The Perfect Match: RPL and RDF Rule Languages

François Bry[1] and Tim Furche[1] and Benedikt Linse[1]

University of Munich,
`http://www.pms.ifi.lmu.de`

Abstract. Path query languages have been previously shown to complement RDF rule languages in a natural way and have been used as a means to implement the RDFS derivation rules. RPL is a novel path query language specifically designed to be incorporated with RDF rules and comes in three flavors: *Node*-, *edge*- and *path*-flavored expressions allow to express conditional regular expressions over the nodes, edges, or nodes *and* edges appearing on paths within RDF graphs. Providing regular *string* expressions and negation, *RPL* is more expressive than other RDF path languages that have been proposed. We give a compositional semantics for *RPL* and show that it can be evaluated efficiently, while several possible extensions of it cannot.

Graph traversal operators play a crucial role in rule and query languages for semi-structured data and for RDF rule languages in particular. This need bas been acknowledged by the development of languages like Versa [11] SPARQLeR [7] and nSPARQL [12] and underlined in [2]. Moreover, the need for traversal of semi-structured data in general, and XML in particular is underscored by the huge success of XPath, arguably the most prominent XML query language. In [12] it has been shown that SPARQL augmented with conditional (in the sense of [9]) regular path expressions is expressive enough to query RDF graphs under the RDFS semantics without computing the closure of the graph under the RDFS entailment rules.

Most path query languages proposed up until now are unfit for clean integration with RDF rule languages for the following reasons: (i) their use of variables interferes with the use of logical variables already present in rule languages, (ii) they do not always evaluate to pairs of nodes and thus cannot be safely used at the place of RDF predicates in query patterns and (iii) they lack negation, regular string expressions and often also conditional operators.

We propose the RDF path language RPL, that is designed for easy integration with RDF rule languages such as SPARQL [14], XCERPT$^{\text{RDF}}$ [5, 13] and RDFLog [4, 3]. RPL is an orthogonal extension to RDF rule languages in that it sets out to extend RDF rule languages by features they lack, and in that it tries to avoid duplication of features they already provide. RPL expressions always evaluate to pairs of nodes within an RDF graph, and can thus be safely used at the place of predicates within the body of RDF rules. Despite of this restriction, SPARQL extended with RPL predicates is capable, just as nSPARQL, to query RDF graphs under the RDFS semantics without computing the closure of the queried graphs under the RDFS entailment rules. RPL is more expressive than previously proposed RDF query languages in that it provides regular string expressions and negation.

RDF Path Expressions (RPEs) come in three flavors: *node-restricting*, *edge-restricting* and *path-restricting*, identified by the keywords NODES, EDGES, PATH, respectively. *Node-restricting* (*edge-restricting*) RPEs only place restrictions on the nodes (edges) appearing within a path. *Path-restricting* expressions may place restrictions on both, nodes and edges. RPEs evaluate to sets of pairs of nodes – i.e. binary relations over the set N of nodes of an RDF graph. The three unrestrictive RPEs [PATH (_ _)∗], [EDGES _∗] and [NODES _∗] evaluate to $N \times N$.

This paper is organized as follows: Section 1 informally introduces the semantics of RPL by example, before its syntax and semantics is formally defined in Sections 2 and 3. Section 5 compares RPL to related path query languages and comes up with first complexity results. Section 6 shows the tractability of RPL as a whole, and the intractability of node and edge flavored path RPL expressions augmented with *unordered* paths.

The contributions of this paper are as follows: (i) We formalize the syntax and semantics of RPL expressions, and (ii) show that RPL can express all relevant RDFS queries. (iii) We show that RPL can be evaluated efficiently, and (iv) that also nSPARQL could be extended by regular string expressions and negation without sacrificing tractability. (v) Finally we show that extensions of RPL and nSPARQL to unordered paths results in the loss of the tractability of both languages.

1 RPL by Example

Before introducing RPL, we define the notions of RDF triples, graphs, and paths in RDF graphs.

Definition 1 (RDF triple, graph). *Let U, B, L be three disjoint sets of URIs, blank node identifiers and RDF literals. Then $t = (s, p, o) \in U \cup B \times U \times U \cup B \cup L$ is an* RDF triple*, and $t_g \in U \times U \times U \cup L$ is a* ground RDF triple*. s, p, o are the* subject*,* predicate *and* object *of t, respectively. A (ground)* RDF graph *is a set of (ground) RDF triples. The set of* nodes *N of an RDF graph G are all elements in $U \cup B \cup L$ that appear in subject or object position of a triple in G.*

Definition 2 (Path in an RDF graph). *Let G be an RDF graph. The sequence $n_1, \ldots, n_k$ is a* path *in G, iff the triples (n_1, n_2, n_3), (n_3, n_4, n_5), $\ldots$, (n_{k-2}, n_{k-1}, n_k) are in G.*

Example 1. [PATH (_ eg:/.∗/)∗ rdf:type]: All pairs (n_1, n_2) of nodes connected over intermediate nodes of the namespace eg. Additionally, the last edge on the connecting path must correspond to the qualified name rdf:type. This first example demonstrates the following points:

- RPEs start with an opening square bracket followed by one of the keywords PATH, EDGES and NODES specifying the flavor of the path expression, and end with a closing square bracket.
- As in SPARQL, XPath, XQuery, XSLT and XCERPT$^{\mathrm{RDF}}$, URIs may be abbreviated by qualified names.
- Wildcards (_) and regular expressions (e.g. /.∗/) play an important role within RPEs. Together with qualified names, URIs and literals, they constitute the atomic building blocks of RPEs, called *atomic RPEs*.
- From atomic RPEs, *compound* RPEs can be built via *sequencing* (denoted by whitespace), *alternation* (`|`), *Kleene closure* (`*` and `+`), *optionality* (`?`), and *negation* (`not(...)`).

Example 2. The expression [PATH (>eg:p ^_[not(PATH eg:p1])])∗ eg:p] collects all pairs of nodes connected over a path with at least one predicate with URI eg:p. All intermediate nodes must not have an outgoing eg:p1 edge.

This second example introduces *path directions* and *path predicates* and demonstrates the following points:

- URIs, regular expressions or qualified names within RPEs may be modified by one of the directions '>' (*forward predicate*), '<' (*reverse predicate*) or '^' (*node*). If an atomic RPE is prefixed with '<' ('^') then it must match with a reverse edge (node) on the path connecting the nodes n_1 and n_2. If an atomic RPE is undirected or prefixed by '>', then it must match a *forward* edge on the path connecting n_1 and n_2.
- Path expressions may be nested via *path predicates*, which roughly correspond to XPath predicates. While URIs, qualified names or regular expressions within RPEs represent *local restrictions* only, predicates allow the specification of *non-local* restrictions, i.e. restrictions that are not directly enforced on nodes or edges on the path, but on nodes or edges connected via a nested path expression.

Example 3. The edge-flavored expression [EDGES rdf:type (rdfs:subClassOf)∗] evaluates to all pairs of nodes connected via one rdf:type edge and zero or more rdfs:subClassOf edges (in this order).

This query determines the indirect class membership of resources under the RDFS semantics. Note that also for many other RDF queries, only the edges along a path are relevant. The query [EDGES (<rdfs:subClassOf)∗ <rdf:type] evaluates to the reverse relation.

