HAL
open science

Answering SPARQL queries modulo RDF Schema with paths

Faisal Alkhateeb, Jérôme Euzenat

To cite this version:

Faisal Alkhateeb, Jérôme Euzenat. Answering SPARQL queries modulo RDF Schema with paths. [Research Report] RR-8394, INRIA. 2013, pp.46. hal-00904961

HAL Id: hal-00904961

https://inria.hal.science/hal-00904961

Submitted on 15 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Answering SPARQL queries modulo RDF Schema with paths

Faisal Alkhateeb, Jérôme Euzenat

RESEARCH REPORT
N° 8394
November 2013
Project-Teams Exmo

ISSN 0249-6399 ISRN INRIA/RR--8394--FR+ENG

Answering SPARQL queries modulo RDF Schema with paths

Faisal Alkhateeb*, Jérôme Euzenat†

Project-Teams Exmo

Research Report n° 8394 — November 2013 — 46 pages

Abstract: SPARQL is the standard query language for RDF graphs. In its strict instantiation, it only offers querying according to the RDF semantics and would thus ignore the semantics of data expressed with respect to (RDF) schemas or (OWL) ontologies. Several extensions to SPARQL have been proposed to query RDF data modulo RDFS, i.e., interpreting the query with RDFS semantics and/or considering external ontologies.
We introduce a general framework which allows for expressing query answering modulo a particular semantics in an homogeneous way. In this paper, we discuss extensions of SPARQL that use regular expressions to navigate RDF graphs and may be used to answer queries considering RDFS semantics. We also consider their embedding as extensions of SPARQL. These SPARQL extensions are interpreted within the proposed framework and their drawbacks are presented. In particular, we show that the PSPARQL query language, a strict extension of SPARQL offering transitive closure, allows for answering SPARQL queries modulo RDFS graphs with the same complexity as SPARQL through a simple transformation of the queries. We also consider languages which, in addition to paths, provide constraints. 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.

Key-words: Semantic web – Query language – Query modulo schema – Resource Description Framework (RDF) – RDF Schema – SPARQL – Regular expression – Constrained regular expression – Path – PSPARQL – NSPARQL – nSPARQL – CPSPARQL – cpSPARQL.

This research report sums up our knowledge about the various ways to query RDF graph modulo RDFS, providing improvements on all the existing proposals. Part of it will be published as [6].

* Computer Science Department, Yarmouk University, Jordan
† INRIA & LIG, Grenoble, France

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Évaluation de requêtes SPARQL en fonction d'un schéma RDF par les chemins

Résumé : SPARQL est le langage de requête standard pour interroger des graphes RDF. Dans son instanciation stricte, il ne propose que des requêtes en fonction de la sémantique de RDF et n'interprète donc pas les vocabulaires exprimés en RDFS ou OWL. Plusieurs extensions de SPARQL ont été proposées pour interroger les données RDF en fonction de vocabulaires RDFS et d'ontologies OWL. Par ailleurs, les extensions de SPARQL qui utilisent des expressions régulières pour naviguer dans les graphes RDF peuvent être utilisées pour répondre aux requêtes sous la sémantique de RDFS. Nous introduisons un cadre général qui permet d'exprimer d'une manière homogène l'interprétation de SPARQL en fonction de différentes sémantiques. Les extensions de SPARQL sont interprétées dans ce cadre et leurs inconvénients sont présentés. En particulier, nous montrons que le langage de requête PSPARQL, une extension stricte de SPARQL, permet de répondre aux requêtes SPARQL sous la sémantique de RDFS avec la même complexité que SPARQL par une transformation des requêtes. Nous considérons également CPSPARQL, une extension de PSPARQL, qui permet de poser des contraintes sur les chemins. Nous montrons que CPSPARQL est suffisamment expressif pour répondre aux requêtes PSPARQL et CPSPARQL sous la sémantique de RDFS. Nous présentons également nSPARQL, un langage de chemins inspiré de XPath permettant d'évaluer des requêtes sous la sémantique de RDFS. Nous comparons l'expressivité et la complexité de nSPARQL et le fragment correspondant de CPSPARQL, que nous appelons cpSPARQL. Les deux langages ont la même complexité bien que cpSPARQL, étant une extension stricte de SPARQL, soit plus expressif que nSPARQL.

Mots-clés : Web Sémantique – langage de requêtes – RDF – RDFS – SPARQL – expressions régulières – expressions régulières avec contraintes – Chemin - PSPARQL - NSPARQL - nSPARQL - CPSPARQL - cpSPARQL.

1 Introduction

RDF (Resource Description Framework [22]) is a standard knowledge representation language dedicated to the description of documents and more generally of resources within the semantic web.

SPARQL [35] 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) [12] together with OWL [26] are two ontology languages recommended by the W3C for defining the vocabulary used in RDF graphs. Extending SPARQL for dealing with RDFS and OWL semantics when answering queries is thus a major issue. Recently, SPARQL 1.1 entailment regimes have been introduced to incorporate RDFS and OWL semantics [18]. We consider here the case of RDF Schema (RDFS) or rather a large fragment of RDF Schema [28].

Query answering with regard to RDFS semantics can be specified by indicating the inference regime of the SPARQL evaluator, but this does not tell how to implement it. It is possible to implement a specific query evaluator, embedding an inference engine for a regime or to take advantage of existing evaluators. For that purpose, one very often transforms the data or the query to standard languages. Two main approaches may 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. These approaches are not exclusive, as shown by [30], though no hybrid approach has been developed so far for SPARQL.

There already have been proposals along the second approach. For instance, [33] provides a query language, called nSPARQL, allowing for navigating graphs in the style of XPath. Then queries are rewritten so that query evaluation navigates the data graph for taking the RDF Schema into account. One problem with this approach is that it does not preserve the whole SPARQL: not all SPARQL queries are nSPARQL queries.

Other attempts, such as SPARQ2L [7] and SPARQLeR [24] are not known to address queries with respect to RDF Schema. SPARQL-DL [36] addresses OWL but is restricted with respect to SPARQL. [25] provides a sound and complete algorithm for implementatiing the OWL 2 DL Direct Semantics entailment regime of SPARQL 1.1. This regime does not consider projection (SELECT) within queries and answering queries is reduced to OWL entailment.

On our side, we have independently developed an extension of SPARQL, called PSPARQL [5], which adds path expressions to SPARQL. Answering SPARQL queries modulo RDF Schema could be achieved by transforming them into PSPARQL queries [4]. 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.

This report mainly contributes in two different ways, of different nature, to the understanding of SPARQL query answering modulo ontologies.

First, it introduces a framework in which SPARQL query answering modulo a logical theory can be expressed. This allows for comparing different approaches that can be used on a semantic basis. For that purpose, we reformulate previous work and definitions and show their equivalence. This also provides a unified strategy for proving properties that we use in the proof section.

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

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

- CPSPARQL can answer full SPARQL (and CPSPARQL) queries modulo RDFS (Section 6.4);
- cpSPARQL has the same complexity as nSPARQL and there exist an efficient algorithm for answering cpSPARQL queries (Section 6.3);
- Any nSPARQL triple pattern can be expressed as a cpSPARQL triple pattern, but not vice versa (Section 7).

Outline. The remainder of the report is organized as follows. In Section 2, we introduce RDF and the SPARQL language. Then we present RDF Schema and the existing attempts at answering queries modulo RDF Schemas (Section 3). The PSPARQL language is presented with its main results in Section 4, and we show how to use them for answering SPARQL queries modulo RDF Schemas. Section 5 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 6 and we show how to use them for answering SPARQL and CPSPARQL queries modulo RDF Schemas. Complexity results as well as a comparison between the expressiveness of cpSPARQL and nSPARQL are presented in Section 7. We report on preliminary implementations (Section 8) and discuss more precisely other related work (Section 9). Finally, we conclude in Section 10.

2 Querying RDF with SPARQL

The Resource Description Framework (RDF [22]) is a W3C recommended language for expressing data on the web. We introduce below the syntax and the semantics of the language as well as that of its recommended query language SPARQL.

2.1 RDF

This section introduces Simple RDF (RDF with simple semantics, i.e., without considering RDFS semantics of the language [22]).

2.1.1 RDF syntax

RDF graphs are constructed over sets of URI references (or urirefs), blanks, and literals [15]. Because we want to stress the compatibility of the RDF structure with classical logic, we will use the term variable instead of that of "blank" which is a vocabulary specific to RDF. The specificity of blanks with regard to variables is their quantification. Indeed, a blank in RDF is a variable existentially quantified over a particular graph. We prefer to retain this classical interpretation which is useful when an RDF graph is placed in a different context. To simplify notations, and without loss of generality, we do not distinguish here between simple and typed literals.

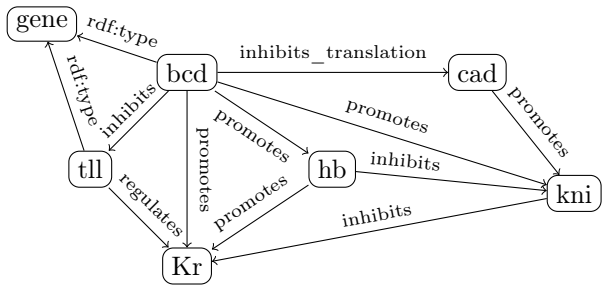

Figure 1: An RDF graph representing some interactions between genes within the early development of drosophila embryo.

In the following, we treats blank nodes in RDF simply as constants (as if they were URIs) as done in the official specification of SPARQL without considering their existential semantics. However, if the existential semantics of blank nodes is considered when querying RDF, the results of this paper indirectly apply by using the graph homomorphism technique [11].

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` or simply `gene`).

This terminology grounds the definition of RDF graphs and GRDF graphs. Basically, an RDF graph is a set of triples of the form $\langle subject, predicate, object \rangle$ whose domain is defined in the following definition.

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

So, every RDF graph is a GRDF graph. If G is an RDF graph, we use $\mathcal{T}(G)$, $\mathcal{U}(G)$, $\mathcal{L}(G)$, $\mathcal{B}(G)$, $\mathcal{V}(G)$ or $voc(G)$ to denote the set of terms, urirefs, literals, variables or names appearing in at least one triple of G.

In a triple $\langle s, p, o \rangle$, s is called the subject, p the predicate and o the object. It is possible to associate to a set of triples G a labeled directed multi-graph, such that the set of nodes is the set of terms appearing as a subject or object at least in a triple of G, the set of arcs is the set of triples of G, i.e., if $\langle s, p, o \rangle$ is a triple, then $s \xrightarrow{p} o$ is an arc (see Figure 1 and 2). By drawing these graphs, the nodes resulting from literals are represented by rectangles while the others are represented by rectangles with rounded corners. In what follows, we do not distinguish the two views of the RDF syntax (as sets of triples or labeled directed graphs). We will then mention interchangeably its nodes, its arcs, or the triples it is made of.

Example 1 (RDF)**.** *RDF can be used for exposing a large variety of data. For instance, Figure 1 shows the RDF graph representing part of the gene regulation network acting in the fruitfly (Drosophila melanogater) embryo. Nodes represent genes and properties express regulation links, i.e., the fact that the expression of the source gene has an influence on the expression of the target gene. The triples of this graph are the following:*

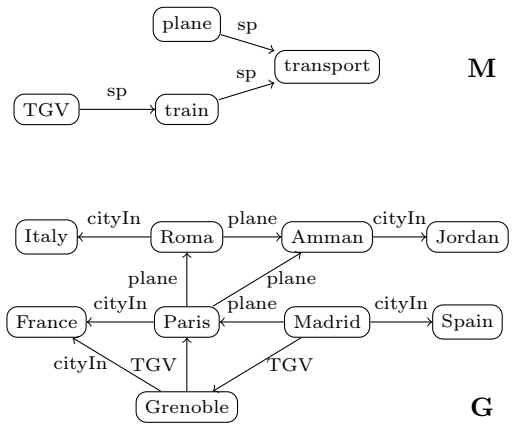

Figure 2: An RDF graph (G) with its schema (M) representing information about transportation means between several cities.

```
dm:bcd rdf:type rn:gene.
dm:bcd rn:inhibits_translation dm:cad.
dm:bcd rn:promotes dm:hb.
dm:bcd rn:promotes dm:kni.
dm:bcd rn:promotes dm:Kr.
dm:bcd rn:inhibits dm:tll.
dm:cad rn:promotes dm:kni.
dm:hb rn:inhibits dm:kni.
dm:hb rn:promotes dm:Kr.
dm:kni rn:inhibits dm:Kr.
dm:tll rn:regulates dm:Kr.
dm:tll rdf:type rn:gene.
```

This example uses only urirefs.

Example 2 (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 2:*

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

For instance, a triple $\langle$`Paris`, `plane`, `Amman`$\rangle$ *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 [22]. The usual notions of validity, satisfiability and consequence are entirely determined by these conditions.

Definition 2 (RDF Interpretation)**.** *Let* $V \subseteq (\mathcal{U} \cup \mathcal{L})$ *be a vocabulary. An* RDF interpretation *of* V *is a tuple* $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ *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;*
- $I_{EXT} : I_P \rightarrow 2^{I_R \times I_R}$ *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 $V \subseteq \mathcal{V}$ be a vocabulary, and G be an RDF graph such that $voc(G) \subseteq V$. An RDF interpretation $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ of V is an* RDF model *of G iff there exists a mapping $\iota' : \mathcal{T}(G) \to I_R$ that extends ι, i.e., $t \in V \cap \mathcal{T}(G) \Rightarrow \iota'(t) = \iota(t)$, such that for each triple $\langle s, p, o \rangle \in G$, $\iota'(p) \in I_P$ and $\langle \iota'(s), \iota'(o) \rangle \in I_{EXT}(\iota'(p))$. The mapping ι' 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 $G \models_{RDF} P$) if and only if each RDF model of G is also an RDF model of P.*

The entailment of RDF graphs (respectively, GRDF graphs) can be characterized in terms of subset for the case of graphs without variables (respectively, in terms of homomorphism when the graphs have variables).

Proposition 1 (RDF, GRDF entailment [22])**.** *$G \models_{RDF} P$ iff $\forall \langle s, p, o \rangle \in P$, then $\langle s, p, o \rangle \in G$. An RDF graph G* RDF-entails *a GRDF graph P iff there exists a mapping $\sigma : \mathcal{T}(P) \to \mathcal{T}(G)$ such that $\forall \langle s, p, o \rangle \in P$, $\langle \sigma(s), \sigma(p), \sigma(o) \rangle \in G$.*

2.2 SPARQL

SPARQL is the RDF query language developed by the W3C [35]. SPARQL query answering is characterized by defining a mapping (shortened here as "map") from the query to the RDF graph to be queried.

