Response letter of the first round of the review

September 19, 2012

Ms. Ref. No.: 12-205
Old title of the paper: Comparison of constrained regular expressions for answering RDF-path queries modulo RDFS
New title of the paper: Constrained regular expressions for answering RDF-path queries modulo RDFS
Authors: Faisal Alkhateeb and Jérôme Euzenat

First of all, we would like to thank the editor and the reviewers for their careful reading of the paper and for providing many useful comments.

We have taken into account all the comments from the reviewers. We treated the typos and the minor comments as well as the inconsistencies and we provide in the following the changes that we have made to address the reviewers’ major comments. It occurs that the main problems pointed out by the reviewers are: i) the proof of theorem 4 is not complete; ii) weaken claims about non-expressiveness or provide proofs; iii) explain the novelty of theorem 3; iv) consider changing the title, since it does not seem that language comparison is the main focus. So, the main answers and the modifications are concentrated on these comments. In particular, we have mainly made the following in the revised version:

- We have totally revised the proof of theorem 4.
- We have weaken some claims and provide the proof of corollary 1 (now it is Theorem 5).
- We have explained the novelty of Theorem 3.
- We have changed the title of the paper

Review 1 by Jorge Pérez
1A) Theorem 3 seems to follow almost directly from the results in [27,28]. If it is the case, the authors should explicitly mention this fact and make the proof shorter. If it is not the case, the authors should clearly explain what is the major difference between the two proofs. Moreover, Algorithm 1 seems to be almost exactly the algorithm proposed in [27,28] for a similar purpose (regarding nested regular expressions). The authors should also mention this connection if the connection exists, and if not, they should clearly explain what the main differences are.

Regarding Theorem 3:
It has been presented in [Alkhateeb, 2008] that PSPARQL and CPSPARQL can encode the core RDFS semantics but without the proof. Moreover, it is shown in [27,28] that this encoding, and in particular using PSPARQL, requires the use of the projection operator (SPARQL SELECT clause) and thus makes the PSPARQL query evaluation an NP-hard problem. This Theorem gives the proof that CPSPARQL can encode the core RDFS semantics but without the need to the projection operator.

We agree with the referee that the proof follows from the results in [27,28] except for the last step. So, we have added a paragraph before Theorem 3 explaining this connection.

Also, we have added a paragraph before Algorithm 1 about its connection with the one presented in [27,28] used for evaluating nested regular expressions.

1B) Regarding the main complexity result in Theorem 4, I think that the proof is incomplete and, as currently presented, cannot be considered as enough evidence to prove the correctness of the result. When defining cpSPARQL, the authors allow for CPRDF-constraints expressions that mention the same variable (Def 22). For example, an expression of the form self::(?x : { <?x, R, ?x> }) with R a regular expression, is a valid cpSPARQL expression. In the proof of Theorem 4, the authors say that they first label every node in G according to whether they satisfy "psi" or not, in this case, whether they satisfy ?x:{ <?x, R, ?x> }. Notice that this step is used to construct the product graph (GxA in the proof). The authors claim that this construction can be done in linear time (\(|G|\times|R|\)), by just saying that the automaton for R can be constructed in NLOGSPACE. I agree that an automaton for R can be efficiently constructed (linear time), but it is not clear to me whether this automaton can be used to label in linear time the nodes "n" such that there is a path satisfying R from "n" to "n". In fact, I think that it cannot be done in time linear w.r.t. \(|G|\) although I do not have a formal proof of this and haven’t found a reference. In any case, the automaton \(A_R\) cannot be used in the standard way to label nodes satisfying the expression <?x, R, ?x> so the proof of Theorem
4 is at least incomplete. (Notice that the straightforward way of labeling would need time \(|G|^2\): for every node "n" in G check whether a node \((n, q_0)\) can reach a node \((n, q_f)\) in \(G \times A_R\), with \(q_0\) the initial state of \(A_R\) and \(q_f\) a final state in \(A_R\). Since this should be repeated for every node in G one obtains total time \(|G| \cdot |G| \cdot |R|\). The proof of a similar result regarding nested regular expressions presented in [27,28] does not suffer from this problem since variables are not allowed inside nested regular expressions, and thus, one never needs to check for the existence of path from a node "n" to the same "n". As proved in [27,28] for the case where there are no variables the labeling can actually be accomplished in linear time.

We have totally revised and completed the proof of Theorem 4.

1C) Related to the above comment, notice that the translation from nested regular expressions to constrained regular expressions presented after Corollary 1 uses a repeated variable. So the flaw in the proof of Theorem 4 somehow invalidates two of the most important results in the paper: linear time complexity of cpSPARQL regular expressions and that cpSPARQL regular expressions are at least as expressive as nested regular expressions (as this result makes sense in the story only if cpSPARQL expressions can be evaluated as efficiently as nested regular expressions). In any case, I think that the use of the repeated variable is not strictly needed when encoding nested regular expressions as cpSPARQL expressions and thus the result can actually be adapted to hold.

The use of repeated variables in constrained regular expressions does not change the time complexity of evaluation constrained regular expressions (see the proof of Theorem 4 in the revised version).

1D) Finally, and as one of the mayor weak points of the paper is the lack of formal proofs when comparing the expressiveness of the languages studied. When the authors compare the expressiveness of both languages they should provide formal proofs of inexpressibility. Notice that as stated in the title this is the main topic of the paper, and thus not having these proofs is a mayor drawback. In particular, Corollary 1 does not follow from any of the results in the paper. I would advice the authors to restate Corollary 1 as a Theorem and provide a formal proof of inexpressibility. Moreover, I would also suggest the authors to focus more on the expressiveness of the languages, and, in particular, on how the two languages (cpSPARQL and nSPARQL) compare to each other for different fragments. For example, an interesting comparison would be to consider cpSPARQL regular expressions versus
full nSPARQL queries. That is, can (at least some features of) cpSPARQL expressions be simulated with nested regular expressions plus SPARQL operators. If the answer is "no" then it would be an interesting argument in favor of cpSPARQL expressions.

We have restated Corollary 1 as a Theorem and provide a formal proof of inexpressibility by providing a counter-example that could not be expressed in nSPARQL.

1E) I think that Definition 2 are 3 are not needed and only complicates things not adding any significant to the paper. I would advice the authors to better provide a characterization of entailment of RDF graphs in terms of subset for the case of graphs without bnodes/variables, and in terms of homomorphism when the graphs have bnodes/variables. We have deleted Def 2 and 3 and provided characterization of entailment of RDF graphs in terms of subset for the case of graphs without bnodes/variables, and in terms of homomorphism when the graphs have bnodes/variables in Definition 4 (now it is Definition 3).

