INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

RPL ready to use: An embeddable and expressive, yet efficient RDF Path Language

Harald Zauner

AUFGABENSTELLER	Prof. Dr. François Bry
BETREUER	Dr. Benedikt Linse Dr. Tim Furche
ABGABETERMIN	14. März 2010

ERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 14. März 2010

Harald Zauner

ABSTRACT

SPARQL is so far the only standardized RDF query language. Surprisingly, it is designed in the way of relational languages like SQL, and pays only little attention to the graph-based data model of RDF.

RPL (RDF Path Language) is a novel and expressive, yet efficient RDF query language intended to close this gap. It is inspired from XPath, the dominant XML path language, and from the nested regular expressions of nSPARQL, a navigational language for RDF. RPL is set apart from the latter in that (1) it is designed from the start to be easily integrated into host languages such as SPARQL, (2) allows for (even negated) predicates that express non-local conditions (i.e. conditions on branches rooted at a node or an edge of the parent path), and (3) provides expressive label tests via regular expressions.

After presenting the syntax and set-based semantics, this thesis elaborates on two approaches for evaluating RPL: A first approach is based on translating RPL into a dialect of nSPARQL's nested regular expressions (as a side effect, an implementation of these is also provided) and is shown to have quadratic data and linear query time complexity. The second approach relies on linking each subexpression e of a RPL query with a boolean matrix, holding (the endpoints of) all paths satisfying e. Although faster on small datasets and in embedded mode (under certain conditions), this approach is shown to also have linear query, but cubic data time complexity. The complexity bounds of both approaches are experimentally verified.

In addition to the core implementation, an entire tool set for working with RPL is presented in this thesis:

First, as an *embeddable language*, RPL can be easily integrated into SPARQL at predicate position and used to imitate the essential core of the RDFS semantics. In this context, allowing RPL to use SPARQL variables is a natural extension.

Second, *RPLgen* – an RDF data generator based on RPL queries – applies RPL as a *generative language* to create richly structured, path-based RDF data, as already done during the experimental evaluation.

Third, *visRPL* illustrates that RPL can also be seen as a *visual language*. visRPL is an Eclipse plugin aiming to ease the authoring of RPL queries by visually composing them (and, at the same time, learning RPL's textual syntax by a two-way synchronization with a textual view of the query).

In summary, its efficient and stable implementation together with its entire tool set make RPL ready for use – even in real-world applications.

ZUSAMMENFASSUNG

SPARQL ist die bisher einzige, standardisierte Anfragesprache für RDF. Seltsamerweise ist diese jedoch im Stil von relationalen Sprachen wie SQL gehalten, berücksichtigt also kaum das graph-basierte Datenmodell von RDF.

RPL (RDF Path Language) ist eine neuartige, ausdrucksstarke, und doch effiziente Pfadanfragesprache für RDF. Es orientiert sich an XPath, der dominierenden Anfragesprache für XML, und nSPARQL's nested regular expressions. Von Letzteren grenzt sich RPL ab, indem es (1) von Anfang an als einfach einzubettende Sprache, in Hostsprachen wie z.B. SPARQL, konzipiert wurde, (2) Prädikate (sogar in negierter Form) erlaubt, welche nicht-lokale Bedingungen ausdrücken (d.h. Bedingungen über Äste, die an einem Knoten oder einer Kante des Elternpfades angreifen), und (3) ausdrucksstarke Knoten- und Kantentests mit Hilfe von regulären Ausdrücken ermöglicht.

Im Anschluß an die Syntax und mengenbasierte Semantik stellt diese Arbeit zwei Ansätze für die Auswertung von RPL vor. Ein erster Ansatz besteht darin, RPL in einen Dialekt von nSPARQL's nested regular expressions zu übersetzen (als Nebeneffekt entsteht dabei eine Implementierung derselben), welcher zu einer in der Größe der Daten quadratischen, und in der Größe der Anfrage linearen Laufzeit führt. Der zweite Ansatz verknüpft jeden Teilausdruck e einer RPL Anfrage mit einer booleschen Matrix, die (die Endpunkte) aller e erfüllenden Pfade enthält. Obgleich schneller auf kleinen Daten, ist die Laufzeit dieses Ansatzes ebenfalls linear in der Größe der Anfrage, jedoch kubisch in der Größe der Daten. Beide Laufzeitschranken werden experimentell bestätigt.

Neben der reinen Implementierung umfasst diese Arbeit eine Reihe von Werkzeugen für den Einsatz von RPL:

Erstens kann RPL als *einbettbare Sprache* leicht an Prädikatsposition in SPARQL eingebunden werden, und dabei die RDFS Semantik imitieren.

Zweitens dient RPL als *generative Sprache* innerhalb des Datengenerators *RPLgen*. Dieser ist in der Lage, stark strukturierte, pfadbasierte RDF Daten zu erzeugen – wie bereits in der experimentellen Auswertung erfolgt.

Drittens erlaubt *visRPL*, ein Eclipse Plugin, das graphische Zusammensetzen von RPL Anfragen, setzt RPL folglich als *visuelle Sprache* ein. Zur gleichen Zeit unterstützt visRPL das Erlernen der textuellen Syntax, denn die graphische und textuelle Ansicht können synchronisiert werden – in beide Richtungen.

Zusammenfassend ist RPL durch seine effiziente und stabile Implementierung, zusammen mit seinen Werkzeugen, bereit für den Einsatz in realen Anwendungen.

ACKNOWLEDGMENTS

I would like to thank Dr. Benedikt Linse for a firm foundation of RPL and his, in every sense truly excellent, supervision of this thesis, even after having finished his doctorate studies.

I am also grateful to Dr. Tim Furche who, like Benedikt, subjected this thesis to close scrutiny and helped much to improve its quality. In numerous discussions, he was an inexhaustible source of inspiration for me.

The same holds true for Prof. Dr. François Bry. Moreover, I am indebted to him for offering the topic of this thesis, and to his entire research unit for providing me a comfortable work environment.

Finally, I would like to thank my family for their understanding and continuous support.

CONTENTS

1

INTRODUCTION

1.1 THE SEMANTIC WEB VISION

Tim Berners-Lee, the inventor of the World Wide Web, expressed his vision of the Semantic Web as follows.

> I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize.
>
> Tim Berners-Lee, 1999

Since 1999, the Semantic Web has continuously gained momentum, so part of this vision has already become reality. The development of the Semantic Web follows a layered approach [4], with each layer (except the lowest) building on top of another. Figure 1 shows a current view of the architecture of such a Semantic Web[1]. Though many of its (upper) components are yet to be developed, it shows the prominent role that RDF, its semantic extension RDFS, and its query language SPARQL play in this vision.

1 see http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#%2824%29

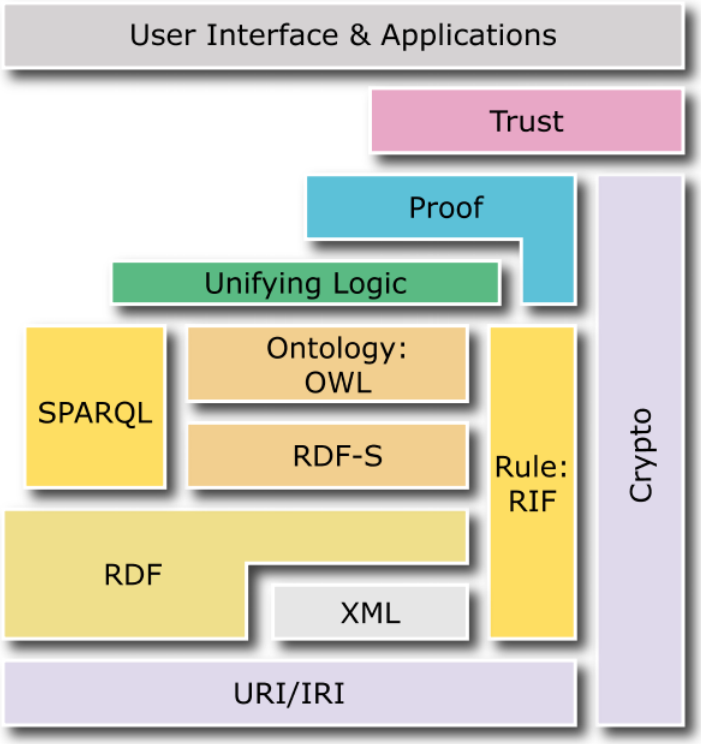


Figure 1: The Semantic Web Stack

1.2 MOTIVATION

As we have seen in the previous section, the Semantic Web Vision is still quite far from being reality. *RPL* (pronounced "ripple"), a novel RDF path language, might help us to speed up this process. Consider for example an application aiming at discovering potential cross-university research partners, that are also interested in Semantic Web topics. This information might lead to new research partnerships, from which the Semantic Web is likely to benefit. More precisely, we want to solve the following analysis task with RPL.

> Retrieve all people from university U, that (directly or indirectly) know people at a university different from U, who are interested in the Semantic Web (i.e. they are interested in documents whose topic contains the string "Semantic Web"). The retrieved connections have to be materialized by using a `:potentialResearchPartner` predicate for further processing.

For this scenario, it is assumed that the FOAF data of the considered universities and people has already been aggregated. Though the promise of linked data and projects like FOAF specifically provide for this kind of scenario, most current analysis and querying tools for RDF are not up to this task. SPARQL fails as it can only query persons that are connected by a path of fixed length. However, when using SPARQL on top of an entailment regime that treats `foaf:knows` as a transitive predicate, SPARQL can solve this task — but just as long as we do not have to add further restrictions on the intermediate persons, e.g. that all of them should be computer scientists.

Not only SPARQL fails at such analysis tasks: there is no language among the RDF rule and query languages surveyed in [15] that can solve this analysis task and is not either impractical for large datasets (as NP- or Turing-complete languages as SPARQLeR [22] and TRIPLE) or only informally specified and now abandoned (as Versa). The more recent nSPARQL [27], an extension of SPARQL with nested regular expressions, can only solve a part of the analysis task (it does not support negation and thus cannot express the "at a university different from U" part).

```
1  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2  PREFIX : <http://example.org/>
3  CONSTRUCT { ?x :potentialResearchPartner ?y } WHERE {
4  ?x [
5      PATH [PATH _ <foaf:member U]
6           (>foaf:knows _)* >foaf:knows
7           [PATH _ >foaf:interest _ >foaf:topic /.*Semantic Web.*/][
8           !PATH _ <foaf:member U]
9     ] ?y
10 }
```

Listing 1: Discovering potential research partners with RPL

Listing 1 shows a SPARQL query that embeds a RPL query (lines 5 – 8) at predicate position within its single triple pattern (lines 4 – 9). This SPARQL query will finally solve our analysis problem.

The RPL query (lines 5 – 8) returns all pairs of nodes (x, y) such that the first node x is a `foaf:member` of U (line 5) and is connected to the second node y via the specified path. Starting from x, an *outgoing* (indicated by `>` in line 6) edge labeled with `foaf:knows` is traversed, leading to an arbitrary node (indicated by `_` in line 6). Arbitrarily many such edge-node pairs can be traversed (indicated

by `(...)*` in line 6), followed by another outgoing `foaf:knows` edge leading to the second node y, which has to satisfy the following two restrictions.

1. It must have a `foaf:interest` edge that leads to some node, whose `foaf:topic` contains the string "Semantic Web" (in line 7, `/.*Semantic Web.*/` is a regular expression over ordinary strings, enclosed in slashes)

2. It must not be a `foaf:member` of university U (this negation is indicated by `!` in line 8)

Once the RPL query (lines 5 – 8) is evaluated, the first node of each result pair is bound to the SPARQL variable `?x` in line 4, and the second node is bound to the SPARQL variable `?y` in line 9. Finally, `?x` and `?y` are used within the SPARQL **`CONSTRUCT`** query form and connected to each other via a `:potentialResearchPartner` predicate (line 3), as desired.

Solving this analysis task in RPL is also *efficient* even on large RDF graphs. One of the two demonstrated evaluation approaches for RPL is the first implementation of the bottom-up labeling algorithm for nested regular expressions [27] on RDF data. RPL extends both nested regular expressions and the labeling algorithm with several important analysis features such as negation and regular expressions over strings to match node and edge labels. This extended labeling algorithm is shown to have linear query and quadratic data time complexity.

1.3 CONTRIBUTIONS

This thesis is organised along the following contributions:

1. After introducing RPL via some sample queries, Chapter 2 presents RPL's concrete and abstract syntax. The latter is also compared to a former version of RPL's abstract syntax from [9]. RPL's concrete syntax can be expressed in $LL(1)$ normal form and serves as input for a parser based on JavaCC.

   Chapter 2 ends with a short introduction into RDF and an excursus on (the implementation of) regular expressions over ordinary strings.

2. A major part of this thesis, namely the implementation of RPL, is covered in Chapter 3. The first contribution is concerned with the syntactic and semantic analysis, which precedes the actual evaluation of RPL queries. During syntactic analysis, the various flavors of RPL are converted into

path-flavored RPL queries, which have proven to easily incorporate edge- and the various node-flavored expressions. The set-based semantics of path-flavored RPL expressions q is finally given on the basis of q's abstract syntax tree, which has been enriched during semantic analysis (e.g. resolving prefixes).

Translating RPL expressions into ENREs (Extended Nested Regular Expressions, an extension of the NREs from nSPARQL [27]) is presented as the first of two evaluation approaches. Therefore, the syntax and semantics of ENREs are defined and the latter is proved to coincide with the semantics of RPL expressions (this also shows the correctness of the translation function). In this context, it is also shown that both semantics indeed just return a set of node pairs of an RDF graph, as desired in [9]. The graph labeling algorithm for NREs (Nested Regular Expressions) [27] is realized via Tarjan's algorithm (Section A.2), after it has been extended to allow for ENREs' additional axes `self_node`, `self_edge`, and `next_or_next`$^{-1}$, for regular expressions as node and edge label tests, and for the negation of nested expressions. Finally, it is shown that the time complexity for constructing the result does not decline compared to the one of NREs — its combined complexity stays quadratic in the size of the RDF graph and linear in the size of the RPL query (both sizes are measured in characters). As a side effect of this approach, the first fully functional, freely available NRE implementation is also provided with this thesis.

The second evaluation approach is denoted as Path-based evaluation, and relies on the notion of partial path matrices, together with their concatenate and union operations. Each subexpression of a RPL query is associated with a partial path matrix containing (the endpoints of) all paths satisfying it. Two variants of this evaluation approach are presented. A first variant applies a fixpoint-based approach when resolving the Kleene star and plus operators, but could only be shown to have a time complexity bound exponential in the size of the RDF graph. Instead, the second variant is shown to have a time complexity cubic in the size of the RDF graph and linear in the size of the RPL query.

3. RPL was designed to be easily embeddable into already existing RDF query languages. In Chapter 4, it is exemplary embedded into SPARQL [29] at predicate position, which preserves the $LL(1)$ property of the SPARQL grammar. As a natural consequence, RPL is extended to import variables from its host language. It is shown that the expressivity of

SPARQL and RPL together is powerful enough to imitate the logical core fragment of the RDFS semantics [20] without computing the RDFS closure of an RDF graph.

4. Chapter 5 experimentally evaluates both evaluation approaches presented in Chapter 3. Several datasets of the same structure, but of different sizes are generated by *RPLgen*, a RPL-based data generator that uses RPL queries in a generative way. The complexity bounds of both evaluation approaches are experimentally verified; boths variants of the Path-based evaluation approach seem to have cubic data complexity (although only an exponential bound has been proved for the first variant in Chapter 3).

5. Three tools are developed to lower the learning curve and to ease the authoring of RPL expressions. They are briefly described in Chapter 6 and can be accessed online at `http://rpl.pms.ifi.lmu.de/`.

   A demo web-page targets users that are curious about seeing RPL in action. Therefore, it ships with two application scenarios along with several predefined RPL queries on these datasets (nevertheless, individual RDF data and individual RPL queries can be entered into the web forms).

   visRPL, an Eclipse-based visual editor for RPL allows to graphically compose RPL queries for users not yet being familiar with RPL's textual syntax. However, visRPL is at any time able to generate a textual RPL query out of this graphical representation and the other way round. Apart from this feature referenced as *roundtripping* [19], visRPL offers all amenities expected from today's editors: undoing and redoing commands, dragging and dropping as well as copying and pasting elements, zooming in and out, syntax highlighting, an outline view, and so forth.

   Finally, another Eclipse-based, but textual editor is available for more experienced users of RPL, which offers syntax completion and an abstract syntax tree view and thus supports the authoring of RPL expressions.