We define in the following subsections the syntax and the semantics of SPARQL. For a complete description of SPARQL, the reader is referred to the SPARQL specification [35] or to [32, 34] for its formal semantics. Unless stated otherwise, the report concentrates on SPARQL 1.0, but some features considered in the presented languages are planned to be integrated in SPARQL 1.1.

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, the union (UNION) of graph patterns, an optional (OPT) graph pattern, or a constraint (FILTER).

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 `AND` P'), (P `UNION` P'), (P `OPT` P'), and (P `FILTER` K) are SPARQL graph patterns.*

A SPARQL constraint K is a boolean expression involving terms from $(\mathcal{V} \cup \mathcal{B})$, e.g., a numeric test. We do not specify these expressions further (see [28] for a more complete treatment).

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

Example 3 (Query)**.** *The following query searches, in the regulatory network of Figure 1, a gene ?x which inhibits a product that regulates a product that ?x promotes, and returns these three entities:*

```
SELECT ?x, ?y, ?z
FROM ...
WHERE {
  ?x rn:inhibits ?y
  ?x rn:promotes ?z
  ?y rn:regulates ?z
  ?x rdf:type rn:gene.
  }
```

Example 4 (Query)**.** *The following query searches in the RDF graph of Figure 2 if there exists a direct plane between a city in France and a city in Jordan:*

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

SPARQL provides several other query forms that can be used for formatting query results: CONSTRUCT which can be used for building an RDF graph from the set of answers, ASK which returns TRUE if there is an answer to a given query and FALSE otherwise, and DESCRIBE which can be used for describing a resource RDF graph. We concentrate on the SELECT query form and modify SPARQL basic graph patterns, leaving the remainder of the query forms unchanged.

2.2.2 SPARQL semantics

In the following, we characterize query answering with SPARQL as done in [32]. The approach relies upon the correspondence between GRDF entailment and maps from RDF graph of the query graph patterns to the RDF knowledge base. SPARQL query constructs are defined through algebraic operations on maps: assignments from a set of variables to terms that preserve names.

Definition 6 (Map)**.** *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 $\sigma : V_1 \to V_2$ such that $\forall x \in (V_1 \cap \mathcal{V})$, $\sigma(x) = x$.*

Operations on maps. If σ is a map, then the domain of σ, denoted by $dom(\sigma)$, is the subset of $\mathcal{T}$ on which σ is defined. The restriction of σ to a set of terms X is defined by $\sigma|_X = \{\langle x, y\rangle \in \sigma |\ x \in X\}$ and the completion of σ to a set of terms X is defined by $\sigma|^X = \sigma \cup \{\langle x, \texttt{null}\rangle |\ x \in X$ and $x \notin dom(\sigma)\}$[1].

If P is a graph pattern, then $\mathcal{B}(P)$ is the set of variables occurring in P and $\sigma(P)$ is the graph pattern obtained by the substitution of $\sigma(b)$ to each variable $b \in \mathcal{B}(P)$. Two maps σ_1 and σ_2 are *compatible* when $\forall x \in dom(\sigma_1) \cap dom(\sigma_2)$, $\sigma_1(x) = \sigma_2(x)$. Otherwise, they are said to be incompatible and this is denoted by $\sigma_1 \perp \sigma_2$. If σ_1 and σ_2 are two compatible maps, then we denote by $\sigma = \sigma_1 \oplus \sigma_2 : T_1 \cup T_2 \to \mathcal{T}$ the map defined by $\forall x \in T_1, \sigma(x) = \sigma_1(x)$ and $\forall x \in T_2, \sigma(x) = \sigma_2(x)$. The *join* and *difference* of two sets of maps Ω_1 and Ω_2 are defined as follows [32]:

- *(join)* $\Omega_1 \bowtie \Omega_2 = \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in \Omega_1, \sigma_2 \in \Omega_2$ are compatible$\}$;
- *(difference)* $\Omega_1 \setminus \Omega_2 = \{\sigma_1 \in \Omega_1 \mid \forall \sigma_2 \in \Omega_2, \sigma_1$ and σ_2 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 RDF-entailment. Answers to compound graph patterns are obtained through the operations on maps.

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

Definition 7 (Answers to compound graph patterns)**.** *Let $\models_{RDF}$ be the RDF entailment relation on basic graph patterns, P, P' be SPARQL graph patterns, K be a SPARQL constraint, and G be an RDF graph. The set $\mathcal{S}(P,G)$ of* answers *to P in G is the set of maps from $\mathcal{B}(P)$ to $\mathcal{T}(G)$ defined inductively in the following way:*

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

The symbol $\models_{RDF}$ 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, $\sigma(K) = \top$ means that this function is evaluated to true once the variables in K are substituted by σ. If not all variables of K are bound, then $\sigma(K) \neq \top$. One particular operator that can be used in SPARQL filter conditions is "*bound*($?x$)". This operator returns true if the variable $?x$ is bound and in this case $\sigma(K)$ 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 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* `SELECT` $\vec{B}$ `FROM` u `WHERE` P *be a SPARQL query, G be the RDF graph identified by the URI u, and $\mathcal{S}(P,G)$ be the set of answers to P in G, then the* answers *$\mathcal{A}(\vec{B},G,P)$ to the query are the restriction and completion to $\vec{B}$ of answers to P in G, i.e.,*

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

The completion to null does not prevent that blank nodes remain in answers: null values only replace unmatched variables. [34] defines a different semantics for the join operation when the maps contain null values. This semantics could be adopted instead without changing the remainder of this paper.

Example 5 (Query evaluation)**.** *The evaluation of the query of Example 4 against the RDF graph of Example 1 returns only one answer:*

$$\langle \texttt{dm:bcd}, \texttt{dm:tll}, \texttt{dm:Kr} \rangle$$

In the following, we use an alternate characterization of SPARQL query answering that relies upon the correspondence between GRDF entailment and maps from the query graph patterns to the RDF graph [32]. The previous definition can be rewritten in a more semantic style by extending entailment to compound graph patterns modulo a map σ [5].

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 σ is defined inductively by:*

$$\begin{aligned} G \models \sigma(P \texttt{ AND } P') &\text{ iff } G \models \sigma(P) \text{ and } G \models \sigma(P') \\ G \models \sigma(P \texttt{ UNION } P') &\text{ iff } G \models \sigma(P) \text{ or } G \models \sigma(P') \\ G \models \sigma(P \texttt{ OPT } P') &\text{ iff } G \models \sigma(P) \text{ and } [G \models \sigma(P') \text{ or } \forall \sigma'; G \models \sigma'(P'), \sigma \bot \sigma'] \\ G \models \sigma(P \texttt{ FILTER } K) &\text{ iff } G \models \sigma(P) \text{ and } \sigma(K) = \top \end{aligned}$$

The following proposition is an equivalent characterization of SPARQL answers, closer to a semantic definition.

Proposition 2 (Answer to a SPARQL query [5])**.** *Let* `SELECT` $\vec{B}$ `FROM` u `WHERE` P *be a SPARQL query with P a SPARQL graph pattern and G be the (G)RDF graph identified by the URI u, then the set of answers to this query is*

$$\mathcal{A}(\vec{B}, G, P) = \{\sigma|^{\vec{B}}_{\vec{B}}|\ G \models_{(G)RDF} \sigma(P)\}.$$

The proof given in [5] starts from a slightly different definition than Definition 7 based on RDF-entailment. However, in the case of RDF entailment of a ground triple, these are equivalent thanks to the interpolation lemma [22].

In order to evaluate the complexity of query answering, we use the following problem, usually named query evaluation but better named ANSWER CHECKING:
Problem: $\mathcal{A}$-ANSWER CHECKING
Input: an RDF graph G, a SPARQL graph pattern P, a tuple of variables $\vec{B}$, and a map σ.
Question: Does $\sigma \in \mathcal{A}(\vec{B}, G, P)$?

This problem has usually the same complexity as checking if an answer exists. For SPARQL, the problem has been shown to be PSPACE-complete.

Proposition 3 (Complexity of $\mathcal{A}$-ANSWER CHECKING [32])**.** $\mathcal{A}$-ANSWER CHECKING *is PSPACE-complete.*

The complexity of checking RDF-entailment and GDRF-entailment is NP-complete [21]. This means that $\mathcal{A}$-ANSWER CHECKING when queries are reduced to basic graph patterns is NP-complete. In fact, the addition of AND, FILTER, and UNION does not increase complexity which remains NP-complete. This complexity comes from the addition of the OPT construction [32].

Hence, for every language in which entailment is NP-complete, used as a basic graph pattern language, the problem for this language will have the same complexity since Definition 9 shows the independence of subquery evaluation.

3 Querying RDF modulo RDF Schema

RDF Schema (or RDFS) [12] together with OWL [26] are formal logics recommended by the W3C for defining the vocabulary used in RDF graphs.

RDF Schema further constrains RDF interpretations. Thus RDF graphs when interpreted under a schema may have less models, and thus more consequences. This provides more answers to a query relying on RDFS semantics. However, SPARQL query answers (under RDF entailment) ignore this semantics.

Example 6 (RDF and RDFS entailment)**.** *For instance, in RDF, it is possible to deduce* ⟨`rn:inhibits rdf:type rdf:Property`⟩ *from* ⟨`dm:hb rn:inhibits dm:kni`⟩*; in RDFS, one can deduce* ⟨`dm:hb rn:regulates dm:kni`⟩ *from* {⟨`dm:hb rn:inhibits dm:kni`⟩, ⟨`rn:inhibits rdfs:subPropertyOf rn:regulates`⟩}.

rdfs:domain[dom]	rdfs:Container[cont]	rdf:Bag[bag]
rdfs:range[range]	rdfs:isDefinedBy[isDefined]	rdf:Seq[seq]
rdfs:Class[class]	rdfs:Literal[literal]	rdf:List[list]
rdf:value[value]	rdfs:subClassOf[sc]	rdf:Alt[alt]
rdfs:label[label]	rdfs:subPropertyOf[sp]	rdf: 1[1]
rdf:nil[nil]	rdfs:comment[comment]	...
rdf:type[type]	rdf:predicate[pred]	rdf: i[2]
rdf:object[obj]	rdf:Statement[stat]	rdf:first[first]
rdf:subject[subj]	rdfs:seeAlso[seeAlso]	rdf:rest[rest]
rdf:Property[prop]	rdfs:Datatype[datatype]	rdfs:member[member]
rdfs:Resource[res]	rdf:XMLLiteral[xmlLit]	
	rdfs:ContainerMembershipProperty[contMP]	

Table 1: The RDF Schema vocabulary.

3.1 RDF Schema

This section focusses on RDF and RDFS as extensions of the Simple RDF language presented in Section 2.1. Both extensions are defined in the same way:

- They consider a particular set of urirefs of the vocabulary prefixed by `rdf:` and `rdfs:`, respectively.
- They add additional constraints on the resources associated to these terms in interpretations.

We present them together below.

3.1.1 RDF Schema vocabulary

There exists a set of reserved words, the RDF Schema vocabulary [12], designed to describe relationships between resources like classes, e.g., `dm:gap rdfs:subClassOf rn:gene`, and relationships between properties, e.g., `rn:inhibits rdfs:subPropertyOf rn:regulates`. The RDF Schema vocabulary is given in Table 1 as it appears in [22]. The shortcuts that we will use for each of the terms are given in brackets. We use $rdfsV$ to denote the RDF Schema vocabulary.

We will consider here a core subset of RDFS, ρdf [28], also called the description logic fragment of RDFS [13]. It contains the following vocabulary:

$$\rho\text{df} = \{\texttt{sc}, \texttt{sp}, \texttt{type}, \texttt{dom}, \texttt{range}\}$$

Example 7 (RDFS)**.** *The RDF graph of Example 1 can be expressed in the context of an RDF Schema which provides more information about the vocabulary that it uses. It specifies subtypes of genes and subtypes of regulation relations.*

```
dm:maternal rdfs:subClassOf rn:gene.
dm:gap rdfs:subClassOf rn:gene.
rn:regulates rdfs:domain rn:gene.
rn:regulates rdfs:range rn:gene.
rn:inhibits rdfs:subPropertyOf rn:regulates.
rn:promotes rdfs:subPropertyOf rn:regulates.
rn:inhibits_translation rdfs:subPropertyOf rn:inhibits.
rn:inhibits_transcription rdfs:subPropertyOf rn:inhibits.
```

```
dm:kni rdf:type dm:gap.
dm:hb rdf:type dm:gap.
dm:Kr rdf:type dm:gap.
dm:tll rdf:type dm:gap.
dm:bcd rdf:type dm:maternal.
dm:cad rdf:type dm:maternal.
```

3.1.2 RDF Schema semantics

In addition to the usual interpretation mapping, a special mapping is used in RDFS interpretations for interpreting the set of classes which is a subset of I_R.

Definition 10 (RDFS interpretation)**.** *An* RDFS interpretation *of a vocabulary V is a tuple $\langle I_R, I_P, Class, I_{EXT}, I_{CEXT}, Lit, \iota\rangle$ such that:*

- *$\langle I_R, I_P, I_{EXT}, \iota\rangle$ is an RDF interpretation;*
- *$Class \subseteq I_R$ is a distinguished subset of I_R identifying if a resource denotes a class of resources;*
- *$I_{CEXT} : Class \rightarrow 2^{I_R}$ is a mapping that assigns a set of resources to every resource denoting a class;*
- *$Lit \subseteq I_R$ is the set of literal values, Lit contains all plain literals in $\mathcal{L} \cap V$.*

Specific conditions are added to the resources associated to terms of RDFS vocabularies in an RDFS interpretation to be an RDFS model of an RDFS graph. These conditions include the satisfaction of the RDF Schema axiomatic triples as appearing in the normative semantics of RDF [22].

