G) \setminus S(P', G))
\]

\[
S(P \mid K, G) = \{\mu \in S(P, G) | \mu(K) = \top\}
\]

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 \( \mu(K) \neq \top \).

As usual for this kind of query language, an answer to a query is an assignment of the distinguished variables (those variables in the \texttt{SELECT} part of the query). Such an assignment is a map 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 8 (Answers to a SPARQL query). Let \( \text{select } B \text{ from } u \text{ where } P \text{ be a SPARQL query, } G \text{ be the RDF graph identified by the URI } u \text{, and } S(P, G) \text{ be the set of answers to } P \text{ in } G \text{, then the answers } A(B, G, P) \text{ to the query are the restriction and completion to } B \text{ of answers to } P \text{ in } G \text{, i.e.,}

\[
A(B, G, P) = \{\sigma|_{B(G)} | \sigma \in S(P, G)\}.
\]

The following definition can be rewritten in a more semantic style by extending entailment to compound graph patterns modulo a map \( \sigma \).

Definition 9 (Compound graph pattern entailment). Let \( \models \) be an entailment relation on basic graph patterns, \( P, P' \) be SPARQL graph patterns, \( K \) be a SPARQL constraint, and \( G \) be an RDF graph, graph pattern entailment by an RDF graph modulo a map \( \sigma \) is defined inductively by:

\[
G \models \sigma(P \land P') \text{ iff } G \models \sigma(P) \text{ and } G \models \sigma(P')
\]

\[
G \models \sigma(P \lor P') \text{ iff } G \models \sigma(P) \text{ or } G \models \sigma(P')
\]

\[
G \models \sigma(P \rightarrow P') \text{ iff } G \models \sigma(P) \text{ and } [G \models \sigma(P') \text{ or } \forall \sigma' ; G \models \sigma'(P'), \sigma \perp \sigma']
\]

\[
G \models \sigma(P \mid K) \text{ iff } G \models \sigma(P) \text{ and } \sigma(K) = \top
\]

3. \text{nSPARQL}

nSPARQL is a query language that uses nested regular expressions in predicate position of graph patterns for navigating the RDF graph [27].
3.1. nSPARQL syntax

**Definition 10** (Regular expression). A regular expression is an expression built from the following grammar:

\[
exp ::= \text{axis} \mid \text{axis}:a \mid \text{exp}\exp \mid \exp\exp \mid \exp *
\]

with \(a \in \mathcal{U}\) and \(\text{axis} \in \{\text{self}, \text{next}, \text{next}^{-1}, \text{edge}, \text{edge}^{-1}, \text{node}, \text{node}^{-1}\}\).

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, axis denote the type of node object which is selected at each step, respectively, the current node (\(\text{self}\) or \(\text{self}^{-1}\)), the nodes reachable through an outbound triple (\(\text{next}\)), the nodes that can reach the current node through an incident triple (\(\text{next}^{-1}\)), the properties of incident triples (\(\text{edge}\)), the properties of outbound triples (\(\text{edge}^{-1}\)), the object of a predicate (\(\text{node}\)) and the predicate of an object (\(\text{node}^{-1}\)). This is illustrated by Figure 2.

![Fig. 2. nSPARQL axis.](image)

**Definition 11** (Nested regular expression). A nested regular expression is an expression built from the following grammar:

\[
exp ::= \text{axis} \mid \text{axis}:a \mid \text{axis}::[\exp] \mid \exp \mid \exp\exp \mid \exp *
\]

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 particular in predicate position, to define nSPARQL triple patterns.