Example 4. The node-flavored expression [NODES (eg:a eg:b)] finds all pairs of nodes that are connected over nodes eg:a and eg:b (in this order), with arbitrary predicates on the path. The query [NODES (eg:/.∗/ | foaf:/.∗/)∗] on the other hand, finds all pairs of nodes connected over a path of length zero or more which contains only intermediate nodes belonging to the namespaces eg or foaf. The predicates on the path are irrelevant, as indicated by the keyword NODES.

Example 5 (RDFS querying with RDFLog augmented by RPL). This example shows how RDF rule languages can be augmented by RPL path expressions to immitate the RDFS semantics. We choose RDFLog as the extended rule language because of its simplicity. The RDFLog rule

$$\forall x\ p\ y\ p_1\ z\ .\ (x\ p\ y) \leftarrow (x\ p_1\ z), (p_1\ [\text{EDGES subprop* }]\ y) \qquad (1)$$

can be used to materialize the extension of the predicate p under the RDFS semantics. In a backward chaining evaluation of an RDFLog program, materialization is only carried out *on demand*, and is thus more efficient than computing the RDFS closure of the queried graph. If only single rules or queries are allowed (such as in SPARQL), then the body of Equation 1 can simply be used in the query at the place of p.

The extension of predicates with a special semantics under the RDFS model theory deserve special treatment. E.g the extension of rdf:type is computed by the following RDFLog rules with RPL predicates:

$$\begin{aligned}\forall x\ y\ .\ (x\ \mathit{type}\ y) &\leftarrow (x\ [\text{EDGES type subclass*}]\ y)\\ \forall x\ y\ p_1\ z.\ (x\ \mathit{type}\ y) &\leftarrow (x\ p_1\ z), (p_1\ [\text{EDGES subprop* dom subclass*}]\ y)\\ \forall x\ y\ p_1\ z.\ (x\ \mathit{type}\ y) &\leftarrow (z\ p_1\ x), (p_1\ [\text{EDGES subprop* range subclass*}]\ y)\end{aligned}$$

It can be shown that also extensions of the remaining RDFS predicates subclasss, subProperty, domain and range can be encoded as RDFLog or SPARQL rule bodies augmented with RPL. The encoding is analogous to the one presented in [12] and is omitted here for the sake of brevity.

2 RPL Syntax

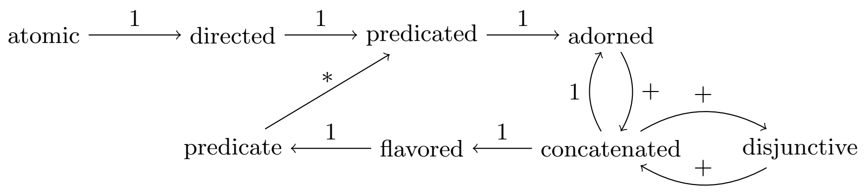
Definition 3 (Abstract syntax of RPEs). *The abstract syntax of RPL, is recursively defined as follows:*

- *A URI u, regular expression re, qualified name q, literal l and wild card* _ *is an* atomic RPE. *Moreover, a qualified name $prefix$:localpart where localpart is a regular expression, is an atomic RPE.*
- *If p is an atomic path expression, then p, $< p$, $> p$ and $\hat{}p$ are* directed *path expressions.*
- *if p_1 is an atomic RPE, and $q_1, \ldots q_n$ are RPL predicates (see below), then p_1 and $p_1[q_1]\ldots[q_n]$ are* predicated *RPEs.*
- *If p is a predicated or concatenated RPE, then p, $p*$, $p+$ and $p?$ are* adorned *RPEs.*
- *If $p_1, \ldots p_n$ are adorned or disjunctive (see below) RPEs, then $(p_1 \ldots p_n)$ with $n \geq 1$ is a* concatenated *RPE.*
- *If $p_1, \ldots p_n$ are concatenated RPEs, $(p_1 \mid \ldots \mid p_n)$ with $n \geq 1$ is a* disjunctive *RPE.*

– *If p is a concatenated RPE, PATH p, EDGES p, NODES p are* flavored *RPEs. They are called path-restricting, edge-restricting and node-restricting expressions, respectively.*
– *If p is a flavored RPE, then p and not(p) are RPL predicates.*

Figure 1 summarizes the relationships between the different types of subexpressions in RPL. An arrow labeled with 1, + or $*$ from type A to type B means that expressions of type B are made up of exactly one, at least one, or zero or more expression of type A, respectively. It holds that any atomic RPL expression is also a directed subexpression, which are in turn also predicated subexpressions, which are in turn adorned subexpressions. As in XQuery, a concatenated expression (called sequence in XQuery) of one element is equivalent to the element itsself. Also an disjunctive RPE of one element is equivalent to the element itsself. Only flavored RPL expressions can be embedded in rule languages.

Fig. 1. Relationships among subexpressions of RPEs


$$\begin{tikzcd}
\text{atomic} \arrow[r, "1"] & \text{directed} \arrow[r, "1"] & \text{predicated} \arrow[r, "1"] & \text{adorned} \arrow[d, bend left, "+"] & \\
& \text{predicate} \arrow[ur, "*"] & \text{flavored} \arrow[l, "1"'] & \text{concatenated} \arrow[l, "1"'] \arrow[u, bend left, "1"] \arrow[r, bend left, "+"] & \text{disjunctive} \arrow[l, bend left, "+"]
\end{tikzcd}$$

The following remarks clarify Definition 3.

– Atomic RPEs correspond to the building blocks of ground RDF graphs with the following exceptions: (i) qualified names are allowed as shorthand notations for URIs, (ii) regular expressions are allowed as a means for matching URIs and Literals[1], (iii) the local part of a qualified name may be expressed by a regular expression, (iv) wildcards can be used to match any blank node, URI or literal.
– RPEs do not provide any means for selecting RDF literals based on their types or based on their language tags, other than using a regular expression for this purpose.
– Just as with ordinary regular (string) expressions, parentheses are used to influence operator precedence. The operators Kleene star (`*`), Kleene plus (`+`), optionality (`?`) are mutually exclusive and have precedence over all other operators. The concatenation operator (denoted by whitespace) binds stronger

[1] matching blank nodes with regular expressions is not allowed, since this would mean *syntactic* matching of RDF graphs, i.e. the semantics of an RPE would be dependent on the syntactic representation of the RDF graph that is being queried.

than the disjunctive operator $|$, i.e. $a\ b \mid c$ is equivalent to $(a\ b) \mid c$. Parentheses may be omitted, if they do not alter operator precedence.

3 RPL Compositional Semantics

The intuitive presentation of the RPEs is now formalized by a compositional semantics, which is given by the function $[\![\cdot]\!]$ and its four helper functions $[\![\cdot]\!]^P$ for path-restricting expressions, $[\![\cdot]\!]^E$ for edge-restricting expressions, $[\![\cdot]\!]^N$ for node-restricting expressions and $[\![a]\!]^V$ for atomic expressions a that are evaluated in *vertex position*. While the functions $[\![\cdot]\!]$, $[\![\cdot]\!]^P$, $[\![\cdot]\!]^E$ and $[\![\cdot]\!]^N$ evaluate to subsets of $N \times N$, i.e. binary relations on the set N of nodes of the queried RDF graph, the function $[\![\cdot]\!]^V$ evaluates to subsets of N.

In order to present the semantics in an easily digestible manner, we split the entire definition according to the flavor of the RPE to be formalized. Definition 4 gives the semantics for *edge-restricting* RPEs, Definitions 5, 6 and 7 add the necessary equations for *node-restricting*, *path-restricting* and arbitrary RPEs, respectively. The three flavors of RPEs differ in the way subexpressions are concatenated. In contrast, most equations for evaluating atomic RPEs, alternatives and Kleene closures are independent of the flavor and are only given once.