1F) Notice that interpretation (Def 2) and model (Def 3) are defined for RDF graphs (not general GRDF graphs). Later in Definition 4 the notion of $|=_{R} P$ is used for a general graph $P$, and thus Definition 4 is underspecified. Everything would be more clear if just characterizations for entailment are provided. See the answer to 1E.

1G) Is Definition 9 really needed for the main goal of the paper? It seems that it is not used anymore in the rest of the paper. If that is the case, I would advice the authors to not include it. It is deleted.

1H) Example 4 should be written a bit more carefully. In particular, when it says "it must return the following set of pairs:" it is not clear why it must return this set. The authors should be more clear stating that they are trying to evaluate the pattern according to RDFS semantics, so the reader can realize the importance of the example. The final sentence of the example says "is correct and complete with regard to entailment". Please say what "entailment" you mean here. We have made this example more precise.
1I) Definition 19: it says that "psi a set of triples *and* FILTER-expressions ". The meaning of the *and* is not clear. Please clarify.
We have modified Definition 19 (now it is Definition 16).

1J) Definition 22: it is not clear what "FILTER(?x)" is. Actually, later FILTER is used with some operators inside. Please rewrite this definition to clarify.
We have rewritten this definition to clarify the definition of "FILTER(?x)" (now it is Definition 19).

1K) Example 7: it says "Example 5 provides that is out of the reach of nSPARQL " please make this claim more precise, ideally, including a reference where this claim is formally proved.
We deleted this repeated claim (see response to 1U)

1L) Definition 23: the notion of "C^x_s" being equal to "T" is not defined.
We have added its definition (now it is Definition 20).

1M) Definition 23 and 24: I’m really confused with these two definitions. Although the syntax of cpSPARQL is recursive (an expression exp can be part of a CPRDF-constraint) the semantics is not defined recursively. Def 23 defines the semantics of "psi = x:C" where "C" is a graph or a SPARQL constraint. Then Def 24 defines the semantics for "axis::[psi]" and it is not clear at all what is the semantics if "psi" is a graph with cpSPARQL expressions inside. From my point of view, the semantics of cpSPARQL is underspecified. Am I missing something here?
We have redefined Definition 23 (now it is Definition 20) to make it more clear.

In the following, we have treated the minor comments of reviewer 1:

1N) pag1-c012-par2: "SPARQL2L" should be "SPARQ2L"
done
1O) pag1-col2-par2: the references for SPARQ2L and SPARQLser are interchanged.
done

1P) pag2-col1-par3: it says that PSPARQL has the same expressiveness as the path language that is being standardized in SPARQL 1.1. This is not true as SPARQL 1.1 considers bag semantics (see reference [7] for more details). Please make this sentence more precise.
We have modified this sentence and made it more precise in the implementation and conclusion parts.

1Q) Def 7: It seems that $S(P,G)$ depends on the entailment relation used, thus I would advice to use the entailment relation as a sub index (you can use it once, and then say that when it is clear from the context you will not mention it).
We have used the entailment relation as a sub index and we have added a text after the definition (now it is Definition 5) to cover this comment.

1R) Def 7: one particular operator used in SPARQL filter conditions is "bound(?X)". The definition of answer (Def 7) does not cover this case (since $\mu(K)$ is not true whenever a variable is unbounded). Please change the definition or explicitly state that "bound(?X)" is not covered.
We have made this definition (now it is Definition 5) more precise to cover the bound operator.

1S) Def 14: this definition is from [27], so please cite it in the definition itself (Something like "Definition 14 [27]"

done (now it is Definition 11)

1T) Def 16: the notion of "basic nSPARQL graph pattern" is not defined
We have defined the notion of basic graph pattern in section 2.2.1 in the first paragraph.

1U) pag6-col2 after Theo 1: it says "it cannot be used to find nodes connected with transportation mean that is not a bus". I think that nSPARQL can answer such a query for a fixed vocabulary (just construct
a query and list all transportation means that are not bus). Please make this claim a bit more specific (are you considering RDFS? do you need a query that works independent of the vocabulary?) and include an idea of why this is not possible, ideally, including a reference where this claim is formally proved. Similar for the other claim about "Air France".

We have made this claim more precise as well as we have provided an idea why this is not possible. Also, we have provided in Corollary 1 (now it is Theorem 5) a proof of a similar example.

1V) pag6-col2: par1 of Sec 3.3. "of do not" --> "do not"
done

1W) Def 18: for the first (and unique?) time in the paper the notation "rho df" is used (last equality). Either explain what does it mean or delete it.

We have replaced "rho df" by its meaning (now it is Definition 15).

1X) Example 5: Please make the first line a complete sentence.
done

1Y) Example 6: Please make the first line a complete sentence.
done

1Z) pag9-col2: par1 of Sec 4.2, please fix the font of "next:::",(and also in several other places in the paper)
done

1a) Def 25: "mu(R)" is not defined
we have defined mu(R) (now it is Definition 22).

1b) pag10-col1: par before Example 8, add a ")" after Definition 25.
done
Review 2 by Emanuele Della Valle

2A) The weakest point of the paper is the implementation experience (section 7) where I would expect some experimental evidence of the implementability of CPSPARQL and a comparison with nSPARQL, or at least with SPARQL 1.1 engines that support the path expressions. However, the theoretical contribution of the paper is significant enough even without experimental evidences.

First of all, thanks to the referee. The theoretical contribution of the paper is significant enough even without experimental evidences. In particular, the particle comparison is useful when one language admits more efficient algorithm than the other, which is not the case as shown in the paper. Regarding the expressivity of the two languages, they are compared in section 6.

Contrary to CPSPARQL, nSPARQL language is not implemented at the moment, so we must leave this interesting point for future work. PSCPARQL (and its extension CPSPARQL) and SPARQL 1.1 engines that support the path expressions are incomparable in terms of the semantics for path expressions see [Arenas et al., 2012]. Moreover, PSCPARQL, which is the mother language of CPSPARQL, has been already compared with SPARQL 1.1 engines that support the path expressions in terms of efficiency [Arenas et al., 2012].

We mentioned this in the revised version, and in particular in the implementation section.

