A RPL through RDF: Expressive Navigation in RDF Graphs

Harald Zauner[1], Benedikt Linse[1,2], Tim Furche[1,3], and François Bry[1]

[1] Institute for Informatics, University of Munich,
Oettingenstraße 67, 80538 München, Germany
[2] Thomson Reuters, Landsberger Straße 191a, 80687 München, Germany
[3] Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England

http://rpl.pms.ifi.lmu.de/

Abstract. RPL (pronounced "ripple") is the most expressive path language for navigating in RDF graphs proposed to date that can still be evaluated with polynomial combined complexity. RPL is a lean language well-suited for integration into RDF rule languages. This integration enables a limited form of recursion for traversing RDF paths of unknown length at almost no additional cost over conjunctive triple patterns.
We demonstrate the power, ease, and efficiency of RPL with two applications on top of the RPL Web interface. The demonstrator implements RPL by transformation to extended nested regular expressions (NREs). For these extended NREs we have implemented an evaluation algorithm with polynomial data complexity. To the best of our knowledge, this demo is the first implementation of NREs (or similarly expressive RDF path languages) with this complexity.

1 Motivation

With the promise of exciting "new kinds of usage scenarios", you finally got your boss at company C to embrace linked data and connect your community forum and contact database to other online communities and FOAF profiles of your contacts. Your boss now wants to put that technology to use: *"I want to cooperate with X on topic Y! Can you get me the name of any person that works at X and that's connected to us via people that are also interested in Y (so that they have an interest in connecting us). Oh, and none of the intermediates should be our competitor Z."*

Though the linked data movement and related initiatives like FOAF or SIOC provide specifically for this kind of scenario, most current analysis and query tools for RDF are not up to this task: SPARQL can only compute persons connected via fixed length paths due to the lack of any form of recursion. Under an (e.g., OWL-based) entailment regime that treats foaf:knows (the FOAF property used to build social networks) as a transitive property, SPARQL can compute all connected persons, but can not ensure that all intermediate persons share the same interest. The recent extension of SPARQL with property paths (to be incorporated into SPARQL 1.1) also fails at this task, as it only allows local restrictions on the traversed edges, but not on the traversed nodes, and no repetition ($*$) over paths with restrictions on nodes *and* edges.

Not only SPARQL fails at such analysis tasks on social networks or similarly interlinked data: Among the many RDF query and rule languages surveyed in [5], there is no language that can solve this analysis task and is not either impractical for large datasets (as NP- or Turing-complete languages such as SPARQLeR and TRIPLE) or only informally specified and now abandoned (as Versa). The more recent nSPARQL [8], an extension of SPARQL with nested regular expressions, can only solve parts of the above analysis task (but not the last sentence) and is not implemented in any publicly available tool.

In this demonstration, we show how to solve this and similar analysis tasks with a novel RDF path language, called RPL. The following RPL path expression solves our analysis problem (f being the FOAF namespace):

```
PATH [PATH _ <f:member C]
     (>f:knows [!PATH _ <f:member Z][PATH _ >f:interest _ >f:topic Y])*
      >f:knows [PATH _ <f:member X]
```

It returns all pairs of nodes such that the first node is an f:member of C and is connected to the second node via the specified path: It first traverses *outgoing* (indicated by >) f:knows edges and nodes that (1) are not members (employees) of Z and (2) have an f:interest edge that leads to some node that has Y as f:topic. It traverses arbitrarily many such edge-node pairs (indicated by *). The last node must also have an *incoming* (indicated by <) f:member edge from X.

Solving this analysis task in RPL is also *efficient* even on large RDF graphs: The demonstrated RPL implementation is, to the best of our knowledge, the first implementation of the bottom-up labeling algorithm for nested regular expressions from [8] on RDF data. It extends both nested regular expressions and the labeling algorithm with several important analysis features such as negation and regular expressions on literals and URIs. The extended labeling algorithm has been shown in [3] to have polynomial combined and data complexity.

With the analysis task solved, your job is save, your boss is impressed, and the Semantic Web vision is closer to reality.

2 A RPL through RDF Graphs

RPL is inspired by XPath, the dominant XML path language, in that it allows *nested predicates* on paths. Predicates allow a RPL user to express, in addition to *local* conditions on the path between two nodes, also *non-local* conditions on branches starting at a node on the path. Like XPath, RPL does not allow variables in path expressions and thus, all RPL queries are tree queries. RPL adapts XPath style path navigation to RDF and goes beyond XPath by replacing XPath's fixed closure axis with closure operators **?**, *, and **+** (as in [1] and [7]). RPL is set apart by three properties:

(1) RPL is designed from start off to be easily integrated into RDF rule and query languages such as XCERPT$^{\text{RDF}}$ [4] by allowing RPL expressions to appear in place of RDF predicates. Thus, a RPL expression e evaluates to a set of node pairs such that, between each pair of nodes, there is a path that matches e.