2

PRELIMINARIES: INTRODUCING RPL

2.1 RPL BY EXAMPLE

RPL (RDF Path Language, pronounced "ripple") is inspired by XPath [11] and NREs (Nested Regular Expressions) [27] in that it allows *predicates* (called nested expressions in the case of NREs) on paths. Thus, in addition to *local* conditions on the path between two RDF nodes, also *non-local* conditions on branches starting at a node or an edge on the path can be expressed via predicates. A RPL expression exp evaluates to a set of node pairs (a, b) such that node a is connected to node b via a path that satisfies exp.

Before we introduce a concrete example, we briefly present the various flavors and the overall structure of RPL expressions in the following paragraphs.

RPL expressions can appear in three *flavors*: *node-* and *edge-flavored* expressions only place restrictions on the nodes and edges of the traversed path, respectively — while *path-flavored* expressions apply conditions for both its nodes and edges.

There are three kinds of node-flavored expressions. `NODES>` a b c describes all paths beginning at some node a' (with a' satisfies a) that has an arbitrary *outgoing* edge to a node b' (with b' satisfies b) that in turn has an arbitrary *outgoing* edge to a node c' (with c' satisfies c). Hence, all node pairs (a', c') are in the evaluation of `NODES>` a b c. `NODES<` a b c reverses the direction of the edges between a' and b' (i.e. a' now has an arbitrary *incoming* edge from b'), and b' and c' (i.e. b' now also has an arbitrary *incoming* edge from c'). Finally, `NODES` a b c allows for arbitrarily directed (i.e. *outgoing and incoming*) edges between a' and b', and b' and c'.

Edge-flavored expressions allow for ad-hoc computation of RDFS semantics: for instance, `EDGES >rdfs:subPropertyOf+` retrieves all pairs of nodes (n, p) such that n is a direct or indirect subproperty of p according to RDFS semantics. Similar as in node-flavored expressions, the character `>` indicates

a forward-oriented edge (i.e. its preceding node has an *outgoing* edge to its following node). The subexpression `>rdfs:subPropertyOf+` is called an *adorned* expression, and `+` is its *multiplicity* (`*` and `?` are available as well).

Path-flavored expressions are the most expressive flavor, as they restrict both nodes and edges. **PATH** _ (<*e* | >*f*) *g* starts with an arbitrary node (_ serves as *wildcard*) that either (a) has an incoming edge whose label satisfies *e* from a node whose label satisfies *g* or that (b) has an outgoing edge whose label satisfies *f* to a node whose label satisfies *g*. The subexpression _ (<*e* | >*f*) *g* is called a *concatenated* expression, consisting of the subexpressions _ (the wildcard), (<*e* | >*f*) (a *disjunctive* expression), and *g* (an *atomic* expression). Atomic expressions are used to match the labels of single nodes or edges from an RDF graph.

Let us start demonstrating RPL via a concrete application scenario: transportation services between cities. The RDF graph shown in Figure 6 is identical to the nSPARQL transportation services example [27] and serves as sample data for this introduction and within this thesis. It describes the various transportation services (and their hierarchy) 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 `PATH :Paris >_ _` (or equivalent: `NODES> :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).

As we want to get rid of that (`:Paris`, `:France`) result pair, we need to ensure that just transport edges are followed. This is where RPL *predicates* come into play. Predicates are flavored expressions that are enclosed in square brackets; they can have either positive or negative (denoted by `!`) *sign*. The predicate `[PATH (_ >rdfs:subPropertyOf)* :transport]` follows an arbitrary number of `rdfs:subPropertyOf` edges, until a `:transport` node is reached. This predicate is now used instead of the outgoing edge from `:Paris`, which results in the following RPL query *q*

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

It specifies that we follow edges from `:Paris` that are labeled with `:transport` or any of its subproperties.

As none of the directly reachable cities interests us any more, we decide to allow intermediate stops. In order to query all cities that are directly and indirectly reachable over at least one transport edge, we simply have to enclose everything after `:Paris` with `+` multiplicity. This yields the following query

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

Evaluating this query on the RDF graph shown in Figure 6 results in the set {(:Paris, :Calais), (:Paris, :Dijon), (:Paris, :Dover), (:Paris, :Hastings), (:Paris, :London)}.

As we are not keen on traveling by bus, we would like to exclude those transport edges. Again, the modification is straightforward: we add another predicate (now with a negative sign, indicated by !) that does not allow :bus edges (and their subproperties) to be traversed. The adapted query is

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

and evaluates to the set {(:Paris, :Calais), (:Paris, :Dijon), (:Paris, :Dover)}, as :London and :Hastings are just reached via bus from :Dover.

All of the presented example queries can also be accessed and run online at http://rpl.pms.ifi.lmu.de/ (Chapter 6).

2.2 SYNTAX OF RPL

After having seen some example queries, we give a formal definition of RPL's syntax in this section. We present the concrete syntax (which is relevant when writing query strings), as well as the abstract syntax of RPL (which is an internal representation of a query string). This section closes with a short comparison of the RPL syntaxes presented here and those given in [9].

2.2.1 *Concrete Syntax*

Definition 1 (Syntax of RPL)
The concrete syntax of RPL is defined by the following grammar.

⟨*flavored*⟩ ::= ⟨*flavor*⟩ ⟨*concatenated*⟩

⟨*flavor*⟩ ::= '**EDGES**' | '**NODES**' | '**NODES<**' | '**NODES>**' | '**PATH**'

⟨*concatenated*⟩ ::= ⟨*adorned*⟩+

⟨*adorned*⟩ ::= ⟨*adornable*⟩ ⟨*multiplicity*⟩

⟨*adornable*⟩ ::= ⟨*directed*⟩ | '(' ⟨*disjunctive*⟩ ')'

⟨*multiplicity*⟩ ::= ε | '**?**' | '*****' | '**+**'

⟨*directed*⟩ ::= ⟨*direction*⟩ ⟨*directable*⟩

⟨*direction*⟩ ::= ε | '**<**' | '**>**'

⟨*directable*⟩ ::= ⟨*atomic*⟩ | ⟨*predicates*⟩

⟨*disjunctive*⟩ ::= ⟨*concatenated*⟩ ('**|**' ⟨*concatenated*⟩)*

⟨*predicates*⟩ ::= '**[**' ⟨*predicate*⟩ ('**][**'⟨*predicate*⟩)* '**]**'

⟨*predicate*⟩ ::= ⟨*sign*⟩ ⟨*flavored*⟩

⟨*sign*⟩ ::= ε | '**!**'

⟨*atomic*⟩ ::= ⟨*STRING_LITERAL1*⟩ | ⟨*STRING_LITERAL2*⟩ | ⟨*REGEXP*⟩ | '**_**' | ⟨*IRI_REF*⟩ | ⟨*PNAME_NS*⟩ | ⟨*PNAME_LN*⟩ | ⟨*PNAME_RX*⟩

This grammar is in $LL(1)$ normal form, if all uppercased rules (e.g. ⟨*REGEXP*⟩ and ⟨*IRI_REF*⟩) are used as tokens as follows.

- ⟨*STRING_LITERAL1*⟩ and ⟨*STRING_LITERAL2*⟩ are ordinary strings enclosed in single and double quotes, respectively. More precisely, these rules correspond to rules [87] and [88] in the W3C SPARQL Recommendation [29].
- ⟨*REGEXP*⟩ is an arbitrary, ordinary regular expression enclosed in slashes (e.g. `/.*/`). Inner slashes have to be escaped by `\`.
- ⟨*IRI_REF*⟩ corresponds to rule [70] in [29] and is used to express IRIs (Internationalized Resource Identifiers) (e.g. `<http://example.org>`).
- ⟨*PNAME_NS*⟩ and ⟨*PNAME_LN*⟩ correspond to rules [71] and [72] in [29], respectively. ⟨*PNAME_NS*⟩ consists of a namespace prefix only (e.g. `rdf:`), while ⟨*PNAME_LN*⟩ consists of a namespace prefix and a (non-empty) local name (e.g. `rdf:type`). The namespace prefix may also consist of only a colon in both cases (e.g. `:` and `:type` are allowed).
- ⟨*PNAME_RX*⟩ allows for regular expressions relative to namespace prefixes (e.g. `rdf:/.*/`).

RDF *plain literals* annotated with a *language tag* (e.g. `"cat"@en` and `"chat"@fr`), and RDF *typed literals* (e.g. `"42"^^xsd:integer`) can be matched by embedding them into a regular expression, i.e. a ⟨*REGEXP*⟩ token (e.g. `/"cat"@en/`, `/"chat"@fr/`, and `/"42"^^http:\/\/www.w3.org\/2001\/XMLSchema#integer/`, respectively).

RPL does not come with any predefined functions. However, SPARQL tests like *isBlank(·)* and *isLiteral(·)* can also be carried out on a syntactic level via regular expressions. For instance, the regular expression `/_:.*/` matches all blank nodes, and `/".*"(@.*)?/` matches all plain literals with or without a language tag (indicated by `(@.*)?`).

Furthermore, any whitespace between tokens is ignored.

2.2.2 *Abstract Syntax*

Figure 2 shows the abstract syntax of RPL by use of a UML (Unified Modeling Language) class diagram. Each rule of the grammar presented in Definition 1 is mapped to either a class, an interface, or an enumeration. An enumeration (interface) is used, when the rule is an alternative of tokens (rules). An ordinary class is used in all remaining cases.

Table 1 shows how tokens of the concrete syntax are mapped to enum literals of the abstract syntax. Although there are no structural differences between the concrete and abstract syntax of RPL, we will mainly stick to the abstract syntax when discussing RPL in greater detail. All classes (interfaces) implement (extend) a common interface `RPE`, which is omitted for the sake of clarity in Figure 2. When talking just of an expression, we usually refer to an arbitrary class or interface of the abstract syntax tree. When talking of an RPE (RPL Expression), a RPL query, or a flavored expression, we refer to an instance of the `FlavoredRPE` class. Similar, when e.g. talking of an adornable expression, we refer to an instance implementing the `AdornableRPE` interface.

Figure 3 shows the most important operations of the `RPE` interface. These are getters and setters for its length (Section 3.1.3), its position (Section 3.1.4), and its partial path matrix (Section 3.4.1). The operation *accept* is part of the *visitor pattern* [17], and is used whenever the abstract syntax tree of an RPE has to be traversed. The visitor pattern is frequently used in compiler construction, as related operations (e.g. for pretty printing or simplifying a query) can be defined in a concrete visitor class (e.g. a `PrettyPrintVisitor` or a `SimplifyVisitor`) instead of "polluting" all abstract syntax tree classes.

Later in this thesis, we use Java's dot notation for accessing member variables of abstract syntax tree classes. For instance, when a is an `AtomicRPE`, we access its position by $a.getPosition()$ (this operation is available via the `RPE` interface, see Figure 3). When writing $a.getRegExp()$, we access the regular expression that is associated with a (i.e. according to Java Beans Conventions, we are accessing the property `regExp` of the class `AtomicRPE`, see Figure 2).

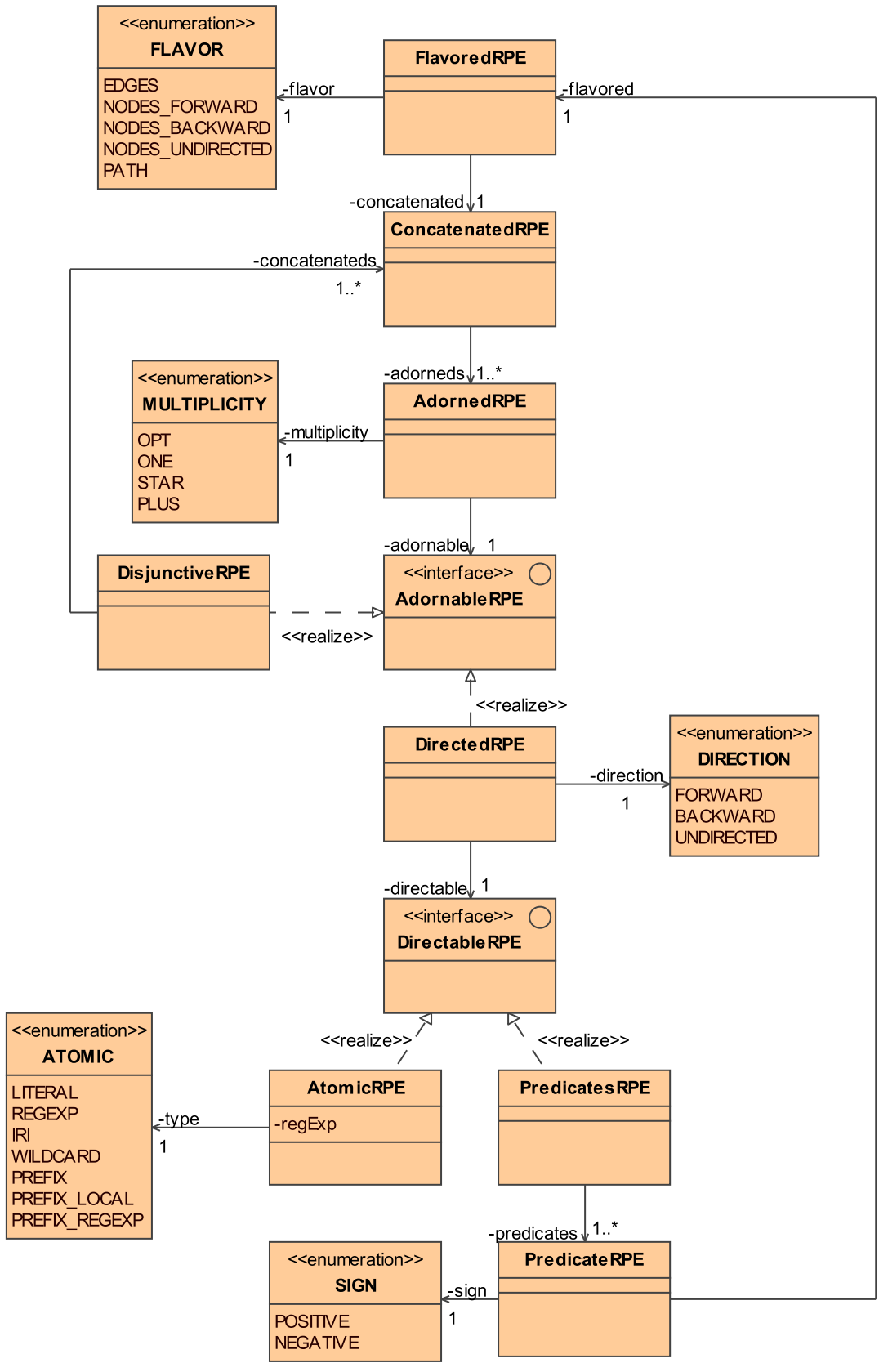


Figure 2: UML class diagram showing the abstract syntax of RPL

⟨*flavor*⟩	FLAVOR
EDGES	EDGES
NODES	NODES_UNDIRECTED
NODES>	NODES_FORWARD
NODES<	NODES_BACKWARD
PATH	PATH

⟨*multiplicity*⟩	MULTIPLICITY
ε	ONE
?	OPT
*	STAR
+	PLUS

⟨*direction*⟩	DIRECTION
ε	UNDIRECTED
>	FORWARD
<	BACKWARD

⟨*sign*⟩	SIGN
ε	POSITIVE
!	NEGATIVE

⟨*atomic*⟩	ATOMIC
⟨*STRING_LITERAL1*⟩	LITERAL
⟨*STRING_LITERAL2*⟩	LITERAL
⟨*REGEXP*⟩	REGEXP
⟨*IRI_REF*⟩	IRI
_	WILDCARD
⟨*PNAME_NS*⟩	PREFIX
⟨*PNAME_LN*⟩	PREFIX_LOCAL
⟨*PNAME_RX*⟩	PREFIX_REGEXP

Table 1: Mapping from tokens to enum literals

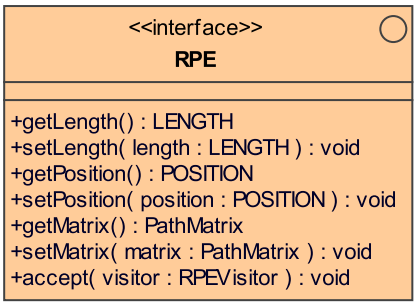


Figure 3: UML class diagram showing the RPE interface and its operations

2.2.3 *Syntax Comparison*

Both syntaxes presented here compare to the syntax definition given in [9] as follows.