**Definition 12** (nSPARQL triple pattern). An nSPARQL triple pattern is a triple \((s, p, o)\) 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:

\[P = (?\text{city}_1,(\text{next} :: [(\text{next} :: \text{sp}^*)/\text{self} :: \text{transport}]^*)+,?\text{city}_2)\]

nSPARQL is rather 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 13** (nSPARQL graph pattern). An nSPARQL graph pattern is defined inductively by:

\[- \text{every nSPARQL triple pattern is an nSPARQL graph pattern;}
\]
\[- \text{if } P_1 \text{ and } P_2 \text{ are two nSPARQL graph patterns and } K \text{ is a SPARQL constraint, then } (P_1 \text{ AND } P_2), (P_1 \text{ UNION } P_2), (P_1 \text{ OPT } P_2), \text{ and } (P_1 \text{ FILTER } K) \text{ are nSPARQL graph patterns.}
\]

However, for theoretical complexity reasons the designers of the nSPARQL language choose to define a more restricted language than SPARQL [28]. 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 [27].

**Definition 14** (Nested path interpretation). Given a nested path \(p\) and an RDF graph \(G\), the interpretation
The evaluation of a nested regular expression

\[ G = \{ y | \langle X, Y \rangle \in [R]_G \} \]

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

The evaluation of such basic graph patterns is measured with the usual evaluation problem:

**Problem:** Evaluation problem for regular expressions

**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 [28].

**Theorem 1** (Complexity of nSPARQL evaluation [28]). The evaluation problem for a nested regular expression \( R \) over an RDF graph \( G \) can be solved in time \( O(\|G\| |R|) \).

Clearly, nSPARQL is a good navigational language, but there still are useful queries that could not be expressed. For example, it cannot be used to find nodes connected with transportation mean that is not a bus or transportation means belonging to Air France, i.e., containing the URI of the company.

### 3.3. Querying RDFS with nSPARQL

[23] has introduced the reflexive relaxed semantics for RDFS in which rdfs:subPropertyOf and rdfs:subClassOf 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 reflexivity 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 in the predicate position.

However, when issuing queries involving these relations, e.g., with a graph pattern like \( \langle ?x \text{ sp } ?y \rangle \), all properties in the graph will be answers. Since this
would clutter results, we assume, as done in [23], 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 $\text{closure}(G)$, which is defined by the graph obtained by saturating $G$ with all triples that can be deduced using rules of Table 1 [23].

**Definition 17.** The evaluation of an nSPARQL triple pattern $t = \langle X, R, Y \rangle$ over an RDF graph $G$ modulo RDFS is defined as the following set of maps:

$$\langle t \rangle^\text{RDFS} = \{ \mu | \text{dom}(\mu) = \{ X, Y \} \cap B$$

$$\land \langle X, \mu(Y) \rangle \in [R]_{\text{closure}(G)} \}$$

**Definition 18** (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:

$$\mathcal{A}^P(G, P) = \{ \mu | (\mu(X), \mu(Y)) \in [R]_{G}^{\text{RDFS}}$$

$$\land \forall \langle X, R, Y \rangle \in P \}$$

As presented in [27], nSPARQL can evaluate queries with regard to RDFS by transforming the queries with rules [23]:

- $\phi(\text{sc}) = (\text{next}::\text{sc})^*$
- $\phi(\text{sp}) = (\text{next}::\text{sp})^*$
- $\phi(\text{dom}) = \text{next}::\text{dom}$
- $\phi(\text{range}) = \text{next}::\text{range}$
- $\phi(\text{type}) = \text{next}::\text{type}/\text{next}::\text{sc}^*$
  - $|\text{edge}/\text{next}::\text{sp}*/\text{next}::\text{dom}/\text{next}::\text{sc}^*$
  - $|\text{node}^{-1}/\text{next}::\text{sp}*/\text{next}::\text{range}/\text{next}::\text{sc}^*$
- $\phi(p) = \text{next}[(\text{next}::\text{sp})/\text{self}::\text{sp}] \ (p \not\in \rho_{\text{df}})$