Definition 11 (RDFS Model)**.** *Let G be an RDFS graph, and $I = \langle I_R, I_P, Class, I_{EXT}, I_{CEXT}, Lit, \iota\rangle$ be an RDFS interpretation of a vocabulary $V \subseteq rdfsV \cup \mathcal{V}$ such that $\mathcal{V}(G) \subseteq V$. Then I is an* RDFS model *of G if and only if I satisfies the following conditions:*

1. *Simple semantics:*
   a) *there exists an extension ι' of ι to $\mathcal{B}(G)$ such that for each triple $\langle s, p, o\rangle$ of G, $\iota'(p) \in I_P$ and $\langle \iota'(s), \iota'(o)\rangle \in I_{EXT}(\iota'(p))$.*
2. *RDF semantics:*
   a) *$x \in I_P \Leftrightarrow \langle x, \iota'(\texttt{prop})\rangle \in I_{EXT}(\iota'(\texttt{type}))$.*
   b) *If $\ell \in term(G)$ is a typed XML literal with lexical form w, then $\iota'(\ell)$ is the XML literal value of w, $\iota'(\ell) \in Lit$, and $\langle \iota'(\ell), \iota'(\texttt{xmlLit})\rangle \in I_{EXT}(\iota'(\texttt{type}))$.*
3. *RDFS Classes:*
   a) *$x \in I_R$, $x \in I_{CEXT}(\iota'(\texttt{res}))$.*
   b) *$x \in Class$, $x \in I_{CEXT}(\iota'(\texttt{class}))$.*
   c) *$x \in Lit$, $x \in I_{CEXT}(\iota'(\texttt{literal}))$.*
4. *RDFS Subproperty:*
   a) *$I_{EXT}(\iota'(\texttt{sp}))$ is transitive and reflexive over I_P.*
   b) *if $\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{sp}))$ then $x, y \in I_P$ and $I_{EXT}(x) \subseteq I_{EXT}(y)$.*
5. *RDFS Subclass:*
   a) *$I_{EXT}(\iota'(\texttt{sc}))$ is transitive and reflexive over $Class$.*
   b) *$\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{sc}))$, then $x, y \in Class$ and $I_{CEXT}(x) \subseteq I_{CEXT}(y)$.*
6. *RDFS Typing:*
   a) *$x \in I_{CEXT}(y)$, $(x, y) \in I_{EXT}(\iota'(\texttt{type}))$.*
   b) *if $\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{dom}))$ and $\langle u, v\rangle \in I_{EXT}(x)$ then $u \in I_{CEXT}(y)$.*
   c) *if $\langle x, y\rangle \in I_{EXT}(\iota'(\texttt{range}))$ and $\langle u, v\rangle \in I_{EXT}(x)$ then $v \in I_{CEXT}(y)$.*

$$\frac{\langle p, \texttt{sp}, q\rangle \quad \langle q, \texttt{sp}, r\rangle}{\langle p, \texttt{sp}, r\rangle} \qquad \frac{\langle A, \texttt{sc}, B\rangle \quad \langle B, \texttt{sc}, C\rangle}{\langle A, \texttt{sc}, C\rangle} \qquad \frac{\langle p, \texttt{dom}, A\rangle \quad \langle x, p, y\rangle}{\langle x, \texttt{type}, A\rangle}$$

$$\frac{\langle p, \texttt{sp}, q\rangle \quad \langle x, p, y\rangle}{\langle x, q, y\rangle} \qquad \frac{\langle A, \texttt{sc}, B\rangle \quad \langle x, \texttt{type}, A\rangle}{\langle x, \texttt{type}, B\rangle} \qquad \frac{\langle p, \texttt{range}, A\rangle \quad \langle x, p, y\rangle}{\langle y, \texttt{type}, A\rangle}$$

Table 2: RDFS inference rules (from [28]).

7. *RDFS Additionals:*
   a) *if* $x \in Class$ *then* $\langle x, \iota'(\texttt{res})\rangle \in I_{EXT}(\iota'(\texttt{sc}))$.
   b) *if* $x \in I_{CEXT}(\iota'(\texttt{datatype}))$ *then* $\langle x, \iota'(\texttt{literal})\rangle \in I_{EXT}(\iota'(\texttt{sc}))$.
   c) *if* $x \in I_{CEXT}(\iota'(\texttt{contMP}))$ *then* $\langle x, \iota'(\texttt{member})\rangle \in I_{EXT}(\iota'(\texttt{sp}))$.

Definition 12 (RDFS entailment)**.** *Let G and P be two RDFS graphs, then G* RDFS-entails *P (denoted by $G \models_{RDFS} P$) if and only if every RDFS model of G is also an RDFS model of P.*

[28] has introduced the reflexive relaxed semantics for RDFS in which `rdfs:subClassOf` and `rdfs:subPropertyOf` do not have to be reflexive. The entailment relation with this semantics is noted $\models^{\text{nrx}}_{\text{RDFS}}$.

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. For instance, ⟨`dm:hb rn:inhibits dm:kni`⟩ only entails ⟨`rn:inhibits sp rn:inhibits`⟩ and variations of this triple in which occurrences of `rn:inhibits` are replaced by variables. 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` predicates. Therefore, it is assumed that elements of the RDFS vocabulary appear only in the predicate position. We will call *genuine*, RDFS graphs which do not constrain the elements of the ρdf vocabulary (and thus these two predicates), and restrict us to querying genuine RDFS graphs.

However, when issuing queries involving these relations, e.g., with a graph pattern like ⟨`?x sp ?y`⟩, all properties in the graph will be answers. Since this would clutter results, we assume, as done in [28], that queries use the reflexive relaxed semantics. It is easy to recover the standard semantics by providing the additional triples when `sp` or `sc` are queried.

In the following, we use the closure graph of an RDF graph G, denoted by $closure(G)$, which is defined by the graph obtained by saturating G with all triples that can be deduced using rules of Table 2.

The SPARQL specification [35] introduces the notion of entailment regimes. These regimes contain several components (query language, graph language, inconsistency handling) [19]. We concentrate here on the definition of answers which, in particular, replace simple RDF entailment for answering queries. It is possible to define answers to SPARQL queries modulo RDF Schema, by using RDFS entailment as the entailment regime.

Definition 13 (Answers to a SPARQL query modulo RDF Schema)**.** *Let* `SELECT` $\vec{B}$ `FROM` u `WHERE` P *be a SPARQL query with P a GRDF graph and G be the RDFS graph identified by the URI u, then the set of* answers *to this query* modulo RDF Schema *is:*

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \{\sigma|^{\vec{B}}_{\vec{B}} | G \models^{nrx}_{RDFS} \sigma(P)\}$$

This definition is justified by the analogy between RDF entailment and RDFS entailment in the definition of answers to queries (see [5]). It does not fully correspond to the RDFS entailment regime defined in [19] since we do not restrict the answer set to be finite. This is not strictly necessary, however, the same restrictions as in [19] can be applied.

The problem is to specify a query engine that can find such answers.

3.2 Querying against ter Horst closure

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 information that can be deduced using a set of predefined rules called RDFS rules, before evaluating the query over the closure graph.

Definition 14 (RDFS closure)**.** *Let G be an RDFS graph on an RDFS vocabulary V. The* RDFS closure *of G, denoted $\hat{G}$, is the smallest set of triple containing G and satisfying the following constraints:*

[RDF1] *all RDF axiomatic triples [22] are in $\hat{G}$;*
[RDF2] *if $\langle s, p, o\rangle \in \hat{G}$, then $\langle p, \texttt{type}, \texttt{prop}\rangle \in \hat{G}$;*
[RDF3] *if $\langle s, p, \ell\rangle \in \hat{G}$, where ℓ is an $\texttt{xmlLit}$ typed literal and the lexical representation s is a well-formed XML literal,*
then $\langle s, p, xml(s)\rangle \in \hat{G}$ and $\langle xml(s), \texttt{type}, \texttt{xmlLit}\rangle \in \hat{G}$;
[RDFS 1] *all RDFS axiomatic triples [22] are in $\hat{G}$;*
[RDFS 6] *if $\langle a, \texttt{dom}, x\rangle \in \hat{G}$ and $\langle u, a, y\rangle \in \hat{G}$, then $\langle u, \texttt{type}, x\rangle \in \hat{G}$;*
[RDFS 7] *if $\langle a, \texttt{range}, x\rangle \in \hat{G}$ and $\langle u, a, v\rangle \in \hat{G}$, then $\langle v, \texttt{type}, x\rangle \in \hat{G}$;*
[RDFS 8a] *if $\langle x, \texttt{type}, \texttt{prop}\rangle \in \hat{G}$, then $\langle x, \texttt{sp}, x\rangle \in \hat{G}$;*
[RDFS 8b] *if $\langle x, \texttt{sp}, y\rangle \in \hat{G}$ and $\langle y, \texttt{sp}, z\rangle \in \hat{G}$, then $\langle x, \texttt{sp}, z\rangle \in \hat{G}$;*
[RDFS 9] *if $\langle a, \texttt{sp}, b\rangle \in \hat{G}$ and $\langle x, a, y\rangle \in \hat{G}$, then $\langle x, b, y\rangle \in \hat{G}$;*
[RDFS 10] *if $\langle x, \texttt{type}, \texttt{class}\rangle \in \hat{G}$, then $\langle x, \texttt{sc}, \texttt{res}\rangle \in \hat{G}$;*
[RDFS 11] *if $\langle u, \texttt{sc}, x\rangle \in \hat{G}$ and $\langle y, \texttt{type}, u\rangle \in \hat{G}$, then $\langle y, \texttt{type}, x\rangle \in \hat{G}$;*
[RDFS 12a] *if $\langle x, \texttt{type}, \texttt{class}\rangle \in \hat{G}$, then $\langle x, \texttt{sc}, x\rangle \in \hat{G}$;*
[RDFS 12b] *if $\langle x, \texttt{sc}, y\rangle \in \hat{G}$ and $\langle y, \texttt{sc}, z\rangle \in \hat{G}$, then $\langle x, \texttt{sc}, z\rangle \in \hat{G}$;*
[RDFS 13] *if $\langle x, \texttt{type}, \texttt{contMP}\rangle \in \hat{G}$, then $\langle x, \texttt{prop}, \texttt{member}\rangle \in \hat{G}$;*
[RDFS 14] *if $\langle x, \texttt{type}, \texttt{datatype}\rangle \in \hat{G}$, then $\langle x, \texttt{sc}, \texttt{literal}\rangle \in \hat{G}$.*

It is easy to show that this closure always exists and can be obtained by turning the constraints into rules, thus defining a closure operation.

Example 8 (RDFS Closure)**.** *The RDFS closure of the RDF graph of Example 1 augmented by the RDFS triples of Example 7 contains, in particular, the following assertions:*

```
dm:bcd rn:inhibits dm:cad. // [RDFS 9]
dm:hb rn:regulates dm:kni. // [RDFS 9]
dm:hb type rn:gene.        // [RDFS 6]
```

Because of axiomatic triples, this closure may be infinite, but a finite and polynomial closure, called *partial closure*, has been proposed independently in [10] and [38].

Definition 15 (Partial RDFS closure)**.** *Let G and H be two RDFS graphs on an RDFS vocabulary V, the* partial RDFS closure *of G given H, denoted $\hat{G}\backslash H$, is obtained in the following way:*

1. *let k be the maximum of i's such that $\texttt{rdf:_}i$ is a term of G or of H;*
2. *replace the rule [RDF 1] by the rule*
   [RDF 1P] *add all RDF axiomatic triples [22] except those that use $\texttt{rdf:_}i$ with $i > k$;*
   In the same way, replace the rule [RDFS 1] by the rule
   [RDFS 1P] *add all RDFS axiomatic triples except those that use $\texttt{rdf:_}i$ with $i > k$;*
3. *apply the modified rules.*

Applying the partial closure to an RDFS graph permits to reduce RDFS entailment to simple RDF entailment.

Proposition 4 (Completeness of partial RDFS closure [22])**.** *Let G be a satisfiable RDFS graph and H an RDFS graph, then $G \models_{RDFS} H$ if and only if $(\hat{G} \backslash H) \models_{RDF} H$.*

The completeness does not hold if G is not satisfiable because in such a case, any graph H is a consequence of G and $\models_{RDF}$ does not reflect this (no RDF graph can be inconsistent). In case G is unsatisfiable, the RDFS entailment regime allows for raising an error [19]. An RDFS graph can be unsatisfiable only if it contains datatype conflicts [38] which can be found in polynomial time.

Since queries must adopt the reflexive relaxed semantics, we have to further restrict this closure. It can be obtained by suppressing constraints RDFS8a and RDFS12a from the closure operation. We denote the partial non reflexive closure $\hat{G} \backslash\backslash H$.

Proposition 5 (Completeness of partial non reflexive RDFS closure)**.** *Let G be a satisfiable genuine RDFS graph and H an RDFS graph, then $G \models^{nrx}_{RDFS} H$ if and only if $(\hat{G} \backslash\backslash H) \models_{RDF} H$.*

This has the following corollary:

Corollary 1.

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \mathcal{A}(\vec{B}, \hat{G} \backslash\backslash P, P)$$

Example 9 (SPARQL evaluation modulo RDFS)**.** *If the query of Example 4 is evaluated against the RDFS closure of Example 8, it will return the three expected answers:*

$$\{\langle \texttt{dm:hb}, \texttt{dm:kni}, \texttt{dm:Kr}\rangle$$
$$\langle \texttt{dm:bcd}, \texttt{dm:tll}, \texttt{dm:Kr}\rangle$$
$$\langle \texttt{dm:bcd}, \texttt{dm:cad}, \texttt{dm:kni}\rangle\}$$

This can be obtained easily from the simple graph of Example 1 augmented by the triples of Example 8.

This shows the correctness and completeness of the closure approach. This approach has several drawbacks which limit its use: It still tends to generate a very large graph which makes it not very convenient, especially if the transformation has to be made on the fly, i.e., when the query is evaluated; It takes time proportional to $|H| \times |G|^2$ in the worst case [28]; Moreover, it is not applicable if one has no access to the graph to be queried but only to a SPARQL endpoint. In this case, it is not possible to compute the closure graph.

Since the complexity of the partial closure has been shown to be polynomial [38], $\mathcal{A}^{\#}$-Answer checking remains PSPACE-complete.

Proposition 6 (Complexity of $\mathcal{A}^{\#}$-Answer checking)**.** $\mathcal{A}^{\#}$-Answer checking *is PSPACE-complete.*

4 The PSPARQL query language

In order to address the problems raised by querying RDF graphs modulo RDF Schemas, we first present the PSPARQL query language for RDF which we introduced in [5]. Unlike SPARQL, PSPARQL can express queries with regular path expressions instead of relations and variables on the edges. For instance, it allows for finding all maternal genes regulated by the `dm:bcd` gene through an arbitrary sequence of inhibitions.

We will, in the next section, show how the additional expressive power provided by PSPARQL can be used for answering queries modulo RDF Schema.

The added expressiveness of PSPARQL is achieved by extending SPARQL graph patterns, hence any SPARQL query is a PSPARQL query. SPARQL graph patterns, based on GRDF graphs, are replaced by PRDF graphs that are introduced below through their syntax (§4.1) and semantics (§4.2). They are then naturally replaced within the PSPARQL context (§4.3).