- The concrete syntax (Definition 1) is suitable for parsing purposes as it is in $LL(1)$ normal form and establishes the precedence rules that have been formulated in [9].
- A direction modifier `^` is not needed and thus omitted.
- The negation of predicates is denoted by `!` instead of `not(·)`.
- As predicates are already expressive enough, a directable subexpression is either an atomic expression or a (non-empty) sequence of predicates (in [9] and XPath [11], it is made up of both). As a consequence, no whitespace is allowed between the closing and opening brackets of predicates (Definition 1).
- Three kinds of node-flavored RPEs are allowed. **`NODES>`** and **`NODES<`** express that all edges (which will be added between the specified nodes, see Chapter 3.1.2) are forward- and backward-directed, respectively. **`NODES`** is already allowed in [9], but now expresses undirected instead of forward-directed edges.

2.3 IMPLEMENTING A RPL PARSER

In the previous section, we have presented both the concrete and abstract syntax of RPL. This section will discuss the missing link in between: a parser that establishes the transformation from the concrete into the abstract syntax.

In a first attempt, the popular ANTLRv3[1] parser generator was used to recognize and parse RPL expressions. However, it has turned out that its lexer component has difficulties to differentiate between the cases, in which the character < appears in the input string. Besides in the token **`NODES<`**, this character is used as a direction modifier (see the ⟨*direction*⟩ rule in Definition 1), and as first character of the ⟨*IRI_REF*⟩ token.

1 see http://www.antlr.org/

```
1 grammar RPE;
2 rpe: direction atomic;
3 direction: ( '<' | '>' )? ;
4 atomic: IRI_REF ;
5 IRI_REF:
6 '<' (
7       ~('<'|'>'|'"'|'{'|'}'|'|'|'^'|'\\'|'`'|'\u0000'..'\u0020')
8     )+
9 '>' ;
```

Listing 2: ANTLRv3 grammar file for RPL (reduced to the conflicting parts)

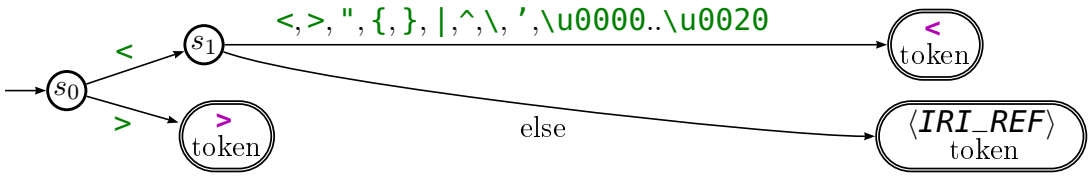


Figure 4: ANTLRv3's lexer DFA, generated from the grammar in Listing 2

Listing 2 shows an ANTLRv3 grammar file for RPL that has been reduced to the conflicting tokens ⟨*IRI_REF*⟩, and the two direction tokens `<` and `>`.

Figure 4 is based on the lexer DFA visualized by ANTLRWorks[2] from Listing 2. It shows that the first < character seen in the input string causes the DFA to change its state from s_0 to s_1. In state s_1, only one further character is considered: if this character matches one of the allowed inner characters of the ⟨*IRI_REF*⟩ token (see line 7 of Listing 2, `~(...)` serves as negation), an ⟨*IRI_REF*⟩ token is already emitted this early — although only two characters have been considered by now.

When e.g. trying to lex the input string `<rdfs:subPropertyOf`, its first two characters < and `r` (`r` is not forbidden in line 7) lead to the emission of an ⟨*IRI_REF*⟩ token, which in turn leads to an exception, as all ⟨*IRI_REF*⟩ tokens are expected to end with a > character, as can be seen in line 9 of Listing 2. Compare also Figures 26 and 27 in order to see the effect of this faulty decision.

ANTLRv3's behavior is quite surprising, as its lexer is said to be $LL(*)$[3] (this means $LL(k)$ for any $k \geq 1$), and not just $LL(2)$ as above.

2 see http://www.antlr.org/works/index.html
3 see http://www.antlr.org/wiki/display/~admin/ANTLR+v4+lexers

In contrast, JavaCC[4] is able to cleanly keep all tokens apart. An additional advantage over ANTLRv3 is that it does not rely on a separate runtime component — all needed classes are directly created during JavaCC parser generation. Figure 5 gives an overview over JavaCC's setup for parsing RPL. The grammar file `RPE.jj`, which is the only input to `javacc`, contains both the lexer and parser rules.

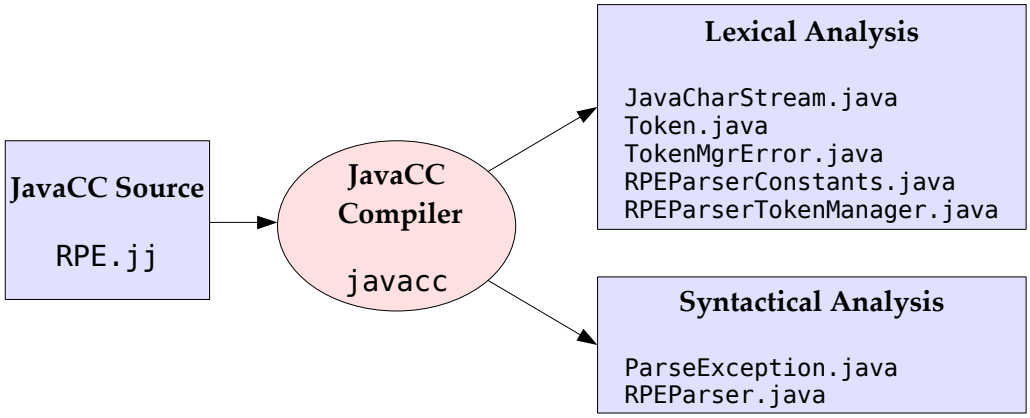


Figure 5: Overview of JavaCC parser generation, similar to [30]

2.4 THE RESOURCE DESCRIPTION FRAMEWORK

The Resource Description Framework (RDF) is a "language for representing information about resources in the World Wide Web" [10]. However, RDF is not only able to represent metadata about Web resources, but also arbitrary graph-shaped data.

DEFINITION 2 (RDF Triple, RDF Graph, Subject, Predicate and Object of an RDF Triple)
Let U, B, and L be pairwise disjoint infinite sets of *RDF URI references*, *RDF blank nodes*, and *RDF literals*, respectively. An *RDF triple* $t = (s, p, o)$ over U, B, and L is an element of $(U \cup B) \times U \times (U \cup B \cup L)$, where s is called *subject*, p is called *predicate*, and o is called *object* of t. An *RDF graph*, for the scope of this thesis, is a *finite* set of RDF triples.

4 see https://javacc.dev.java.net/

Definition 3 (Nodes, Edges, and Terms of an RDF Graph)
Let G be an RDF graph. The set of *nodes* of G is the set of subjects and objects of G's RDF triples, $nodes(G) := \{s \mid (s,p,o) \in G\} \cup \{o \mid (s,p,o) \in G\}$. Similarly, the set of *edges* of G is defined as $edges(G) := \{p \mid (s,p,o) \in G\}$, and the set of *terms* of G as $terms(G) := nodes(G) \cup edges(G)$.

Definition 4 (Paths in an RDF Graph)
For the scope if this thesis, a *path* p within an RDF graph G is a sequence $(n_1, \ldots, n_{2k+1})$, $k \geq 0$ where the RDF triples $(n_{2i-1}, n_{2i}, n_{2i+1})$ are contained in G for all $1 \leq i \leq k$. The length of p is defined as the number of edges that p contains, i.e. $|p| := k$.

An RDF graph G can also be interpreted as a directed labeled multigraph $\mathcal{M}(G)$ [5]. When drawing an RDF graph, each node $n \in nodes(G)$ is represented by its label which is drawn inside an oval (if $n \in U \cup B$) or a rectangle (if $n \in L$). Each edge e belonging to an RDF triple (s,e,o) is drawn as a directed, labeled arrow from the figure of s to the figure of o.

When discussing RPL, we always refer to the RDF graph presented in Figure 6 unless stated otherwise. It consists of several cities that are connected to each other via different means of transport. Additionally, Figure 6 makes use of the RDFS vocabulary [20], in order to express e.g. `rdfs:subPropertyOf` (which is abbreviated as `rdfs:sp`) and `rdfs:subClassOf` relationships.

RDF Serializations

RDF offers a plethora of serialization formats, of which only the currently most commonly used ones, *RDF/XML* [6] and *Turtle* [7] shall be mentioned here. RDF/XML, however, is a verbose syntax and difficult to read.

Listing 3 shows a Turtle serialization of the RDF graph shown in Figure 6. The Turtle syntax is quite close to the RDF data model, as it is made up of single triples that are separated by a dot character from each other. If a triple t is instead ended by a comma (semicolon), the subject and predicate (subject) of t will be reused for the subsequent triple.

Definition 5 (Size of an RDF Graph)
For time complexity considerations (Chapter 3), we define the size of an RDF graph G (denoted by $|G|$) as the size (in characters) of its minimal representation in Turtle syntax, by not making use of any namespace prefix declarations or abbreviations.

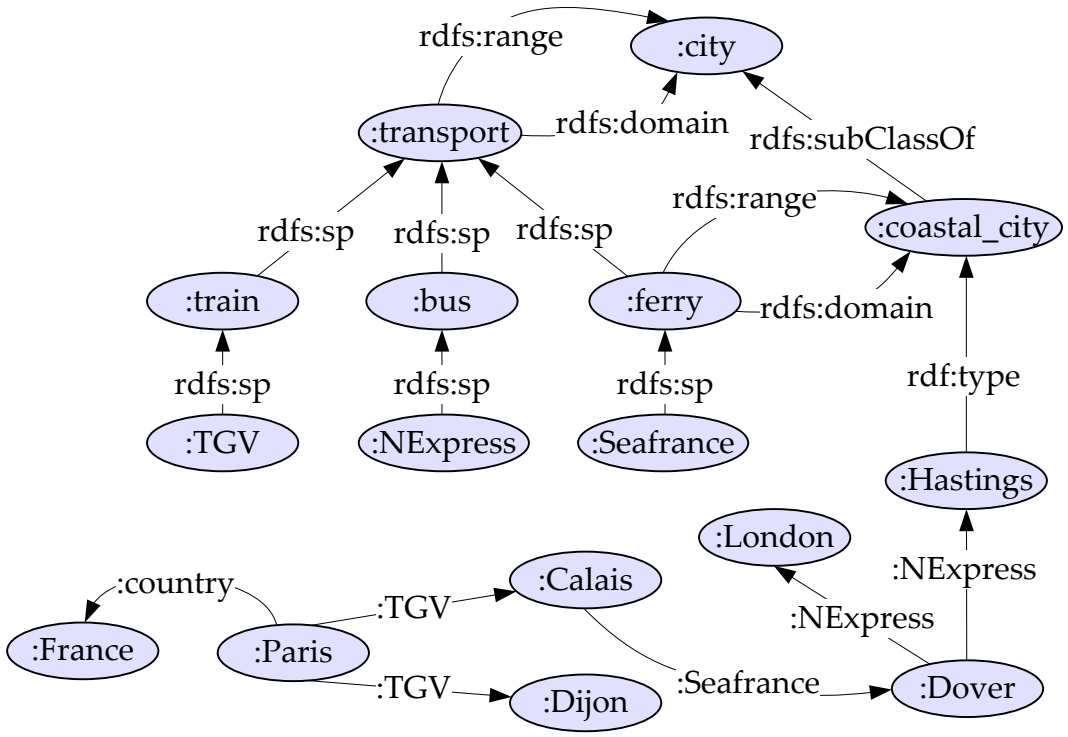


Figure 6: RDF graph about transportation services between cities [27]

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://example.org/> .

:Paris :TGV :Calais , :Dijon ; :country :France .
:Calais :Seafrance :Dover .
:Dover :NExpress :Hastings , :London .

:TGV rdfs:subPropertyOf :train .
:Seafrance rdfs:subPropertyOf :ferry .
:NExpress rdfs:subPropertyOf :bus .

:train rdfs:subPropertyOf :transport .
:ferry rdfs:subPropertyOf :transport .
:bus rdfs:subPropertyOf :transport .

:ferry rdfs:range :coastal_city ; rdfs:domain :coastal_city .
:Hastings a :coastal_city .

:transport rdfs:range :city ; rdfs:domain :city .
:coastal_city rdfs:subClassOf :city .
```

Listing 3: Turtle serialization of the RDF graph from Figure 6

2.5 EXCURSUS: REGULAR EXPRESSIONS OVER STRINGS

Regular Expressions over ordinary strings are used in conjunction with

1. `AtomicRPEs`, which serve as edge or node label tests, and
2. ENREs (Definition 10) of the form $\langle axis \rangle$`::`$\langle regexp \rangle$.

[13] states that regular expressions are poorly implemented in most modern programming languages. Concerning Java, this claim can be verified using a snippet like the one shown below.

```
public static void main(String[] args) {
    StringBuilder regexp = new StringBuilder();
    StringBuilder string = new StringBuilder();
    for (int i = 1; i <= 28; i++) {
        regexp.insert(0, "a?");
        regexp.append("a");
        string.append("a");
        long before = System.nanoTime();
        string.toString().matches(regexp.toString());
        long after = System.nanoTime();
        double seconds = (after - before) / 1000000000.;

        System.out.println("Matching (a?)^" + i + "a^" + i + " against a^" + ↩
            i + " took " + seconds + " s.");
    }
}
```