**Example 4** (nSPARQL query 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 from France and the other from Jordan:

$$\{(?\text{city}_1, (\text{next} :: \text{transport})^*, (?\text{city}_2)$$

$$\text{(??city} _1, \text{next :: cityIn,France})$$

$$\text{(??city}_2, \text{next :: cityIn, Jordan})\}$$

When evaluating this graph pattern against the RDF graph of Figure 1, it returns empty set in the time it must return the following set of pairs:

$$\{(??\text{city}_1 \leftarrow \text{Paris}, \text{??city}_2 \leftarrow \text{Amman}), (??\text{city}_1 \leftarrow \text{Grenoble}, ??\text{city}_2 \leftarrow \text{Amman})\}$$

To do so, the above graph pattern could be transformed to the following nSPARQL graph pattern:

$$\{(??\text{city}_1, (\text{next} :: (\text{next} :: \text{sp})^*/\text{self} :: \text{transport})^*, (?\text{city}_2)$$

$$\text{(??city} _1, \text{next :: cityIn, France})$$

$$\text{(??city}_2, \text{next :: cityIn, Jordan})\}$$

This encoding is correct and complete with regard to entailment.

**Theorem 2** (Completeness of $\phi$ [27] (Theorem 3)). Let $\langle X, p, Y \rangle$ be a SPARQL triple pattern with $X, Y \in (U \cup B)$ and $p \in U$, then $\langle X, p, Y \rangle^{\text{RDFS}} = \langle X, (\phi(p), Y) \rangle^G$ for any RDF graph $G$.

4. **CPSPARQL and cpSPARQL: syntax and semantics**

In this section, we present CPSPARQL and cpSPARQL. 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.
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, we adopt the axis borrowed from XPath, with which the reader may be more familiar, as done for nSPARQL. This also will allow us to better compare cpSPARQL and nSPARQL. Additionally, in the original proposal, ALL and EXISTS keywords are 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 RDFS semantics, i.e., the fragment still captures 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 19 (Constrained regular expression). A constrained regular expression is an expression built from the following grammar:

\[ \text{exp} ::= \text{axis} \mid \text{axis}::a \mid \text{axis}::[x : \psi] \mid \text{axis}::]x : \psi[ \mid \text{exp} \mid \text{exp}\text{exp} \mid \text{exp}\text{exp} \mid \text{exp}* \]

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

Constrained regular expressions allow for constraining 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 map) 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.

The constraints used to define constrained regular expressions involve constrained regular expressions. This will allow nesting of constraints. That is, a constrained regular expression may contain a constraint which in turn uses constrained regular expressions in its graph pattern as done in the following example.

Example 5 (Constrained regular expression). The following constrained regular expression:

\[ (\text{next} :: [?p : \{(?p,(\text{next} :: sp)^*,\text{transport}) \mid \text{Filter}(?pl = \text{bus})\}])^+ \]

Could be used to find nodes connected by transportation means that is not a 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 particular in predicate position, to define CPSPARQL.

Definition 20 (CPSPARQL triple pattern). A CPSPARQL triple pattern is a triple \( (s,p,o) \) such that \( s \in T \), \( o \in T \) and \( p \) is a constrained regular expression.

Definition 21 (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 \((P_1 \cup P_2),(P_1 \cup P_2),(P_1 \cup P_2),(P_1 \cup P_2)\) and \((P_1 \cup P_2),(P_1 \cup P_2)\) are CPSPARQL graph patterns.

Example 6 (CPSPARQL graph pattern). The following CPSPARQL graph pattern:

\[ \{(?\text{city}_1, (\text{next} :: [?p : \{(?p,(\text{next} :: sp)^*,\text{transport}) \mid \text{Filter}(?pl = \text{bus})\}])^+, ?\text{city}_2) \]

\[ (?\text{city}_1, \text{next} :: \text{cityIn}, \text{France}) \]

\[ (?\text{city}_2, \text{next} :: \text{cityIn}, \text{Jordan}) \]

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. If open square brackets were used, this graph pattern would, in addition, provide the value of the \(?p\) variable, 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 cpSPARQL regular expression as predicate)
Definition 22 (cpSPARQL regular expression). A cpSPARQL regular expression is an expression built from the following grammar:

\[ \text{exp} ::= \text{axis} | \text{axis}:a | \text{axis}::\text{?x} : \text{TRUE} \]

\[ | \text{axis}::\text{?x} : \{\{\text{?x,exp,v}\}\{\text{FILTER}\{\text{?x}\}\}} \]

\[ | \text{exp}\text{exp}\exp | \exp\exp | \exp* \]

such that \( v \) is either \( \text{?x} \) or a constant (an element of \( U \cup L \)).