4.1 PRDF syntax

Let Σ be an alphabet. A *language* over Σ is a subset of Σ^*: its elements are sequences of elements of Σ called *words*. A (non empty) word $\langle a_1, \ldots, a_k \rangle$ is denoted $a_1 \cdot \ldots \cdot a_k$. If $A = a_1 \cdot \ldots \cdot a_k$ and $B = b_1 \cdot \ldots \cdot b_q$ are two words over Σ, then $A \cdot B$ is the word over Σ defined by $A \cdot B = a_1 \cdot \ldots \cdot a_k \cdot b_1 \cdot \ldots \cdot b_q$.

Definition 16 (Regular expression pattern)**.** *Let Σ be an alphabet, X be a set of variables, the set $\mathcal{R}(\Sigma, X)$ of* regular expression patterns *is inductively defined by:*

- $\forall a \in \Sigma$, $a \in \mathcal{R}(\Sigma, X)$ *and* $!a \in \mathcal{R}(\Sigma, X)$*;*
- $\forall x \in X$, $x \in \mathcal{R}(\Sigma, X)$*;*
- $\epsilon \in \mathcal{R}(\Sigma, X)$*;*
- *If* $A \in \mathcal{R}(\Sigma, X)$ *and* $B \in \mathcal{R}(\Sigma, X)$ *then* $A|B$, $A \cdot B$, A^*, $A^+ \in \mathcal{R}(\Sigma, X)$.

A regular expression over $(\mathcal{U}, \mathcal{B})$ can be used to define a language over the alphabet made of $\mathcal{U} \cup \mathcal{B}$. PRDF graphs are GRDF graphs where predicates in the triples are regular expression patterns constructed over the set of URI references and the set of variables.

Definition 17 (PRDF graph)**.** *A* PRDF triple *is an element of* $\mathcal{U} \cup \mathcal{B} \times \mathcal{R}(\mathcal{U}, \mathcal{B}) \times \mathcal{T}$. *A PRDF graph is a set of PRDF triples.*

Hence all GRDF graphs are PRDF graphs.

Example 10 (PRDF Graph)**.** *PRDF graphs can express interesting features of regulatory networks. For instance, one may observe that* `dm:bcd` *promotes* `dm:Kr` *without knowing if this action is direct or indirect. Hence, this can be expressed by* `dm:bcd rn:promotes+ dm:Kr`.

A generalized version of the graph pattern of the query of Example 4 can be expressed by:

```
?x rn:inhibits.rn:regulates* ?z.
?x rn:promotes+ ?z.
?x rdf:type rn:gene.
```

4.2 PRDF semantics

To be able to define models of PRDF graphs, we have first to express path semantics within RDF semantics to support regular expressions.

Definition 18 (Support of a regular expression)**.** *Let $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ be an interpretation of a vocabulary $V = \mathcal{U} \cup \mathcal{L}$, ι' be an extension of ι to $B \subseteq \mathcal{B}$, and $R \in \mathcal{R}(\mathcal{U}, B)$, a pair $\langle x, y \rangle$ of $(I_R \times I_R)$* supports *R in ι' if and only if one of the two following conditions are satisfied:*

(i) the empty word $\epsilon \in L^*(R)$ and $x = y$;

(ii) there exists a word of length $n \geq 1$ $w = w_1 \cdot \ldots \cdot w_n$ where $w \in L^*(R)$ and a sequence of resources of I_R $x = r_0, \ldots, r_n = y$ such that $\langle r_{i-1}, r_i \rangle \in I_{EXT}(\iota'(w_i))$, $1 \leq i \leq n$.

Instead of considering paths in RDF graphs, this definition considers paths in the interpretations of PRDF graphs, i.e., paths are now relating resources. This is used in the following definition of PRDF models in which it replaces the direct correspondences that exist in RDF between a relation and its interpretation, by a correspondence between a regular expression and a sequence of relation interpretations. This allows for matching regular expressions, e.g., r^+, with variable length paths.

Definition 19 (Model of a PRDF graph)**.** *Let G be a PRDF graph, and $I = \langle I_R, I_P, I_{EXT}, \iota\rangle$ be an interpretation of a vocabulary $V \supseteq \mathcal{V}(G)$, I is a* PRDF model *of G if and only if there exists an extension ι' of ι to $\mathcal{B}(G)$ such that for every triple $\langle s, R, o\rangle \in G$, $\langle \iota'(s), \iota'(o)\rangle$ supports R in ι'.*

This definition extends the definition of RDF models, and they are equivalent when all regular expression patterns R are reduced to atomic terms, i.e., urirefs or variables. PRDF entailment is defined as usual:

Definition 20 (PRDF entailment)**.** *Let P and G be two PRDF graphs, G* PRDF-entails *P (noted $G \models_{PRDF} P$) if and only if all models of G are models of P.*

It is possible to define the interpretation of a regular expression evaluation as those pairs of resources which support the expression in all models.

Definition 21 (Regular expression interpretation)**.** *The interpretation $[\![R]\!]_G$ of a regular expression R in a PRDF graph G is the set of nodes which satisfy the regular expression in all models of the graph:*

$$[\![R]\!]_G = \{\langle x, y\rangle |\ \forall I \text{ PRDF model of } G, \langle x, y\rangle \text{ supports } R \text{ in } I\}$$

It is thus possible to formulate the regular expression evaluation problem:
Problem: Regular expression evaluation
Input: An RDF graph G, a regular expression R, and a pair $\langle a, b\rangle$
Question: Does $\langle a, b\rangle \in [\![R]\!]_G$?

We will use this same problem with different type of regular expressions.

4.3 PSPARQL

PSPARQL is an extension of SPARQL introducing the use of paths in SPARQL graph patterns. PSPARQL graph patterns are built on top of PRDF in the same way as SPARQL is built on top of GRDF.

Definition 22 (PSPARQL graph pattern)**.** *A* PSPARQL graph pattern *is defined inductively by:*

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

A PSPARQL query for the select form is `SELECT` $\vec{B}$ `FROM` u `WHERE` P such that P is a PSPARQL graph pattern.

Analogously to SPARQL, the set of answers to a PSPARQL query is defined inductively from the set of maps of the PRDF graphs of the query into the RDF knowledge base. The definition of an answer to a PSPARQL query is thus identical to Definition 9, but it uses PRDF entailment.

Definition 23 (Answers to a PSPARQL query)**.** *Let* `SELECT` $\vec{B}$ `FROM` u `WHERE` P *be a PSPARQL query with P a PRDF graph pattern, and G be the RDF graph identified by the URI u, then the set of* answers *to this query is:*

$$\mathcal{A}^{\star}(\vec{B}, G, P) = \{\sigma|^{\vec{B}}_{\vec{B}} | G \models_{PRDF} \sigma(P)\}$$

4.4 SPARQL queries modulo RDFS with PSPARQL

To overcome the limitations of previous approaches when querying RDF graphs modulo an RDF Schema, we provide a new approach which rewrites a SPARQL query into a PSPARQL query using a set of rules, and then evaluates the transformed query over the graph to be queried. In particular, we show that every SPARQL query Q to evaluate over an RDFS graph G can be transformed into a PSPARQL query $\tau(Q)$ such that evaluating Q over $\hat{G}$, the closure graph of G, is equivalent to evaluating $\tau(Q)$ over G.

The query rewriting approach is similar in spirit to the query rewriting methods using a set of views [31, 14, 20]. In contrast to these methods, our approach uses the data contained in the graph, i.e., the rules are inferred from RDFS entailment rules. We define a rewriting function τ from RDF graph patterns to PRDF graph patterns through a set of rewriting rules over triples (which naturally extends to basic graph patterns and queries). $\tau(Q)$ is obtained from Q by applying the possible rule(s) to each triple in Q.

Definition 24 (Basic RDFS graph pattern expansion)**.** *Given an RDF triple t, the* RDFS expansion *of t is a finite PSPARQL graph pattern $\tau(t)$ defined as:*

$$\begin{aligned}
\tau(\langle s, \texttt{sc}, o\rangle) =& \{\langle s, \texttt{sc}^+, o\rangle\}\\
\tau(\langle s, \texttt{sp}, o\rangle) =& \{\langle s, \texttt{sp}^+, o\rangle\}\\
\tau(\langle s, p, o\rangle) =& \{\langle s, ?x, o\rangle, \langle ?x, \texttt{sp}^*, p\rangle\}(p \notin \{\texttt{sc}, \texttt{sp}, \texttt{type}\})\\
\tau(\langle s, \texttt{type}, o\rangle) =& \{\langle s, \texttt{type} \cdot \texttt{sc}^*, o\rangle\}\\
& \texttt{UNION}\ \{\langle s, ?p, ?y\rangle, \langle ?p, \texttt{sp}^* \cdot \texttt{dom} \cdot \texttt{sc}^*, o\rangle\}\\
& \texttt{UNION}\ \{\langle ?y, ?p, s\rangle, \langle ?p, \texttt{sp}^* \cdot \texttt{range} \cdot \texttt{sc}^*, o\rangle\}
\end{aligned}$$

The first rule handles the transitive semantics of the subclass relation. Finding the subclasses of a given class can be achieved by navigating all its direct subclasses. The second rule handles similarly the transitive semantics of the subproperty relation. The third rule tells that the subject-object pairs occurring in the subproperties of a given property are inherited to it. Finally, the fourth rule expresses that the instance mapped to s has for type the class mapped to o (we use the word "mapped" since s and/or o can be variables) if one of the following conditions holds:

1. the instance mapped to s has for type one of the subclasses of the class mapped to o by following the subclass relationship zero or several times. The zero times is used since s can be directly of type o;
2. there exists a property of which s is subject and such that the instances appearing as a subject must have for type one of the subclasses of the class mapped to o;
3. there exists a property of which s is object and such that the instances appearing as an object must have for type one of the subclasses of the class mapped to o.

The latter rule takes advantage of a feature of PSPARQL: the ability to have variables in predicates.

Example 11 (PSPARQL Query)**.** *The result of transforming the query of Example 4 with τ is:*

```
SELECT ?x, ?y, ?z
FROM ...
WHERE {
  ?x ?r ?y. ?r sp* rn:inhibits.
  ?y ?t ?z.?t sp* rn:regulates.
  ?x ?s ?z. ?s sp* rn:promotes.
  ( ?x rdf:type.sc* rn:gene.
  UNION
   { ?x ?u ?v. ?u sp*.dom.sc* rn:gene.}
  UNION
   { ?v ?u ?x. ?u sp*.range.sc* rn:gene.}
  )
  }
```

This query provides the correct set of answers for the RDF graph of Example 1 modulo the RDF Schema of Example 7 (given in Example 9).

Any SPARQL query can be answered modulo an RDF Schema by rewriting the resulting query in PSPARQL and evaluating the PSPARQL query.

Proposition 7 (Answers to a SPARQL query modulo RDF Schema by PSPARQL)**.**

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \mathcal{A}^{\star}(\vec{B}, G, \tau(P))$$

This transformation does not increase the complexity of PSPARQL which is the same as the one of SPARQL:

Proposition 8 (Complexity of $\mathcal{A}^{\star}$-Answer checking)**.** $\mathcal{A}^{\star}$-Answer checking *is PSPACE-complete.*

5 nSPARQL and NSPARQL

An alternative to PSPARQL was proposed with the nSPARQL language, a simple query language based on nested regular expressions for navigating RDF graphs [33]. We present it as well as NSPARQL, an extension more comparable to PSPARQL.

5.1 nSPARQL syntax

Definition 25 (Regular expression)**.** *A* regular expression *is an expression built from the following grammar:*

$$re ::= axis \mid axis{::}a \mid re \mid re/re \mid re|re \mid re^*$$

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

In the following, we use the positive closure of a path expression R denoted by R^+ and defined as $R^+ = R/R^*$.

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

The model underlying nSPARQL is that of XPath which navigates within XML structures. Hence, the axis denotes the type of node object which is selected at each step, respectively, the current node (`self` or $\texttt{self}^{-1}$), the nodes reachable through an outbound triple (`next`), the nodes that can reach the current node through an incident triple ($\texttt{next}^{-1}$), the properties of outbound triples (`edge`), the properties of incident triples ($\texttt{edge}^{-1}$), the object of a predicate (`node`) and the predicate of an object ($\texttt{node}^{-1}$). This is illustrated by Figure 3.

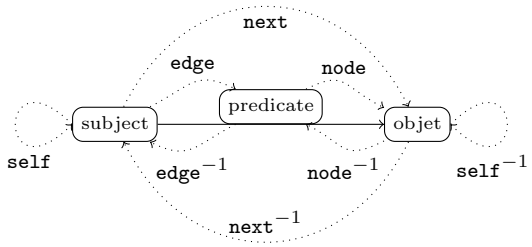

Figure 3: nSPARQL axes.

Definition 26 (Nested regular expression)**.** *A* nested regular expression *is an expression built from the following grammar (with $a \in \mathcal{U}$):*

$$nre ::= axis \mid axis{::}a \mid axis{::}[nre] \mid nre \mid nre/nre \mid nre|nre \mid nre^*$$

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

Example 12 (nSPARQL triple pattern)**.** *Assume that one wants to retrieve the pairs of cities such that there is a way of traveling by any transportation mean. The following nSPARQL pattern expresses this query:*

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

This pattern expresses a sequence of properties such that each property (predicate) is a sub-property of the property "transport".

Example 13 (nSPARQL triple pattern)**.** *In the context of molecular biology, nSPARQL expressions may be very useful. For instance, part of the graph patterns used for the query of Example 4 can be expressed by:*

```
?x next::rn:inhibits / next::rn:regulates ?z
```

which finds all pairs of nodes such that the first one inhibits a regulator of the second one. It can be further enhanced by using transitive closure:

```
?x next::rn:inhibits / next::rn:regulates+ ?z
```

expressing that we want a path between two nodes going through a first inhibition and then an arbitrary non null number of regulatory steps (`+` is the usual notation such that `a+` corresponds to `a/a*`). Nested expressions allow for going further by constraining any step in the path. So,

```
?x next::rn:inhibits[ next::rdf:type/self::dm:gap ] / next::rn:regulates+ ?z
```

requires, in addition, the second node in the path to be a gap gene.

From nSPARQL triple patterns, it is also possible to create a query language from nSPARQL triple patterns. As in SPARQL, a set of nSPARQL triple patterns is called an nSPARQL basic graph pattern. nSPARQL graph patterns may be defined in the usual way, i.e., by replacing triple patterns by nSPARQL triple patterns.