```
...
Matching (a?)^20a^20 against a^20 took 0.180791973 s.
Matching (a?)^21a^21 against a^21 took 0.424027634 s.
Matching (a?)^22a^22 against a^22 took 0.671791247 s.
Matching (a?)^23a^23 against a^23 took 1.575839667 s.
Matching (a?)^24a^24 against a^24 took 3.72857081 s.
Matching (a?)^25a^25 against a^25 took 5.844788933 s.
Matching (a?)^26a^26 against a^26 took 13.787329395 s.
Matching (a?)^27a^27 against a^27 took 32.909819265 s.
Matching (a?)^28a^28 against a^28 took 50.436544589 s.
```

Java's native regular expression implementation `java.util.regex` (initiated by a call of `matches(String regex)` on a `String` object in the snippet above) is based on a backtracking approach. So, when trying to match a string against the pattern `a?`, this backtracking implementation tries first for the pattern `a`, and then for ε. The patterns used in the snippet are a concatenation of the sub-patterns `a?` (n times) and `a` (n times). So, for evaluating the n-th pattern,

$\mathcal{O}(2^n)$ possibilities have to be considered, and only the very last (choosing ε for all `a?` sub-patterns) will lead to a match (cf. [13]).

However, by constructing an ε-NFA $\mathcal{A}$ from a regular expression exp, the time complexity of matching a string s to the pattern exp is $\mathcal{O}(n \cdot m)$, where n is the length of s and m is the number of $\mathcal{A}$'s states. m, however, is at most equal to the length of exp by Thompson's construction [32].

For this reason, we prefer the `dk.brics.automaton` implementation [24] over Java's native implementation and modify it accordingly, so that no determinisation takes place (which, in the worst case, would also require exponential time as it can lead to a DFA having $\mathcal{O}(2^m)$ states, cf. [28]).

3

IMPLEMENTING RPL

After presenting RPL's syntactic and semantic analysis in Section 3.1, this chapter first gives a set-based semantics of path-flavored RPL queries.

Section 3.3 presents an evaluation approach that is based on translating RPL into an intermediate language, ENREs (Extended Nested Regular Expressions). After presenting the syntax and semantics of ENREs, we show the correctness of the translation function and present an adapted graph labeling algorithm for ENREs that is shown to have linear query and quadratic data time complexity.

Section 3.4 presents two variants of the Path-based evaluation approach. After introducing the notion of partial path matrices and the basic idea of the algorithm (linking each subexpression e of a RPL query with a matrix that stores all paths satisfying e), the second variant is shown to have linear query and cubic data time complexity.

3.1 SYNTACTIC & SEMANTIC ANALYSIS

Many query languages include syntactic constructs that do not have any effect on the expressive power of the language, but allow users to express the same (i.e. semantically equivalent) query in an alternative, often shorter or more intuitive way (consider e.g. the abbreviated syntax of XPath [11]). Such syntactic constructs are generally known as *syntactic sugar*[1] and are eliminated within a compiler phase called *syntactic analysis*.

On the other hand, not all preconditions for the evaluation of a query can be expressed on a purely syntactic level in most query languages (e.g. the existence of all tables and columns that are referenced within an SQL query, or the declaration of all namespace prefixes that are used within a SPARQL [29] query). These semantic checks are carried out within another compiler phase called *semantic analysis*.

1 see http://en.wikipedia.org/wiki/Syntactic_sugar

An overview of RPL's syntactic and semantic analysis is given in Figure 7.

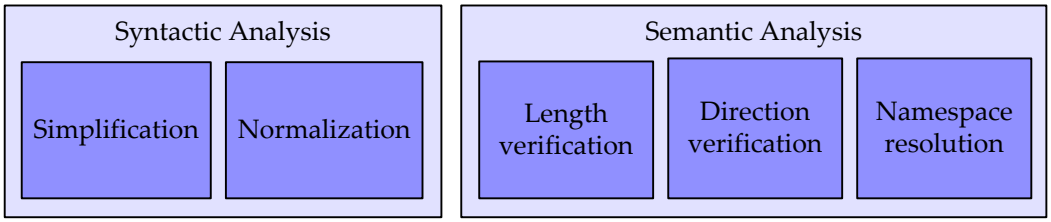


Figure 7: RPL's syntactic and semantic analysis phases

The syntactic analysis phase of RPL is organized in two passes. During *simplification*, its first pass, all unnecessary parentheses are removed from a RPL query. The second pass, called *normalization*, then converts node- and edge-flavored RPL queries into path-flavored ones (hence, node- and edge-flavored queries can be considered as syntactic sugar).

Figure 8 demonstrates the syntactic analysis of the query `NODES> :a (_)+`.

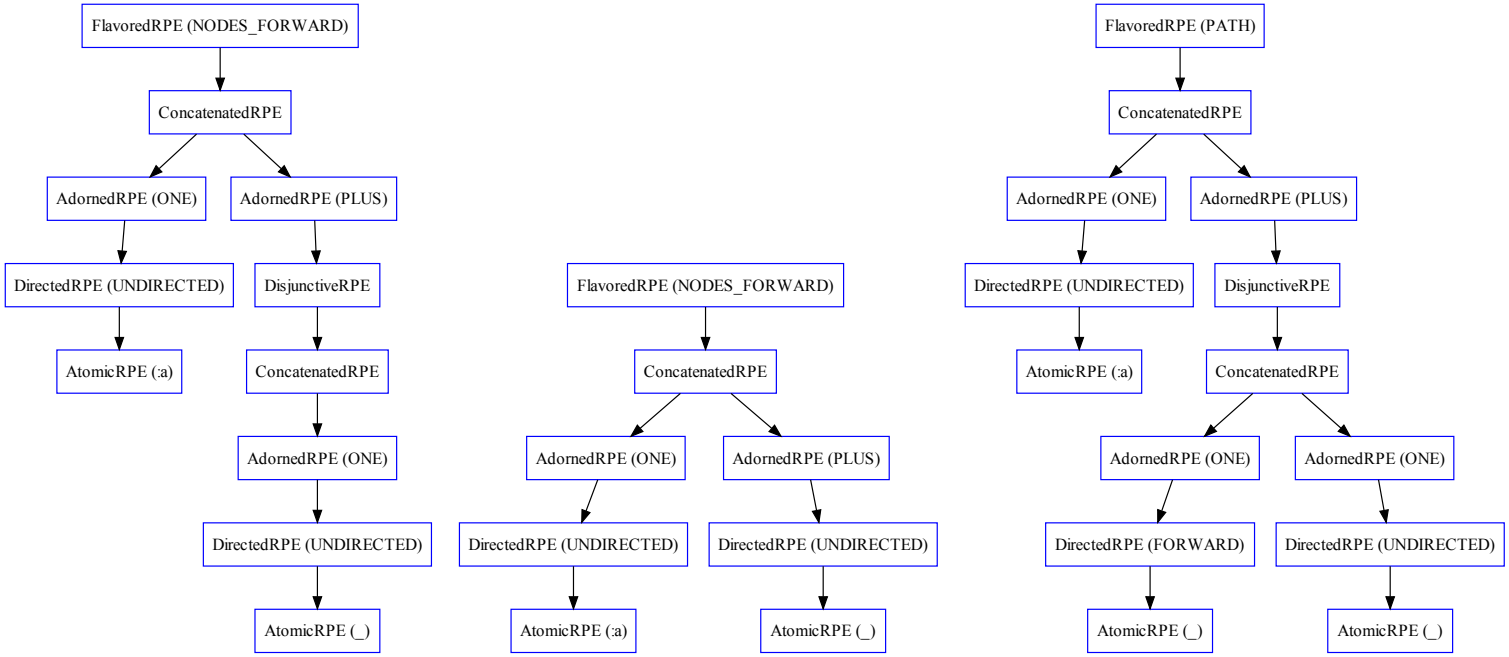
The semantic analysis phase consists of three passes, of which each enriches the abstract syntax tree of a RPL query. We call a RPL query *valid* iff it has passed all checks of the semantic analysis.

Its first pass, *length verification*, checks e.g. that the length of all possible paths (when counting each node and edge as 1) defined by a (path-flavored, due to normalization) RPL query is an odd number, as every path must start and end with a node. For instance, this check fails on `PATH :a >:b`, as it describes a path starting with a node `:a` that has an outgoing edge `:b`, but no target node.

The second pass of the semantic analysis is called *direction verification*. Using the length information that has been stored on each node of the abstract syntax tree during the preceding pass, direction verification computes if the path a subexpression e is expected to match either starts with a node or an edge — we say that e appears at node or edge position. In this context, it is also checked that each `DirectableRPE` appearing at node position (i.e. it is expected to match a node) is not directed by any of the direction modifiers `<` and `>`. For instance, `PATH :a >:b <:c` fails direction verification, as `<:c` is directed by `<`, but is expected to match a node.

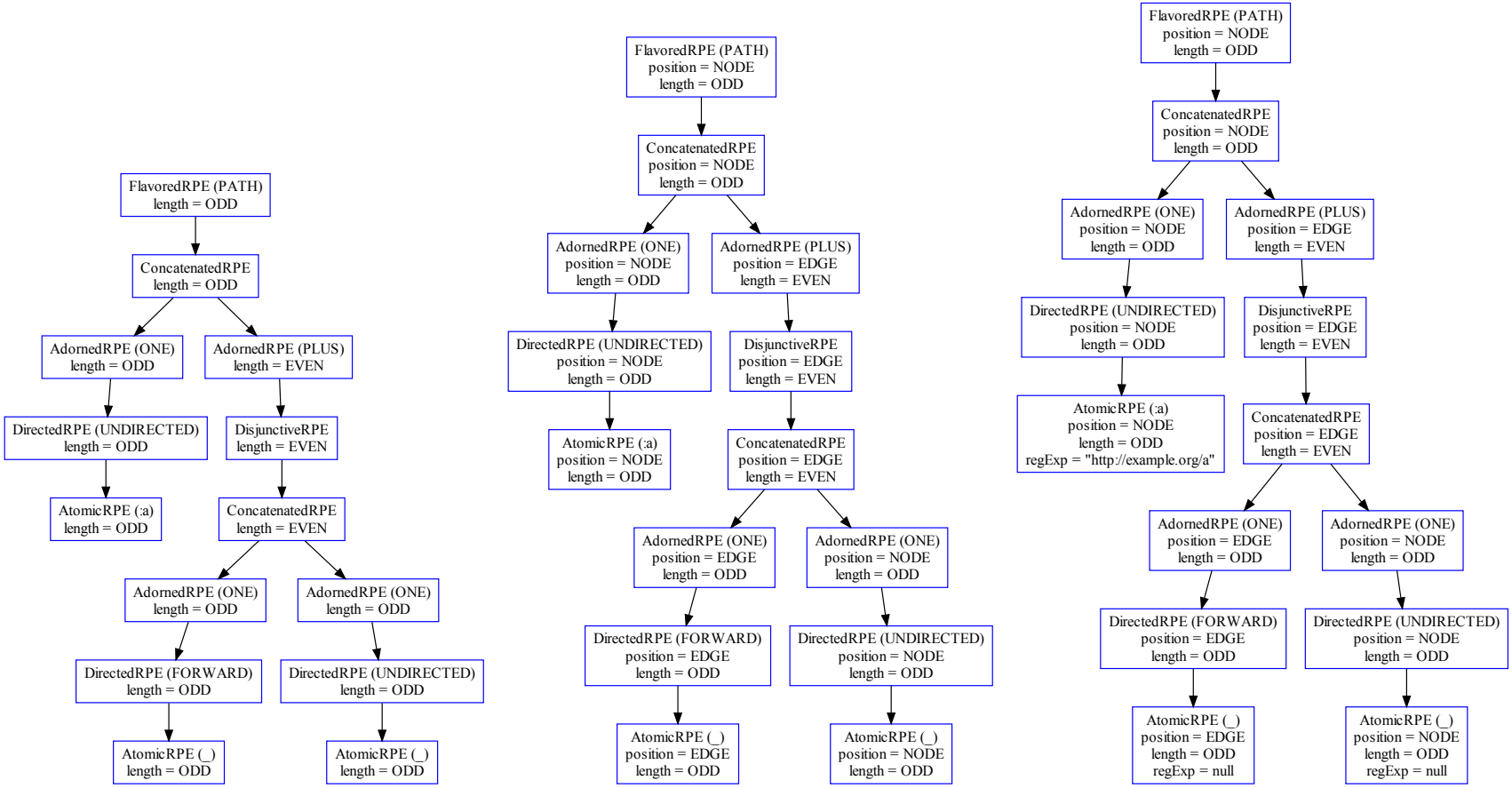
Namespace resolution is the third pass within the semantic analysis of RPL. It resolves namespace prefixes that are defined in the underlying RDF graph, on which the RPL query is to be evaluated. Furthermore, all `AtomicRPEs` are enriched with a regular expression that corresponds to their string value.

Figure 9 demonstrates all passes of the semantic analysis on the RPL query `PATH :a (>_ _)+`, as returned by the syntactic analysis on the original query `NODES> :a (_)+` (Figure 8).



(a) Initial query: `NODES> :a (_)+` (b) After simplification: `NODES> :a _+` (c) After normalization: `PATH :a (>_ _)+`

Figure 8: Syntactic analysis of the query `NODES> :a (_)+` consists of removing unnecessary parentheses during simplification, and converting it into a path-flavored query during normalization



(a) After length verification (b) After direction verification (c) After namespace resolution

Figure 9: Enrichment of the abstract syntax tree in Figure 8c with lengths during length verification, positions during direction verification, and regular expressions for `AtomicRPE`s during namespace resolution

3.1.1 *Simplification*

Simplification is the first pass during the syntactic analysis of RPL queries. It minimizes the abstract syntax tree of arbitrary RPL queries by removing all unnecessary parentheses.

EXAMPLE 1

```
PATH ((_ >rdfs:subPropertyOf)* :transport)
```

gets simplified to

```
PATH (_ >rdfs:subPropertyOf)* :transport
```

Figure 10 shows the query of Example 1 before and after simplification.

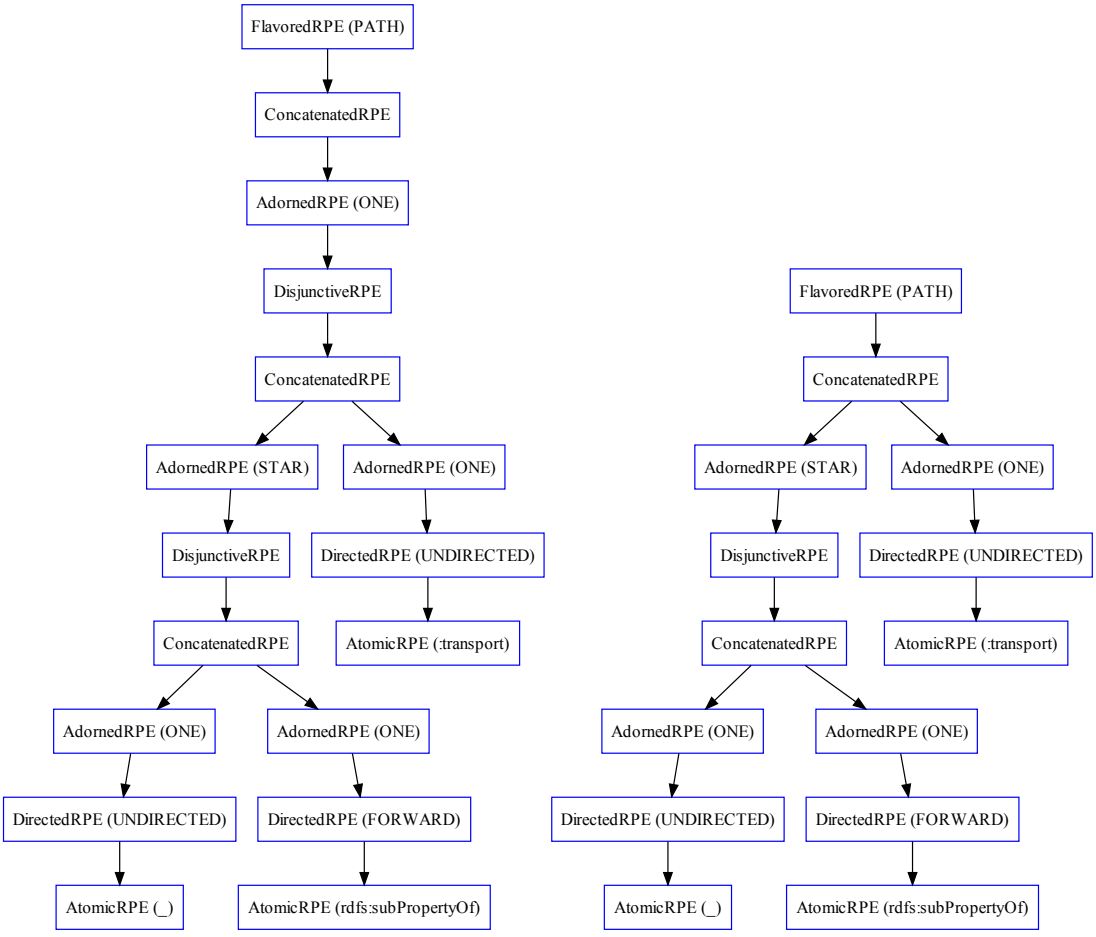


Figure 10: Abstract syntax tree before (left) and after (right) simplification

No further simplification is done beyond this. For instance,

```
PATH (_ >rdfs:subPropertyOf)? (_ >rdfs:subPropertyOf)* :transport
```

could as well be simplified to the resulting query of Example 1. The reason for this behaviour is that the simplification pass is just considered as a preparation step for normalization, the following pass within the syntactic analysis phase.

3.1.2 *Normalization*

To ease the authoring of RPL queries, node-, edge-, and path-flavored RPEs are allowed. However, for evaluation purposes, it is more convenient to deal with only one flavor of RPEs. Hence, node- and edge-flavored RPEs are converted into path-flavored ones during *normalization*.

EXAMPLE 2

```
EDGES >[PATH (_ >rdfs:subPropertyOf)* :transport]
```

is an edge-flavored RPE whose single edge is restricted by a predicate. The equivalent path-flavored RPE is obtained by inserting a wildcard before and after this single edge.