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 \( \text{?x} \) as the only variable and/or a SPARQL \text{FILTER} constraint.

Deciding if a CPSPARQL triple is a cpSPARQL triple can be decided 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:

\( \{\text{?city1}, \{\text{next}::\{\text{?p, (next :: sp)}*, \text{transport}\}\}\}^+ \)

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

Example 5 provides a cpSPARQL regular expression that is out of reach of nSPARQL since it is not possible to apply constraints, such as SPARQL constraints, in the nodes traversed or expressed by nested regular expressions. Only navigational constraints can be expressed. By contrast, CPSPARQL graph patterns allow for queries like:

\( \{\text{next}::\{\text{?p, (next :: sp)}*, \text{?z}\}, \{\text{?q, (next :: sp)}*, \text{?z}\}, \{\text{?p, owl :: inverseOf}\, \text{?q}\}, \text{Filter}(\text{regex}\{\text{?z, iata.org}\})\} \)

which is not a cpSPARQL regular expression since it uses free variables (here \( \text{?z} \) and \( \text{?q} \)).

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 \( \text{next}::[\psi] \) (where \( \psi = \{\text{?p} : \{\{\text{?p, sp}\}, \text{transport}\}\} \) is equivalent to \( \text{next}::p \) if \( p \) satisfies the constraint \( \psi \). That is, \( p \) should be a sub-property of \( \text{transport} \) (when \( p \) is substituted to the variable \( \text{?p} \)).

Definition 23 (Satisfied constraint in an RDF graph). Let \( G \) be an RDF graph, \( s \) a term of \( G \) and \( \psi = x : C \) be a constraint, then \( s \) satisfies \( \psi \) in \( G \) (denoted \( s \in \{\psi\}_G \)) if one of the following conditions is satisfied:

- \( C \) is a graph, and there exists \( \mu \in S(C,G) \) such that \( \mu(x) = s \);
- \( C \) is a SPARQL constraint and \( C^*_s \) is \( \top \) (where \( C^*_s \) denotes the constraint obtained by the substitution of \( s \) to each occurrence of the variable \( x \) in \( C \)).

As done 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 14 to take into account the semantics of terms with constraints.

Definition 24 (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 14; otherwise the interpretation of \( P \) in \( G \) is defined as:

\[ \{\text{self} :: [\psi]_G = \{\langle x, x \rangle \mid x \in \text{voc}(G) \land x \in \{\psi\}_G\} \}
\[ \{\text{next} :: [\psi]_G = \{\langle x, y \rangle \mid \exists z : \langle x, z, y \rangle \in G \land z \in \{\psi\}_G\} \}
\[ \{\text{edge} :: [\psi]_G = \{\langle x, y \rangle \mid \exists z : \langle x, y, z \rangle \in G \land z \in \{\psi\}_G\} \}
\[ \{\text{node} :: [\psi]_G = \{\langle x, y \rangle \mid \exists z : \langle x, y, z \rangle \in G \land z \in \{\psi\}_G\} \}
\[ \{\text{axis}^{-1} :: [\psi]_G = \{\langle x, y \rangle \mid \langle y, x \rangle \in \{\text{axis} :: [\psi]\}_G\} \}

Definition 25 (Answer to a CPSPARQL triple pattern). The evaluation of a CPSPARQL triple pattern \( t = (X, R, Y) \) over an RDF graph \( G \) is defined as the following set of maps:

\[ [t]_G = \{\mu : \text{dom}(\mu) = \{X, Y\} \cap B \cup B(R) \land \langle \mu(X), \mu(Y) \rangle \in [\mu(R)]_G \} \]
Of course, this semantics applies to cpSPARQL graph patterns. Note that $\mathcal{B}(R)$ is the set of variables occurring as the head variable of an open bracket constraint in $R$.