Definition 28 (nSPARQL graph pattern)**.** *An* nSPARQL graph pattern *is defined inductively by:*

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

For time complexity reasons the designers of the nSPARQL language choose to define a more restricted language than SPARQL [33]. 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.

5.2 nSPARQL semantics

In order to define the semantics of nSPARQL, we need to know the semantics of nested regular expressions [33]. Here we depart from the semantics given in [33] by adding a third variable in the interpretation whose sole purpose is to compact the set of rules. Both definitions are equivalent.

Definition 29 (Nested path interpretation)**.** *Given a nested path p and an RDF graph G, the interpretation of p in G (denoted $[\![p]\!]_G$) is defined by:*

$$
\begin{aligned}
[\![\texttt{self}]\!]_G &= \{\langle x,x,x\rangle; x \in \mathcal{T}\}\\
[\![\texttt{next}]\!]_G &= \{\langle x,y,z\rangle; \exists z; \langle x,z,y\rangle \in G\}\\
[\![\texttt{edge}]\!]_G &= \{\langle x,z,z\rangle; \exists z; \langle x,z,y\rangle \in G\}\\
[\![\texttt{node}]\!]_G &= \{\langle z,y,z\rangle; \exists z; \langle x,z,y\rangle \in G\}\\
[\![nre\texttt{::a}]\!]_G &= \{\langle x,y,a\rangle \in [\![nre]\!]_G\}\\
[\![nre^{-1}]\!]_G &= \{\langle y,x,z\rangle; \langle x,y,z\rangle \in [\![nre]\!]_G\}\\
[\![nre_1\texttt{[}nre_2\texttt{]}]\!]_G &= \{\langle x,y,z\rangle \in [\![nre_1]\!]_G\,; \exists w,k; \langle z,w,k\rangle \in [\![nre_2]\!]_G\}\\
[\![nre_1\texttt{/}nre_2]\!]_G &= \{\langle x,y,z\rangle; \langle x,w,k\rangle \in [\![nre_1]\!]_G \,\&\, \langle w,y,z\rangle \in [\![nre_2]\!]_G\}\\
[\![nre_1\texttt{|}nre_2]\!]_G &= [\![nre_1]\!]_G \cup [\![nre_2]\!]_G\\
[\![nre\texttt{*}]\!]_G &= [\![\texttt{self}]\!]_G \cup [\![nre]\!]_G \cup [\![nre\texttt{/}nre]\!]_G \cup [\![nre\texttt{/}nre\texttt{/}nre]\!]_G \cup \ldots
\end{aligned}
$$

The evaluation of a nested regular expression R over an RDF graph G is defined as the sets of pairs $\langle a,b\rangle$ of nodes in G, such that b is reachable from a in G by following a path that conforms to R. We will write $\langle x,y\rangle \in [\![R]\!]_G$ as a shortcut for $\exists z$ such that $\langle x,y,z\rangle \in [\![R]\!]_G$.

Definition 30 (Satisfaction of a nSPARQL triple pattern)**.** *Given a basic nested path graph pattern $\langle s,p,o\rangle$ and an RDF graph G, $\langle s,o\rangle$ satisfies p in G (denoted $G \models_{nSPARQL} \langle s,p,o\rangle$) if and only if $\exists\sigma$; $\langle \sigma(s),\sigma(o)\rangle \in [\![p]\!]_G$*

This nested regular expression evaluation problem is solved efficiently through an effective procedure provided in [33].

Theorem 1 (Complexity of nested regular expression evaluation [33]). *The evaluation problem for a nested regular expression R over an RDF graph G can be solved in time $O(|G| \times |R|)$.*

Answers to nSPARQL queries follow the same definition as for SPARQL (Definition 9) but with maps satisfying nSPARQL triple patterns.

Definition 31 (Evaluation of a nSPARQL triple pattern). *The evaluation of a nSPARQL triple pattern $\langle x, R, y\rangle$ over an RDF graph G is:*

$$[\![\langle x, R, y\rangle]\!]_G = \{\sigma | dom(\sigma) = \{x, y\} \cap \mathcal{B} \text{ and } \langle \sigma(x), \sigma(y)\rangle \in [\![R]\!]_G\}$$

[33] shows that avoiding the projection operator (SELECT), keeps the complexity of nSPARQL *basic*, i.e., conjunctive, graph pattern evaluation to polynomial and mentions that adding projection would make it NP-complete.

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.

5.3 NSPARQL

It is also possible to create a query language from nSPARQL triple patterns by simply replacing SPARQL triple patterns by nSPARQL triple patterns. We call such a language NSPARQL for differentiating it from the original nSPARQL [33]. However, the merits of the approach are directly inherited from the original nSPARQL.

A NSPARQL query for the select form is `SELECT` $\vec{B}$ `FROM` u `WHERE` P such that P is a nSPARQL graph pattern (see Definition 28). Hence, NSPARQL graph patterns are built on top of nSPARQL in the same way as SPARQL is built on top of GRDF and PSPARQL is built on top of PRDF.

Answers to NSPARQL queries are based on the extension of $\models_{nSPARQL}$ nSPARQL graph patterns following Definition 9 using $\models_{nSPARQL}$ as the entailment relation.

Definition 32 (Answers to an NSPARQL query). *Let* `SELECT` $\vec{B}$ `FROM` u `WHERE` P *be a NSPARQL query with P a nSPARQL graph pattern and G be the (G)RDF graph identified by the URI u, then the set of* answers *to this query is:*

$$\mathcal{A}^o(\vec{B}, G, P) = \{\sigma|_{\vec{B}}^{\vec{B}} |\ G \models_{nSPARQL} \sigma(P)\}.$$

The complexity of NSPARQL query evaluation is likely to be PSPACE-complete as SPARQL (see §2.2.2).

5.4 SPARQL queries modulo RDFS with nSPARQL

Definition 33. *The evaluation of an nSPARQL triple pattern $\langle x, R, y\rangle$ over an RDF graph G modulo RDFS is defined as*

$$[\![\langle x, R, y\rangle]\!]_G^{rdfs} = [\![\langle x, R, y\rangle]\!]_{closure(G)}$$

Definition 34 (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:*

$$\mathcal{S}^o(P, G) = \bigcap_{t \in P} [\![t]\!]_G^{rdfs}$$

As presented in [33], nSPARQL can evaluate queries with respect to RDFS by transforming the queries with rules [28]:

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

Example 14 (nSPARQL evaluation modulo RDFS)**.** *The following nSPARQL graph pattern could be used as a query to retrieve the set of pairs of cities connected by a sequence of transportation means such that one city is from France and the other city is from Jordan:*

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

When evaluating this graph pattern against the RDF graph of Figure 2, it returns the empty set since there is no explicit "transport" property between the two queried cities. However, considering the RDFS semantics, it should return the following set of pairs:

$$\{\langle ?city_1 \leftarrow Paris, ?city_2 \leftarrow Amman\rangle, \langle ?city_1 \leftarrow Grenoble, ?city_2 \leftarrow Amman\rangle\}$$

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

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

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

Theorem 2 (Completeness of ϕ on triples [33] (Theorem 3))**.** *Let $\langle x, p, y\rangle$ be a SPARQL triple pattern with $x, y \in (\mathcal{U} \cup \mathcal{B})$ and $p \in \mathcal{U}$, then for any RDF graph G:*

$$[\![\langle x, p, y\rangle]\!]_G^{rdfs} = [\![\langle x, \phi(p), y\rangle]\!]_G$$

This results uses the natural extension of ϕ to nSPARQL graph patterns to answer SPARQL queries modulo RDFS.

Proposition 9 (Completeness of ϕ on graph patterns [33])**.** *Given a basic graph pattern P and an RDFS graph G,*

$$G \models_{RDFS}^{nrx} P \text{ iff } G \models_{nSPARQL} \phi(P)$$

5.5 SPARQL queries modulo RDFS with NSPARQL

These results about nSPARQL can be transferred to NSPARQL query answering:

Proposition 10 (Answers to a SPARQL query modulo RDFS by NSPARQL)**.** *Let* ***SELECT*** $\vec{B}$ ***FROM*** *u* ***WHERE*** *P be a SPARQL query with P a SPARQL graph pattern and G the RDFS graph identified by the URI u, then the set of answers to this query is:*

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \{\sigma|_{\vec{B}}^{\vec{B}} \mid \sigma \in \mathcal{S}^o(\phi(P), G)\}$$

Example 15 (NSPARQL Query)**.** *The result of transforming the query of Example 4 with ϕ is:*

```
SELECT ?x, ?y, ?z
FROM ...
WHERE {
  ?x next[(next::sp)*/self::rn:inhibits] ?y.
  ?y next[(next::sp)*/self::rn:regulates] ?z.
  ?x next[(next::sp)*/self::rn:promotes] ?z.
  ?x next::rdf:type/next::sc*
    |edge/next::sp*/next::dom/next::sc*
    |node-1/next::sp*/next::range/next::sc* rn:gene.
    }
```

This query provides the correct set of answers for the RDF graph of Example 1 modulo the RDF Schema of Example 7 (given in Example 9).

The main problem with NSPARQL is that, because nested regular expressions do not contain variables, it does not preserve SPARQL queries. Indeed, it is impossible to express a query containing the simple triple $\langle ?x\ ?y\ ?z\rangle$. This may seem like a minor problem, but in fact NSPARQL graph patterns prohibits dependencies between two variables as freely as it is possible to express in SPARQL. For instance, querying a gene regulation network for self-regulation cycles (such that a product which inhibits another product that indirectly activates itself or which activates a product which indirectly inhibit itself), can be achieved with the following SPARQL query:

```
SELECT ?a
WHERE {
    ?a ?p ?b.
    ?b rn:promotes ?c.
    ?c ?q ?a.
    ?a rdf:type rn:gene.
    ?b rdf:type rn:gene.
    ?c rdf:type rn:gene.
    ?p sp rn:regulates.
    ?q sp rn:regulates.
    ?p owl:inverseOf ?q.
    }
```

Such queries are representative of queries in which two different predicates (`?p` and `?q`) are dependent of each others[2]. They are not expressible by a language like NSPARQL. An nSPARQL expression can express (abstracting for the type constraints):

```
?a next[(next::sp)*/self::rn:regulates] / next[(next::sp)*/self::rn:regulates]
        / next[(next::sp)*/self::rn:regulates] ?a
```

but it cannot express the `owl:inverseof` constraints because it defines a dependency between two places in the path (this is not anymore a regular path).

The same type of query can be used for querying a banking system for detecting money laundering in which a particular amount is moving and coming back through an account through buy/sell or debit/credit operations with an intermediary transfer.

[2]Here through `owl:inverseOf`, but the use of OWL vocabulary is not at stake here, it could have been `rdfs:subPropertyOf` or any other property.

6 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 second one 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 [6], a language using CPSPARQL graph patterns in the same way as nSPARQL does.

6.1 CPSPARQL syntax

The notation that we use 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 `axes` 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 express 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 35 (Constrained regular expression)**.** *A* constrained regular expression *is an expression built from the following grammar:*

$$cre ::= axis \mid axis{::}a \mid axis{::}[?x : \psi] \mid axis{::}]?x : \psi[\mid cre \mid cre/cre \mid cre|cre \mid cre^*$$

with ψ a set of triples belonging to $\mathcal{U} \cup \mathcal{B} \cup \{?x\} \times cre \times \mathcal{T} \cup \{?x\}$ and FILTER-expressions over $\mathcal{B} \cup \{?x\}$. ψ 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.

We use $\mathcal{B}(R)$ for the set of variables occurring as the head variable of an open bracket constraint in R.

Constraint nesting is allowed because constrained regular expressions may be used in the graph pattern of another constrained regular expression as in Example 16.

Example 16 (Constrained regular expression)**.** *The following constrained regular expression could be used to find nodes connected by transportation means that are not buses:*

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

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

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

Definition 36 (CPSPARQL triple pattern)**.** *A* CPSPARQL triple pattern *is a triple* $\langle s, p, o\rangle$ *such that* $s \in \mathcal{T}$*,* $o \in \mathcal{T}$ *and* p *is a constrained regular expression.*

Definition 37 (CPSPARQL graph pattern)**.** *A* CPSPARQL graph pattern *is defined inductively by:*

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

Example 17 (CPSPARQL graph pattern)**.** *The following CPSPARQL graph pattern could be used to retrieve the set of pairs of cities connected by a sequence of transportation means (which are not buses) such that one city in France and the other one in Jordan:*

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

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

By restricting CPRDF constraints, it is possible to define a far less expressive language. cpSPARQL is such a language.

Definition 38 (cpSPARQL regular expression [6])**.** *A* cpSPARQL regular expression *is an expression built from the following grammar:*

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

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

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

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

Example 18 (cpSPARQL triple patterns)**.** *The query of Example 12 could be expressed by the following cpSPARQL pattern:*

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

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

Example 16 provides another cpSPARQL regular expression. By contrast, CPSPARQL graph patterns allow for queries like:

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

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

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

6.2 CPSPARQL semantics

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

Definition 39 (Satisfied constraint in an RDF graph)**.** *Let G be an RDF graph, s and o be two nodes of G and $\psi = x{:}C$ be a constraint, then s and o satisfies ψ in G (denoted $\langle s, o\rangle \in [\![\psi]\!]_G$) if and only if one of the following conditions is satisfied:*

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

As for nested regular expressions, the evaluation of a constrained regular expression R over an RDF graph G is defined as a binary relation $[\![R]\!]_G$, by a pair of nodes $\langle a, b\rangle$ such that a is reachable from b in G by following a path that conforms to R. The following definition extends Definition 29 to take into account the semantics of terms with constraints.

Definition 40 (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 29, otherwise the interpretation of P in G is defined as:*

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

Definition 41 (Answer to a CPSPARQL triple pattern)**.** *The evaluation of a CPSPARQL triple pattern $\langle x, R, y\rangle$ over an RDF graph G is defined as the following set of maps:*

$$[\![\langle x, R, y\rangle]\!]_G = \{\sigma \mid dom(\sigma) = \{x, y\} \cap \mathcal{B} \cup \mathcal{B}(R) \text{ and } \langle \sigma(x), \sigma(y)\rangle \in [\![\sigma(R)]\!]_G\}$$

[3]Except for the case of bound (see Definition 7 and the discussion after it).

such that $\sigma(R)$ is the constrained regular expression obtained by substituting the variable $?x$ appearing in a constraint with open brackets in R by $\sigma(?x)$.