Together with the omission of variables, this ensures polynomial time and space data complexity (and polynomial combined) for RPL. This contrasts to most RDF path languages that either *(a)* allow the extraction of entire paths from RDF graphs (like SPARQLeR [6]) and have therefore exponential data complexity, or *(b)* allow variables to bind with nodes on the path (like PSPARQL [2]) again at the price of exponential combined complexity.

(2) RPL's rich syntax scales with the needs and abilities of the user: Beginners can describe paths only through the traversed edges (*edge-flavored* RPL) or only through the traversed nodes (*node-flavored* RPL), advanced users can place restrictions on both nodes and edges (*path-flavored* RPL). Together with RPL's predicates this is the main difference to the SPARQL property path extension that only allows restrictions on the edges traversed by a path.

(3) Expressive label tests with regular expressions and predicates with negation push RPL to the limits of RDF path languages with polynomial data complexity. E.g., the edge-flavored expression `EDGES >/.*train.*/*` traverses an arbitrary path whose forward-directed edges contain the keyword "train".

Predicates allow non-local restrictions and are arbitrary RPL expressions in square brackets. They can have either positive or negative sign (denoted by `!`).

3 System Description

Syntax and Semantics of RPL: To give a solid foundation for the discussion of the RPL implementation, we first briefly sketch its syntax:

⟨*rpl-expr*⟩ ::= ⟨*flavor*⟩ ⟨*adorned*⟩+
⟨*flavor*⟩ ::= '`EDGES`' | '`NODES`' | '`NODES<`' | '`NODES>`' | '`PATH`'
⟨*adorned*⟩ ::= (⟨*directed*⟩ | '`(`' ⟨*disjunctive*⟩ '`)`') ('`?`' | '`*`' | '`+`')?
⟨*directed*⟩ ::= ('`<`' | '`>`')? (⟨*labeltest*⟩ | ⟨*predicates*⟩)
⟨*disjunctive*⟩ ::= ⟨*adorned*⟩+ ('`|`' ⟨*adorned*⟩+)*
⟨*predicates*⟩ ::= '`[`' '`!`'? ⟨*rpl-expr*⟩ ('`][`' '`!`'? ⟨*rpl-expr*⟩)* '`]`'
⟨*labeltest*⟩ ::= '`_`' | ⟨*LITERAL*⟩ | ⟨*IRI_REF*⟩ | ⟨*REGEXP*⟩
| ⟨*PN_PREFIX*⟩? '`:`' (⟨*PN_LOCAL*⟩ | ⟨*REGEXP*⟩)?

⟨*adorned*⟩ expressions form expression sequences and can be adorned by multiplicities, ⟨*directed*⟩ expressions correspond to XPath node tests. We borrow most of the token classes from SPARQL (e.g. ⟨*PN_PREFIX*⟩ and ⟨*PN_LOCAL*⟩), but also allow regular expressions (enclosed in slashes) via ⟨*REGEXP*⟩.

The semantics of RPL is specified by translation into ENREs, an extended version of nSPARQL's nested regular expressions (NREs) [8]. We have chosen this semantics to closely reflect our implementation that also translates RPL expressions into ENREs and gives the, to the best of our knowledge, first implementation of (E)NREs on RDF data.

Consider, e.g., the RPL expression `PATH` p `(>`t `_)+` that returns pairs of p together with any node reached from p over one or more intermediate nodes via t edges. This expression is translated into the ENRE

```
self_node::p/(next::t/self_node)+
```

ENREs have been tailored to fit the peculiarities of RPL and extend NREs from [8] by regular expressions for label tests, the negation of nested expressions (indicated by `!`), and three new navigation axes: `self_node` and `self_edge` act like `self` but only on nodes and edges, respectively; `next_or_next`$^{-1}$ is used for edges, where no direction is specified. Negation of nested expressions is realized as complement relation (i.e. $[\![\texttt{[!}exp\texttt{]}]\!]_G := \overline{[\![\texttt{[}exp\texttt{]}]\!]_G}$) and thus does not affect the polynomial complexity bounds.

Implementation: In the course of the demonstration, we demonstrate both RPL and how RPL queries are transformed into ENREs.