4.3. Querying RDFS with CPSPARQL

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

**Definition 26 (RDFS triple pattern expansion).** Given an RDF triple $t$, the RDFS expansion of $t$, denoted by $\tau(t)$, is defined as:

- $\tau((s, sc, o)) = \langle s, next::sc^+, o \rangle$
- $\tau((s, sp, o)) = \langle s, next::sp^+, o \rangle$
- $\tau((s, dom, o)) = \langle s, next::dom, o \rangle$
- $\tau((s, range, o)) = \langle s, next::range, o \rangle$
- $\tau((s, type, o)) = \langle s, next::type/next::sc^1\rangle$
- $\tau((s, p, o)) = \langle s, (next::[?x: (?x, (next::sp)^*, p)]) \rangle$
  \[ p \notin \{sp, sc, type, dom, range\} \]

It is clear that 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 $\text{dom}(\mu)$, i.e., only variables occurring as a subject or an object in a CPSPARQL triple pattern are considered in maps (see Definition 25). Therefore, the SELECT operator (projection) is not needed 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
```

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

```
SELECT ?X ?Y
```

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 an extra variable "?P" is used in the transformation. It is argued in [28] 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 and thus 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.

**Theorem 3.** Let $\langle X, p, Y \rangle$ be a SPARQL triple pattern with $X, Y \in (U \cup B)$ and $p \in P$, then $\llangle (X, p, Y) \rrangle_G^{df}$ is $\llangle (X, \tau(p), Y) \rrangle_G$ 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 [27]. That is $\langle \mu(X), \mu(Y) \rangle \llangle (X, p, Y) \rrangle_G^{df}$ iff $\langle \mu(X), \mu(Y) \rangle \llangle (X, \tau(p), Y) \rrangle_G$.

- $\implies$ Suppose that $\langle \mu(X), \mu(Y) \rangle \llangle (X, p, Y) \rrangle_G^{df}$. In this case, there exists $p_1$ such that $\langle p_1, sp, ... sp, p_n = p \rangle$ and $\langle \mu(X), \mu(Y) \rangle \llangle (X, p_1, Y) \rrangle_G$ as well as $\langle \mu(X), \mu(Y) \rangle \llangle (X, next::p_1, Y) \rrangle_G$. Let us consider now the transformed triple $\tau(t) = \langle X, \psi \rangle$ (where $\psi = \llangle (?p, (next::sp)^*, p) \rrangle$). The maps for the variable $?p$ will be $\llangle (?p, p) \rrangle$ for $i = 1, ..., n$ (since $\llangle \psi \rrangle_G = \llangle (p_1, p) \rrangle = \llangle p_1, p \rrangle$).

Now according to Definitions 25 and 24, $\langle \mu(X), \mu(Y) \rangle \llangle (X, \psi) \rrangle_G^{df}$ iff $\langle \mu(X), \mu(Y) \rangle \llangle (X, \tau(p), Y) \rrangle_G^{df}$, and this condition holds.

- $\impliedby$ We have to prove that if $\langle \mu(X), \mu(Y) \rangle \llangle (X, \psi) \rrangle_G$, then $\langle \mu(X), \mu(Y) \rangle \llangle (X, p, Y) \rrangle_G^{df}$. Suppose that $\langle \mu(X), \mu(Y) \rangle \llangle (X, \psi) \rrangle_G$. In this case, there exists $p_1$ such that $\langle \mu(X), (next::p_1, Y) \rangle \llangle (X, \psi) \rrangle_G$ and $p_1 \llangle \psi \rrangle_G$, that is $\langle p_1, next::sp, p_2 \rangle, ..., \langle p_{n-1}, next::sp, p_n = p \rangle \llangle \psi \rrangle_G$. Therefore,
\[ (\mu(\mathcal{X}), \mu(\mathcal{Y})) \in \{(\mathcal{X}, p, \mathcal{Y})\}^{dfs}_G \text{ since } \langle p_1, (\text{next}::\text{sp})^* \rangle \text{ and } \langle \mu(\mathcal{X}), \text{next}::p_1, \mu(\mathcal{Y}) \rangle \in G. \]

5. Complexity of evaluating cpSPARQL

The complexity of cpSPARQL is given with respect to the following problem:

Problem: Evaluation problem for cpSPARQL regular expressions

Input: An RDF graph \( G \), a cpSPARQL regular expression \( R \), and a pair \( (a, b) \)

Question: Does \( (a, b) \in [R]_G \)?