In the following, we have treated the minor comments of reviewer 2:

2B) B is probably a set of blank-nodes. To the best of my knowledge variables do not syntactically belong to RDF, but to graph patterns. We have used blank nodes only for graph patterns (see Definition 1).

2C) T appears to contain literals, do GRDF allow for literal as subject? We have eliminated literals from subjects, although they do not change the results in the paper.

2D) blank nodes and IRIs can appear as subjects in RDF, you only allow for IRIs. 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, results of this paper may indirectly apply by using the graph homomorphism technique. So, we decided to not consider them in this paper. In the revised version, we have added a discussion about this issue after Definition 1.

2E) to the best of my knowledge what the authors name "map" is normally named "mapping"
We have replaced "map" by "mapping" everywhere in the paper.

2F) in B(P), B appears to have a different meaning from the B introduced in section 2.1
Indeed, B has been used for one meaning, i.e. to denote the set of variables. B is used to denote the set of variables as well as B(P) is used to denote the set of variables occurring in P. This is explained in paragraph 3, Section 2.2.2.

2G) in definition 8, the sigma notation in the equation is not explained in the document. Moreover, given the context (the authors are using relational algebra symbols to explain sparql semantics) I would not use sigma since it recalls the selection operator in relational algebra
We have replaced sigma by mu (now it is Definition 6).

2I) in Example 3 I would provide a short text explaining the expression. It would allow readers to check if they are understanding.
We have added an explanation text to Example 3.

2J) the referring to the initial formulation of CPSPARQL [2] does not help the readers (who may have not read [2]) to follow the presentation. I recommend to present CPSPARQL as if [2] does not exist and later on summarize the differences.
We think that this will add extra notations and make the paper difficult to follow, as the paper already has two many notations. So, we prefer not to include it in this paper.

2K) in definition 19 I cannot correctly interpret B since it was defined twice in Section 2.1 and at page 4 (see above)
We replied to this comment (see response to 2F ).
2L) the symbol sigma is used with a different meaning of the symbol introduced in definition 8. Moreover, in terms of relational algebra it is a projection, thus why not using pi? We have replaced sigma by mu.

2M) while the whole CPSPARQL language *in* in theory as efficient as SPARQL is -> while the whole CPSPARQL language *is*, in theory, as efficient as SPARQL.
done.

Review 3: by anonymous reviewer

The title of the paper indicates a comparison of languages for answering RDF-path queries modulo RDFS. However, only one and a half pages out of 15 are concerned with a comparison. The paper first presents RDF, RDFS and nSPARQL. Then it introduces CPSPARQL, which extends PSPARQL by constraints, where PSPARQL is a language from the same authors already published in Journal of Web Semantics. The main topic of the paper then is to demonstrate that the extended PSPARQL can express certain queries which are not expressible by nSPARQL. This is certainly of interest, however to some extend the comparison seems to be not fair. nSPARQL was introduced for answering queries considering RDFS semantics. So when you have broader applications in mind, you should demonstrate, that nSPARQL could not be extended in a canonic way by a constraints language. It is not convincing to be able to claim more expressiveness when it is only because nSPARQL needs a SELECT clause - which SPARQL provides anyway - and your language does not, respectively, because your constraints are more general. You should comment on this and in particular improve the structure of the paper by focusing on the main points of interest.

First of all, CPSPARQL and nSPARQL are two languages that have been developed independently. nSPARQL is a good navigational language, but it does not preserve the atomic construct of SPARQL (the triple pattern) while cpSPARQL does. Although it is shown in [27,28] that SPARQL triple patterns can be encoded by nested regular expressions, triple patterns with three variables (subject, predicate, object) could not be expressed. The reader may
wonder if 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".

This is not the only limitation of nSPARQL. nSPARQL cannot be used to express constraints (using for example SPARQL FILTER constraints) on the nodes traversed by paths even with the use of SPARQL SELECT clause. This is due to the fact that nSPARQL does neither allow such constraints nor variables inside the nested regular expressions. To do so, it is required to redefine its syntax and semantics to allow such constructs (see also response to ID). So, it is not only "because nSPARQL needs a SELECT clause - which SPARQL provides anyway - and your language does not, respectively, because your constraints are more general". Moreover, the use of the projection operator (SELECT clause), which is studied well in [27,28] makes the query evaluation an NP-hard problem.

Since these issues, which indeed are in the main comparison goal of the paper were not clear enough, we have revised "section 6 On the expressiveness of cpSPARQL and nSPARQL" by adding a discussion about these issues and we have added a proof to Corollary 1 containing a counter-example that cannot be expressed in nSPARQL (in the revised paper, Theorem 5).

Despite this discussion, we have changed the title of the paper.

References


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 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 [21]) 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) [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 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 τ(G) returns the answer, while the lazy approach transforms the query so that the transformed query τ(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 [28]. It consists of providing 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.

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

∗Corresponding author.
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 as well as they are a strict extension of PSPARQL graph patterns. 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 [21]) is a W3C recommended language for expressing data on the semantic web. We introduce below the syntax and the semantics (simple semantics [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).

**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 $\text{voc}(G)$ 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:
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 \(G\).

**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 \(\mu : V_1 \rightarrow V_2\) such that \(\forall x \in (V_1 \cap V), \mu(x) = x\).

In the following definition, we provide a characterization of 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 \(G \models_{\text{RDF}} P\)) if \(\langle s, p, o \rangle \in P\), then \(\langle s, p, o \rangle \in G\). An RDF graph \(G\) RDF-entails a GRDF graph \(P\) (denoted by \(G \models_{\text{GRDF}} P\)) if there exists a mapping \(\mu : T(P) \rightarrow T(G)\) such that if \(\langle s, p, o \rangle \in P\), then \(\langle \mu(s), \mu(p), \mu(o) \rangle \in G\).

### 2.2. SPARQL

SPARQL is the RDF query language developed by the W3C [30]. 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 set of triple patterns such as a GRDF graph in SPARQL, the union of graph patterns, an optional graph pattern, or a constraint (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 \(P \land P'\), \((P \lor P')\), \((P \Rightarrow P')\), \((P \land K)\), \((P \land K)\), and \((P \land \overline{K})\) 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 \(\text{SELECT } \vec{B} \text{ FROM } u \text{ WHERE } P\) 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:

\[
\text{SELECT } ?\text{city1} ?\text{city2} \\
\text{FROM } ... \\
\text{WHERE } \\
?\text{city1} \text{ cityIn } ?\text{city2}. \\
?\text{city1} \text{ cityIn } \text{France}. \\
?\text{city2} \text{ cityIn } \text{Jordan}. 
\]

#### 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 \(\text{dom}(\mu)\), is the subset of \(\mathcal{T}\) on which \(\mu\) is defined. The restriction of \(\mu\) to a set of terms \(X\) is defined by \(\mu|_X = \{ (x, y) \in \mu | x \in X \}\) and the completion of \(\mu\) to a set of terms \(X\) is defined by \(\mu|_X = \mu \cup \{ (x, \text{null}) | x \in X \text{ and } x \notin \text{dom}(\mu) \}\).