In the following, let G be an RDF graph over the vocabulary $U \cup B \cup L$, u a URI, l an RDF Literal, re a regular expression, a an atomic RPE, pe a predicated RPE, $f_1, \ldots f_k$ flavored RPEs, and e, $e_1, \ldots, e_k$ arbitrary RPEs.

Definition 4 (Semantics of edge-restricting RPEs). *The semantics of edge-restricting RPEs is given by the function $[\![\cdot]\!]^E$ defined as follows:*

$$[\![u]\!]^{E,P} = \{(n_1, n_2) \mid (n_1, u, n_2) \in G\} \tag{2}$$

$$[\![\_]\!]^{E,P} = \{(n_1, n_2) \mid \exists p \ .\ (n_1, p, n_2) \in G\} \tag{3}$$

$$[\![/re/]\!]^{E,P} = \{(n_1, n_2) \mid \exists p \in \mathcal{L}(re)\ .\ (n_1, p, n_2) \in G\} \tag{4}$$

$$[\![{>}pe]\!]^X = [\![pe]\!]^X \textit{ for } X \in \{E, P\} \tag{5}$$

$$[\![{<}pe]\!]^X = \{(n_2, n_1) \mid (n_1, n_2) \in [\![{>}pe]\!]^X\} \textit{ for } X \in \{E, P\} \tag{6}$$

$$[\![e_1\ \ldots\ e_k]\!]^E = \{(n_1, n_{k-1}) \mid \exists n_2, \ldots n_k\ .\ \forall 1 \le i \le k\ ((n_i, n_{i+1}) \in [\![e_i]\!]^E)\} \tag{7}$$

$$[\![(e_1 \mid \ldots \mid e_k)]\!]^X = [\![e_1]\!]^X \cup \ldots \cup [\![e_k]\!]^X \textit{ for } X \in \{P, E, N\} \tag{8}$$

$$[\![a[f_1] \ldots [f_k]]\!]^E = \bigcup_{a' \in [\![a[f_1]\ldots[f_k]]\!]^V} [\![a']\!]^E \tag{9}$$

$$[\![\epsilon]\!]^{E,P} = \{(n, n) \mid n \in N\} \tag{10}$$

$$[\![e+]\!]^X = [\![e]\!]^X \cup [\![e\ e+]\!]^X \textit{ for } X \in \{P, E, N\} \tag{11}$$

$$[\![e*]\!]^X = [\![\epsilon]\!] \cup [\![e+]\!]^X \textit{ for } X \in \{P, E, N\} \tag{12}$$

$$[\![e?]\!]^X = [\![\epsilon]\!] \cup [\![e]\!]^X \textit{ for } X \in \{P, E, N\} \tag{13}$$

The centerpiece of Definition 4 is Equation 7. It states that the semantics of a sequence of edge-restricting RPEs is a binary relation of nodes (n_1, n_2) such that there is a path from n_1 to n_2 over arbitrary intermediate nodes $n_2, \ldots n_{k-1}$ such that these intermediate nodes are connected via the subexpressions $e_1, \ldots, e_k$.

The other equations in Definition 4 do not only hold for edge-restricting RPEs, but also for path-restricting ones, and some hold also for node-restricting expressions, as indicated by X.

Equations 2, 3 and 4 establish that a URI u evaluates to the pairs of nodes connected via a predicate of name u, the wildcard character to all pairs of nodes connected via an arbitrary predicate, and a regular string expression re to those pairs of nodes which are connected via a predicate that is in the language $\mathcal{L}(re)$ defined by re. Note that when part of a node-restricting expression, the semantics of URIs, wildcards and regular string expressions is different (see Definition 5).

Equations 5 and 6 formalize the specification of edge traversal in forward or reverse direction with the directions $<$ and $>$. If no direction is given, then Equations 2, 3 and 4 hold, i.e. forward traversal is assumed.

Equations 10, 11, 12 and 13 define the semantics of the Kleene star, Kleene plus and optional parts of RPEs.

Formalizing the semantics of predicates within edge-restricting expressions, Equation 9 references Definition 5. Here the idea is to also allow the formulation of queries that use the same URI in predicate and subject or object position. An example for such queries from [12] is finding all pairs of cities that are connected via some transportation service, given a hierarchy of transportation services and connections among cities using instances of this hierarchy.

Definition 5 (Semantics of node-restricting RPEs). *The semantics for node-restricting RPEs is defined as follows:*

$$\begin{aligned}
[\![\_]\!]^V &= N && (14)\\
[\![/re/]\!]^V &= N \cap \mathcal{L}(re) && (15)\\
[\![u]\!]^V &= \{u\} \cap N && (16)\\
[\![l]\!]^V &= \{l\} \cap N && (17)\\
[\![pe]\!]^N &= \{(n,n) \mid n \in [\![pe]\!]^V\} && (18)\\
[\![a[f_1]\ldots[f_k]]\!]^V &= [\![a]\!]^V \cap \{n_1 \mid \exists n_2 \,.\, (n_1,n_2) \in [\![f_1]\!]\} \cap && (19)\\
&\quad \ldots \cap \{n_1 \mid \exists n_2 \,.\, (n_1,n_2) \in [\![f_k]\!]\} && (20)\\
[\![e_1 \ \ldots \ e_k]\!]^N &= \{(n_1, n_{2k}) \mid \exists n_2, \ldots n_{2k-1}, p_1, \ldots, p_{k-1} \,. && (21)\\
&\quad \forall 1 \le i \le k \ ((n_{2i-1}, n_{2i}) \in [\![e_i]\!]^N) \wedge \\
&\quad \forall 1 \le i \le k-1 \ ((n_{2i}, p_i, n_{2i+1}) \in G)\}
\end{aligned}$$

While many RDFS queries only respect the predicates on a path between two resources and are therefore best expressed as edge-restricting RPEs, some path queries may only be interested in the traversed nodes and are better expressed as node-restricting RPEs. An example for this type of query is finding all pairs of persons in a social graph that are somehow connected over the resources anna and new_york. This query could be answered by the RPE

NODES (anna new_york) | (new_york anna). Definition 5 formalizes node-restricting RPEs.

In this setting, a URI, regular string expression, wildcard or qualified name is evaluated in node position (Equation 18), and is thus treated differently from the evaluation within edge-restricting RPEs (Equation 2).

The core of Definition 5 is the formalization of node concatenation in Equation 21. Concatenations may involve arbitrary RPEs, i.e. atomic, predicated, directed, grouped path expressions, alternatives, Kleene closures and concatenations themselves. While the nodes on the path described by a node-restricting concatenation are given by the subexpressions of the concatenation, the predicates are arbitrary. Equation 21 makes use of the binary helper function $[\![\cdot]\!]^N$ defined on subexpressions, and the unary function $[\![\cdot]\!]^V$, which is part of the formalization of path-restricting RPEs.

Path-restricting RPEs are needed whenever constraints shall be laid both on the predicates and nodes on a path within an RDF graph. Equation 23 is the centerpiece of Definition 6. Path-restricting RPEs are expected to start and end with restrictions on the first and last *edge* of an RDF graph, because they are designed for easy integration with RDF query languages such as SPARQL and Xcerpt$^{\text{RDF}}$ where they are used at the place of RDF predicates. If the first and/or last restriction is laid on a node instead, this must be indicated with a '^' symbol, and Equations 24 and 25 apply.