This semantics also applies to cpSPARQL graph patterns.

6.3 Evaluating cpSPARQL regular expressions

In order to establish the complexity of cpSPARQL we follow [33] to store an RDF graph as an adjacency list: every $u \in voc(G)$ is associated with a list of pairs $\alpha(u)$. For instance, if $\langle s, p, o\rangle \in G$, then $\langle \texttt{next}::p, o\rangle \in \alpha(s)$ and $\langle \texttt{edge}^{-1}::o, s\rangle \in \alpha(p)$. Also, $\langle \texttt{self}::u, u\rangle \in \alpha(u)$, for $u \in voc(G)$. The set of terms of a constrained regular expression R, denoted by $\mathcal{T}(R)$, is constructed as follows:

$$\begin{aligned}\mathcal{T}(R) =& \{R\} \text{if } R \text{ is either } \texttt{axis}, \texttt{axis}::\text{a, or } \texttt{axis}::\psi\\ \mathcal{T}(R_1/R_2) =& \mathcal{T}(R_1|R_2) = \mathcal{T}(R_1) \cup \mathcal{T}(R_2)\\ \mathcal{T}(R_1^*) =& \mathcal{T}(R_1)\end{aligned}$$

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

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

Algorithm 2 (Eval) solves the evaluation problem for a constrained regular expression R over an RDF graph G. This algorithm is almost the same as the one in [33] which solves the evaluation problem for nested regular expressions R over an RDF graph G. The Eval algorithm calls the Algorithm 1 (LABEL), which is an adaptation of the LABEL algorithm of [33] in which we modify only the first two steps. These two steps are based on the transformation rules from nSPARQL expressions to cpSPARQL expressions (see §7.4).

Algorithm 1 LABEL(G, exp):

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

The algorithm as the same $O(|G| \times |R|)$ time complexity as usual regular expressions [39, 27] and nested regular expressions [33] evaluation.

Theorem 3 (Complexity of cpSPARQL regular expression evaluation)**.** *Eval solves the evaluation problem for constrained regular expression in time $O(|G| \times |R|)$.*

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

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

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

6.4 SPARQL queries modulo RDFS with CPSPARQL

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

Definition 42 (RDFS triple pattern expansion [2])**.** *Given an RDF triple t, the* RDFS expansion *of t, denoted by $\tau(t)$, is defined as:*

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

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

The extra variable $?x$ introduced in the last item of the transformation, is only used inside the constraint of the constrained regular expression and so it is not considered to be in $dom(\sigma)$, i.e., only variables occurring as a subject or an object in a CPSPARQL triple pattern are considered in maps (see Definition 41). Therefore, the projection operator (SELECT) 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 19 (SPARQL query transformation)**.** *Consider the following SPARQL query that searches pairs of nodes connected with a property p*

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

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

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

The evaluation of the above PSPARQL query is the map $\{?X \leftarrow a, ?P \leftarrow b, ?Y \leftarrow c\}$. 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 [33] that including the projection (SELECT) 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 τ function of Definition 42, to the following cpSPARQL query (in which there is no need for the projection operator):

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

Since the variable $?z$ is used inside the constraint, the answer to this query will be $\{?X \leftarrow a, ?Y \leftarrow b\}$ (see Definition 40).

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

7 On the respective expressiveness of cpSPARQL and nSPARQL

In this section, we compare the expressiveness of cpSPARQL with that of 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. These are the core results of [6].

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

Although it is explained in [33] that SPARQL triple patterns can be encoded by nested regular expressions, triple patterns with three variables (subject, predicate, object) could not be expressed by nested regular expressions since variables are not allowed in nested regular expressions. The reader may wonder whether this is useful or not. The following query is a useful example:

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

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

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

7.2 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 projection (SELECT):

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

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

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

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

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

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

So, the above query cannot be expressed in nSPARQL without the use of SELECT, which is not allowed in nSPARQL [33]. Besides, any SPARQL query that uses SELECT over a set of variables such that there exists at least one existential variable, i.e., a variable not in the SELECT clause, used in a FILTER constraint cannot be expressed by nSPARQL graph patterns.

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

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

7.3 nSPARQL cannot express all cpSPARQL, even with SELECT

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

If one wants to restrict the query of Example 12 such that every stop is a city in the same country (for example, France), then the following nested regular expression expresses this query:

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

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

$$\langle ?city_1, (next :: [\psi_1]/self :: [\psi_2])^+, ?city_2\rangle$$

where $\psi_1 = ?x : \{\langle ?x, (next :: sp)^*, transport\rangle\}$
and $\psi_2 = ?x : \{\langle ?x, next :: cityIn, France\rangle\}$

If one wants that each stop satisfies a specific constraint, e.g., cities with a population size larger than 20,000 inhabitants, and each transportation mean belongs to Air France, i.e., its

URI is in the airfrance domain name. Then this query is expressed by the following constrained regular expression:

$$P = \langle ?city_1, (next :: [\psi_1]/self :: [\psi_2])^+, ?city_2 \rangle$$

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

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

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

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

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

This capability to express constraints on values in path expressions, available in XPath as well, is invaluable for selecting exactly those paths that are useful instead of being constrained to resort to a posteriori selection. This provides interesting computational properties discussed in Section 8.

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

Example 20. *Consider the following RDF graph representing flights belonging to different airline companies and other transportation means between cities:*

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

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

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

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

7.4 cpSPARQL can express all nSPARQL

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

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

In the last clause of this transformation, when the nested regular expression $R = exp_1 :: [exp_2]$, it is required to check the existence of two pairs of nodes that satisfies the sub-expression exp_2 (see Definition 29). Similarly, in cpSPARQL it is necessary to express this nested regular expression as a triple in which the constraint is satisfied by the existence of two pairs of nodes that replace the variables $?x$ and $?y$.

This transformation process is illustrated by the following example.

Example 21 (From nSPARQL to cpSPARQL)**.** *Consider the following nested regular expression:*

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

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

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

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

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

8 Implementation

CPSPARQL has been implemented in order to evaluate its feasibility[4]. cpSPARQL does not 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` clause).

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

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 different RDF graph sizes from 5, 10, . . . , up to 100,000 triples in memory graphs). In particular, it showed the capability at stake here: answering SPARQL queries with the RDFS semantics.

The implementation has been also tested thoroughly in [8] and the results show that PSPARQL had better performances than other implementations of SPARQL with paths[5].

It has not been possible to us to compare the performance of our CPSPARQL implementation with other proposals. Indeed, contrary to CPSPARQL, nSPARQL is not implemented at the moment, so we must leave the experimental comparison for future work.

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 path expressions with nodes satisfying constraints while matching (on the fly instead of filtering them a posteriori). The implemented prototype follows this natural strategy, thus reducing the search space. This strategy promises to be always more efficient than a strategy which applies constraints a posteriori. More details are available in [2].

9 Related work

The closest work to ours, nSPARQL, has been presented and compared in detail in Section 5 [33]. However, there are other work which may be considered relevant.

RQL [23] attempts to combine the relational algebra with some special class hierarchies. It supports a form of transitive expressions over RDFS transitive properties, i.e., `rdfs:subPropertyOf` and `rdfs:subClassOf`, for navigating through class and property hierarchies. Versa [29], RxPath [37] are all path-based query languages for RDF that are well suited for graph traversal. SPARQLeR [24] 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 two given nodes. This extension offers functionalities like testing the length of paths and testing if a given node is in the found paths. SPARQ2L [7] 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 [27] (see also [8]).

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 [17]. Recently [13] extended Datalog in order to cope with querying modulo ontologies. Ontologies are in DL-Lite and, in particular DL-Lite$_{\mathcal{R}}$ which contains the fragment of RDFS considered here. However, this work only considers conjunctive queries which is not sufficient for evaluating SPARQL queries which contains constructs such as UNION, OPT and constraints (FILTER) which are not found in Datalog. [9] 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

[5]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/`

and cpSPARQL, the complexity is far higher (coNP).

Standardization efforts by the W3C SPARQL working group have defined the notion of inference regime for SPARQL [19, 18]. This notion is relevant to query evaluation modulo RDFS that is exhibited by CPSPARQL and is obviously less relevant to cpSPARQL and nSPARQL. One main difference is that we have departed from the strict definition of "matching graph patterns" with the use of path for exploring the graph, and specifically the graph entailed by RDFS. This avoids the use of RDF graph closure on which strict matching is applied. CPSPARQL and nSPARQL use query rewriting for answering queries modulo RDFS, but, unlike DL-Lite rewriting strategies, queries are rewritten by preserving their structure instead of producing unions of conjunctive queries.

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

The language RPL extends nested regular expressions [40] to allow boolean node tests. However, using variables in nested regular expressions of nSPARQL requires extending its syntax and semantics. Hence, comparison between variables and values as well as triple patterns with variables in subject, predicate and object are not allowed (see examples in Section 7).

10 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 [19].

One possible approach for querying an RDFS graph G in a sound and complete way is by computing the closure graph of G, i.e., the graph obtained by saturating G with all informations that can be deduced using a set of predefined rules called RDFS rules, then evaluating the query Q over the closure graph. However, this approach takes time proportional to $|Q| \times |G|^2$ in the worst case [28].

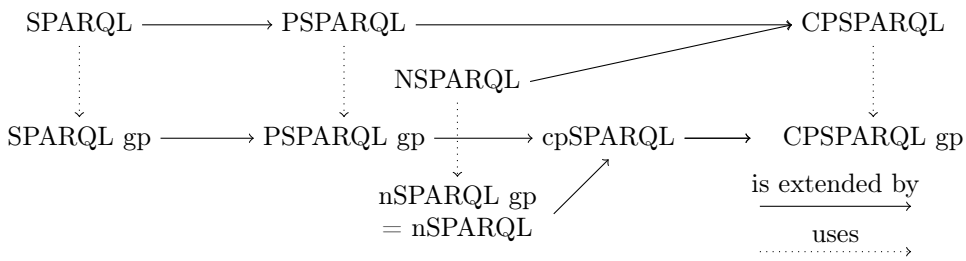

Figure 4: Query languages and their graph patterns.

Over the years, several other languages, i.e., PSPARQL [5], nSPARQL [33] and CPSPARQL [2], have been shown able to deal with RDFS graphs without computing the closure. They all use different variations of path regular expressions as triple predicates and adopt a semantics based on checking the existence of paths (without counting them) in the RDF graph. In order

to ease their comparison, we defined cpSPARQL as very close to nSPARQL and NSPARQL as very close to CPSPARQL.

Figure 4 shows the position of the various languages. nSPARQL and cpSPARQL are good navigational languages for RDF(S). However, cpSPARQL is an extension of SPARQL graph patterns, while nSPARQL does not contain all SPARQL graph patterns. Moreover, using such a path language within the SPARQL structure allows for properly extending SPARQL. Some features (such as filtering nodes inside expressions) are very simple to add to the syntax and semantics of nested regular expressions.

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

This work can also be extended in the schema or ontology language on which it is applied. OWL 2 opens the door to many OWL fragments for which it should be possible to design query evaluation procedures.

A Proofs

A.1 Induction lemma

All completeness proofs below follow the same style: because most of the results that we use are based on entailment of graphs, i.e., basic graph patterns, we need to promote these results to all graph patterns. This is done easily by defining query-structure preserving transformations and using Lemma 1.

Definition 43 (Query-structure preserving transformation)**.** *A transformation ψ on RDFS graphs and SPARQL graph patterns is said to be* query structure preserving *if and only if:*

$$\psi(G \models P) = \psi(G) \models \psi(P) \tag{1}$$
$$\psi(P \ \texttt{AND} \ P') = \psi(P) \ \texttt{AND} \ \psi(P') \tag{2}$$
$$\psi(P \ \texttt{UNION} \ P') = \psi(P) \ \texttt{UNION} \ \psi(P') \tag{3}$$
$$\psi(P \ \texttt{OPT} \ P') = \psi(P) \ \texttt{OPT} \ \psi(P') \tag{4}$$
$$\psi(P \ \texttt{FILTER} \ K) = \psi(P) \ \texttt{FILTER} \ K \tag{5}$$

As its name indicates, such a transformation preserves the structure of graph patterns. This is the case of most transformations proposed here since they are defined on basic graph patterns and extended to queries by applying them to basic graph patterns. The structure of P must be preserved, but not be isomorphic to that of $\psi(P)$. For instance, the transformation τ may introduce extra `UNION` in the resulting pattern.

The induction lemma shows that if an entailment relation is complete for basic graph patterns, then it is complete for all graph patterns.

Lemma 1 (Induction lemma)**.** *Let ψ be a query-structure preserving transformation, if for all RDFS graph G, basic graph pattern B and map σ, $\psi(G) \models \sigma(\psi(B))$ iff $G \models \sigma(B)$, then for all graph pattern P, $\psi(G) \models \sigma(\psi(P))$ iff $G \models \sigma(P)$.*

Proof of Lemma 1. The lemma itself is proved by induction:

Base step For B a basic graph pattern and G an RDFS graph, $\psi(G) \models \sigma(\psi(B))$ iff $G \models \sigma(B)$, by hypothesis,

Induction step If for P and P', graph patterns, and G RDFS graph and σ map, $\psi(G) \models$

$\sigma(\psi(P))$ iff $G \models \sigma(P)$ and $\psi(G) \models \sigma(\psi(P'))$ iff $G \models \sigma(P')$, then:

$$\begin{aligned}
G \models \sigma(P \texttt{ AND } P') &\text{ iff } G \models \sigma(P) \text{ and } G \models \sigma(P') \\
&\text{ iff } \psi(G) \models \psi(\sigma(P)) \text{ and } \psi(G) \models \psi(\sigma(P')) \\
&\text{ iff } \psi(G) \models \sigma(\psi(P \texttt{ AND } P')) \\
G \models \sigma(P \texttt{ UNION } P') &\text{ iff } G \models \sigma(P) \text{ or } G \models \sigma(P') \\
&\text{ iff } \psi(G) \models \sigma(\psi(P)) \text{ or } G \models \sigma(\psi(P')) \\
&\text{ iff } \psi(G) \models \sigma(\psi(P \texttt{ UNION } P')) \\
G \models \sigma(P \texttt{ OPT } P') &\text{ iff } G \models \sigma(P) \text{ and } [G \models \sigma(P') \\
&\text{ or } \forall \sigma'; G \models \sigma'(P'), \sigma \perp \sigma'] \\
&\text{ iff } \psi(G) \models \sigma(\psi(P)) \text{ and } [\psi(G) \models \sigma(\psi(P')) \\
&\text{ or } \forall \sigma'; \psi(G) \models \sigma'(\psi(P')), \sigma \perp \sigma'] \\
&\text{ iff } \psi(G) \models \sigma(\psi(P \texttt{ OPT } P')) \\
G \models \sigma(P \texttt{ FILTER } K) &\text{ iff } G \models \sigma(P) \text{ and } \sigma(K) = \top \\
&\text{ iff } \psi(G) \models \sigma(\psi(P)) \text{ and } \sigma(K) = \top \\
&\text{ iff } \psi(G) \models \sigma(\psi(P \texttt{ FILTER } K))
\end{aligned}$$

The only difficult point is in the OPT part, but since the induction step strictly preserves the set of maps satisfying a graph pattern, the universal quantification holds. □

A.2 Completeness of partial non reflexive RDFS closure

We first have to extend the completeness proof of the partial closure (Proposition 4) to the non reflexive case.