RPL is implemented in Java and uses Sesame 2.2.4 for accessing RDF data. We use the event based RDFHandler interface for fast and space efficient parsing of RDF triples into an in-memory graph representation (we choose this as it is an open question whether the bottom-up labeling algorithm used for evaluating ENREs and thus RPL can be implemented on a stream of RDF triples).

Through a number of normalization and verification steps, RPL queries are transformed into an equivalent ENRE. This ENRE is evaluated by an extended version of the bottom-up graph labeling algorithm from [8]. The algorithm recursively labels every node and edge v of the RDF graph with all nested expressions that v satisfies (i.e., there is a path beginning at v which satisfies the nested expression). The result is a product automaton $\mathcal{P} := \mathcal{G} \times \mathcal{A}$, where $\mathcal{G}$ is the RDF graph seen as an NFA (each node and each edge of the RDF graph is both an initial and final state, and the transitions are given by its triples), and $\mathcal{A}$ is the NFA that is induced by the ENRE seen as regular expression (Thompson's construction). $\mathcal{P}$ is used in a second phase to compute all node pairs (a, b), such that $(a, \cdot)$ is an initial state from which a final state $(b, \cdot)$ is reached in $\mathcal{P}$.

4 Riding RPL (Demo Description)

We demonstrate RPL using a Web-based, interactive interface: it provides a set of predefined application scenarios for a quick take-up of RPL. The application scenarios include the queried RDF graph, a visualization of that data, as well as a number of predefined queries that illustrate the strengths of RPL for analysing the respective data. The interface also allows users to enter their own RDF data (in any of the common RDF serializations Turtle, RDF/XML, N-Triples and N3) and their own RPL queries.

Furthermore, RPL queries can be comfortably authored in two separate Eclipse plugins: *visRPL* allows users to graphically compose RPL queries (see Figure 1) for new users, and another textual editor offers syntax highlighting and completion for more experienced users of RPL.

Application Scenario: Transportation Services. This application scenario is based on the nSPARQL [8] transportation services example. Both the data and the queries discussed here are available to the user as part of the Web-based interface. Figure 2 gives an impression of the data used in this scenario (we abbreviate rdfs:subPropertyOf by rdfs:sp). The left hand of the graph shows

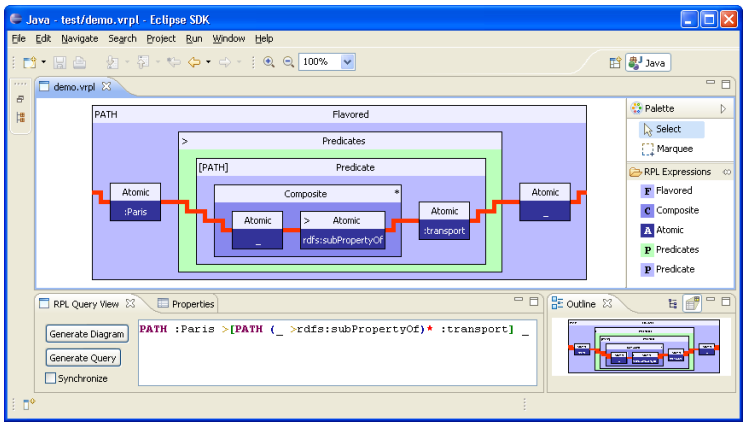

Fig. 1. Visual Eclipse editor visRPL for RPL

a simple ontology for various connection types between cities, the right hand shows connections between a few European cities.

Imagine you are in Paris and want to find out which cities can be *directly* reached via any transportation service. You might start with a RPL query like `NODES> :Paris _` (or equivalent `PATH :Paris >_ _`), which will return all nodes that :Paris has an outgoing edge to. Hence, the pair (:Paris, :France) will also be part of the result set (due to the :country edge between them). To ensure that only transport edges are followed, we might be tempted to enumerate all types (train, bus, ferry) of transport edges in our data. Not only is such a solution clumsy, it also forces us to change our query whenever new types of transport edges are introduced. Thus, we use instead the RPL predicate that is shown in Figure 1. The query `PATH :Paris >[PATH (_ >rdfs:subPropertyOf)* :transport] _` specifies that we only follow edges from :Paris that are labeled as :transport or any of its sub-properties.

Once we submit this query, an AJAX request is sent to the RPL Web service which evaluates the RPL query and returns *(a)* a simplified and normalized, i.e. path-flavored RPL query, *(b)* an equivalent ENRE, and *(c)* the result of the evaluation (or an error message). The Web interface displays this information together with timing information in a dialog as shown in Figure 3.