```
PATH _ >[PATH (_ >rdfs:subPropertyOf)* :transport] _
```

As can be seen in Example 2, the fundamental idea is to insert wildcards at the proper positions in node- and edge-flavored RPEs. More precisely, wildcards have to be inserted at all positions of a node-flavored (edge-flavored) RPE, where an edge (a node) would be required in the corresponding path-flavored RPE. If `NODES>` or `NODES<` is used, these wildcards also have to be directed by `>` or `<`, respectively.

In the case of node-flavored RPEs, it is crucial that the simplification pass has already happened — otherwise, the algorithm that establishes the insertion of wildcards at the appropriate positions would not function correctly.

Example 3 demonstrates that adorned subexpressions, whose multiplicity is `?`, `*`, or `+`, require special treatment.

EXAMPLE 3

We reuse the query of Example 2 by just adding the multiplicity `+`

```
EDGES >[PATH (_ >rdfs:subPropertyOf)* :transport]+
```

Converting this edge-flavored RPE results in two syntactically different (but semantically equivalent) path-flavored RPEs, depending on where parentheses are introduced.

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

The implementation of the normalization algorithm will return the first result.

One special case during normalization deserves attention: when converting node-flavored RPEs, the size of the query might increase exponentially under certain conditions. If all children of a concatenated expression are adorned subexpressions with multiplicity +, then one of these adorned subexpressions must be unrolled as shown in the UML object diagram in Figure 11. As is the case with ordinary regular expressions, unrolling has no effect on the semantics of the query.

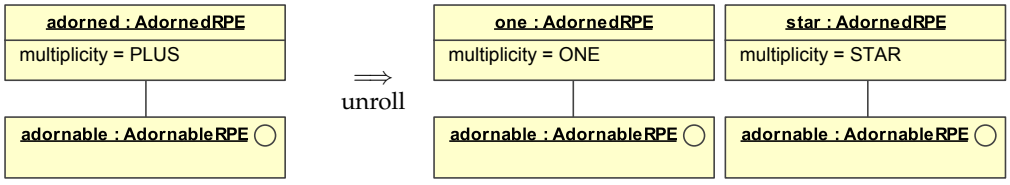


Figure 11: Unrolling +-adorned expressions

The exponential increase of the query size is reached when a query nests several +-adorned expressions into one another, like Example 4 demonstrates.

However, there is no way to bypass unrolling in this case, as both evaluation algorithms (Sections 3.3 and 3.4) are designed to only work on path-flavored RPEs.

Example 4

In order to illustrate this special case, consider the following query

```
NODES> (:a [NODES> :b+])+
```

Unrolling transforms it to

```
NODES> (:a [NODES> :b :b*]) (:a [NODES> :b :b*])*
```

The final path-flavored RPE is

```
PATH (:a >_ [PATH :b (>_ :b)*]) (>_ (:a >_ [PATH :b (>_ :b)*]))*
```

3.1.3 *Length Verification*

Length verification is the first pass of the semantic analysis phase. It calculates the length of all possible paths that are defined by a RPL query and its subexpressions. However, for this calculation, it is not necessary to count the total number of nodes and edges that have to be traversed — it is sufficient to know if a path has odd or even length.

After these lengths have been calculated and stored in the abstract syntax tree (Figures 2 and 3) , several checks are carried out. For instance, as path-flavored queries always start and end with a node, the length of its possible paths has to be odd in any case. Let us study the example query `PATH :a >:b`. It defines a path starting at a node `:a` that has an outgoing edge `:b`, but no target node. Although we are already sure that this query is incorrect, we try to intuitively determine its length. `:a` matches a single node, so its length is odd, and `>:b` matches a single edge, so its length is odd as well. The length of the concatenation `:a >:b` is obtained by "adding" the length of its children. Concatenating two paths of odd length however yields a path of even length (as is the case when adding two odd numbers), and this will cause length verification to fail on `PATH :a >:b`.

The following Definitions 6 and 7 formalize this intuition.

Definition 6 (Lengths, Addition of Lengths)
Let $lengths := \{\texttt{EVEN}, \texttt{ODD}\}$ be the set of lengths. The addition of two lengths $length_1, length_2 \in lengths$ is defined as

$$length_1 + length_2 := \begin{cases} \texttt{EVEN} & \text{if } length_1 = length_2 \\ \texttt{ODD} & \text{else} \end{cases}$$

Definition 7 (Length of RPEs)
Let $lengths := \{\texttt{EVEN}, \texttt{ODD}\}$ be the set of lengths. The length of an `RPE`, $length : \texttt{RPE} \mapsto lengths \cup \{\bot\}$ is inductively defined as follows. The notation used within the following boxes is explained in Section 2.2.2.

Let *atomic* be an `AtomicRPE`. $length(atomic) := \texttt{ODD}$

Let *predicate* be a `PredicateRPE`.

$$length(predicate) := \begin{cases} \bot & \text{if } length(predicate.getFlavored()) = \bot \\ \texttt{ODD} & \text{else} \end{cases}$$

Let p be a `PredicatesRPE` and $children := p.getPredicates()$.

$$length(p) := \begin{cases} \bot & \text{if } \exists child \in children : length(child) = \bot \\ \texttt{ODD} & \text{else} \end{cases}$$

Let $directed$ be a `DirectedRPE`.

$$length(directed) := length(directed.getDirectable())$$

Let d be a `DisjunctiveRPE` and $children := d.getConcatenateds()$.

$$length(d) := \begin{cases} len & \text{if } \forall child \in children : length(child) = len \\ \bot & \text{else} \end{cases}$$

Let a be an `AdornedRPE` and $adornable := a.getAdornable()$.

$$length(a) := \begin{cases} \texttt{EVEN} & \text{if } length(adornable) = \texttt{EVEN} \\ \texttt{ODD} & \text{if } length(adornable) = \texttt{ODD} \wedge \\ & \quad a.getMultiplicity() = \texttt{ONE} \\ \bot & \text{else} \end{cases}$$

Let c be a `ConcatenatedRPE` and $children := c.getAdorneds()$.

$$length(c) := \begin{cases} \sum_{child \in children} length(child) \\ \quad \text{if } \forall child \in children : length(child) \neq \bot \\ \bot \quad \text{else} \end{cases}$$

Let f be a `FlavoredRPE`.

$$length(f) := \begin{cases} \texttt{ODD} & \text{if } length(f.getConcatenated()) = \texttt{ODD} \\ \bot & \text{else} \end{cases}$$

There are three cases within Definition 6, where $\bot$ is returned as the *length* of an `RPE` (Figure 3), while all of its children have a length different from $\bot$.

1. All children of a `DisjunctiveRPE` need to have either `ODD` or `EVEN` length. Hence, `(:a :b :c | :d)` is allowed, while `(:a :b | :c)` is not (the `ConcatenatedRPEs` `:a :b` and `:c` have `EVEN` and `ODD` length, respectively).

2. If the length of an `AdornableRPE` is `ODD`, then the multiplicity of its parent `AdornedRPE` must be `ONE`. In all other cases, the length of this `AdornedRPE`

is undefined. For instance, `:a+` would be of `ODD` length, if the multiplicity `+` is expanded only once (resulting in `:a`), but it would be of `EVEN` length, if `+` is expanded twice (resulting in `:a :a`).

3. A `ConcatenatedRPE`, which is child of a `FlavoredRPE`, must be of `ODD` length, as every path is of `ODD` length. E.g., `PATH :a >:b` describes a path starting with a node `:a` that has an outgoing edge `:b`, but no target node. (`PATH :a >:b _` would be a valid expression)

From an implementation point of view, an `RPELengthException` is thrown if the *length* function returns $\bot$. This yields an error message as shown in Example 5.

Example 5
Length verification of `PATH (:a :b | :c)` is unsuccessful (according to the first of the above three cases) and yields the following error message.

```
Expression ":c" has odd length:

PATH (:a :b | :c)
              ^^
```

3.1.4 *Direction Verification*

In the previous pass of the semantic analysis, we have calculated the length of a RPL query and all of its subexpressions. When further processing these lengths, we can determine if a subexpression e has to start with a node or an edge label test by calculating the length of its preceding subexpressions. This enables us to check if e is allowed to be directed by `<` or `>`, as direction modifiers should only be used on edge label tests[2].

Example 6
The RPL query `PATH :a >:b <:c` consists of three atomic expressions, where `:a` is a node test, `>:b` is an edge test and `<:c` in turn is a node test that is invalid, as it is directed by `<`. Hence, the *direction verification* of `PATH :a >:b <:c` fails with the following error message.

2 It is arguable if really an error should be thrown if the user tries to apply a direction modifier on node label tests, as the RPL query could still be evaluated by just ignoring this modifier. However, we believe that giving the user a chance to reconsider his query by rejecting it is the best option.

```
Expression ":c" appears at NODE position and cannot be directed:

PATH :a >:b <:c
             ^^
```

We say that a subexpression e appears at `NODE` and `EDGE` position, if the path defined by e has to start with a node and edge label test, respectively. Formally, a position can be defined as an element of the set of *positions* $:= \{\texttt{EDGE}, \texttt{NODE}\}$, and these positions can be efficiently computed via a single top-down traversal as follows.

1. The position of a `FlavoredRPE` is always `NODE`.
2. If $adorned_1, \ldots, adorned_n$ are the children of a `ConcatenatedRPE` c, their position is defined as
   $$position(adorned_i) := position(c) + \sum_{j=0}^{i-1} length(adorned_j)$$
   $(1 \leq i \leq n$, and $\sum_{j=0}^{0} length(adorned_j) := \texttt{EVEN})$
3. In all remaining cases, the position of an `RPE` is inherited from its parent.

In the second case, we add a length to a position, which is defined as follows. The idea is that a position stays unchanged when adding an `EVEN` length, and changes (from `EDGE` to `NODE` or from `NODE` to `EDGE`) when adding an `ODD` length.

Definition 8 (Addition of lengths to positions)

The addition of a position $pos \in \{\texttt{EDGE}, \texttt{NODE}\}$ and a length $len \in \{\texttt{EVEN}, \texttt{ODD}\}$ is defined as

$$pos + len := \begin{cases} pos & \text{if } len = \texttt{EVEN} \\ \texttt{EDGE} & \text{if } len = \texttt{ODD} \wedge pos = \texttt{NODE} \\ \texttt{NODE} & \text{if } len = \texttt{ODD} \wedge pos = \texttt{EDGE} \end{cases}$$

Once the position of all elements in the abstract syntax tree has been computed, the remaining check is to verify that no `DirectableRPE` (which can be an `AtomicRPE` or a `PredicatesRPE`) appears at `NODE` position and is directed by `<` or `>` (i.e. the direction of its parent `DirectedRPE` must be equal to `DIRECTION.UNDIRECTED`).

If this check fails, an `RPEDirectionException` like shown in Example 6 is thrown.

3.1.5 *Namespace Resolution*

So far, the order of the presented passes within the syntactic and semantic analysis was determined by the way they rely on each other. The *namespace resolution* of `AtomicRPEs` in contrast can be done at any stage during this analysis.

Each namespace prefix that is used in a RPL query has to be declared within the RDF data against it is to be evaluated. Otherwise, an `RPENamespaceException` will be thrown. If using RPL as embedded language (Chapter 4), the namespace prefix declarations from the enclosing host query will be taken into account instead.

Example 7
Namespace resolution fails with an error message, if any undeclared namespace prefix is used. In the expression below, the prefix `rdfs:` has not been declared in the corresponding RDF data.

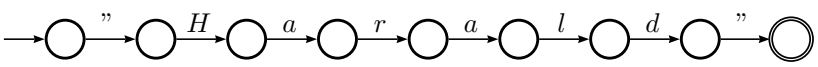
```
Namespace prefix "rdfs:" cannot be resolved:

PATH _ >rdfs:range _
        ^^^^^^^^^^
```

Apart from namespace resolution, a second task is done during this pass: the generation of (ordinary) regular expressions from all kinds (`LITERAL`, `WILDCARD`, `IRI`, ..., see Figure 2) of `AtomicRPEs`.

This is possible, since regular expressions are powerful enough to embed all those kinds. Furthermore, matching strings against a regular expression created from e.g. a ⟨*STRING_LITERAL1*⟩ token (Definition 1) has the same complexity as comparing these two strings character by character, as the resulting automaton is already deterministic.

Example 8
The string literal `"Harald"` (interpreted as a regular expression) translates to the following automaton.



3.2 SEMANTICS OF RPL

In this section, we define the semantics of RPL. A compositional semantics of RPL has already been given in [9]. We have chosen not to adapt these definitions to the current version of RPL, as their correctness has not been shown. Instead, we give a purely set-based semantics of RPL.

Definition 9 (Semantics of RPL)

The semantics of a path-flavored RPE f over an RDF graph G (Definition 2) is given as $[\![f]\!]_G$, where $[\![\cdot]\!]_G$ is inductively defined as follows. The notation used within the following boxes is explained in Section 2.2.2.

Let a be an `AtomicRPE`, $pos := a.getPosition()$, $rx := a.getRegExp()$, and dir a `DIRECTION`.
Case pos = `NODE`
$[\![a]\!]_G := \{(n,n) \mid \exists n \in nodes(G) \wedge n \in \mathcal{L}(rx)\}$
Case pos = `EDGE`

$$[\![a]\!]_G^{dir} := \begin{cases} \{(s,o) \mid \exists p : (s,p,o) \in G \wedge p \in \mathcal{L}(rx)\} & \text{if } dir = \texttt{FORWARD} \\ \{(o,s) \mid \exists p : (s,p,o) \in G \wedge p \in \mathcal{L}(rx)\} & \text{if } dir = \texttt{BACKWARD} \\ [\![a]\!]_G^{\texttt{FORWARD}} \cup [\![a]\!]_G^{\texttt{BACKWARD}} & \text{if } dir = \texttt{UNDIRECTED} \end{cases}$$

Let p be a `PredicateRPE`, $f := p.getFlavored()$, $pos := p.getPosition()$, and $s := p.getSign()$.
Case pos = `NODE`

$$[\![p]\!]_G := \begin{cases} \{(n,n) \mid n \in nodes(G) \wedge \exists q : (n,q) \in [\![f]\!]_G\} & \text{if } s = \texttt{POSITIVE} \\ \{(n,n) \mid n \in nodes(G) \wedge \nexists q : (n,q) \in [\![f]\!]_G\} & \text{if } s = \texttt{NEGATIVE} \end{cases}$$

Case pos = `EDGE`

$$[\![p]\!]_G := \begin{cases} \{(e,e) \mid e \in edges(G) \wedge \exists q : (e,q) \in [\![f]\!]_G\} & \text{if } s = \texttt{POSITIVE} \\ \{(e,e) \mid e \in edges(G) \wedge \nexists q : (e,q) \in [\![f]\!]_G\} & \text{if } s = \texttt{NEGATIVE} \end{cases}$$

Let p be a `PredicatesRPE`, $p_1, p_2, \ldots, p_n$ $(n \geq 1)$ its children, $pos := p.getPosition()$, and dir a `DIRECTION`.
Case pos = `NODE`
$[\![p]\!]_G := \{(n,n) \mid n \in nodes(G) \wedge (n,n) \in [\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G\}$
Case pos = `EDGE`
$[\![p]\!]_G^{dir} :=$

$$\begin{cases} \{(s,o) \mid \exists (s,e,o) \in G \wedge (e,e) \in [\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G\} & \text{if } dir = \texttt{FORWARD} \\ \{(o,s) \mid (s,o) \in [\![p]\!]_G^{\texttt{FORWARD}}\} & \text{if } dir = \texttt{BACKWARD} \\ [\![p]\!]_G^{\texttt{FORWARD}} \cup [\![p]\!]_G^{\texttt{BACKWARD}} & \text{if } dir = \texttt{UNDIRECTED} \end{cases}$$

Let d be a `DirectedRPE` and $direction := d.getDirection()$.
$[\![d]\!]_G := [\![d.getDirectable()]\!]_G^{direction}$

Let d be a `DisjunctiveRPE` and $c_1, c_2, \ldots, c_n (n \geq 1)$ its children.
$[\![d]\!]_G := [\![c_1]\!]_G \cup [\![c_2]\!]_G \cup \ldots \cup [\![c_n]\!]_G$

Let a be an `AdornedRPE`, $ad := a.getAdornable()$, and $mult := a.getMultiplicity()$.

$$[\![a]\!]_G := \begin{cases} [\![ad]\!]_G & \text{if } mult = \texttt{ONE} \\ \{(v,v) \mid v \in terms(G)\} \cup [\![ad]\!]_G & \text{if } mult = \texttt{OPT} \\ \{(v,v) \mid v \in terms(G)\} \cup [\![ad+]\!]_G & \text{if } mult = \texttt{STAR} \\ [\![ad]\!]_G \cup [\![ad]\!]_G \circ [\![ad]\!]_G \cup \ldots & \text{if } mult = \texttt{PLUS} \end{cases}$$

Let c be a `ConcatenatedRPE` and $a_1, a_2, \ldots, a_n (n \geq 1)$ its children.
$[\![c]\!]_G := [\![a_1]\!]_G \circ [\![a_2]\!]_G \circ \ldots \circ [\![a_n]\!]_G$

Let $flavored$ be a `FlavoredRPE`.
$[\![flavored]\!]_G := [\![flavored.getConcatenated()]\!]_G$

with $X \circ Y := \{(x,z) \mid \exists y : (x,y) \in X \wedge (y,z) \in Y\}$ (this operator is used for `AdornedRPEs`, `ConcatenatedRPEs` and `PredicatesRPEs`).

3.3 ENRE-BASED EVALUATION

NREs (Nested Regular Expressions) have been presented in [27] as a means to describe and query regular paths in RDF graphs. The proposed graph labeling

algorithm for the evaluation of NREs has been shown to have polynomial combined, and linear data and query time complexity, when considering the complexity of the associated decision problem.

In this thesis, we present ENREs, an extended version of NREs designed to address the characteristics of RPL: regular expressions (over ordinary strings) as node and edge label tests, as well as the negation of predicates (which are called nested expressions in the case of ENREs).

The syntax and semantics of ENREs is given in Section 3.3.1.

In Section 3.3.2, we show that RPL queries can be translated into ENREs in linear time, and that this translation is correct in regard to the semantics of RPL (Section 3.2) and ENREs. It is also shown that the semantics of ENREs, which have been constructed by translating a RPL query, is a subset of $nodes(G) \times nodes(G)$, with G being an RDF graph (Definitions 2 and 3).

In Section 3.3.3, we present an adapted version of the graph labeling algorithm for NREs, which relies on constructing product automata from the underlying RDF graph for each nested expression of an ENRE.

Finally, we formally prove that the time complexity (i.e. the complexity which is actually needed when constructing the result set of a query, see [1] — rather than the complexity of just deciding whether a certain pair of RDF terms is in the result set or not) is quadratic in the size of the data and linear in the size of the query, as is the case for NREs [27]. Hence, evaluating RPL by using ENREs as an intermediate language is as efficient as evaluating NREs (which are less expressive).

3.3.1 *Syntax and Semantics of ENREs*

Definition 10 (Abstract Syntax of ENREs)
The abstract syntax of ENREs is defined by the following grammar.

⟨*enre*⟩ ::= ⟨*enre*⟩ '**/**' ⟨*enre*⟩ |
⟨*enre*⟩ '**|**' ⟨*enre*⟩ |
⟨*enre*⟩ ('**?**' | '*****' | '**+**') |
⟨*axis*⟩ |
⟨*axis*⟩ '**::**' ⟨*regexp*⟩ |
⟨*axis*⟩ '**::**' ⟨*nested*⟩

⟨*nested*⟩ ::= '**!**'? '**[**' ⟨*enre*⟩ '**]**'

⟨*axis*⟩ ::= '**next**' | '**next**$^{-1}$' | '**next_or_next**$^{-1}$' |
'**self_node**' | '**self_edge**'

Definition 10 gives just an abstract syntax, hence it is not targeted at parsing ENREs. The optionality operator and the Kleene star and plus operators (`?`, `*`, and `+`) bind more strongly than the concatenation operator `/`, which in turn binds more strongly than the alternative operator `|`. Parentheses will be used, whenever these precedence rules need to be overridden.

Definition 10 both extends and limits the definition given in [27].

- The *navigation axes* (see the ⟨*axis*⟩ rule) **`self`**, **`edge`**, **`edge`**$^{-1}$, **`node`**, and **`node`**$^{-1}$ are omitted, as they are not needed when evaluating RPL. On the other hand, new axes are introduced to express the peculiarities of RPL: **`self_node`** and **`self_edge`** act like **`self`**, but filter for nodes and edges, respectively. The new axis **`next_or_next`**$^{-1}$ is used for undirected edge navigation and is satisfied iff **`next`** or **`next`**$^{-1}$ holds.
- Like RPEs, ENREs support the negation (indicated by `!`) of *nested expressions* (see the ⟨*nested*⟩ rule), which are called predicates in RPL (both nested expressions and predicates are enclosed by brackets).
- Via the ⟨*regexp*⟩ rule, ENREs allow to use regular expressions (over ordinary strings) for node and edge label tests, while NREs only allowed for IRIs.

Example 9

Let us consider the following example query, to be evaluated on an RDF graph G (Definition 2).