Definition 6 (Semantics of path-restricting RPEs). *The semantics of path-restricting RPEs is defined as follows:*

$$[\![\hat{}a]\!] = [\![a]\!]^V \tag{22}$$

$$\begin{aligned}[\![e_1\ \dots\ e_k]\!]^P = \{(n_1, n_j) \mid \exists n_2, \dots, n_{j-1}\ .\ (n_1, n_2) \in [\![e_1]\!]^P \wedge n_2 \in [\![e_2]\!]^V \wedge \\ \dots \wedge n_{j-1} \in [\![e_{k-1}]\!]^V \wedge (n_{j-1}, n_j) \in [\![e_k]\!]^P\}\end{aligned} \tag{23}$$

$$[\![\hat{}pe\ e]\!]^P = \{(n_1, n_2) \in [\![e]\!]^P \mid n_1 \in [\![pe]\!]^V\} \tag{24}$$

$$[\![e\ \hat{}pe]\!]^P = \{(n_1, n_2) \in [\![e]\!]^P \mid n_1 \in [\![pe]\!]^V\} \tag{25}$$

Definition 7 (Semantics of flavored RPEs).

$$[\![PATH\ e]\!] = [\![e]\!]^P \tag{26}$$

$$[\![EDGES\ e]\!] = [\![e]\!]^E \tag{27}$$

$$\begin{aligned}[\![NODES\ e]\!] = \{(n_1, n_4) \mid \\ \exists n_2, n_3, p_1, p_2\ .\ (n_2, n_3) \in [\![e]\!]^N \wedge (n_1, p_1, n_2), (n_3, p_2, n_4) \in G\}\end{aligned} \tag{28}$$

$$[\![not(u)]\!] = [\![\_]\!] \setminus [\![u]\!] \tag{29}$$

4 RPL Restrictions and Extensions

In order to compare RPL to other regular path languages over ordinary graphs and RDF graphs, and to study the complexity of RPL fragments, we introduce the following set of sublanguages:

Definition 8 (RPL sublanguages). *Besides the operators* `+`, `?` *and* `*`, *RPL makes use of the following features:*

- *regular string expressions (denoted by RSE)*
- *the EDGE keyword (denoted by $\rightarrow$)*
- *the NODE keyword (denoted by $\circ$)*
- *the PATH keyword (denoted by $\dashrightarrow$)*
- *predicates (denoted by $[]$)*
- *concatenation (denoted by $/$)*
- *disjunction (denoted by $|$)*
- *predicate negation (denoted by $\neg$)*
- *direction modifiers (denoted by μ)*

$RPL^{f_1,\ldots,f_k}$ with $f_1,\ldots,f_k \in \{RSE, \rightarrow, \circ, \dashrightarrow, [], /, |, \neg, \mu\}$ denotes the sublanguage of RPL making use of the operators `+`, `?`, *and* `*` *and the features $f_1,\ldots,f_n$ only.*

Languages such as XPath and Xcerpt allow queries to be incompletely specified in depth, or with respect to order. Incompleteness in depth is specified via the descendant axis in XPath and via the `desc` keyword in Xcerpt. Incompleteness with respect to order is the default querying mode in XPath and can be overridden by using the `<<` operator; in Xcerpt it is specified via curly braces.

An obvious extension of RPL is thus to introduce *unordered* and *incomplete* paths. While the order in Xcerpt query terms is enforced/relaxed with respect to the *sibling* axis of an XML document, the order in RPEs may be relaxed with respect to the paths traversed, i.e. the *descendant* axis. Also the concept of *incomplete* specification of *siblings* in Xcerpt query terms may be transfered to the *descendant* axis by allowing double brackets within RPL. We denote the extensions of the sublanguages of RPL by unordered paths, incomplete paths and both by adding the symbols $\{\}$, $[[]]$ or both to the feature list of the sublanguage. The RPL expression `Nodes { x y z }` thus evaluates to all pairs of nodes that are connected by a path containing only the intermediate nodes `x`, `y`, and `z` in an abritrary order. The RPL expression `Nodes [[ x y ]]` on the other hand evaluates to all pairs of nodes that are connected via a path that contains the nodes x and y with x appearing before y, and an arbitrary number of nodes before x, between x and y and following y.

The semantics of $\{\}$ is formalized by the functions $[\![\cdot]\!]^{UN}$, $[\![\cdot]\!]^{UE}$, and $[\![\cdot]\!]^{UP}$ for unordered node-flavored, edge-flavored and path-flavored expressions, respectively. The semantics of $[[]]$ is given by the functions $[\![\cdot]\!]^{IN}$, $[\![\cdot]\!]^{IE}$, and $[\![\cdot]\!]^{IP}$.

Definition 9 (Semantics of unordered and incomplete RPEs).

$$[\![e]\!]^{UN} = \bigcup_{p \in Perm(e)} [\![p]\!]^{N} \tag{30}$$

$$[\![e]\!]^{UE} = \bigcup_{p \in Perm(e)} [\![p]\!]^{E} \tag{31}$$

$$[\![e]\!]^{UP} = \bigcup_{p \in Perm(e)} [\![p]\!]^{P} \tag{32}$$

$$[\![e]\!]^{IN} = \bigcup_{c \in Comp(e)} [\![c]\!]^{N} \tag{33}$$

$$[\![e]\!]^{IE} = \bigcup_{c \in Comp(e)} [\![c]\!]^{E} \tag{34}$$

$$[\![e]\!]^{IP} = \bigcup_{c \in Comp(e)} [\![c]\!]^{P} \tag{35}$$

A completion *of a sequence $e := e_1, \ldots, e_n$ is a sequence c that contains all elements of e plus an arbitrary number of wildcards. A completion of e is called* order-respecting, *iff for $e_i, e_j \in e$ with $i < j$, e_i appears in c before e_j. $Perm(e)$ and $Comp(e)$ denotes the set of all permuations and order respecting completions of e, respectively.*

Both extensions of RPL – to unordered paths and to incomplete paths – are mere syntactic sugar. The RPE `Nodes { x y }` can be rewritten to the equivalent RPE `Nodes (x y) | (y x)` and the RPE `Nodes [[ x y ]]` can be rewritten to `Nodes _* x _* y _*`. Observe that whereas the rewriting of incomplete path expressions is linear in the size of the original expression, the rewriting of unordered paths is exponential in the size of the original expression. We chose not to include incomplete RPEs in standard RPL, since one can easily do without them. On the other hand we chose not to include unordered RPEs in standard RPL, because it would make evaluation of RPL NP-hard as shown in Section 6.

The semantics of RPEs that are both unordered and incomplete (denoted by $\{\{\}\}$) is easily defined at the aid of non-order-respecting permutations. For the sake of brevity, we omit this extension of RPL.

5 RPL compared to Lorel, SPARQLeR and nSPARQL

[?] extends SPARQL by regular expression patterns which may occur at the place of predicates in RDF graphs. These regular expression patterns include amongst others kleene closure, disjunction, concatenation, but not predicate negation and regular string expressions. Moreover, node labels are are not considered part of the path to be matched by the regular expression pattern.

The Lorel query language[1] is an offspring of the XML database system Lore, but can be used to query all kinds of semi-structured data. It has received considerable attention in the research community, partially due to its incorporation of regular path expressions.