---

**Fig. 1.** An RDF graph \((G)\) with its schema \((M)\) representing information about transportation means between several cities.
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 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 $\mu = \mu_1 \oplus \mu_2 : T_1 \cup T_2 \rightarrow T$ 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]:

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

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 $\models_{rdf} B$ 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 $S(P, G)$ of answers to $P$ in $G$ is defined inductively in the following way:

$$S(P, G) = \{\mu | B(P) \models_{rdf} \mu(P)\}$$

if $P$ is a basic graph pattern

$$S((P \text{ AND } P'), G) = S(P, G) \times S(P', G)$$

$$S(P \text{ UNION } P', G) = S(P, G) \cup S(P', G)$$

$$S(P \text{ OPT } P', G) = (S(P, G) \times S(P', G))$$

$$\cup (S(P', G) \times S(P, G))$$

$$S(P \text{ FILTER } K, G) = \{\mu | \mu \in S(P, G) \mid \mu(K) = T\}$$

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, $\mu(K) = T$ 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 $\mu(K) \neq T$. One particular operator that can be used in SPARQL filter conditions is "$\text{bound}(?x)$". 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 $\text{SELECT } \vec{B} \text{ FROM } u \text{ WHERE } P$ be a SPARQL query, $G$ be the RDF graph identified by the URI $u$, and $S(P, G)$ be the set of answers to $P$ in $G$, then the answers $A(\vec{B}, G, P)$ to the query are the restriction and completion to $\vec{B}$ of answers to $P$ in $G$, i.e.,

$$A(\vec{B}, G, P) = \{\mu | \vec{B} \models \mu \mid \mu \in S(P, G)\}.$$

### 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:

$$\text{exp ::= axis } | \text{axis:a } | \text{exp | expexp | expexp } | \text{exp}^*$$

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

Regarding the precedence among the regular expression operators, it is as follows: $*, /$, then $\text{l}$. 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 (self or 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.
Definition 8 (Nested regular expression). A nested regular expression is an expression built from the following grammar:
\[
\text{exp} ::= \text{axis} | \text{axis}:a | \text{axis}:[\text{exp}] | \text{exp} | \text{exp} / \text{exp} | \text{exp}^* 
\]

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 \( s \in T \), \( o \in T \) 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:
\[
P = \langle ?city_1, (\text{next} :: (\text{next} :: \text{sp})^*/\text{self} :: \text{transport})^+, ?city_2 \rangle
\]

This pattern expresses a sequence of properties such that each property is a sub-property of the property “transport”.

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 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 \( (P_1 \text{ AND } P_2) \), \( (P_1 \text{ UNION } P_2) \), \( (P_1 \text{ OPT } P_2) \), and \( (P_1 \text{ FILTER } K) \) are nSPARQL graph patterns.

However, for theoretical 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 $\overline{p}_G$) is defined by:

$$\overline{\text{self}}_G = \{ \langle x, x \rangle | x \in \text{voc}(G) \}$$

$$\overline{\text{a}}_G = \{ \langle a, a \rangle \}$$

$$\overline{\text{next}}_G = \{ \langle x, y \rangle | \exists z; \langle x, z, y \rangle \in G \}$$

$$\overline{\text{edge}}_G = \{ \langle x, y \rangle | \exists z; \langle x, a, y \rangle \in G \}$$

$$\overline{\text{node}}_G = \{ \langle x, y \rangle | \exists z; \langle z, x, y \rangle \in G \}$$

$$\overline{\text{axis}^{-1}}_G = \{ \langle x, y \rangle | \langle y, x \rangle \in [\text{axis}]_G \}$$

$$\overline{\text{a}}^{-1}_G = \{ \langle y, x \rangle | \langle x, y \rangle \in [\text{axis}]_G \}$$

$$\overline{\text{exp1}}_G = \{ \overline{\text{exp1}}_G \cup \overline{\text{exp2}}_G \}$$

$$\overline{\text{exp1}}_G = \{ \langle x, y \rangle | \exists z; \langle x, z, y \rangle \in \overline{\text{exp1}}_G \}$$

$$\overline{\text{exp}}_G = \{ \text{self}_G \cup \overline{\text{exp1}}_G \cup \overline{\text{exp2}}_G \}$$

$$\overline{\text{exp}}_G = \{ \langle x, y \rangle | \exists z; \langle y, x \rangle \in \overline{\text{exp1}}_G \}$$

$$\overline{\text{exp}}_G = \{ \text{self}_G \cup \overline{\text{exp}}_G \cup \overline{\text{exp1}}_G \}$$

$$\overline{\text{exp}}_G = \{ \text{self}_G \cup \overline{\text{exp}}_G \cup \overline{\text{exp1}}_G \}$$

$$\overline{\text{exp}}_G = \{ \text{self}_G \cup \overline{\text{exp}}_G \cup \overline{\text{exp1}}_G \}$$

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 $t = \langle X, R, Y \rangle$ over an RDF graph $G$ is:

$$[t]_G = \{ \mu \text{ dom}(\mu) = \{ X, Y \} \cap B \text{ and } \langle \mu(X), \mu(Y) \rangle \in [R]_G \}$$

Answers to nSPARQL queries follow the same definition as for SPARQL but this time it is 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:

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

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 [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 $O(|G|.|R|)$.

Clearly, nSPARQL is a good navigational language, but there are still useful queries that could not be expressed. However, we conjecture that it cannot be used to find nodes connected by a property that is not $p$ (e.g., with transportation mean that is not a bus or transportation means belonging to Air France, i.e., containing the URI of the company). Although nSPARQL can answer such query for a fixed vocabulary by listing all properties that are not $p$ (e.g. by listing all properties that are not bus), a query that works independent of the vocabulary cannot be expressed. This is because it is required to know all properties that are used in the vocabulary other than $p$ in order to use them in the query. Moreover, properties that are sub-properties of $p$ should be excluded from the query when considering RDFS (see the Appendix section for a proof outline of a similar example).