We follow [27] to store an RDF graph as an adjacency list. That is, every \( u \in \text{voc}(G) \) is associated with a list of pairs \( \alpha(u) \). For instance, if \( (s, p, o) \in G \), then \( (\text{next}::p, o) \in \alpha(s) \) and \( (\text{edge}^{-1}::o, s) \in \alpha(p) \). Also, \( (\text{self}::u, u) \in \alpha(u) \), for \( u \in \text{voc}(G) \). The set of terms of a constrained regular expression \( R \), denoted by \( \mathcal{T}(R) \), is constructed as follows:

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

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

\[ \mathcal{T}(R_1^*) = \mathcal{T}(R_1) \]

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

\[ - \langle u, q \rangle \in \text{voc}(G) \times Q, \text{ for every } u \in \text{voc}(G) \text{ and } q \in Q; \]

\[ - \langle v, q \rangle \in \delta'((u, p), s) \text{ iff } q \in \delta(p, s); \text{ and one of the following conditions satisfied:} \]

\[ * s = \text{axis} \text{ and there exists } a \text{ s.t. } \langle \text{axis}::a, v \rangle \in \alpha(u) \]

\[ * s = \text{axis}::a \text{ and } \langle \text{axis}::a, v \rangle \in \alpha(u) \]

\[ * s = \text{axis}::\psi \text{ and there exists } b \text{ s.t. } \langle \text{axis}::b, v \rangle \in \alpha(u) \text{ and } b \in [v]_G \]

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 and \( (a, b) \) be a pair of nodes. Let \( \{\psi_1, \ldots, \psi_n\} \) be the set of constraints of \( R \) with \( (\psi_i = x_i : C) \), and \( w_{\psi_i} = [\psi_i]_G = \{n_1, \ldots, n_m\} \) be the set of nodes satisfying \( \psi_i \) in \( G \). Now, for each \( n \in [\psi_i]_G \), we label the node \( n \) of \( G \) with a pair \( \langle \psi_i, \text{true} \rangle; \langle \psi_i, \text{false} \rangle \), otherwise. This way, once the constraints are evaluated and the nodes are labeled as described above, checking if a node \( n \) satisfies a constraint \( \psi_i \) can be done in \( O(1) \). Note that also constructing the automaton of \( R \) can be done in \( \text{NLOGSPACE} \) as in the usual automaton [32,22] (with the alphabet described in the text above). So, constructing the product automaton \( G \times \mathcal{A}_R \) can be done in time \( O(|G|, |R|) \). Hence, checking if the pair \( (a, b) \in [R]_G \) is equivalent to checking if the language accepted by \( (G \times \mathcal{A}_R) \) is not empty, which can be done in \( O(|G|, |R|) \) (as in the case of usual regular expressions [32,22]). Moreover, the evaluation of each \( \psi_i \) over an RDF graph \( G \) can be done once (either a priori or once the constraint is reached for the first time) and in time \( O(|G|, |\psi_i|) \) since each \( \psi_i \) is itself a constraint with a triple containing a cpSPARQL constrained regular expression and/or a SPARQL constraint (see Definition 22).

An efficient algorithm that solves the evaluation problem for nested regular expressions \( R \) over an RDF graph \( G \) in time \( O(|G|, |R|) \) is presented in [27]. So, both constrained and nested regular expressions admit an efficient algorithm.
6. On the expressiveness of cpSPARQL and nSPARQL

In this section, we compare the expressiveness of cpSPARQL with nSPARQL. In particular, we show that:

1. nSPARQL patterns (i.e., nested regular expressions) can be transformed to an equivalent cpSPARQL patterns.
2. there exists a cpSPARQL regular expression that cannot be expressed in a nested regular expression.
3. for some queries, which can be expressed by CPSPARQL, the projection operator (the SELECT operator) is required to be expressed in nSPARQL.
4. some natural and useful queries that can be expressed in CPSPARQL patterns and cannot expressed in nSPARQL patterns even with the SELECT operator.

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

\[
\{(?person_1,(next::ascendant)^+) / (\text{next}^{-1}::\text{ascendant})^+, ?\text{person}_2), \\