RPEs compare to Lorel path expressions as follows:

- The data model of Lorel is an edge-labeled graph, without node labels. Therefore Lorel does not distinguish the three flavors of RPEs.
- Both languages provide the unary operators Kleene plus (`+`), Kleene star (`*`) and optionality (`?`), and the binary operators concatenation (denoted by '`.`' in Lorel), and alternative.
- Lorel allows the use of the character '`%`' to match 0 or more characters within a label. RPL on the other hand allows regular string expressions. Wildcards for entire labels are denoted by '`#`' in Lorel and '`_`' in RPL.
- Lorel allows the extraction of values from traversed paths by so-called path variables. RPEs do not use variables since they may be embedded in RDF query language such as SPARQL or XCERPT$^{\mathrm{RDF}}$, that provide themselves variables.
- RPEs allow the restriction of paths based on path predicates, Lorel does not. Hence query 2 is not expressible in Lorel.

In [10] the evaluation of regular expressions over the alphabet σ of an edge-labeled graph g is studied. Compared to RPEs, [10] considers the labels of edges to be atomic, i.e. they do not consider regular string expressions on node or edge-level. Moreover, non-local restrictions on paths (i.e. predicates) and traversal in reverse direction are not expressible. Since nodes in the queried graphs are unlabelled, only the edge labels are relevant, i.e. the path expressions in [10] correspond to a subset of edge-flavored RPEs.

[10] considers the problems *Regular Simple Path*, *Fixed Regular Path (R)*, and *Regular Path*. The problem Regular Simple Path takes a regular expression e, a graph g over the same alphabet Σ, and a pair of nodes (x, y) as input, and returns true iff g contains a directed simple path from x to y that satisfies e. A path is called *simple*, if it does not contain the same vertex twice. The problem Fixed Regular Path is the same as regular simple path, but e is not considered as input. Regular Path is the same as Regular Simple Path, but the path is not required to be simple.

[10] show that Fixed Regular Simple Path is NP-complete and Regular Simple Path is NP-hard by a simple reduction from the problems Even Path and Disjoint Paths treated in [8] and [6], respectively. Regular Path, however, is decidable in time $O(|E|\,|D|)$, where $|E|$ is the size of the regular path expression and $|D|$ is the size of the data – shown by the construction of a product automaton of the NFA of a regular path expression and the database graph interpreted as a NFA. In RPL we choose to accept arbitrary paths, including non-simple paths as possible connections among two nodes. RPEs are more expressive than the regular path expressions of [10] in three respects: (i) They allow the specification of predicates on nodes, (ii) regular expressions for matching edge and node labels, and (iii)

in that they take into account also the labels of *nodes*. Therefore, the results of [10] leave the question, if there is a polynomial time algorithm for the evaluation problem of RPEs, open. The following result for the complexity of $RPL^{\rightarrow,/,|,\mu}$ expressions is a direct consequence of the complexity *Regular Path*.

Corollary 1. *$RPL^{\rightarrow,/,|,\mu}$ can be evaluated in time $O(|E|\,|G|)$, where $|E|$ is the size of the path expression and $|G|$ is the size of the queried RDF graph.*

[12] propose the regular path language nSPARQL with the following syntax:

$$exp := \text{ axis } \mid \text{ axis::}a\ (a \in U) \ \mid \text{ axis::}[exp] \mid exp/exp \mid exp|exp \mid exp^{*} \quad (36)$$

where $axis \in \{self, next, next^{-1}, node, node^{-1}, edge, edge^{-1}\}$ and U denotes the set of URIs. The axes *next*, *edge* and *node* are used to navigate from one node in an RDF graph to an adjacent one, from a node to one of its outgoing edges and from an edge to its sink. If the starting node is left unspecified, *next*, *edge* and *node* can be interpreted as binary relations over an RDF graph G. Node tests following the axes *next*, *edge* and *node* constrain the label of a traversed edge, the object of an arc, and the subject, respectively. The semantics of the predicates [], alternatives |, Kleene star *, and concatenation / are as expected.

In this section we briefly give an intuitive semantics of nSPARQL path expressions by translating Examples 1, 2, 3 and 4 to nSPARQL.

We abbreviate URIs in nSPARQL path expressions by qualified names to shorten the examples.

Example 6 (nSPARQL path expressions).

- Example 1 is *contained* in the nSPARQL path expression (next)*/next::rdf:type. An exact translation is not possible due to the absence of regular string expressions for matching nodes or edges of RDF graphs.
- Example 2 is *contained* in the nSPARQL path expression (next::eg:p)$^{+}$. An exact translation is not possible due to the absence of negation in nSPARQL predicates.
- Example 3 is *equivalent* to next::rdf:type/(next::rdfs:subClassOf)*.
- The first RPL expression in Example 4 is *equivalent* to *next*/self::eg:a/*next*/self::eg:b in nSPARQL.
- The nSPARQL path expression

$$\text{next::a/(next::[next::a/self::b])}^{*}\text{/(next::[node::b] | next::a)}^{+} \qquad (37)$$

from [12] is *contained* in the RPE [EDGES a(_[PATH a b]) $*$ _]. An exact translation to an RPE is not possible, since RPEs always evaluate to pairs of *nodes* of an RDF graph. In contrast, nSPARQL expressions may also evaluate to pairs of edges and nodes, as the subexpression "node::b" of Expression 6 does. Expression 6 can, however, be translated to an equivalent Xcerpt$^{\text{RDF}}$ query term or SPARQL query pattern that makes use of a single RPE.

Given an nSPARQL path expression *exp*, an RDF graph G, and a pair of nodes (n_1, n_2), the problem whether there is a path from n_1 to n_2 matching *exp* within G, can be decided in $O(|G| \cdot |exp|)$.

Corollaries 2 and 3 shed light on the expressive relationship between fragments of RPL and nSPARQL. An immediate consequence of corollary 2 is corollary 4.

Corollary 2. *Any RPE* $r \in RPL^{\rightarrow,\circ,\dashrightarrow,[],/,|,\mu}$ *can be translated to an equivalent nSPARQL path expression of length* $\mathcal{O}(|p_c|)$.

Proof. The translation function from $\text{RPL}^{\rightarrow,\circ,\dashrightarrow,[],/,|,\mu}$ to nSPARQL is given in Listing 1.1. Obviously, the size of `to_nSPARRQL(exp)` is linear in the size of `exp` for any RPL expression in $\text{RPL}^{\rightarrow,\circ,\dashrightarrow,[],/,|,\mu}$.