### 3.3. Querying RDFS with nSPARQL

[24] has introduced the reflexive relaxed semantics for RDFS in which rdps:subPropertyOf and rdps:subClassOf of do not have to be reflexive. The reflexive relaxed semantics does not change much RDFS. Indeed, from the standard (reflexive) seman-
tions, 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 in the predicate position.

However, when issuing queries involving these relations, e.g., with a graph pattern like \( (\forall x \ rdfs:p ?y) \), all properties in the graph will be answers. Since this 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 \( rdfs:p \) or \( rdfs:c \) 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 [24].

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

\[
\begin{align*}
[\mu]^{\text{RDFS}}_G = \{ & \mu|\text{dom}(\mu) = \{X, Y\} \cap B \\
& \land (\mu(X), \mu(Y)) \in [R]_{\text{closure}(G)} \}
\end{align*}
\]

**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:

\[
\mathcal{A}^o(G, P) = \{ \mu \mid (\mu(X), \mu(Y)) \in [R]^{\text{RDFS}}_G, \\
\forall (X, R, Y) \in P \}
\]

As presented in [28], nSPARQL can evaluate queries with regard to RDFS by transforming queries with the rules [24]:

- \( \phi(sc) = (\text{next::sc})+ \)
- \( \phi(sp) = (\text{next::sp})+ \)
- \( \phi(dom) = \text{next::dom} \)
- \( \phi(range) = \text{next::range} \)
- \( \phi(type) = \text{next::type/next::sc}+ \)
- \( \phi(p) = \text{next::sp/next::dom/next::sc}+ \)

\[
\begin{align*}
|\text{edge}/\text{next::sp}/\text{next::dom}/\text{next::sc}+ | \\
|\text{node}^{-1}/\text{next::sp}/\text{next::range}/\text{next::sc}+ | \\
(p \not\in \{\text{sp, sc, type, dom, range}\})
\end{align*}
\]

**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:

\[
\begin{align*}
\{&(\text{city}1, (\text{next::transport})^+, \text{city}2) \\
&(\text{city}1, \text{next::city}In, \text{France}) \\
&(\text{city}2, \text{next::city}In, \text{Jordan})
\}
\end{align*}
\]

When evaluating this graph pattern against the RDF graph of Figure 1 and considering the RDFS semantics, it returns the empty set instead of the following set of pairs:

\[
\begin{align*}
\{&(\text{city}1 \leftarrow \text{Paris}, \text{city}2 \leftarrow \text{Amman}), (\text{city}1 \leftarrow \text{Grenoble}, \text{city}2 \leftarrow \text{Amman})
\}
\end{align*}
\]

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

\[
\begin{align*}
\{&(\text{city}1, (\text{next::sp})^*/\text{self::transport})^+, \text{city}2) \\
&(\text{city}1, \text{next::city}In, \text{France}) \\
&(\text{city}2, \text{next::city}In, \text{Jordan})
\}
\end{align*}
\]

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

**Theorem 2** (Completeness of \( \phi \) [28] (Theorem 3)). Let \( (X, p, Y) \) be a SPARQL triple pattern with \( X, Y \in R \).
its head variable.

Definition 16
(Constrained regular expression)
those whose nodes satisfy encountered constraint.
versed by constrained regular expressions and select
features RDFS semantics.
spect to RDFS semantics, i.e. the fragment still cap-
do not use these keywords in the fragment presented
the traversed path that satisfies the given constraint. We
traversed nodes or to check the existence of a node in
words are used to allow expressing constraints on all

4. CPSPARQL and cpSPARQL: syntax and
semantics

CPSPARQL has been defined for addressing two
main issues. The first one comes from the need to ex-
tend PSPARQL and thus to allow for expressing con-
straints 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 de-
formed 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, re-
spectively. In this paper, we adopt the axis borrowed
from XPath, with which the reader may be more famil-
lar, as done for nSPARQL. This will also allow us to
better compare cpSPARQL and nSPARQL. Addition-
ally, in the original proposal, ALL and EXISTS key-
words 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 re-
spect to RDFS semantics, i.e. the fragment still cap-
tures RDFS semantics.

Constraints act as filters for paths that must be tra-
versed 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:

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

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

Constrained regular expressions allow for constrain-
ing the item in one axis to satisfy a particular con-
straint, 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 distin-
guishing between constraints which export their vari-
able (it may be assigned by the mapping) and con-
straints which do not export it (the variable is only no-
tional). This is equivalent to the initial CPSPARQL
formulation, in which the variable was always ex-
ported, since CPSPARQL can ignore such variables
through projection.

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

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

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

In contrast to nested regular expressions, con-
strained 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 \(\{s, p, o\}\) such that \(s \in T, o \in T\)
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 pat-
terns and \(K\) is a SPARQL constraint, then \((P_{1} \\text{UNION} P_{2}), (P_{1} \\text{OPT} P_{2}), \) and \((P_{1} \\text{FILTER} K)\) are CPSPARQL graph patterns.

Example 6 (CPSPARQL graph pattern). The fol-
lowing 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:

\[
\{\langle \text{city}_1, \text{next} :: ?p : \{\langle ?p, (\text{next} :: sp)\}, \text{transport}\}
\text{Filter}(\langle ?p ! = \text{bus}\rangle)\}^+, \langle \text{city}_2 \rangle
\]

\[
\{\langle \text{city}_1, \text{next} :: \text{cityIn}, \text{France}\}
\{\langle \text{city}_2, \text{next} :: \text{cityIn}, \text{Jordan}\}\}
\]

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:

\[
\text{exp} ::= \text{axis} | \text{axis} : a | \text{axis} :: ?x : \text{TRUE}[
| \text{axis} :: ?x : \{\text{exp} : \text{FILTER}(\langle ?x \rangle)\}]
| \text{exp} | \text{exp} | \text{exp}^*
\]

such that v is either ?x or a constant (an element of \(U \cup L\)) and FILTER(\langle ?x \rangle) is the usual SPARQL filter condition containing at most one variable (the variable ?x).