\text{Filter}(!(?\text{person}_1 = ?\text{person}_2)) \}
\]

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:

\[
\{(?\text{person}_1,(next::\text{ascendant})^+, ?\text{ancestor}), \\
(?\text{person}_2,(next::\text{ascendant})^+, ?\text{ancestor}), \\
\text{Filter}(!(?\text{person}_1 = ?\text{person}_2)) \\
\& \& (\text{regex}(?\text{ancestor}, "\text{alkhateeb}")\})
\]

Notice that the evaluation of this graph pattern is the mapping \{?\text{person}1 \leftarrow p1, ?\text{ancestor} \leftarrow p3, ?\text{person}2 \leftarrow p2\}. Therefore, to obtain the desired result, the projection operator must be performed:

\[
\sigma_{?\text{person}_1,?\text{person}_2}( \\
\{(?\text{person}_1,(next::\text{ascendant})^+, ?\text{ancestor}), \\
(?\text{person}_2,(next::\text{ascendant})^+, ?\text{ancestor}), \\
\text{Filter}(!(?\text{person}_1 = ?\text{person}_2) \\
\& \& (\text{regex}(?\text{ancestor}, "\text{alkhateeb}")\})
\}
\]

So, the above query could not be expressed in nSPARQL without the use of SELECT, which is not allowed in nSPARQL [28]. Besides, any SPARQL query that use the 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:

\[
\{(?\text{person}_1,(next::\text{ascendant})^+) / \text{self}::(?\text{ancestor} : \\
\text{Filter}(\text{regex}(?\text{ancestor}, "\text{alkhateeb}"))) \\
(\text{next}^{-1}::\text{ascendant})^+, ?\text{person}_2), \\
\text{Filter}(!(!(?\text{person}_1 = ?\text{person}_2))\})
\]

Additionally, 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:

\[
(\text{city}_1, (next :: [(\text{next} :: \text{sp})* / \text{self} :: \text{transport}] / \text{self} :: [next :: \text{cityIn} / \text{self} :: \text{France}])^{+}, \text{city}_2)
\]

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

\[
(\text{city}_1, (next :: [\psi_1] / \text{self} :: [\psi_2])^{+}, \text{city}_2)
\]

where:

\[
\psi_1 = x : \{(?x, (next :: \text{sp})^{+}, \text{transport})\}, \text{and} \psi_2 = x : \{(?x, next :: \text{cityIn}, \text{France})\}
\]

Now, if one wants that each stop satisfies a specified constraint (as a simple example, cities with a population size larger than 20,000 inhabitants) and each transportation mean belongs to Air France (i.e., contains the URL name-space of Air France company). Then this query is expressed by the following constrained regular expression:

\[
P = (\text{city}_1, (next :: [\psi_1] / \text{self} :: [\psi_2])^{+}, \text{city}_2),
\]

where:

\[
\psi_1 = x : \{(?x, (next :: \text{sp})^{+}, \text{transport}). \text{FILTER}(\text{regex}(x,"www.AirFrance.fr")\}, \text{and} \psi_2 = x : \{(?x, next :: \text{size}, \text{size}). \text{FILTER}(\text{size} > 20,000)\}
\]

However, this query could not be expressed by a nested regular expression, since it is not possible to apply constraints, such as SPARQL constraints, in the nodes traversed or expressed by nested regular expressions. Only navigational constraints can be expressed. Additionally, \psi_2 could be rewritten with a FILTER expression:

\[
\psi_2 = x : (?x, next :: size[?size : \text{FILTER}(?size > 20,000)]/\text{next}^{-1} :: \text{size}, ?x).
\]
In this case, the ?size variable is not exported. Hence, the above query could be expressed by a cpSPARQL regular expression without requiring the SELECT operation. This cannot be expressed by a nested regular expression.

**Corollary 1.** There exists a constrained regular expression $R_1$ such that there is no nested regular expression $R_2$ with $[R_1]_G = [R_2]_G$ for every RDF graph $G$.

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

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