```
self_node:::Paris/next::![self_edge::TGV]
```

It expresses a path that starts at a node of G (because of the **`self_node`** axis) labeled with `:Paris`, followed by a **`next`** move. The **`next`** axis moves from the subject to the object of an RDF triple (Definition 2), and is able to express restrictions on its predicate (see also Figure 28). In our case, this restriction consists of the nested expression **`![self_edge::TGV]`**, which matches every edge of G *not* (indicated by `!`) labeled with `:TGV`.

Evaluating this ENRE on the RDF graph from Figure 6 will lead to the result set {(`:Paris`, `:France`)}, as the node `:Paris` has an outgoing edge labeled with `:country` that reaches the node `:France`. The other two outgoing edges of `:Paris`, labeled with `:TGV`, are not followed as they do not satisfy the nested expression **`![self_edge::TGV]`**.

In the preceding example, we have already (informally) come across the semantics of ENREs, which is now formally given in Definition 11.

Definition 11 (Semantics of ENREs)

The semantics of an ENRE exp over an RDF graph G is given as $[\![exp]\!]_G$, where $[\![\cdot]\!]_G$ is inductively defined as follows. In general, the result of $[\![\cdot]\!]_G$ is a subset of $terms(G) \times terms(G)$ (Definition 2).

$$[\![\texttt{self\_node}]\!]_G := \{(n,n) \mid n \in nodes(G)\}$$
$$[\![\texttt{self\_node::}\alpha]\!]_G := \{(n,n) \mid n \in nodes(G) \wedge cond^G_\alpha(n)\}$$

$$[\![\texttt{self\_edge}]\!]_G := \{(p,p) \mid p \in edges(G)\}$$
$$[\![\texttt{self\_edge::}\alpha]\!]_G := \{(p,p) \mid p \in edges(G) \wedge cond^G_\alpha(p)\}$$

$$[\![\texttt{next}]\!]_G := \{(s,o) \mid \exists p : (s,p,o) \in G\}$$
$$[\![\texttt{next::}\alpha]\!]_G := \{(s,o) \mid \exists p : (s,p,o) \in G \wedge cond^G_\alpha(p)\}$$

$$[\![\texttt{next}^{-1}]\!]_G := \{(o,s) \mid (s,o) \in [\![\texttt{next}]\!]_G\}$$
$$[\![\texttt{next}^{-1}\texttt{::}\alpha]\!]_G := \{(o,s) \mid (s,o) \in [\![\texttt{next::}\alpha]\!]_G\}$$

$$[\![\texttt{next\_or\_next}^{-1}]\!]_G := [\![\texttt{next}]\!]_G \cup [\![\texttt{next}^{-1}]\!]_G$$
$$[\![\texttt{next\_or\_next}^{-1}\texttt{::}\alpha]\!]_G := [\![\texttt{next::}\alpha]\!]_G \cup [\![\texttt{next}^{-1}\texttt{::}\alpha]\!]_G$$

$$\text{where } cond^G_\alpha(x) := \begin{cases} x \in \mathcal{L}(regexp) & \text{if } \alpha = regexp \\ \exists q \in terms(G) : (x,q) \in [\![exp]\!]_G & \text{if } \alpha = \texttt{[}exp\texttt{]} \\ \nexists q \in terms(G) : (x,q) \in [\![exp]\!]_G & \text{if } \alpha = \texttt{![}exp\texttt{]} \end{cases}$$

$$[\![exp_1\texttt{|}exp_2]\!]_G := [\![exp_1]\!]_G \cup [\![exp_2]\!]_G$$
$$[\![exp_1\texttt{/}exp_2]\!]_G := [\![exp_1]\!]_G \circ [\![exp_2]\!]_G, \text{ where}$$
$$X \circ Y := \{(x,z) \mid \exists y : \ (x,y) \in X \wedge (y,z) \in Y\}$$

$$[\![exp\texttt{?}]\!]_G := [\![\texttt{self\_node}]\!]_G \cup [\![\texttt{self\_edge}]\!]_G \cup [\![exp]\!]_G$$
$$[\![exp\texttt{*}]\!]_G := [\![\texttt{self\_node}]\!]_G \cup [\![\texttt{self\_edge}]\!]_G \cup [\![exp\texttt{+}]\!]_G$$
$$[\![exp\texttt{+}]\!]_G := [\![exp]\!]_G \cup [\![exp\texttt{/}exp]\!]_G \cup [\![exp\texttt{/}exp\texttt{/}exp]\!]_G \cup \ldots$$

3.3.2 *Translating RPL into ENREs*

After having defined the syntax and semantics of ENREs, the missing link between RPL and ENREs is established: a translation function *trans* that maps RPEs to ENREs. This translation should be easy to compute (i.e. in linear time of the length of its source query), and, of course, correct in the sense that the semantics of a RPL query q (Definition 9) coincides with the semantics of $trans(q)$ (Definition 11).

The following Example might give an idea of how such a translation can be constructed.

EXAMPLE 10
Consider the following RPL query q

`PATH _ >`p `_` with $p :=$ `[PATH (_ >rdfs:subPropertyOf)* :transport]`

During semantic analysis (Section 3.1), the `AtomicRPEs` `rdfs:subPropertyOf` and `:transport` got enriched with their corresponding regular expressions $r_1 :=$ `http://www.w3.org/2000/01/rdf-schema#subPropertyOf` and $r_2 :=$ `http://example.org/transport`, respectively.

The translation function (Definition 13) will return the following ENRE e

`self_node/next::[`α`]/self_node` with
$\alpha :=$ `self_edge::[`β`]`
$\beta :=$ `(self_node/next::`r_1`)*/self_node::`r_2

The ENRE β corresponds to `PATH (_ >rdfs:subPropertyOf)* :transport`, the flavored child of q's predicate p. p appears at `EDGE` position within q, so β is used as a nested expression to restrict the `self_edge` move, and thus we get to the ENRE α.

q itself just consists of a wildcard (at `NODE` position) that has an outgoing edge (at `EDGE` position) satisfying p to another wildcard (again at `NODE` position). Each of the wildcards can be translated via a `self_node` move, and jumping from the first to second wildcard is done via a `next` move satisfying α (see also Figure 28).

Having seen this example translation, we continue by giving a formal definition of the translation function *trans* (Definition 13), preceded by its helper function *getAxis* (Definition 12).

Definition 12 (Helper Function *getAxis*)

The helper function *getAxis*, which is used in the following translation function, maps a position (Section 3.1.4) and a `DIRECTION` (`FORWARD`, `BACKWARD`, or `UNDIRECTED`) to an ENRE navigation axis as follows.

$$getAxis(pos, dir) := \begin{cases} \texttt{self\_node} & \text{if } pos = \texttt{NODE} \\ \texttt{next} & \text{if } pos = \texttt{EDGE} \wedge dir = \texttt{FORWARD} \\ \texttt{next}^{-1} & \text{if } pos = \texttt{EDGE} \wedge dir = \texttt{BACKWARD} \\ \texttt{next\_or\_next}^{-1} & \text{if } pos = \texttt{EDGE} \wedge dir = \texttt{UNDIRECTED} \end{cases}$$

Within *trans*, this helper function is used in the case of `DirectableRPEs` (which can be `AtomicRPEs` and `PredicatesRPEs`, compare Figure 2) in order to determine the appropriate ENRE navigation axis.

Definition 13 (Translation Function *trans*)

With the help of Definition 12, we inductively define a translation function *trans* that maps an RPE to an ENRE. The notation used within the following boxes is explained in Section 2.2.2.

Let a be an `AtomicRPE` and dir a `DIRECTION`.
$trans^{dir}(a) := getAxis(a.getPosition(), dir)\texttt{::}a.getRegExp()$

Let p be a `PredicateRPE`, $pos := p.getPosition()$, $f := p.getFlavored()$, and $s := p.getSign()$.

$$trans(p) := \begin{cases} axis\texttt{::[}trans(f)\texttt{]} & \text{if } s = \texttt{SIGN.POSITIVE} \\ axis\texttt{::![}trans(f)\texttt{]} & \text{else} \end{cases}$$

$$\text{where } axis := \begin{cases} \texttt{self\_node} & \text{if } pos = \texttt{NODE} \\ \texttt{self\_edge} & \text{if } pos = \texttt{EDGE} \end{cases}$$

Let p be a `PredicatesRPE`, $p_1, p_2, \ldots, p_n (n \geq 1)$ its children, $pos := p.getPosition()$, and dir a `DIRECTION`.
$trans^{dir}(p) := getAxis(pos, dir)\texttt{::[}trans(p_1)\texttt{/}trans(p_2)\texttt{/}\ldots\texttt{/}trans(p_n)\texttt{]}$

Let d be a `DirectedRPE` and $direction := d.getDirection()$.
$trans(d) := trans^{direction}(d.getDirectable())$

Let d be a `DisjunctiveRPE` and $c_1, c_2, \ldots, c_n (n \geq 1)$ its children.
$trans(d) := trans(c_1)\texttt{|}trans(c_2)\texttt{|}\ldots\texttt{|}trans(c_n)$

Let a be an `AdornedRPE`, $adornable := a.getAdornable()$, and $mult := a.getMultiplicity()$.

$$trans(a) := \begin{cases} trans(adornable) & \text{if } mult = \texttt{MULTIPLICITY.ONE} \\ trans(adornable)\texttt{?} & \text{if } mult = \texttt{MULTIPLICITY.OPT} \\ trans(adornable)\texttt{*} & \text{if } mult = \texttt{MULTIPLICITY.STAR} \\ trans(adornable)\texttt{+} & \text{if } mult = \texttt{MULTIPLICITY.PLUS} \end{cases}$$

Let c be a `ConcatenatedRPE` and $a_1, a_2, \ldots, a_n (n \geq 1)$ its children.
$trans(c) := trans(a_1)\texttt{/}trans(a_2)\texttt{/}\ldots\texttt{/}trans(a_n)$

Let $flavored$ be a `FlavoredRPE`.
$trans(flavored) := trans(flavored.getConcatenated())$

The remaining part of this section shows that the semantics of a valid RPL query over an RDF graph G always evaluates to a subset of $nodes(G) \times nodes(G)$ (Theorem 1 and Corollary 1), and the correctness of the translation function $trans$ (Theorem 2).

The following Lemmas 1 and 2 serve as preparation for Theorem 1.

Lemma 1

Let d be a `DirectableRPE` that appears at `NODE` position, i.e. $d.getPosition() =$ `NODE`. Then $[\![trans(d)]\!]_G \subseteq nodes(G) \times nodes(G)$, with G an RDF graph and $trans$ as in Definition 13.

Proof. d can either be an `AtomicRPE` or a `PredicatesRPE` (Figure 2).

Case d is an `AtomicRPE`
Then, by Definitions 12 and 13, $trans(d) =$ `self_node::`$d.getRegExp()$. By Definition 11, $[\![$`self_node::`$d.getRegExp()]\!]_G \subseteq \{(n,n) \mid n \in nodes(G)\} \subseteq nodes(G) \times nodes(G)$.

Case d is a `PredicatesRPE`
In this case, $trans(d) =$ `self_node::[`$trans(p_1)\texttt{/}trans(p_2)\texttt{/}\ldots\texttt{/}trans(p_n)$`]`, where $p_1, \ldots, p_n$ are the children of d (by Definitions 12 and 13). Then, by Definition 11, $[\![trans(d)]\!]_G \subseteq \{(n,n) \mid n \in nodes(G)\} \subseteq nodes(G) \times nodes(G)$. □

Lemma 2

Let c be a `ConcatenatedRPE` and $a_1, \ldots, a_n$ $(n \geq 1)$ be the adorned children of c. If there exists an $i \in \{1, \ldots, n\}$ such that the ENRE-based semantics of

a_i contains just pairs of nodes of an RDF graph G, the same applies to c, i.e. $[\![trans(a_i)]\!]_G \subseteq nodes(G) \times nodes(G) \Rightarrow [\![trans(c)]\!]_G \subseteq nodes(G) \times nodes(G)$.

Proof. If $n = 1$, we are finished as $trans(c) = trans(a_1)$ by Definition 13, and thus $[\![trans(c)]\!]_G = [\![trans(a_1)]\!]_G \subseteq nodes(G) \times nodes(G)$ by assumption.

Let $n > 1$. Applying the rules from Definitions 11 and 13, we get $[\![trans(c)]\!]_G = [\![trans(a_1)]\!]_G \circ \ldots \circ [\![trans(a_i)]\!]_G \circ \ldots \circ [\![trans(a_n)]\!]_G$, where the *concatenation operator* $\circ$ is defined as $X \circ Y := \{(x,z) \mid \exists y : (x,y) \in X \wedge (y,z) \in Y\}$. As for all $j \in \{1,\ldots,n\}$, $[\![trans(a_j)]\!]_G$ never contains a pair (e,n) or (n,e) with $e \in edges(G) \setminus nodes(G)$ and $n \in nodes(G) \setminus edges(G)$ (there is no case in Definition 11, where such pairs are returned), $\circ$ only connects nodes with nodes and edges and edges. As $[\![trans(a_i)]\!]_G$ is a subset of $nodes(G) \times nodes(G)$, also $[\![trans(a_{i-1})]\!]_G \circ [\![trans(a_i)]\!]_G$ (if $i > 1$) and $[\![trans(a_i)]\!]_G \circ [\![trans(a_{i+1})]\!]_G$ (if $i < n$) are subsets of $nodes(G) \times nodes(G)$. By induction, it follows that $[\![trans(c)]\!]_G \subseteq nodes(G) \times nodes(G)$. □

THEOREM 1

Let f be a valid `FlavoredRPE`, i.e. f has passed all checks of the semantic analysis phase (Section 3.1). The ENRE-semantics of f returns pairs of nodes of an RDF graph G, i.e. $[\![trans(f)]\!]_G \subseteq nodes(G) \times nodes(G)$.

Proof. By assumption, f has `ODD` length (the length verification pass would have failed otherwise, see Section 3.1.3). Let c be the concatenated child of f. As the length of f is defined as the length of its child c, c has also `ODD` length (Definition 7).

Since the length of c is `ODD`, it must have at least one adorned child, whose length is also `ODD` (by Definition 7). Furthermore, its multiplicity is `ONE`, as each other multiplicity would imply `EVEN` length (also by Definition 7).

Case There exists exactly one such adorned child a
a appears at `NODE` position, since all other adorned children have `EVEN` length in this case. However, adding `EVEN` arbitrarily many times to the position `NODE` (which is the initial position of f and c) has no effect at all (Section 3.1.4).

Case There exist several such adorned children
Let a denote the first of these adorned children. In this case, all adorned children of c which appear left of a have `EVEN` length. With the same argument as above, a appears at `NODE` position.

This `AdornedRPE` a has an adornable child, which is a directed or disjunctive subexpression.

Case *a* has a directed child
This directed child is the parent of a directable subexpression d. By Lemma 1, $[\![trans(d)]\!]_G \subseteq nodes(G) \times nodes(G)$.

Case *a* has a disjunctive child
This disjunctive child d consists of at least two concatenated children (this is due to the simplification pass, see Section 3.1.1) – each of them has `ODD` length, and appears at `NODE` position. So, the argument given above for the outermost concatenated expression c can be repeated for each concatenated child of d. Hence, by induction, Lemma 2 can be applied on all children $child_1, \ldots, child_n$ of d yielding $[\![trans(child_i)]\!]_G \subseteq nodes(G) \times nodes(G)$ for all $i \in \{1, \ldots, n\}$. By Definitions 13 and 11, it follows that $[\![trans(d)]\!]_G = \bigcup_{i=1}^{n} [\![trans(child_i)]\!]_G \subseteq nodes(G) \times nodes(G)$.

As a's multiplicity is `ONE`, $[\![trans(a)]\!]_G = [\![trans(d)]\!]_G$ by Definition 13. As a is a child of c, we can now apply Lemma 2 on c, which (together with Definition 13) yields $[\![trans(f)]\!]_G = [\![trans(c)]\!]_G \subseteq nodes(G) \times nodes(G)$, which is what we wanted to show. □

THEOREM 2
Let q be a valid `RPE` and G an arbitrary RDF graph. Its semantics $[\![q]\!]_G$ is equal to translating q into an ENRE and then applying the ENRE-semantics $[\![trans(q)]\!]_G$, i.e. $[\![q]\!]_G = [\![trans(q)]\!]_G$.

Proof. By structural induction, see Section A.1. □

COROLLARY 1
Let f be a valid `FlavoredRPE`. The semantics of f over an RDF graph G, $[\![f]\!]_G$ is a subset of $nodes(G) \times nodes(G)$.

Proof. This is a direct consequence of Theorems 1 and 2. □

3.3.3 *Graph Labeling Algorithm*

The graph labeling algorithm presented in this section is based on the graph labeling algorithm for NREs [27], but has been adapted for ENREs. More precisely, it supports regular expressions over strings and the negation of nested expressions, as well as the additional navigation axes available for ENREs (Section 3.3.1).

The basic idea of the algorithm is that both an RDF graph G and an ENRE exp can be interpreted as NFAs (Definitions 14, 15, and 17) and combined together into a product automaton (Definition 18). Such a product automaton is not only built from G and exp, but from G and all nested expressions of exp in a bottom-up manner (i.e. the product automaton belonging to the innermost nested expression of exp is constructed first). Each node or edge of G gets labeled with all nested expressions that it satisfies (hence the name "graph labeling algorithm"), and these labels are taken into account for constructing all outer product automata. Each constructed product automaton $\mathcal{P}$ is decomposed into its strongly connected components by Tarjan's algorithm (Section A.2), in order to efficiently compute all initial states from which final states are reached in $\mathcal{P}$.

Finally, all these steps are combined into a `LABEL` (Listing 4) and an `EVAL` (Listing 5) method, which together implement the *Graph Labeling Algorithm*.

Definition 14 (Nondeterministic Finite Automaton)

An NFA (Nondeterministic Finite Automaton) $\mathcal{A}$ is a 5-tuple $(S, \Sigma, \delta, I, F)$ consisting of a finite set of states S, a finite set of input symbols (i.e. the alphabet) Σ with $S \cap \Sigma = \emptyset$, a transition function $\delta : S \times \Sigma \to 2^S$, a set of initial states $I \subseteq S$, and a set of final states $F \subseteq S$. By 2^S, we denote the power set of S. An ε-NFA is an NFA that also allows for ε transitions, i.e. $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$.

An alternative definition of NFAs, which only allows for a single initial state q_0, is given in [21]. However, Definition 14 (also given in [28]) is more convenient in our case, as Definition 15 will show.

As the RDF graph from Figure 6 is too large to illustrate the following constructions, we hence rely on the RDF graph from the following Figure 12 for that purpose.

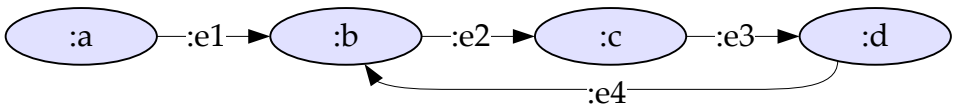


Figure 12: A simple RDF graph

Definition 15 (Interpreting an RDF Graph as NFA)

Let G be an RDF graph. The corresponding NFA $\mathcal{G}$ is defined as $\mathcal{G} := (terms(G), E, \delta_{\mathcal{G}}, terms(G), terms(G))$. E is the set of input symbols and contains ENREs of the form $\langle axis \rangle$`::`$\langle regexp \rangle$, where $\langle axis \rangle \in \{$`self_node`, `self_edge`,

next, **next**$^{-1}$} and $\langle regexp \rangle$ is an arbitrary element of $terms(G)$. The transition function $\delta_{\mathcal{G}}$ is defined as follows.

$$\delta_{\mathcal{G}}(v, e) := \begin{cases} \{v\} & \text{if } v \in nodes(G) \wedge e = \textbf{self\_node::}v \\ \{v\} & \text{if } v \in edges(G) \wedge e = \textbf{self\_edge::}v \\ \{o \mid (v, p, o) \in G\} & \text{if } e = \textbf{next::}p \\ \{s \mid (s, p, v) \in G\} & \text{if } e = \textbf{next}^{-1}\textbf{::}p \end{cases}$$

For later complexity considerations, we define the size of $\mathcal{G}$ as the sum of the string lengths of its transition labels, i.e. $|\mathcal{G}| := \sum_{(v,l,v') \in \delta_{\mathcal{G}}} |l|$.

Example 11

Figure 13 shows the NFA $\mathcal{G}$ corresponding to the RDF graph presented in Figure 12. Each state of $\mathcal{G}$ is both an initial and final state.

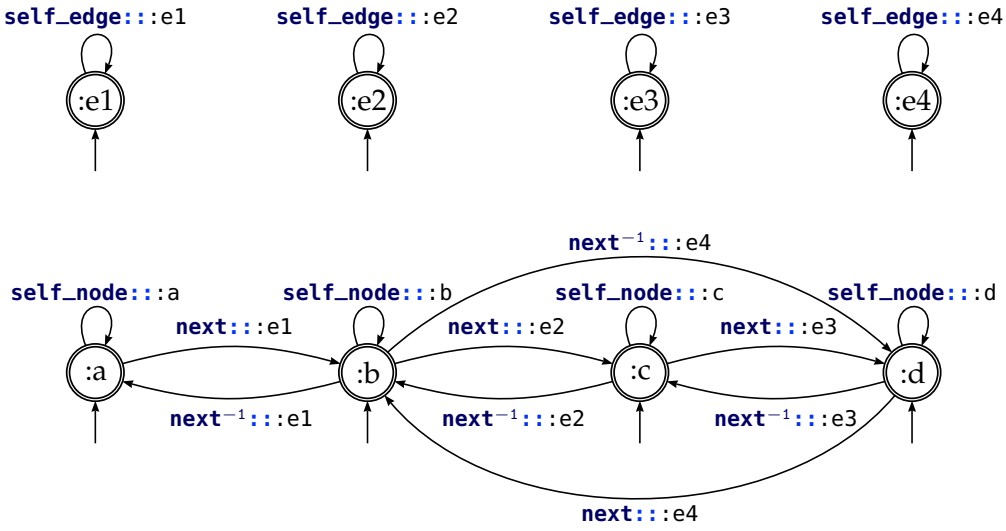


Figure 13: Interpreting the RDF graph from Figure 12 as an NFA

With Definition 15, we already have the first ingredient for the construction of the product automaton (Definition 18). The second ingredient is an ENRE that is converted into another NFA by Thompson's construction [32]. Prior to presenting this construction in Definition 17, we have to recall that an ENRE (Extended Nested *Regular Expression*) exp is indeed a regular expression, namely over the alphabet $\mathbf{D}_0(exp)$, as given in the following Definition 16.

Definition 16 (Depth-0 Terms of an ENRE)

The set of *depth*-0 *terms* of an ENRE exp is inductively defined as follows.

$$\mathbf{D}_0(exp) := \begin{cases} \mathbf{D}_0(e_1) \cup \mathbf{D}_0(e_2) & \text{if } exp = e_1/e_2, \text{ or } exp = e_1|e_2 \\ \mathbf{D}_0(e) & \text{if } exp = e?, exp = e*, \text{ or } exp = e+ \\ \{exp\} & \text{otherwise} \end{cases}$$

It is important to see that $\mathbf{D}_0$ does not decompose the nested expressions of exp (for these, the third case applies).

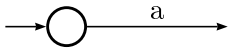
Definition 17 (Interpreting an ENRE as NFA)

Let exp be an ENRE. exp can be interpreted as a regular expression over the alphabet $\mathbf{D}_0(exp)$, which in turn can be translated into an ε-NFA $\mathcal{A}_{exp} = (Q, \mathbf{D}_0(exp), \delta_{\mathcal{A}}, I, F)$.

This translation known as Thompson's construction has already been presented in [32] and [13] for regular expressions over an arbitrary alphabet. It differs from the well-known illustration in [21], as no temporary final states (i.e. states which are not final in the end result) are created during the translation process, which makes it straightforward to implement.

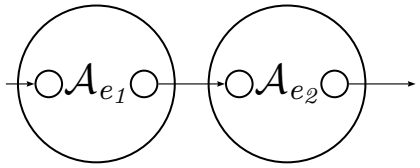
Instead of constructing a complete ε-NFA in each single translation step, we just construct *partial* ε-NFAs for each subexpression. Partial ε-NFAs have no final states, but dangling arrows that are pointing at nothing. After the outermost partial ε-NFA has been constructed, each of its dangling arrows will be connected to a final state, which finally turns it into an ε-NFA.

Let a be an ENRE of the form $\langle axis \rangle$, $\langle axis \rangle$::$\langle regexp \rangle$, or $\langle axis \rangle$::$\langle nested \rangle$. The partial ε-NFA for a is

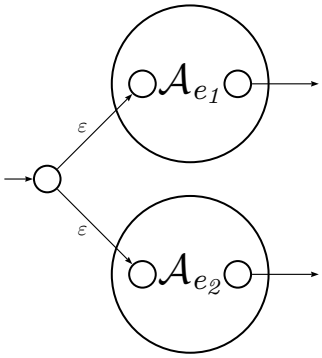


In the following boxes, $\mathcal{A}_e$ denotes the (recursively constructed) partial NFA that belongs to a subexpression e. The inner state drawn left to the label $\mathcal{A}_e$ is $\mathcal{A}_e$'s initial state. The inner state drawn right to the label $\mathcal{A}_e$ is a fictitious state, just as if all of $\mathcal{A}_e$'s dangling arrows originated from this single state.

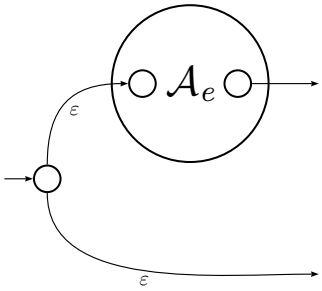
Let e_1 and e_2 be two ENREs. The concatenation e_1/e_2 is translated into the partial ε-NFA



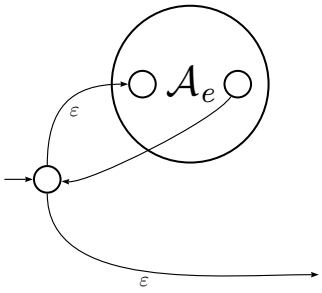
Let e_1 and e_2 be two ENREs. The alternative $e_1|e_2$ is translated into the partial ε-NFA



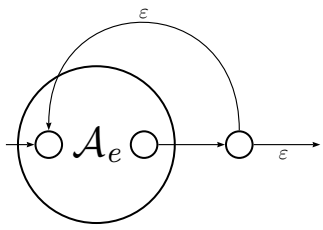
Let e be an ENRE. The optionality $e?$ is translated into the partial ε-NFA



Let e be an ENRE. The Kleene star expression $e*$ is translated into the partial ε-NFA



Let e be an ENRE. The Kleene plus expression $e+$ is translated into the partial ε-NFA



For later complexity considerations (Section 3.3.4), we define the size of $\mathcal{A}_{exp}$ as the sum of the sizes of all its transition labels, i.e.

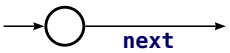
$$|\mathcal{A}_{exp}| := \sum_{(q,l,q') \in \delta_{\mathcal{A}}} |l|$$

where $|l|$ denotes the string length of the transition label l, except in the following two cases. (1) The length of an ε-transition is defined as one, i.e. $|\varepsilon| := 1$, and (2) the length of transitions labeled with nested expressions exp' of the form *axis*`::`*nested* only includes the length of its navigation axis *axis*, i.e. $|exp'| := |axis|$. The latter exception is motivated by the construction of $\mathcal{A}_{exp}$, as nested expressions are treated as single symbols and not further split up. This coincides with the definition of $\mathbf{D}_0$ (Definition 16), which also does not decompose nested expressions.

Example 12

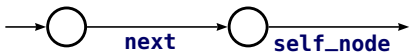
Let us consider the ENRE `self_node::a/(next/self_node)+`. We construct its corresponding NFA $\mathcal{A}_{exp}$ according to Definition 17, in a bottom-up manner.

`next` translates to the partial NFA (1)

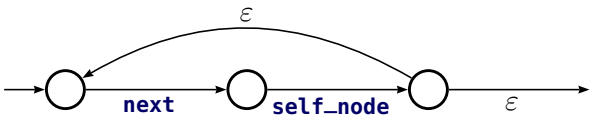


`self_node` translates to the partial NFA (2)

`next/self_node` translates to the partial NFA (3), combining its children (1) and (2)

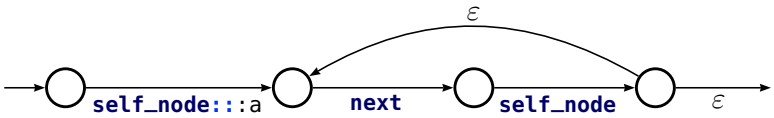


`(next/self_node)+` translates to the partial NFA (4), using its child (3)

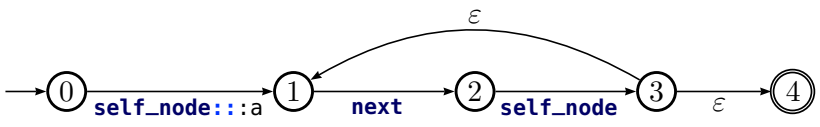


`self_node::a` translates to the partial NFA (5)

`self_node::a/(next/self_node)+` translates to the partial NFA (6) by combining its children (4) and (5)



Finally, the partial NFA (6) is converted into the ε-NFA below. The states have been numbered, as this automaton will be reused in Example 13.



We have seen how to interpret both an RDF graph and an ENRE as NFAs (Definitions 15 and 17). The resulting NFAs $\mathcal{G}$ and $\mathcal{A}_{exp}$ are now combined into a product automaton. This product automaton $\mathcal{P}$ should only perform a transition, if both underlying transitions of $\mathcal{G}$ and $\mathcal{A}_{exp}$ "match".

Definition 18 (Product Automaton)

Let $\mathcal{G} := (terms(G), E, \delta_{\mathcal{G}}, terms(G), terms(G))$ be the NFA that corresponds to an RDF graph G (in the sense of Definition 15), and $\mathcal{A}_{exp} = (Q, \mathbf{D}_0(exp), \delta_{\mathcal{A}}, I, F)$ the ε-NFA that corresponds to an ENRE exp (in the sense of Definition 17). The product automaton $\mathcal{P}$ is defined as $\mathcal{P} := (terms(G) \times Q, \mathbf{D}_0(exp), \delta_{\mathcal{P}}, I \times terms(G), F \times terms(G))$. Let $axis \in \{\texttt{next}, \texttt{next}^{-1}, \texttt{next\_or\_next}^{-1}, \texttt{self\_node}, \texttt{self\_edge}\}$, $v \in terms(G)$, and $q \in Q$. $\delta_{\mathcal{P}}$ is defined as follows.

$$\delta_{\mathcal{P}}((v,q), axis) := \bigcup_{\substack{a \in terms(G) \\ axis' \models axis}} \delta_{\mathcal{G}}(v, axis'\texttt{::}a) \times \delta_{\mathcal{A}}(q, axis) \tag{3.1}$$

$$\delta_{\mathcal{P}}((v,q), axis\texttt{::}regexp) := \bigcup_{\substack{a \in \mathcal{L}(regexp) \\ axis' \models axis}} \delta_{\mathcal{G}}(v, axis'\texttt{::}a) \times \delta_{\mathcal{A}}(q, axis\texttt{::}regexp) \tag{3.2}$$

$$\delta_{\mathcal{P}}((v,q), axis\texttt{::[}exp\texttt{]}) := \bigcup_{\substack{a | exp \in labels(a) \\ axis' \models axis}} \delta_{\mathcal{G}}(v, axis'\texttt{::}a) \times \delta_{\mathcal{A}}(q, axis\texttt{::[}exp\texttt{]}) \tag{3.3}$$

$$\delta_{\mathcal{P}}((v,q), axis\texttt{::![}exp\texttt{]}) := \bigcup_{\substack{a | exp \notin labels(a) \\ axis' \models axis}} \delta_{\mathcal{G}}(v, axis'\texttt{::}a) \times \delta_{\mathcal{A}}(q, axis\texttt{::![}exp\texttt{]}) \tag{3.4}$$

$$\delta_{\mathcal{P}}((v,q), \varepsilon) := \{v\} \times \delta_{\mathcal{A}}(q, \varepsilon) \tag{3.5}$$

$\models$ is a binary relation between navigation axes that is defined as follows.

$$axis' \models axis :\Leftrightarrow axis' = axis \vee (axis' \in \{\texttt{next}, \texttt{next}^{-1}\} \wedge axis = \texttt{next\_or\_next}^{-1})$$

The idea behind Equations 3.1 – 3.5 is to ensure that $\mathcal{P}$ has a transition from a source product state to a target product state, iff it coincides with the underlying transitions of $\mathcal{G}$ and $\mathcal{A}_{exp}$.

More precisely, all equations state that, if $\mathcal{A}_{exp}$ has a transition t_1 from a state q to a state q' and $\mathcal{G}$ has a transition t_2 from a state v to a state v', and t_1 and t_2 "match" somehow, then $\mathcal{P}$ should have a transition from the product state (v, q) to the product state (v', q') as well.

Equation 3.1 covers the case that t_1 is labeled just with a navigation axis *axis*, so all matching transitions t_2 must have a navigation axis $axis'$ with $axis' \models axis$. $axis' \models axis$ means that $axis'$ in some way satisfies *axis*, i.e. $axis'$ and *axis* are either equal or in case that $axis =$ **`next_or_next`**$^{-1}$, both **`next`** and **`next`**$^{-1}$ are allowed for $axis'$.

Equation 3.2 covers a more restrictive case, as t_1 is now labeled with an ENRE of the form $axis$`::`$regexp$. For each matching transition t_2 of the form $axis'$`::`a, it must hold that $axis' \models axis$ and that a matches to the regular expression pattern *regexp*, i.e. $a \in \mathcal{L}(regexp)$.

Equation 3.3 deals with transitions t_1 labeled with nested expressions of the form $axis$`::[`exp`]`. Which transitions t_2 of the form $axis'$`::`a coincide with t_1? Of course $axis'$ must satisfy *axis* as in the previous cases. Additionally, a must satisfy the nested expression *exp*, i.e. there must be a pair $(a, \cdot) \in [\![exp]\!]_G$. At this point, we assume that *exp* has already been evaluated (in fact, this is guaranteed by the `LABEL` method, see Listing 4), and, as a consequence, that a has been labeled with *exp* (i.e. $exp \in labels(a)$).

Equation 3.4 is analog to Equation 3.3, but takes care of negated nested expressions of the form $axis$`::![`exp`]`. Again, both axes have to coincide, but, in contrast to Equation 3.3, the nested expression *exp* should not be satisfied, i.e. $exp \notin labels(a)$. It is not necessary to introduce negative labels, in contrast to what has been proposed in [9].

Finally, Equation 3.5 allows for ε-transitions in $\mathcal{A}_{exp}$, i.e. $\mathcal{A}_{exp}$ is allowed to do an ε-transition, while $\mathcal{G}$ does not change at all.

Example 13

We construct the product automaton of $\mathcal{G}$ from Figure 13 and $\mathcal{A}_{exp}$ from Example 12. The result is shown in Figure 14; the color of the first row of states is meant to be ignored for now. It can been seen that $(\texttt{:a}, 0)$ is the only initial state that reaches final states, namely $(\texttt{:b}, 4)$, $(\texttt{:c}, 4)$, and $(\texttt{:d}, 4)$. Hence, we can conclude that the ENRE **`self_node`**`::a/(`**`next/self_node`**`)+` evaluates to $\{(\texttt{:a}, \texttt{:b}), (\texttt{:a}, \texttt{:c}), (\texttt{:a}, \texttt{:d})\}$, which fits our intuition.

As already indicated within Example 13, these product automata are now used to compute the result of an ENRE *exp* over an RDF graph interpreted as NFA $\mathcal{G}$. By the definition of $[\![\cdot]\!]_G$ (Definition 11) and the construction of $\mathcal{P}$

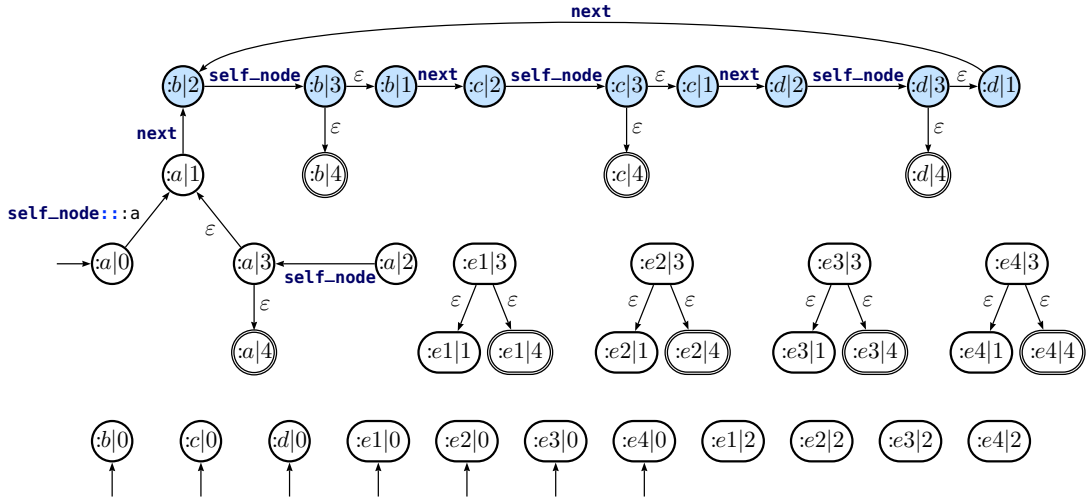


Figure 14: Product automaton $\mathcal{P}$ created by combining Figure 13 and Example 12

(Definition 18), it holds that $(a, b) \in [\![exp]\!]_G$ iff there is an initial state $(a, \cdot)$ from which a final state $(b, \cdot)$ is reached in $\mathcal{P}$. This reachability relation is computed in two phases. In a first phase, Tarjan's algorithm (Section A.2) is applied to decompose $\mathcal{P}$ into its strongly connected components, abbreviated as SCCs from now on. The resulting component graph is a directed, acyclic graph and can be traversed via a depth-first search in a second phase. During this traversal, each SCC s is enriched with a set containing the first component of all final states of $\mathcal{P}$ that are reachable from s, and this reachability information is used to construct the result.

Let us reuse the product automaton $\mathcal{P}$ from Example 13. The first row of nodes (highlighted with blue background in Figure 14) forms the only SCC (denoted by s) of $\mathcal{P}$ that consists of more than one single node. s reaches the SCCs containing the final states $(\texttt{:b}, 4)$, $(\texttt{:c}, 4)$, and $(\texttt{:d}, 4)$. The SCC $(\texttt{:a}, 0)$, however, reaches s over the SCC $(\texttt{:a}, 1)$, which leads to the result already given in Example 13. All other initial states of $\mathcal{P}$ are isolated, so no more result pairs will be found.

In the remaining part of this section, all these ingredients are brought together into the *Graph Labeling Algorithm*, which consists of a `LABEL` (Listing 4) and an `EVAL` (Listing 5) method. In contrast to the version presented in [27], these methods do not work directly on an RDF graph G, but on its associated NFA $\mathcal{G}$ (Definition 15). Furthermore, `EVAL` has only two arguments, as it *constructs*

the result set rather than just *decides* if a pair (a, b) is in the result set (a and b are the two additional arguments of EVAL in [27]).