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 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:

\[
\{\langle \text{city}_1, \langle \text{next} :: ?p : \{\langle ?p, (\text{next} :: sp)\}, \text{transport}\}\rangle\}^+, \langle \text{city}_2 \rangle
\]

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

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

\[
\text{next} :: ?p, (?p, (\text{next} :: sp)^*, ?z),
\]

\[
\langle ?q, (\text{next} :: sp)^*, ?z\rangle,
\]

\[
\langle ?p, \text{owl} : \text{inverseOf}, ?q\rangle,
\]

\[
\text{Filter}(\text{regex}(?z, \text{iata.org}))
\]

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

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

**Definition 20** (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:

1. \(C\) is a triple pattern \(C = \langle X, R, Y \rangle\), and \(\langle x, R, y \rangle \in [R]_G\), where \(K^x_s\) means that \(s\) is substituted to the variable \(x\) if \(K = x\) or \(K\) contains the variable \(x\).
2. \(C\) is a SPARQL filter constraint and \(C^x_s = T\), where \(C^x_s = T\) means that the constraint obtained by the substitution of \(s\) to each occurrence of the variable \(x\) in \(C\) is evaluated to true\(^1\).
3. \(C = \text{FILTER} K\), then 1 and 2 should be satisfied

In contrast to the case of CPSPARQL triple patterns (the reader can refer to [2] for the general case), there is no need for the existence of a mapping because cpSPARQL triple patterns contain only one variable from Definition 7.

\(^1\)Except for the case of bound (see Definition 5 and the discussion after it.)
As done for nested regular expressions, the evaluation of a constrained regular expression \( \tau \) over an RDF graph \( G \) is defined as a binary relation \([\tau]_G\), by a pair of nodes \((a, b)\) such that \( a \) is reachable from \( b \) in \( G \) by following a path that conforms to \( \tau \). The following definition extends Definition 11 to take into account the semantics of terms with constraints.

**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:

\[
[\text{self}::[\psi]]_G = \{ \langle x, x \rangle \mid x \in \text{voc}(G) \land x \in [\psi]_G \}
\]

\[
[next::[\psi]]_G = \{ \langle x, y \rangle \mid \exists z; (x, y, z) \in G \land z \in [\psi]_G \}
\]

\[
[edge::[\psi]]_G = \{ \langle x, y \rangle \mid \exists z; (x, y, z) \in G \land z \in [\psi]_G \}
\]

\[
[node::[\psi]]_G = \{ \langle x, y \rangle \mid \exists z; (x, z, y) \in G \land z \in [\psi]_G \}
\]

\[
[\text{axis}^{-1}::[\psi]]_G = \{ \langle x, y \rangle \mid (y, x) \in [\text{axis}::[\psi]]_G \}
\]

**Definition 22** (Answer to a CPSPARQL triple pattern). The evaluation of a CPSPARQL triple pattern \( \tau \) over an RDF graph \( G \) is defined as the following set of mappings:

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

Of course, this semantics applies to cpSPARQL graph patterns. Note that \( B(R) \) 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 23** (RDFS triple pattern expansion). Given an RDF triple \( t \), the RDFS expansion of \( t \), denoted by \( \tau(t) \), is defined as:

\[
\tau((s, p, o)) = (s, \text{next}::\text{sp}^+, o)
\]

\[
\tau((s, \text{sp}, o)) = (s, \text{next}::\text{sp}^+, o)
\]

\[
\tau((s, \text{dom}, o)) = (s, \text{next}::\text{dom}, o)
\]

\[
\tau((s, \text{range}, o)) = (s, \text{next}::\text{range}, o)
\]

\[
\tau((s, \text{type}, o)) = (s, \text{next}::\text{type}/\text{next}::\text{sp}^+1
\]

\[
\text{edge}/(\text{next}::\text{sp})^*/(\text{next}::\text{dom}/(\text{next}::\text{sp})^+1 \text{node}^{-1}(\text{next}::\text{sp})^*/(\text{next}::\text{range}))(\text{next}::\text{sp})^*, o)
\]

\[
\tau((s, p, o)) = (s, (\text{next}[:?x : \{ (?x, (\text{next}::\text{sp})^+, p )\}], o))
\]

It is clear that the RDF 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 mappings (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 \( p \):

\[
\]

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

\[
\]

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 PPSPARQL 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):

\[
(?X, (\text{next}::*?x: ?x, (\text{next}::\text{sp})^*, p)), ?Y)
\]
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 $(X, p, Y)$ be a SPARQL triple pattern with $X, Y \in (U \cup B)$ and $p \in U$, then $[(X, p, Y)]_{G}^{rdfs} = [(X, \tau(p), Y)]_{G}$ 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 $(\mu(X), \mu(Y)) \in [(X, p, Y)]_{G}^{rdfs}$ iff $(\mu(X), \mu(Y)) \in [(X, \tau(p), Y)]_{G}$.

- $(\Rightarrow)$ Suppose that $(\mu(X), \mu(Y)) \in [(X, p, Y)]_{G}^{rdfs}$. In this case, there exists $p_1$ such that $(p_1 sp p_2 sp \ldots sp p_n = p)$ and $(\mu(X), p_1, \mu(Y)) \in G$ as well as $(\mu(X), next::p_1, \mu(Y)) \in G$. Let us consider now the transformed triple $(\mu(X), (next::p) \psi$, $Y)$ where $\psi = [\{?, p, (next::sp)^* \} \}$). The mappings for the variable $?p$ will be $\{(?, p_i) \mid i = 1, \ldots, n\}$ (since $[\psi]_G = \{(p_i, p) \mid i = 1, \ldots, n\}$). Now according to Definitions 25 and 24, $(\mu(X), \mu(Y)) \in [(X, (next::p) \psi), Y]_{G}$ iff $(\mu(X), \mu(Y)) \in G$ and $p_1 \in [\psi]_G$.

- $(\Leftarrow)$ We have to prove that if $(\mu(X), \mu(Y)) \in [(X, (next::p) \psi), Y]_{G}$ with $\psi = [\{?, p, (next::sp)^* \} \}$, then $(\mu(X), \mu(Y)) \in [(X, p, Y)]_{G}^{rdfs}$. Suppose that $(\mu(X), \mu(Y)) \in [(X, (next::p) \psi), Y]_{G}$.

In this case, there exists $p_1$ such that $(\mu(X), next::p_1, \mu(Y)) \in G$ and $p_1 \in [\psi]_G$, that is, $(p_1, next::sp p_2, \ldots, (p_{n-1}, next::sp, p_n = p) \in G$. Therefore, $(\mu(X), \mu(Y)) \in [(X, p, Y)]_{G}^{rdfs}$ since $(p_1, (next::sp)^* \} \}$ and $(\mu(X), next::p_1, \mu(Y)) \in G$.