Listing 1.1. Translation from RPL to nSPARQL

```
to_nSPARQL(EDGES exp)          = to_nSPARQL(exp, edges)
to_nSPARQL(NODES exp)          = next/to_nSPARQL(exp, nodes)/next
to_nSPARQL(PATH  exp)          = to_nSPARQL(exp, path)
to_nSPARQL(exp*, mode)         = to_nSPARQL(exp, mode)*
to_nSPARQL(exp+, mode)         = to_nSPARQL(exp, mode)+
to_nSPARQL(exp?, mode)         = self | to_nSPARQL(exp, mode)

to_nSPARQL(_, edges)           = next
to_nSPARQL(u, edges)           = next::u
to_nSPARQL(>u, edges)          = next::u
to_nSPARQL(<u, edges)          = next-1::u
to_nSPARQL(u[p1]...[pn], edges)      =
   next::u[to_nSPARQL(p1)]...[to_nSPARQL(pn)]
to_nSPARQL(exp1|...|expn, mode)      =
  to_nSPARQL(exp1, mode)|...|to_nSPARQL(expn, mode)
to_nSPARQL(exp1 ... expn, edges)     =
  to_nSPARQL(exp1, edges)/.../to_nSPARQL(expn, edges)

to_nSPARQL(_, nodes)           = self
to_nSPARQL(u, nodes)           = self::u
to_nSPARQL(exp1 ... expn, nodes)     =
  to_nSPARQL(exp1, nodes)/next/.../next/to_nSPARQL(expn, nodes)
to_nSPARQL(a[p1] ... [pn], nodes)    =
   self::a[to_nSPARQL(p1)] ... [to_nSPARQL(pn)]

to_nSPARQL(^a, path)           = self::a
to_nSPARQL(>a, path)           = next::a
to_nSPARQL(<, path)            = next^-1::a
to_nSPARQL(exp1 ... expn, path)       =
  to_nSPARQL(exp1, edges)/to_nSPARQL(exp2,nodes)/.../
  to_nSPARQL(expn-1,nodes)/to_nSPARQL(expn,edges)
```

nSPARQL does not support the Kleene optionality operator `?`. Nevertheless RPL expressions with `?` can be translated to nSPARQL by using the `self` axis without a node test, which has the same semantics as the empty path expression in RPL.

Note that some syntactically correct RPL expressions are not given a semantics in Section 3. Among these expressions are `Edges ^a`, `Nodes >a`, `Nodes <a` or `Path >a >b`. Similarly, these expressions are not handled by the translation function. For implementations, there are two possible ways of treating such expressions: Raising a syntax error at parse time, or evaluation to the empty relation over all possible input graphs.

Corollary 3. *Any nSPARQL path expression p_n excluding the axes node, node^{-1}, edge, and edge^{-1} can be translated to an equivalent RPE p_c of length $\mathcal{O}(|p_n|)$.*

Proof. For the translation of an nSPARQL expression including only the axes $next$, $next^{-1}$ and $self$, the expression is first normalized by inserting steps along the $self$ axis without node tests. The resulting expression e does not contain consecutive steps along the axes $next$ and $next^{-1}$, but the axis $next$ and $next^{-1}$ on the one hand and the axis $self$ on the other hand alternate. This transformation is done for both the expression itself and for any subexpression appearing within a predicate. For example the nSPARQL expression

$$\text{next}^{-1}\text{::b/next[next::a/next::b]/next}^{-1}\text{::c}$$

is normalized to

$$\text{next}^{-1}\text{::b/self/next[next::a/self/next::b]/self/next}^{-1}\text{::c} \ .$$

Obviously, this transformation preserves the semantics of the expression. Subsequently, the transformed expression is translated to RPL according to the function `to_rpl` in Listing 1.2. Obviously size of the resulting expression is linear in the size of the original.

Listing 1.2. Translation of nSPARQL to RPL.

```
to_rpl(step_1/.../step_n) = PATH to_rpl(step_1) ... to_rpl(step_n)
to_rpl(next) = >_
to_rpl(next::a) = >a
to_rpl(next^-1) = <_
to_rpl(next^-1::a) = <a
to_rpl(self) = ^_
to_rpl(sefl::a) = ^a
```

Corollary 4. *A RPE p_c in RPL$^{\rightarrow,\circ,\dashrightarrow,[],/,|,\mu}$ can be evaluated in $O(|G| \cdot |p_c|)$.*

6 Further Complexity Results

The comparison of RPL to related path query languages in the last section has already brought up some complexity results for sublanguages of RPL. In this section we establish the tractability of RPL as a whole and the intractability of RPL with unordered paths.

Theorem 1 (Tractability of RPL and nSPARQL$^{RSE,\neg}$). *RPL and the extension of nSPARQL by regular string expressions and predicate negation (denoted by nSPARQL$^{RSE,\neg}$) can be evaluated in time $O(|exp| \cdot |G|)$.*

Proof. (Sketch) Theorem 1 builds upon Corollary 4, that establishes that the evaluation of $RPL^{\rightarrow,\circ,\dashrightarrow,[],/,|,\mu}$ is in $O(|exp| \cdot |G|)$. The only features missing in $RPL^{\rightarrow,\circ,\dashrightarrow,[],/,|,\mu}$ when compared to full RPL are predicate negation ($\neg$) and regular string expressions (RSE). The evaluation of regular string expressions is linear. Thus, defining the size of an RDF graph as the total length of the characters appearing within its nodes and edges, the complexity remains in $O(|exp| \cdot |G|)$ when regular string expressions are added to the language.

Showing that predicate negation has no effect on evaluation complexity is a little more tricky: Consider the proof of the tractability of nSPARQL in [12]. It involves the construction of product automata $G \times \mathcal{A}_p$ for each predicate p appearing in the expression exp to be evaluated. We can extend nSPARQL to nSPARQL$^{\neg}$ by allowing predicate negation in the same way as RPL allows predicate negation. A RPE p_c with predicate negation can then be translated to an nSPARQL$^{\neg}$ expression p_n in linear time, such that the size of p_n remains linear in the size of p_c.

It remains to be shown that nSPARQL$^{\neg}$ is in $O(|exp| \cdot |G|)$. For this end, we adapt the algorithm $LABEL(G, exp)$ from [12] to label both positive and negative predicates appearing in exp. For each negative predicate $not(p)$ we introduce the label not_p which is attached to each node n in G *not* matching with p. Then, for each negative predicate $not(p)$ in exp, we substitute $not(p)$ in not_p, thereby obtaining an ordinary nSPARQL expression exp^+. exp^+ evaluates to over G with the adapted labelling algorithm if and only if exp evaluates to true over G with the original labelling algorithm.

Theorem 2 (NP-Completeness of $RPL^{\circ,/,\{\}}$). *The evaluation problem of $RPL^{\circ,/,\{\}}$ is NP-complete.*

Proof. Obviously the evaluation problem for $RPL^{\circ,/,\{\}}$ is in NP. We show its NP-hardness by a reduction from the directed Hamiltonian path problem. Let G be an abritrary RDF graph with nodes $\{n_1, \ldots, n_k\}$. Then G has a directed Hamiltonian path if and only if the RPE { NODES $n_1, \ldots n_k$ } has a non-empty solution over G.

Theorem 3. *The evaluation problem for $RPL^{\rightarrow,/,\{\}}$ is in $O(n \cdot \sigma^w \cdot e)$ where n is the number of nodes of the RDF graph, e the number of edges, σ the number of edge labels, and w is the length of the path expression.*

Corollary 5. *The evaluation problem for $RPL^{\rightarrow,/}$ is in $O(e \cdot w)$ where w is the length of the regular path expression and e is the number of edges in the RDF graph.*

Proof. Theorem 3 only gives an upper bound for the evaluation of $RPL^{\rightarrow,/,\{\}}$, therefore it suffices to give an algorithm that runs in $O(n \cdot \sigma^w \cdot e)$ time.

Let G be an RDF graph, and $p \in RPL^{\rightarrow,/,\{\}}$. The idea of the algorithm is to view G as a non-deterministic finite automaton, and p as a word to be checked by the automaton. p is checked from the first element to the last, and the set of valid states in the automaton is remembered in each step, starting out from the set of all nodes in the RDF graph. For $RPL^{\rightarrow,/}$ (i.e. only ordered edge-flavored expressions), this view gives us an algorithm in $O(e \cdot w)$, where e is the number of edges in G, and w is the length of p (Corollary 5).