It turns out that none of the directly reachable cities interests us today. Thus, we decide that intermediate stops are acceptable and want to adapt our query accordingly. We simply have to add a closure multiplicity (`+`) to get to any place that is reachable by one or more transport edges from :Paris, see Figure 3.

Unfortunately, we get really sick when traveling with a ferry and thus would like to exclude connections that use a ferry. Again, the modification is straight-

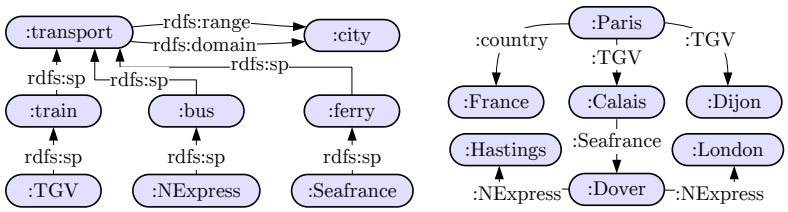

Fig. 2. Transportation services RDF graph

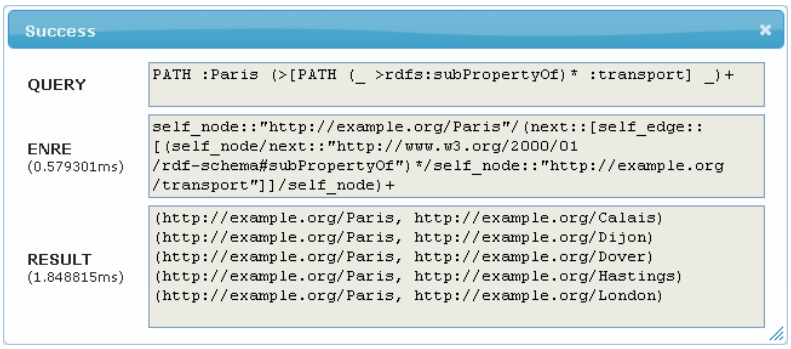

Fig. 3. Result dialog: RPL query, ENRE, result

forward: we add another predicate (`!` indicates negation) that does not allow :ferry edges (and their sub-properties) to be traversed. The adapted query is

```
PATH :Paris (>[PATH (_ >rdfs:subPropertyOf)* :transport][
             !PATH (_ >rdfs:subPropertyOf)* :ferry] _)+
```

It evaluates to the set $\{(:\mathsf{Paris}, :\mathsf{Calais}), (:\mathsf{Paris}, :\mathsf{Dijon})\}$. The cities :Dover, as well as :London and :Hastings (which are only reachable over :Dover), are excluded now as the only transport link from Calais to Dover is a kind of ferry.

Finally, RPL is also able to express a relevant part of the RDFS entailment rules: the following query retrieves all nodes that "are" cities according to RDFS (i.e. all nodes that have an rdf:type edge to :city in the RDFS closure of Fig. 2).

```
NODES ( [PATH _ >rdf:type (_ >rdfs:sc)* :city]
      | [EDGES >[PATH (_ >rdfs:sp)* _ >rdfs:domain (_ >rdfs:sc)* :city]]
      | [EDGES <[PATH (_ >rdfs:sp)* _ >rdfs:range (_ >rdfs:sc)* :city]])
```

Acknowledgements. The research leading to these results has received funding from the *European Research Council* under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 246858.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1), 1997.
2. F. Alkhateeba, J.-F. Baget, and J. Euzenat. Extending SPARQL with regular expression patterns. *J. of Web Semantics*, 2009.
3. F. Bry, T. Furche, and B. Linse. The perfect match: RPL and RDF rule languages. In *RR*, 2009.
4. F. Bry, T. Furche, B. Linse, A. Pohl, A. Weinzierl, and O. Yestekhina. Four lessons in versatility or how query languages adapt to the web. In F. Bry and J. Maluszynski. *Semantic Techniques for the Web, The Rewerse Perspective*, LNCS 5500, 2009.
5. T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF querying: Lang. constructs and eval. methods compared. In *Reasoning Web*, LNCS 4126. 2006.
6. K. Kochut and M. Janik. SPARQLeR: Extended SPARQL for semantic association discovery. In *ESWC*, 2007.
7. M. Marx. Conditional XPath. *ACM Trans. on Database Sys. (TODS)*, 30(4), 2005.
8. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. In *ISWC*, 2008.