□

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 [28] 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 $(next::p, o) \in \alpha(s)$ and $(edge^{-1}::p, s) \in \alpha(p)$.

Also, $(ass::u, u) \in \alpha(u)$, for $u \in \text{voc}(G)$. The set of terms of a constrained regular expression $R$, denoted by $T(R)$, is constructed as follows:

- Let $A_{R} = (Q, T(R), s_0, F, \delta)$ be the $epsilon-NFA$ of $R$ constructed in the usual way using the terms $T(R)$, where $\delta : Q \times (T(R) \cup \{epsilon\}) \rightarrow 2^Q$.

In the evaluation algorithm, we use the product automaton $G \times A_{R}$ (in which $\delta' : \text{voc}(G) \times Q \times (T(R) \cup \{epsilon\}) \rightarrow 2^\text{voc}(G) \times Q$ is its transition function). We construct $G \times A_{R}$ as follows:

- $(\forall u \in \text{voc}(G) \times Q$, for every $u \in \text{voc}(G)$ and $q \in Q$;
- $(\forall q \in \delta'((u, p), s)$ iff $q \in \delta(p, s)$; and one of the following conditions satisfied:

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

Algorithm 1 solves the evaluation problem for a constrained regular expression $R$ over an RDF graph $G$. This algorithm is almost the same as the ones used for similar purposes, e.g., the same as the one used for evaluating nested regular expressions presented in [28] and the one used for evaluating PSPARQL regular expressions presented in [5].

**Theorem 4 (Complexity of cpSPARQL evaluation)**

Eval solves the evaluation problem for constrained regular expression in time $O(|G|\cdot |R|)$.

**Proof.** Let $R$ be the a constrained regular expression, $G$ be an RDF graph, $(a, b)$ be a pair of nodes, $A_{R}$ be the automaton recognising the language of $R$.
construct \( A_R \) (assume \( q_0 \) : initial state and \( F \) : set of final states).

construct the product automaton \( G \times A_R \)

- if a state \( \langle b, q_f \rangle \) with \( q_f \in F \), is reachable from \( \langle a, q_0 \rangle \) in \( G \times A_R \)

  return YES;

else

  return NO;

end if

Algorithm 1 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.

Constructing the automaton of \( R \) can be done in \( \text{NLOGSPACE} \) 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 \( \langle s_i, \text{next} :: p, s_j \rangle \in A_R \) and \( \langle n_i, \text{next} :: p, n_j \rangle \in G \), then add \( \langle s_i, n_j \rangle \) to the product automaton.

- If \( \langle s_i, \text{next} :: [\text{Filter}(?x)], s_j \rangle \in A_R \) and \( \langle n_i, \text{next} :: p, n_j \rangle \in G \), then add \( \langle s_j, n_i \rangle \) 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 \( \psi \) can be done in \( O(1) \).

- If \( \langle s_i, \text{next} :: {?x} :: \{\langle ?x, R_i, v \rangle \}, s_j \rangle \in A_R \) (respectively, \( \langle s_i, \text{next} :: {?x} :: \{\langle ?x, R_i, v \rangle \}, s_j \rangle \in A_R \) and \( \langle n_i, \text{next} :: p, n_j \rangle \in G \), then substitute the only node \( p \) to the variable \(?x\) and construct the sub-automaton for the constrained regular expression \( R_i \), starting from the node \( p \) to the node \( v \) (respectively, starting from the node \( p \) to the node \( v \)).

More precisely, we check only if the pair \( \langle p, v \rangle \) satisfies the sub-expression \( R_i \). Note that when the self axis is used, we check if the pair \( \langle n_j, v \rangle \) satisfies the sub-expression \( R_i \).

- If \( \langle s_i, \text{next} :: {?x} :: \{\langle ?x, R_i, v \rangle \} \text{Filter}(?x), s_j \rangle \in A_R \) (respectively, \( \langle s_i, \text{next} :: {?x} :: \{\langle ?x, R_i, v \rangle \} \text{Filter}(?x), s_j \rangle \in A_R \) and \( \langle n_i, \text{next} :: p, n_j \rangle \in G \), then: if \( p \) satisfies the SPARQL filter constraint, then construct the sub-automaton for the constrained regular expression \( R_i \), starting from the node \( p \) to the node \( v \) (respectively, starting from the node \( p \) to the node \( v \)).

So, constructing the product automaton \( (G \times A_R) \) can be done in time \( O(|G| \cdot |R|) \). Hence, checking if the pair \( \langle a, b \rangle \in [R]_G \) is equivalent to checking if the language accepted by \( (G \times A_R) \) is non-empty, which can be done in \( O(|G| \cdot |R|) \) (as in the case of usual regular expressions [33,23]).

An efficient algorithm that solves the evaluation problem for nested regular expressions \( R \) over an RDF graph \( G \) in time \( O(|G| \cdot |R|) \) is presented in [28]. 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. 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 shown 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 if whether this is useful or not. The following query is a useful example:

```
SELECT *
```