For unordered edge-flavored path expressions, a naive implementation would compute all possible permutations, and check the RDF graph for correspondance with each of these permuations. Since there $w!$ permutations for a path of length w, this procedure has a complexity of $O(w! \cdot e)$. The following algorithm is more efficient:

Again, the RDF graph G is viewed as a finite automaton, which is traversed using symbols occurring in the path expression p. In step i of the computation, each node n in G is labelled with all paths p of length i such that n is reachable over p from some other node m in G. Initially, all nodes are labeled with the empty path ϵ. After w steps (or earlier), the algorithm terminates and exactly the set of labeled nodes in G is reachable over p. In Listing 1.3 we use set notation to represent paths, since the order of traversal is irrelevant; however we must think of paths as multisets, because the same edge label may occur multiple times in p. For this reason, the set difference operator $\setminus$ and the union operator $\cup$ in Listing 1.3 are the set difference and the union operator for *multisets*, not *sets*, respectively.

Listing 1.3. Evaluation algorithm for expressions in $RPL^{\rightarrow,/}$

```
for each node $n$ in $G$ do $labels(n) = \{\epsilon\}$ end
for $i = 1$ to $w$ do              // $w$ is the length of path $p$
  for each $e$ in $E$ do           // follow every edge
    for each $l$ in $labels(source(e))$ do
      if $label(e)$ is in $p \setminus l$ then
        $labels(sink(e))$.add($\{l\} \cup label(e)$)
      end
    end
  end
  remove all labels of length $i-1$
end
```

In the i-th iteration of the outermost loop of Listing 1.3, the set of labels for the nodes in G is bounded by $\sigma^i \cdot |n|$. Thus, the number of edge traversals in step i is bounded by $\sigma^i \cdot |n|$. The total number of edge traversals is thus $\sigma^{w+1} \cdot |n|$ (geometric series).

Theorem 4 (NP-Completeness of $RPL^{\rightarrow,/,\{\}}$). *The evaluation problem of $RPL^{\rightarrow,/,\{\}}$ is NP-complete.*

Proof. For the proof of Theorem 4 we use a reduction from the Hamiltonian Cycle Problem. The idea of the proof is illustrated in Figure 2. Let $G = (V, E)$

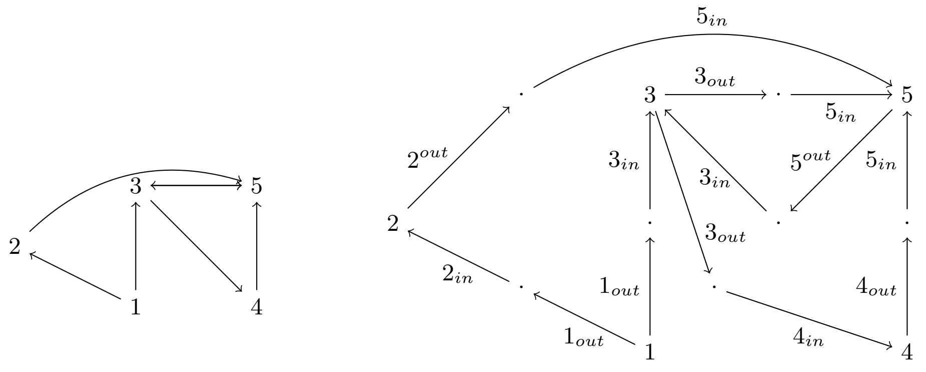
be a directed labeled graph with nodes $\{1, \ldots, k\}$. G has a Hamiltonian Cycle if and only if the RPE $\{$ EDGES $1_{in}, 1_{out}, \ldots, k_{in}, k_{out}$ $\}$ has a non-empty solution over the *edge expansion graph* of G, which is defined as follows:

Definition 10 (Edge expansion graph). *Let $G = (V, E)$ with $V = 1, \ldots, k$ be a graph. The* edge expansion graph *$F = (V', E', \mu)$ of G is an edge labeled graph with the following properties:*

- *$V \subseteq V'$*
- *For each edge $(u, v) \in E$ there is some node n in V' and edges $(u, n), (n, v) \in E'$ with $\mu(u, n) = u_{out}$ and $\mu(n, v) = v_{in}$. There are no other edges in E' involving n.*
- *These are all nodes and edges in F.*

The edge expansion graph F of a given Graph G with v vertices and e edges contains $v + e$ vertices and $2 \cdot e$ edges. Obviously, F can be constructed from G in polynomial time.

Fig. 2. Polynomial reduction from the Hamilton Cycle Problem to $RPL^{\rightarrow,/,\{\}}$ evaluation



7 Implementation of RPL by Compilation to Prolog

In this section we show how RPL can be easily and efficiently implemented by a compilation to Prolog. Before giving the translation, we first hihglight three challenges that must be met by the translation process.

1. Since regular expressions make excessive use of the kleene closure operators `+` and `*`, which must be translated to recursive rules in Prolog, non-termination

must be avoided. Non-termination of transitive closure computations in Prolog can often be resolved by term permutation in rule bodies, or clause permutation in programs. But in the presence of cyclic data, transitive closure computations may still not terminate, due to infinitely many paths between nodes. There are two ways of dealing with this issue: (a) keeping track of the path that is traversed during the computation of the transitive closure or (b) tabling of the predicates that are used for transitive closure computation. Solution (b) eases the translation process, but requires evaluation by a Prolog engine that supports tabling such as XSB.

2. A second challenge in the translation process arises from the use of regular string expressions in RPL. This challenge is most easily met by translation to a Prolog engine that takes care of regular expression matchin such as Ciao Prolog or XSB Prolog.
3. A third challenge arises from the fact that most Prolog engines are not prepared to dealing with RDF data, with the notable exception of SWI Prolog. Again, there are (at least) two solutions for dealing with this situation: (a) use SWI Prolog or (b) assume that RDF graphs are encoded as ternary terms with some distinguished predicate name such as `triple`. This assumption is not as farfetched as it might seem, since the well-known N-Triples serialization of RDF is simply a collection of triples, and can be easily imported into Prolog engines. Moreover, there are conversion utilities from RDF/XML, Notation3 or Turtle to N-Triples.

We deal with the issues 1 and 2 by a translation to XSB Prolog and resolve issue 3 by assuming a native Prolog encoding of RDF graphs as ternary atoms with predicate name `triple`.

Let u be a URI, $pr_1, \ldots$ RPL predicates, $a_1, \ldots$ adorned or disjunctive RPEs, and c a concatenated or predicated RPE. The translation of RPL expressions to Prolog is given by the following function *to_prolog*. Each translation rule yields at least one Prolog rule, and may recursively call other translation rules. The predicate name of the head of the rule to be generated is given as an argument to the translation function.

$$to\_prolog(flavor\ c, p) = \begin{cases} edges(c,p) & \text{if } flavor = \text{EDGES.} \\ path(c,p) & \text{if } flavor = \text{PATH.} \\ nodes(c,p) & \text{if } flavor = \text{NODES.} \end{cases} \tag{38}$$

$$edges(u,p) = \text{p(X,Y) :- triple(X, } u\text{, Y).} \tag{39}$$

$$edges({<}u,p) = p\text{(X,Y) :- } p_1\text{(Y, X).} \quad R \tag{40}$$

with p_1 a fresh predicate name and $edges(u, p_1) = R$. Predicated RPL expressions are translated by Equations 41 and 42.