```
   procedure LABEL(𝒢, exp)
1  for each axis::[exp′] and axis::![exp′] in D_o(exp)
2    LABEL(𝒢, exp′)
3  construct 𝒜_exp
4  construct 𝒫 from 𝒢 and 𝒜_exp
5  for each strongly connected component c in 𝒫
6    for each initial state (v, ·) in c that reaches a final state in 𝒫
7      labels(v) := labels(v) ∪ {exp}
8  return 𝒫
```

Listing 4: The LABEL method

The main differences between the LABEL method presented here and in [27] are that Listing 4 is enabled for ENREs, as it takes care of negated nested expressions ($axis::![exp']$), and that it returns the outermost product automaton $\mathcal{P}$, which later will be reused by the EVAL method (Listing 5). Furthermore, it tells more explicitly how all initial states that reach a final state in $\mathcal{P}$ should be found in linear time: namely by decomposing $\mathcal{P}$ into its SCCs (see line 5), as already mentioned.

The purpose of the LABEL method is to label each state v of $\mathcal{G}$ (Definition 15) with a set $labels(v)$ of nested expressions exp' that v satisfies (i.e. $(v, \cdot) \in [\![exp']\!]_G$). This is done by filtering the depth-0 terms of exp for nested expressions of the form $axis::[exp']$ or $axis::![exp']$ (in line 1) and performing a recursive call of LABEL$(\mathcal{G}, exp')$ for each ENRE exp' (in line 2). These recursive calls ensure that all nested expressions already appear in the label sets of all nodes satisfying them prior to constructing the dependent product automaton $\mathcal{P}$ (in line 4) from $\mathcal{G}$ and $\mathcal{A}_{exp}$ (which is constructed in line 3).

In line 5, $\mathcal{P}$ is decomposed into its strongly connected components via Tarjan's algorithm (Section A.2), as already explained. All initial states $(v, \cdot)$ that reach some final states of $\mathcal{P}$ are retrieved (in line 6) and their first component v (a node or an edge of G) is labeled with exp in line 7, i.e. $labels(v) := labels(v) \cup \{exp\}$. Finally, in the outermost call of LABEL, $\mathcal{P}$ is returned in line 8 for further reuse in the EVAL method.

The evaluation of an ENRE exp over $\mathcal{G}$ is initiated by calling the EVAL method, which is presented in Listing 5.