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 *
```

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:

\[
\{ (?\text{person}1, (\text{next}::\text{ascendant})^+ / \\
(\text{next}^{-1}::\text{ascendant})^+, ?\text{person}2), \\
\text{Filter}(!(?\text{person}1 = ?\text{person}2)) \}
\]

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:

\[
\{ (?\text{person}1, (\text{next}::\text{ascendant})^+, \text{ancestor}), \\
(?\text{person}2, (\text{next}::\text{ascendant})^+, \text{ancestor}), \\
\text{FILTER}(!(?\text{person}1 = ?\text{person}2) \\
\&\& (\text{regex}(\text{ancestor}, "\text{alkhateeb}"))) \}
\]

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

\[
\mu_{\text{person}1, \text{person}2}( \\
\{ (?\text{person}1, (\text{next}::\text{ascendant})^+, \text{ancestor}), \\
(?\text{person}2, (\text{next}::\text{ascendant})^+, \text{ancestor}), \\
\text{FILTER}(!(?\text{person}1 = ?\text{person}2) \\
\&\& (\text{regex}(\text{ancestor}, "\text{alkhateeb}"))) \})
\]

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 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:

\[
\{ (?\text{person}1, (\text{next}::\text{ascendant})^+ \\
/\text{self}::[?\text{ancestor} : \\
\text{FILTER}(\text{regex}(\text{ancestor}, "\text{alkhateeb}"))] \\
/(\text{next}^{-1}::\text{ascendant})^+, ?\text{person}2), \\
\text{FILTER}(!(?\text{person}1 = ?\text{person}2)) \}
\]

This shows that this remark applies even for SPARQL queries which do not require projection!

nSPARQL cannot express all cpSPARQL, even with SELECT

In the following, 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:

\[
\{ (?\text{city}1, (\text{next} :: (\text{next} : sp)^+ /\text{self} :: \text{transport}) /\text{self} :: \text{cityIn} /\text{self} :: \text{France})^+, ?\text{city}2) \}
\]

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

\[
\{ (?\text{city}1, (\text{next} :: [?\text{city}/\text{self} :: [?\text{size}]^+, ?\text{city}2), \\
\text{where:} \\
\psi_1 = ?x : \{(?x, (\text{next} : sp)^+, \text{transport}) \}, \\
\psi_2 = ?x : \{(?x, \text{next} :: \text{cityIn}, \text{France}) \}
\]

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:

\[
P = (?\text{city}1, (\text{next} :: [?\text{city}/\text{self} :: [?\text{size}]^+, ?\text{city}2), \\
\text{where:} \\
\psi_1 = ?x : \{(?x, (\text{next} : sp)^+, \text{transport}) . \text{FILTER} (\text{regex}(?x, "www.AirFrance.fr")]], \\
\psi_2 = ?x : \{(?x, \text{next} :: \text{size}, ?\text{size}) . \text{FILTER} (!\text{size} > 20,000) \}
\]

However, this query cannot be expressed by a nested regular expression, since it is not possible to apply constraints, such as SPARQL constraints, 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:

\[
\psi_2 = ?x : \{(?x, \text{next} :: \text{size}!?\text{size} : \text{FILTER}(!\text{size} > 20,000)) /\text{next}^{-1} :: \text{size}, ?x \}
\]

In this case, the \(\text{size}\) variable is 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. 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 \( G \).

Proof. We use a simple counter-example that cannot be expressed in nSPARQL. Consider the RDF graph \( G \) of Figure 3 representing transportation means between cities and each city has a population size less than 20,000:

If one wants that each stop satisfies a specified constraint, e.g., cities with a population size larger than 20,000 inhabitants, this may be expressed by the following constrained regular expression:

\[
P = (?city1, (next :: transport/self :: [psi1])^+, ?city2),
\]

where:

\[
 psi1 = [\{x :: (?x, next :: size ?size : FILTER(?size > 20,000))/next^+ :: size, ?x\}]
\]

\[
 [P]_G = \emptyset \text{ since the population size for all cities is less than 20,000.}
\]

As nSPARQL does not allow to constrain the values of the nodes traversed by paths, this query could be only expressed in nSPARQL by enumerating the values of the population size that are larger than 20,000 (say \( v_1, \ldots, v_n \)) as in the following nested regular expression (of course by assuming the literals are followed to be used in nested regular expressions which is not the case in the current definition of nSPARQL):

\[
 \text{nextp}_1 = (?city1, (next :: transport/self :: [next :: size/self :: (v_1 \ldots v_n)])^+, ?city2)
\]

However, there are infinite number of values larger than 20,000.

The above query could be partially expressed in nSPARQL without constraining the values of the size as in the following nested regular expression:

\[
 \text{nextp}_2 = (?city1, (next :: transport^+) +, ?city2)
\]

However, \([\text{nextp}_2]_G \neq \emptyset\) and so the above query cannot be expressed by a nested regular expression.

\[ \square \]

This owes to the fact that nSPARQL does neither allow applying constraints (such as SPARQL FILTER constraints) in the nodes traversed or expressed by nested regular expressions nor the variables (it is required to have an extra variable to express the SPARQL FILTER constraints for the size inside the path expression). So, any nested regular expression \( \text{nextp} \) would have the \( [\text{nextp}]_G \neq \emptyset \). Only navigational constraints can be expressed.

The only way to express variables and constraints is outside nested regular expressions, i.e., in the subject and/or object position of triple patterns. This could be expressed by using fixed length nested regular expression to catch the size by a variable "\(?size\)" and then expressing that FILTER constraint on that variable.

**cpSPARQL can express all nSPARQL**

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 \( \text{trans}(R) = \exp_1 :: [\psi] \), where:

\[
\psi = {?x :: \{(?x, \text{trans}(\exp_3), p)\}, \text{if } \exp_2 = \exp_3 / \text{self :: p}}
\]

\[
\psi = {?x :: \{(?x, \text{trans}(\exp_2) / \text{trans}(\exp_2)^{-1}, ?x\}, \text{otherwise}}
\]

The last clause of this expression is only here because cpSPARQL, as nSPARQL, cannot introduce any new variable beside "?x". It is thus necessary to express as a triple that the constraint is satisfied and ends where it started. 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:

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

according to the transformation rules above, the constrained regular expression equivalent to this expression \( R_2 \)

\[
= \text{trans}(R_1)
\]

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

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

\[
= \text{next :: } {?x :: \{(?x, \text{trans}(\text{next :: sp})^*, \text{transport})\}}
\]

\[
= \text{next :: } {?x :: \{(?x, \text{trans}(\text{next :: sp})^*, \text{transport})\}}
\]

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

CPSPARQL has been implemented in order to evaluate its feasibility\(^2\). 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 \textit{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 that PSPARQL had better performances than other implementations of SPARQL with paths\(^3\).

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

\(^2\)The prototype is available at http://exmo.inrialpes.fr/software/psparql.

\(^3\)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/

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] attempt 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\(_R\) 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 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.

[13] 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 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 over the closure graph. However, this approach takes time proportional to \(|H| \times |G|^2\) 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. It should be noticed that CPSPARQL 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 4 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


Fig. 4. Query languages and their graph patterns


Appendix

Example 10. Consider the following RDF graph representing flights belonging to different airline companies and other transportation means between cities:

\[
\{\text{city}_1, \text{airfrance} : \text{flight}_1, \text{city}_2\} \\
\{\text{city}_2, \text{airfrance} : \text{flight}_2, \text{city}_3\} \\
\vdots \\
\{\text{city}_i, \text{anothercompany} : \text{flight}_1, \text{city}_j\}
\]

Suppose that one wants to search pairs of cities connected by a sequence of using flights belonging to air-france 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:

\[(\text{airfrance} : \text{flight}_1 | \ldots | \text{airfrance} : \text{flight}_n)^+\]

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.