$$\begin{aligned} &edges(u[pr_1] \ldots [pr_n], p) = \\ &\qquad p\text{(X,Y) :- triple(X,}u\text{,Y), } p_1(u,\_), \ldots, p_n(u,\_). \quad R_1 \ldots R_n \end{aligned} \tag{41}$$

with p_i fresh predicate names and $edges(pr_i, p_1) = R_i$ for $1 \leq i \leq n$. The corresponding rule for $<u$ is obtained by switching X and Y in the term `triple(X,`u`,Y)`.

$$edges(\_[pr_1] \ldots [pr_n], p) = \qquad (42)$$
$$p(\text{X,Y}) \text{ :- triple(X,P,Y), } p_1(\text{P,\_}), \ldots, p_n(\text{P,\_}). \quad R_1 \ldots R_n$$

with p_i fresh predicate names and $to_prolog(pr_i, p_i) = R_i$ for $1 \leq i \leq n$. The corresponding rule for a regular expression re instead of an underscore is obtained by inserting the term `re_match(`re`, P, _, _, _)` after the term `triple(X, P, Y)` into the rule defining p. Note that this translation only works for XSB Prolog when the module `regmatch` is included. Again, the corresponding rules for the reverse edges $<$ _ or $<$ re is obtained by switching X and Y in the term `triple(X,`u`,Y)`.

$$nodes(u[pr_1] \ldots [pr_n], p) = \qquad (43)$$
$$p(\text{u,u}) \text{ :- node(u), } p_1(u,\_), \ldots, p_n(u,\_). \quad R_1 \ldots R_n$$

with $p_1, \ldots, p_n$ fresh predicate names, and $to\_prolog(pr_i) = R_i$ for $1 \leq i \leq n$. Wildcards and literals at the place of u are translated in the same way. A regular expression re at the place of u requires binding the node n to a variable, and testing if n is in the language defined by re with the XSB predicate `re_match` as follows:

$$nodes(u[pr_1] \ldots [pr_n], p) = \qquad (44)$$
$$p(\text{P,P}) \text{ :- node(P), re\_match(}re\text{, P, \_, \_, \_), } p_1(P,\_), \ldots, p_n(P,\_). \quad R_1 \ldots R_n$$

$$edges(c?, p) = p(\text{X,X}) \text{ :- node(X).} \quad p(\text{X,Y}) \text{ :- } p_1(\text{X,Y}).\ R \qquad (45)$$

with p_1 a fresh predicate name and $edges(c, p_1) = R$, and the predicate `node` defined by the following rules:

$$\text{node(X) :- triple(X, \_, \_).} \quad \text{node(X) :- triple(\_, \_, X).} \qquad (46)$$

$$edges(c+, p) = p(\text{X,Y}) \text{ :- } p_1(\text{X,Y}). \quad p(\text{X,Y}) \text{ :- } p_1(\text{X,Z}), p(\text{Z,Y}). \quad R \qquad (47)$$

with p_1 a fresh predicate name and $edges(c, p_1) = R$. The Kleene star operator `*` is translated in a very similar fashion.

$$edges((a_1 \ldots a_n), p) = \qquad (48)$$
$$p(\text{X,Y}) \text{ :- } p_1(\text{Z}_0\text{,Z}_1), \ldots, p_n(\text{Z}_{n-1}\text{,Z}_n). \quad R_1 \ldots R_n$$

with $p_1, \dots p_n$ fresh predicate names and $edges(a_i, p_i) = R_i$ for $1 \le i \le n$.

$$nodes((a_1 \dots a_n), p) = \begin{array}{l} p(\mathrm{X,Y}) \text{ :- } p_1(\mathrm{Z_0,Z_1}),\ \mathrm{triple(Z_1,\ \_,\ Z_2)},\ \dots, \\ \quad \mathrm{triple(Z_{2n-2},\ \_,\ Z_{2n-1})},\ p_n(\mathrm{Z_{2n-1},Z_{2n}}). \\ R_1\ \dots\ R_n \end{array} \tag{49}$$

with $p_1, \dots p_n$ fresh predicate names and $nodes(a_i, p_i) = R_i$ for $1 \le i \le n$.

$$\begin{array}{l} path((a_1 \dots a_n), p) = \\ \quad p(\mathrm{X,Y}) \text{ :- } p_1(\mathrm{Z_0,Z_1}),\ \dots,\ p_n(\mathrm{Z_{n-1},Z_n}).\quad R_1\ \dots\ R_n \end{array} \tag{50}$$

with $p_1, \dots p_n$ fresh predicate names and $edges(a_i, p_i) = R_i$ for odd i and $nodes(a_i, p_i) = R_i$ for even i in $\{1, \dots, n\}$.

$$\begin{array}{l} edges((c_1 \mid \dots \mid c_n), p) = \\ \quad p(\mathrm{X,Y}) \text{ :- } p_1(\mathrm{X,Y}).\quad \dots\quad p(\mathrm{X,Y}) \text{ :- } p_n(\mathrm{X,Y}).\quad R_1\ \dots\ R_n \end{array} \tag{51}$$

with $p_1, \dots p_n$ fresh predicate names and $edges(c_i, p_i) = R_i$ for $1 \le i \le n$. Disjunctive RPEs within edge- and path-flavored RPEs are translated in exactly the same way.

$$to\_prolog(not(f), p) = p(\mathrm{X,Y}) \text{ :- not } p_1(\mathrm{X,Y}).\quad \mathrm{R} \tag{52}$$

where p_1 is a fresh predicate name and $to\_prolog(f, p_1) = R$.

8 Conclusion and Future Work

This paper describes the novel RDF path language *RPL*. *RPL* is one of the few RDF path languages with a formal semantics. Compared to other query languages it omits features that are rarely used, but includes features such as regular string expressions, direction modifiers, and predicate negation, that may turn out to be extremely useful for query authors. *RPL* can be evaluated efficiently, but extensions of *RPL* with unordered paths or variables cannot. *RPL* is currently being implemented. Future work includes the experimental affirmation of the tractability of *RPE* evaluation.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

2. Renzo Angles, Claudio Gutierrez, and Jonathan Hayes. RDF query languages need support for graph properties. Technical report, 2004.
3. François Bry, Tim Furche, Clemens Ley, and Benedikt Linse. RDFLog—taming existence - a logic-based query language for RDF, 2007.
4. François Bry, Tim Furche, Clemens Ley, Benedikt Linse, and Bruno Marnette. RDFLog: It's like datalog for RDF. In *Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden (30th September–1st October 2008)*, 2008.
5. François Bry, Tim Furche, Benedikt Linse, and Alexander Pohl. XcerptRDF: A pattern-based answer to the versatile web challenge. In *Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden, Germany (30th September–1st October 2008)*, pages 27–36, 2008.
6. S. Fortune, J.E. Hopcroft, and J.C. Wyllie. The Directed Subgraph Homeomorphism Problem. 1978.
7. Krys Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2007.
8. As La Paugh and Ch Papadimitrou. The even-path problem for graphs and digraphs. *Networks(New York, NY)*, 14(4):507–513, 1984.
9. M. Marx. Conditional XPath. *ACM Transactions on Database Systems (TODS)*, 30(4):929–959, 2005.
10. A.O. Mendelzon and P.T. Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24:1235, 1995.
11. Chimezie Ogbuji. Versa: Path-based RDF query language, 2005.
12. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. In Amit P. Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy W. Finin, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2008.
13. Alexander Pohl. RDF Querying in Xcerpt: Language Constructs and Implementation. Deliverable I4-Dx2, REWERSE, 2008.
14. Andy Seaborne and Eric Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.