The last line of the expression is only here because the cpSPARQL language is very close to nSPARQL: it cannot introduce a new variable beside $?x$. However, authorizing a variable as object of the predicate would not change the complexity of the language and would allow to get rid of the extra reverse path.

This transformation process is illustrated by the following example.

**Example 9** (From nSPARQL to cpSPARQL). Consider the following nested regular expression:

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

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

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

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

**7. Implementation**

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

Experiments have been carried out for evaluating the behaviour of the system and test its ability to correctly answer SPARQL, PSPARQL, and CPSPARQL queries in reasonable time (against in memory graphs). In particular, it showed the capability at stake here: answering SPARQL queries with the RDFS semantics.

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. 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 while matching (on the fly and not a posterior filtering for paths) path expressions with nodes satisfying constraints. 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].

The implementation also has been tested thoroughly in [7] and the results show good performance of PSPARQL with respect to some other implementations of SPARQL.

**8. Related work**

The closest work to ours, nSPARQL, has been presented and compared in detail in Section 3 [27]. However, there are other work which may be considered relevant.

RQL [18] attempt to combine the relational algebra with some special class hierarchies. It supports a form of transitive expressions over RDFS transitive prop-

---

1The prototype is available at http://exmo.inrialpes.fr/software/psparql.

2The queries and the RDF data that are used for the experimental results can be found in http://www.dcc.uchile.cl/~jperez/papers/www2012/
Path queries (queries with regular expressions) can be translated into recursive Datalog programs over a ternary relation triple (node, predicate, node), 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 [13]. Recently [10] extended Datalog in order to cope with querying modulo ontologies. Ontologies are in DL-Lite and, in particular DL-LiteR 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 [15,14]. 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 to use an RDF graph closure on which strict matching can applied. CPSPARQL and nSPARQL use query rewriting for answering queries modulo RDFS, but, unlike DL-Lite rewriting strategies, the query are rewritten by preserving their structure instead of producing unions of conjunctive queries.

[12] study 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 μ-calculus formulas and then reducing the containment problem to testing satisfiability in the logic.

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

One possible approach for querying an RDF 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 over the closure graph. However, this approach takes time proportional to |H| × |G|^2 in the worst case [23].

The query language nSPARQL [27] 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 in in theory as efficient as SPARQL is. Moreover, we compared cpSPARQL with nSPARQL and showed that cpSPARQL is strictly more expressive than nSPARQL. It should be noticed that 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 Novem-
Figure 3 shows the position of the various languages. With equivalent complexity, cpSPARQL is arguably a better language than nSPARQL for expressing path queries because it 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.

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.

References