Proof of Proposition 5. The proof can be derived from Proposition 4. Indeed, it should be shown that suppressing rules [RDFS8a] and [RDFS12a] does suppress all and only consequences of the reflexivity of sc and sp.

[RDFS12a] generates $\langle c \texttt{ sc } c\rangle$ which can be further used by [RDFS12b] and [RDFS11]. However, these two rules would only generate triples that are already in their premises. Hence the only new generated triple is the reflexivity triple.

[RDFS8a] generates $\langle p \texttt{ sp } p\rangle$ which can be further used by [RDFS8b], [RDF2] and [RDFS9]. Similarly, these three rules would only generate triples that are already in their premises or in axiomatic triples. Hence the only new generated triple is the reflexivity triple.

Finally, both triples could be consumed by rules [RDFS6] and [RDFS7]. However, these rules would require constraining the sp or sc relations through dom or range statements respectively. This is not possible in genuine RDFS graphs.

In the other direction, the only way a model can contain $\langle p, p\rangle \in I_{EXT}(\iota'(\texttt{sp}))$ or $\langle c, c\rangle \in I_{EXT}(\iota'(\texttt{sc}))$ is through Definition 11 constraints:

- (1a) it is a triple of G. Then it is still in $\hat{G}\backslash\backslash H$;
- (4a) and (5a) by reflexivity;
- (4b), (6b) and (6c) which only apply to generate reflexive sc or sp statements when constraints are added to sc or sp.

□

Proof of Corrolary 1. The proof is a simple consequence of Proposition 5. Proving that:

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \mathcal{A}(\vec{B}, \hat{G}\backslash\backslash P, P)$$

by Definition 13 and Proposition 2, is equivalent to proving that:

$$\{\sigma|_{\vec{B}}^{\vec{B}} | G \models_{RDFS}^{\text{nrx}} \sigma(P)\} = \{\sigma|_{\vec{B}}^{\vec{B}} | \hat{G}\backslash\backslash P \models_{RDF} \sigma(P)\}$$

and Proposition 5 together with Lemma 1 means that this is true if G is satisfiable and genuine. □

A.3 Completeness and complexity of the PSPARQL RDFS query encoding

Proof of Proposition 7. We use the same pattern as before, using Lemma 2 to be proved below. Proving that:

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \mathcal{A}^{\star}(\vec{B}, G, \tau(P))$$

by Definition 13 and Definition 23, is equivalent to proving that:

$$\{\sigma|_{\vec{B}}^{\vec{B}} | G \models_{RDFS}^{\text{nrx}} \sigma(P)\} = \{\sigma|_{\vec{B}}^{\vec{B}} | G \models_{PSPARQL} \sigma(\tau(P))\}$$

and Lemma 2 together with Lemma 1 means that this is true. □

The proof is longer than the previous ones because it does not rely on externally proved propositions. Instead, we need to prove the following lemma

Lemma 2. *Given a basic SPARQL graph pattern P and an RDFS graph G,*

$$G \models_{RDFS}^{nrx} P \text{ iff } G \models_{PSPARQL} \tau(P)$$

The proof relies on the notion of PRDF homomorphism, a particular sort of map [5]:

Definition 44 (PRDF homomorphism)**.** *Let G be a GRDF graph, and H be a PRDF graph. A PRDF homomorphism from H into G is a map π from term(H) into term(G) such that $\forall\langle s, R, o\rangle \in H$, either*

(i) the empty word $\epsilon \in L^*(R)$ and $\pi(s) = \pi(o)$; or

(ii) $\exists\langle n_0, p_1, n_1\rangle, \ldots, \langle n_{k-1}, p_k, n_k\rangle$ in G such that $n_0 = \pi(s)$, $n_k = \pi(o)$, and $p_1 \cdot \ldots \cdot p_k \in L^*(\pi(R))$.

Proof of Lemma 2. Since the answers to a graph pattern can be made by joining the answers to its triple patterns [5], it is sufficient to show that answering a triple t is equivalent to answering its transformed triple $t' = \tau(t)$. Hence we consider each case of τ:

– Let $t = \langle \mathcal{X}, sc, \mathcal{Y}\rangle$, then $t' = \langle \mathcal{X}, sc^+, \mathcal{Y}\rangle$

  (⇒) If $\sigma \in \mathcal{A}^{\#}(\vec{B}, G, t)$, then either $\langle \sigma(\mathcal{X}), sc, \sigma(\mathcal{Y})\rangle \in G$ or $\langle \sigma(\mathcal{X}) = \mathcal{X}_0, sc, \mathcal{Y}_0\rangle$, …, $\langle \mathcal{X}_n, sc, \sigma(\mathcal{Y}) = \mathcal{Y}_n\rangle \in G$. In both cases, $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$.

  (⇐) If $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$, then $\langle \sigma(\mathcal{X}) = \mathcal{X}_0, sc, \mathcal{Y}_0\rangle$, …, $\langle \mathcal{X}_n, sc, \sigma(\mathcal{Y}) = \mathcal{Y}_n\rangle \in G$. In what follows, $\langle \sigma(\mathcal{X}), sc, \sigma(\mathcal{Y})\rangle \in \hat{G}$. So, $\sigma \in \mathcal{A}^{\#}(\vec{B}, G, t)$.

– Let $t = \langle \mathcal{X}, sp, \mathcal{Y} \rangle$, then $t' = \langle \mathcal{X}, sp^+, \mathcal{Y} \rangle$

(⇒) If $\sigma \in \mathcal{A}^{\#}(\vec{B}, G, t)$, then either $\langle \sigma(\mathcal{X}), sp, \sigma(\mathcal{Y}) \rangle \in G$ or $\langle \sigma(\mathcal{X}) = \mathcal{X}_0, sp, \mathcal{Y}_0 \rangle$, ..., $\langle \mathcal{X}_n, sp, \sigma(\mathcal{Y}) = \mathcal{Y}_n \rangle \in G$. In both cases, $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$.

(⇐) If $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$, then $\langle \sigma(\mathcal{X}) = \mathcal{X}_0, sp, \mathcal{Y}_0 \rangle$, ..., $\langle \mathcal{X}_n, sp, \sigma(\mathcal{Y}) = \mathcal{Y}_n \rangle \in G$. In what follows, $\langle \sigma(\mathcal{X}), sp, \sigma(\mathcal{Y}) \rangle \in \hat{G}$. So, $\sigma \in \mathcal{A}^{\#}(\vec{B}, G, t)$.

– Let $t = \langle \mathcal{X}, p, \mathcal{Y} \rangle$, then $t' = \{\langle \mathcal{X}, ?p, \mathcal{Y} \rangle, \langle ?p, \mathtt{sp}^*, p \rangle\}$.

(⇒) If $\sigma \in \mathcal{A}^{\#}(\vec{B}, G, t)$, then either $\langle \sigma(\mathcal{X}), p, \sigma(\mathcal{Y}) \rangle \in G$ or $\langle \sigma_1(\mathcal{X}), \sigma_1(?p) = p_0, \sigma_1(\mathcal{Y}) \rangle$, $\langle p_0, sp, p_1 \rangle$, ..., $\langle p_{n-1}, sp, p_n = p \rangle \in G$. In the first case, the map σ is a PRDF homomorphism from t' into G. In the second case, the map σ_1 is a PRDF homomorphism from t' into G. The restriction of σ_1 to the variables of the query $\vec{B}$ is σ. Hence in both cases, $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$.

(⇐) If $\sigma|_{\vec{B}}^{\vec{B}} \in \mathcal{A}^{\star}(\vec{B}, G, t')$, then $\langle \sigma|_{\vec{B}}^{\vec{B}}(\mathcal{X}), \sigma|_{\vec{B}}^{\vec{B}}(?p) = p_0, \sigma|_{\vec{B}}^{\vec{B}}(\mathcal{Y}) \rangle$, $\langle p_0, sp, p_1 \rangle$, ..., $\langle p_{n-1}, sp, p_n = p \rangle \in G$. In what follows, $\langle \sigma|_{\vec{B}}^{\vec{B}}(\mathcal{X}), p, \sigma|_{\vec{B}}^{\vec{B}}\mathcal{Y}) \rangle \in \hat{G}$. So, $\sigma|_{\vec{B}}^{\vec{B}} \in \mathcal{A}^{\#}(\vec{B}, G, t)$.

– Let $t = \langle \mathcal{X}, type, \mathcal{Y} \rangle$,

(⇒) If $\sigma \in \mathcal{A}^{\#}(\vec{B}, G, t)$, then at least one of the following cases is satisfied:

  – The triples $\langle \sigma(\mathcal{X}), type, o_1 \rangle$, $\langle o_1, sc, o_2 \rangle$, ..., $\langle o_{n-1}, sc, o_n = \sigma(\mathcal{Y}) \rangle$ belong to G. So that σ is a PRDF homomorphism from the first part of t' into G and $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$.
  – The triples $\langle \sigma_1(\mathcal{X}), \sigma_1(?p_1) = p_1, \sigma_1(?y) = y \rangle$, $\langle p_1, sp, p_2 \rangle$, ..., $\langle p_{n-1}, sp, p_n \rangle$, $\langle p_n, dom, o_1 \rangle$, $\langle o_1, sc, o_2 \rangle$, ..., $\langle o_{n-1}, sc, o_n = \sigma_1(\mathcal{Y}) \rangle$ belong to G. So that σ_1 is a PRDF homomorphism from the second part of t' into G. The restriction of σ_1 to the variables of the query $\vec{B}$ is σ. Hence $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$.
  – The triples $\langle \sigma_1(?y) = y, \sigma_1(?p_1) = p_1, \sigma_1(\mathcal{X}) \rangle$, $\langle p_1, sp, p_2 \rangle$, ..., $\langle p_{n-1}, sp, p_n \rangle$, $\langle p_n, range, o_1 \rangle$, $\langle o_1, sc, o_2 \rangle$, ..., $\langle o_{n-1}, sc, o_n = \sigma_1(\mathcal{Y}) \rangle$ belong to G. So that σ_1 is a PRDF homomorphism from the second part of t' into G. The restriction of σ_1 to the variables of the query $\vec{B}$ is σ. Hence $\sigma \in \mathcal{A}^{\star}(\vec{B}, G, t')$.

(⇐) If $\sigma|_{\vec{B}}^{\vec{B}} \in \mathcal{A}^{\star}(\vec{B}, G, t')$, then there exist three cases. The first case is that the triples $\langle \sigma|_{\vec{B}}^{\vec{B}}(\mathcal{X}), type, o_1 \rangle$, $\langle o_1, sc, o_2 \rangle$, ..., $\langle o_{n-1}, sc, o_n = \sigma|_{\vec{B}}^{\vec{B}}(\mathcal{Y}) \rangle$ belong to G. Using the sub-class RDFS rules, $\langle \sigma(\mathcal{X}), type, \sigma(\mathcal{Y}) \rangle \in \hat{G}$. In the second case, the triples $\langle \sigma(\mathcal{X}), \sigma(?p_1) = p_1, y \rangle$, $\langle p_1, sp, p_2 \rangle$, ..., $\langle p_{n-1}, sp, p_n \rangle$, $\langle p_n, dom, o_1 \rangle$, $\langle o_1, sc, o_2 \rangle$, ..., $\langle o_{n-1}, sc, o_n = \sigma(\mathcal{Y}) \rangle$ belong to G. Using the sub-property RDFS rules, the triples $\langle \sigma(\mathcal{X}), p_2, y \rangle$, ..., $\langle \sigma(\mathcal{X}), p_n, y \rangle \in \hat{G}$. So that $\langle \sigma(\mathcal{X}), type, o_1 \rangle$, ..., $\langle \sigma(\mathcal{X}), type, o_n = \sigma(\mathcal{Y}) \rangle \in \hat{G}$ (it is easy to prove the same thing for the third case with `range` instead of `dom`). Hence $\sigma|_{\vec{B}}^{\vec{B}} \in \mathcal{A}^{\#}(\vec{B}, G, t)$.

□

Proof of Proposition 8. The complexity of answering queries modulo RDF Schema through the PSPARQL RDFS encoding is the sequential combination of that of the two components: encoding (τ) and evaluating the resulting query. The complexity of the encoding is linear in terms of triples in the query graph pattern P, moreover its size is not more than seven times that of P (if all triples are type statements). Since the complexity of PSPARQL query answering has been proved to be PSPACE-complete in [5], the complexity of $\mathcal{A}^{\star}$-Answer checking is PSPACE-complete. □

A.4 Completeness of NSPARQL query answering

Proof of Proposition 10. This is similar to the previous proof. The proof is a simple consequence of Proposition 2. Proving that:

$$\mathcal{A}^{\#}(\vec{B}, G, P) = \mathcal{A}^{o}(\vec{B}, G, \phi(P))$$

by Definition 13 and Definition 32, is equivalent to proving that:

$$\{\sigma|_{\vec{B}}^{\vec{B}} | G \models^{\text{nrx}}_{RDFS} \sigma(P)\} = \{\sigma|_{\vec{B}}^{\vec{B}} | G \models_{nSPARQL} \sigma(\phi(P))\}$$

and Proposition 2 together with Lemma 1 means that this is true. □

A.5 Complexity of cpSPARQL evaluation

Proof of Theorem 3. Let R be the a constrained regular expression, G be an RDF graph, $\langle a, b\rangle$ be a pair of nodes, and A_R be the automaton recognizing the language of R.

The automaton of R can be constructed as described in §6.3 in NLOGSPACE (as for the usual automata [39, 27]). For simplicity and without loss of generality, we use the `next` axis to illustrate the construction of the product automaton. This is because the axis determines the node to be checked (subject, predicate or object) and thus does not affect the construction. The construction of the product automaton is done as follows:

- If $R = axis ::]?x : TRUE[$ then checking whether the pair $\langle a, b\rangle$ is in $[\![R]\!]_G$ can be done in $O(|G|)$ since it is sufficient to substitute each node n to $?x$ and check whether $\langle a, n, b\rangle$ is in G (according to the axis).
- Otherwise, call the Eval algorithm (where D_0 is defined as done in [33]). If $\langle s_i, next :: [FILTER(?x)], s_j\rangle \in A_R$ and $\langle n_i, next :: p, n_j\rangle \in G$, then add $\langle s_j, n_j\rangle$ to the product automaton if p satisfies the SPARQL filter constraint by substituting only the node p to the variable $?x$. Checking if a node n satisfies a SPARQL filter constraint can be done in $O(1)$.

  Additionally, if $\langle ?x, next :: p, ?y\rangle.FILTER(?x, ?y) \in A_R$ and $\langle n_i, next :: p, n_j\rangle \in G$, then add $\langle s_j, n_j\rangle$ to the product automaton if $\langle n_i, next :: p, n_j\rangle \in G$ and the SPARQL filter constraint is satisfied by substituting the node n_i to the variable $?x$ and n_j to the variable $?y$.

So, the product automaton $(G \times \mathcal{A}_R)$ can be obtained in time $O(|G| \times |R|)$. Hence, checking if the pair $\langle a, b\rangle \in [\![R]\!]_G$ is equivalent to checking if the language accepted by $(G \times \mathcal{A}_R)$ is not empty, which can be done in $O(|G| \times |R|)$ (as in the case of usual regular expressions [39, 27]). □

A.6 Correctness and completeness of RDFS translation to PSPARQL

The proof of Theorem 4 follows from the results in [33] except the last step.

Proof of Theorem 4. We need to prove only the last step since all other transformation steps are the same as the ones in [33]. That is $\langle \sigma(x), \sigma(y)\rangle \in [\![\langle x, p, y\rangle]\!]_G^{rdfs}$ iff $\langle \sigma(x), \sigma(y)\rangle \in [\![\langle x, \tau(p), y\rangle]\!]_G$.

- ($\Rightarrow$) Assume that $\langle \sigma(x), \sigma(y)\rangle \in [\![\langle x, p, y\rangle]\!]_G^{rdfs}$. In this case, there exists p_1 such that $(p_1\ sp\ p_2\ sp\ \ldots\ sp\ p_n = p)$ and $\langle \sigma(x), p_1, \sigma(y)\rangle \in G$ as well as $\langle \sigma(x), \texttt{next::}p_1, \sigma(y)\rangle \in G$. Let us consider now the transformed triple $\tau(t) = \langle x, (\texttt{next::}\psi), y\rangle$ (where $\psi = [?p : \{\langle ?p, (\texttt{next::sp})^*, p\rangle\}]$). The maps for the variable $?p$ will be $\{\langle ?p, p_i\rangle \mid i = 1, \ldots, n\}$ (since $[\![\psi]\!]_G = \{\langle p_i, p\rangle \mid i = 1, \ldots, n\}$). According to Definitions 40 and 41, $\langle \sigma(x), \sigma(y)\rangle \in [\![\langle x, (\texttt{next::}\psi), y\rangle]\!]_G$ iff $\langle \sigma(x), \sigma(y)\rangle \in G$ and $p_1 \in [\![\psi]\!]_G$, and this condition holds.

– ($\Leftarrow$) We have to prove that, if $\langle\sigma(x),\sigma(y)\rangle \in [\![\langle x, (\texttt{next::}[\psi]), y\rangle]\!]_G$ (with $\psi = ?p : \{\langle ?p, (\texttt{next::sp})^*, p\rangle\}$), then $\langle\sigma(x),\sigma(y)\rangle \in [\![\langle x,p,y\rangle]\!]_G^{rdfs}$. Assume that $\langle\sigma(x),\sigma(y)\rangle \in [\![\langle x, (\texttt{next::}\psi), y\rangle]\!]_G$. In this case, there exists p_1 such that $\langle\sigma(x), \texttt{next::}p_1, \sigma(y)\rangle \in G$ and $p_1 \in [\![\psi]\!]_G$, that is, $\langle p_1, \texttt{next::sp}, p_2\rangle$, ..., $\langle p_{n-1}, \texttt{next::sp}, p_n = p\rangle \in G$. Therefore, $\langle\sigma(x),\sigma(y)\rangle \in [\![\langle x,p,y\rangle]\!]_G^{rdfs}$ since $\langle p_1, (\texttt{next::sp})^*, p\rangle$ and $\langle\sigma(x), \texttt{next::}p_1, \sigma(y)\rangle \in G$.

□

A.7 Expressiveness of nSPARQL and cpSPARQL

Proof of Theorem 5. Consider, without loss of generality, RDF graphs containing a predicate s whose range is the set of integers. If one wants to select nodes which have a s-transition whose value is over 3, this could be expressed by the following constrained regular expression:

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

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

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

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

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

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

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

Proof of Theorem 6. The equivalence of the cpSPARQL encoding of nested regular expressions (trans) is given by induction on the structure of nested regular expressions.

– if R is either `axis` or `axis`::a, then $trans(R) = R$ and thus $[\![R]\!]_G = [\![trans(R)]\!]_G$.

– Now assume that $[\![R1]\!]_G = [\![trans(R1)]\!]_G$ and $[\![R2]\!]_G = [\![trans(R2)]\!]_G$, then $[\![R1|R2]\!]_G = [\![trans(R1)]\!]_G \cup [\![trans(R2)]\!]_G = [\![trans(R1)|\ trans(R2)]\!]_G = [\![trans(R1|R2)]\!]_G$ (based the definition of regular languages). The same applies for the concatenation $[\![R1/R2]\!]_G$ and the closure $(R_1)^*$.

– If $R = R_1 :: [R_2]$, then $trans(R) = R_1 :: [\psi]$, where $\psi = ?x : \{\langle ?x, trans(R_2), ?y\rangle\}$. Based on Definition 29, $[\![R_1 :: [R_2]]\!]_G = \{\langle x,y\rangle \mid \exists z,w \wedge \langle z,w\rangle \in [\![R_2]\!]_G\}$. If $\langle z,w\rangle \in [\![R_2]\!]_G$, then $\langle z,w\rangle \in [\![trans(R2)]\!]_G$ by substituting z and w to the variables $?x$ and $?y$, respectively (Definitions 39 and 40).

□

References

[1] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS)*, pages 122–133, New York, NY, USA, 1997. ACM. pages 34

[2] F. Alkhateeb. *Querying RDF(S) with regular expressions*. Thèse d'informatique, Université Joseph Fourier, Grenoble (FR), 2008. pages 3, 25, 29, 34, 35

[3] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Constrained regular expressions in SPARQL. Research Report 6360, INRIA, Montbonnot (FR), 2007. pages 3, 4

[4] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Constrained regular expressions in SPARQL. In H. Arabnia and A. Solo, editors, *Proc. international conference on semantic web and web services (SWWS), Las Vegas (NV US)*, pages 91–99, 2008. pages 3, 4

[5] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Journal of web semantics*, 7(2):57–73, 2009. pages 3, 9, 10, 13, 15, 29, 35, 39, 40

[6] F. Alkhateeb and J. Euzenat. Constrained regular expressions for answering RDF-path queries modulo RDFS. *International journal of web information systems*, 2014. to appear. pages 1, 25, 26, 30

[7] K. Anyanwu, A. Maduko, and A. Sheth. SPARQ2L: towards support for subgraph extraction queries in RDF databases. In *Proc. 16th international conference on World Wide Web (WWW)*, pages 797–806, 2007. pages 3, 34

[8] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proc. of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 629–638, 2012. pages 34

[9] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The DL-Lite family and relations. *Journal of artificial intelligence research*, 36:1–69, 2009. pages 34

[10] J.-F. Baget. Homomorphismes d'hypergraphes pour la subsomption en RDF. In *Proc. 3e journées nationales sur les modèles de raisonnement (JNMR), Paris (France)*, pages 1–24, 2003. pages 14

[11] J.-F. Baget. RDF entailment as a graph homomorphism. In *Proc. 4th International Semantic Web Conference (ISWC), Galway (IE)*, pages 82–96, 2005. pages 5

[12] D. Brickley and R. Guha. RDF vocabulary description language 1.0: RDF schema. Recommendation, W3C, 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`. pages 3, 10, 11

[13] A. Calì, G. Gottlob, and T. Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proc. 28th ACM Principle of Database Systems conference (PODS), Providence (RI US)*, pages 77–86, 2009. pages 11, 34

[14] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. View-based query processing for regular path queries with inverse. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 58–66, 2000. pages 18

[15] J. Carroll and G. Klyne. RDF concepts and abstract syntax. Recommendation, W3C, February 2004. pages 4

[16] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. PSPARQL query containment. In *Proc. 13th International symposium on database programming languages (DBPL), Seattle (WA US)*, 2011. pages 35

[17] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. 14th International Conference on Data Engineering (ICDE)*, pages 14–23, 1998. pages 34

[18] B. Glimm and M. Krötzsch. SPARQL beyond subgraph matching. In *Proc. 9th International Semantic Web Conference (ISWC), Shanghai (CN)*, pages 59–66, 2010. pages 3, 35

[19] B. Glimm and C. Ogbuji. SPARQL 1.1 entailment regimes. Working draft, W3C, June 2010. `http://www.w3.org/TR/sparql11-entailment`. pages 13, 15, 35

[20] G. Grahne and A. Thomo. Query containment and rewriting using views for regular path queries under constraints. In *Proc. 22nd ACM symposium on Principles of database systems (PODS)*, pages 111–122, New-York (NY US), 2003. ACM. pages 18

[21] C. Gutierrez, C. Hurtado, and A. Mendelzon. Foundations of semantic web databases. In *Proc. 23rd ACM Symposium on Principles of Database Systems (PODS), Paris (FR)*, pages 95–106, 2004. pages 10

[22] P. Hayes. RDF semantics. Recommendation, W3C, February 2004. pages 3, 4, 6, 7, 10, 11, 12, 14

[23] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *Proc. 11th International Conference on the World Wide Web (WWW), Honolulu (HA US)*, 2002. pages 34

[24] K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proc. 4th European Semantic Web Conferenc (ESWC'07)*, pages 145–159, 2007. pages 3, 34

[25] I. Kollia and B. Glimm. Optimizing SPARQL query answering over OWL ontologies. *Journal of artificial intelligence research*, 48:253–303, 2013. pages 3

[26] D. McGuinness and F. van Harmelen. OWL web ontology language overview. Recommendation, W3C, 2004. `http://www.w3.org/TR/owl-features/`. pages 3, 10

[27] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995. pages 28, 34, 41

[28] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and efficient minimal RDFS. *Journal of web semantics*, 7(3):220–234, 2009. pages 3, 7, 11, 13, 15, 23, 35

[29] M. Olson and U. Ogbuji. Versa: Path-based RDF query language, 2002. `http://copia.ogbuji.net/files/Versa.html`. pages 34

[30] J. Pan, E. Thomas, and Y. Zhao. Completeness guaranteed approximation for OWL DL query answering. In *Proc. of the 22nd International Workshop on Description Logics (DL), Oxford (UK)*, September 2009. pages 3

[31] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *Proc. ACM international conference on Management of data (SIGMOD), Philadelphia (PA US)*, pages 455–466, ACM Press, New-York (NY US), 1999. pages 18

[32] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM transactions on database systems*, 34(3):16, 2009. pages 7, 8, 9, 10

[33] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010. `http://www.sciencedirect.com/science/article/B758F-4Y95V3X-1/2/9e5098d690fbe4d05a099f4c90a29a10`. pages 3, 19, 21, 22, 23, 28, 30, 31, 34, 35, 41

[34] A. Polleres. From SPARQL to rules (and back). In *Proc. 16th World Wide Web Conference (WWW)*, pages 787–796, 2007. pages 7, 9

[35] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Recommendation, W3C, January 2008. pages 3, 7, 13

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

[37] A. Souzis. RxPath specification proposal, 2004. `http://rx4rdf.liminalzone.org/RxPathSpec`. pages 34

[38] H. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2):79–115, 2005. pages 14, 15

[39] M. Yannakakis. Graph-theoretic methods in database theory. In *Proc. 9th ACM Symposium on Principles of Database Systems (PODS)*, pages 230–242, 1990. pages 28, 41

[40] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: Expressive navigation in RDF graphs. In *Proc. 4th International Conference on Web reasoning and rule systems (RR), Bressanone/Brixen (IT)*, volume 6333 of *Lecture Notes in Computer Science*, pages 251–257, 2010. pages 35

Contents

1 Introduction **3**

2 Querying RDF with SPARQL **4**
2.1 RDF 4
2.2 SPARQL 7

3 Querying RDF modulo RDF Schema **10**
3.1 RDF Schema 10
3.2 Querying against ter Horst closure 13

4 The PSPARQL query language **15**
4.1 PRDF syntax 16
4.2 PRDF semantics 16
4.3 PSPARQL 17
4.4 SPARQL queries modulo RDFS with PSPARQL 18

5 nSPARQL and NSPARQL **19**
5.1 nSPARQL syntax 19
5.2 nSPARQL semantics 21
5.3 NSPARQL 22
5.4 SPARQL queries modulo RDFS with nSPARQL 22
5.5 SPARQL queries modulo RDFS with NSPARQL 24

6 cpSPARQL and CPSPARQL **25**
6.1 CPSPARQL syntax 25
6.2 CPSPARQL semantics 27
6.3 Evaluating cpSPARQL regular expressions 28
6.4 SPARQL queries modulo RDFS with CPSPARQL 29

7 On the respective expressiveness of cpSPARQL and nSPARQL **30**
7.1 Nested regular expressions (nSPARQL) cannot express all (SPARQL) triple patterns 30
7.2 nSPARQL without SELECT cannot express all CPSPARQL 30
7.3 nSPARQL cannot express all cpSPARQL, even with SELECT 31
7.4 cpSPARQL can express all nSPARQL 33

8 Implementation **33**

9 Related work **34**

10 Conclusion **35**

A Proofs **37**
A.1 Induction lemma 37
A.2 Completeness of partial non reflexive RDFS closure 38
A.3 Completeness and complexity of the PSPARQL RDFS query encoding 39
A.4 Completeness of NSPARQL query answering 41
A.5 Complexity of cpSPARQL evaluation 41
A.6 Correctness and completeness of RDFS translation to PSPARQL 41
A.7 Expressiveness of nSPARQL and cpSPARQL 42

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399