```
   procedure EVAL(𝒢, exp)
1  result := ∅
2  for each state v of 𝒢
3      labels(v) := ∅
4  𝒫 := LABEL(𝒢, exp)
5  for each strongly connected component c in 𝒫
6      for each initial state (a, ·) in c
7          for each final state (b, ·) that is reached from c
8              result := result ∪ {(a, b)}
9  return result
```

Listing 5: The EVAL method

The first line of EVAL initializes a set *result*, which will be used as the return value of EVAL in line 9. Lines 2 – 3 serve as an initialization step for the subsequent call to the LABEL method in line 3: $labels(v)$ is initialized to an empty set for each state v of $\mathcal{G}$. After LABEL has been called (line 4), it holds that $labels(v)$ is the set of all nested expressions that v satisfies. LABEL returns a product automaton $\mathcal{P}$ that has been constructed according to Definition 18, but that has already been enriched with its component graph within the LABEL method. Hence, line 5 does not recalculate the SCCs of $\mathcal{P}$, but just iterates over them. In lines 6 – 8, the final result is constructed by retrieving all initial states $(a, \cdot)$ that reach a final state $(b, \cdot)$ in $\mathcal{P}$ and adding each pair (a, b) to *result*.

3.3.4 *Complexity Analysis*

In contrast to [27], we do not discuss the complexity of the *decision problem* that is associated with the EVAL method (Listing 5), but rather the complexity that is based on the effort which is actually needed to *construct the result* [1]. This also explains why our EVAL method has $\mathcal{G}$ and exp as its only arguments, with $\mathcal{G}$ as in Definition 15 and exp an arbitrary ENRE. The construction of $\mathcal{G}$ from an RDF graph G can be done in time $\mathcal{O}(|G| \cdot \log |G|)$ by sorting $terms(G)$ in a separate precalculation step, which is not taken into account here.

This section is organized as follows: In Lemma 3, we prove several properties of $\mathcal{A}_{exp}$, as given by Definition 17. Lemmas 4 and 5 deal about the time and space complexity of a product automaton $\mathcal{P}$, as given by Definition 18.

Finally, we prove the time complexity of LABEL$(\mathcal{G}, exp)$ to be $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$ in Lemma 6, with exp an arbitrary ENRE and $\mathcal{G}$ as in Definition 15. The time

complexity of LABEL already determines the time complexity of EVAL$(\mathcal{G}, exp)$ to $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$, as shown in Theorem 3.

Lemma 3

Let $\mathcal{A}_{exp}$ be an ε-NFA, constructed from an ENRE exp via Definition 17. Then, (i) the number of $\mathcal{A}_{exp}$'s states and (ii) the number of $\mathcal{A}_{exp}$'s transitions is $\mathcal{O}(|\mathcal{A}_{exp}|)$. Furthermore, (iii) $|\mathcal{A}_{exp}|$ is $\mathcal{O}(|exp|)$, where $|exp|$ denotes the string length of exp.

Proof. (i) In the base case (compare the first box in Definition 17), one new state is created along with one transition whose label l in any case starts with a navigation axis. However, the string length of a navigation axis is greater than one. In the inductive case of a concatenation, i.e. $exp = e_1/e_2$, no new state is created. In all remaining inductive cases, exactly one new state is created, but at the same time, two ε-transitions are introduced, of which each is defined to have length one (Definition 17). All together, this shows that the number of $\mathcal{A}_{exp}$'s states is always less than $|\mathcal{A}_{exp}|$, hence also $\mathcal{O}(|\mathcal{A}_{exp}|)$.

(ii) The number of $\mathcal{A}_{exp}$'s transitions, denoted by t, is less than $|\mathcal{A}_{exp}|$, as each transition contributes at least an amount of one to $|\mathcal{A}_{exp}|$ (remember that ε-transitions are defined to have length one, Definition 17). So, $t \leq |\mathcal{A}_{exp}|$ and hence t is $\mathcal{O}(|\mathcal{A}_{exp}|)$.

(iii) We show that $|\mathcal{A}_{exp}| \leq |exp|$, and hence $|\mathcal{A}_{exp}|$ is $\mathcal{O}(|exp|)$.
In the base case (compare the first box in Definition 17), it trivially holds that $|\mathcal{A}_{exp}| \leq |exp|$. In the inductive case of an optionality operator, i.e. $exp = e?$, we may assume that $|\mathcal{A}_e| \leq |e|$ by induction hypothesis. However, $|\mathcal{A}_{exp}| = |\mathcal{A}_e| + 2$ by construction (Definition 17) and $|exp| = |e| + 1$ (as $|?| = 1$). All together, this shows that $|\mathcal{A}_{exp}| \leq |exp|$ in the case of an optionality operator; the remaining inductive cases are similar. □

Lemma 4

The construction of a product automaton $\mathcal{P}$ is possible in time $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$, with $\mathcal{P}$, $\mathcal{G}$, and $\mathcal{A}_{exp}$ as in Definition 18.

Proof. By Def. 18, $\mathcal{P} = (terms(G) \times Q, \mathbf{D}_0(exp), \delta_{\mathcal{P}}, I \times terms(G), F \times terms(G))$ has a total of $|terms(G)| \cdot |Q|$ states. We construct $\mathcal{P}$ by constructing all outgoing transitions for each product state $(v, q) \in terms(G) \times Q$.

Let (v, q) be an arbitrary product state of $\mathcal{P}$. As $\mathcal{A}_{exp}$ has been recursively built from a regular expression exp over the alphabet $\mathbf{D}_0(exp)$ according to Definition 17, its state q has at most two outgoing transitions (none of $\mathcal{A}_{exp}$'s

construction steps does create a state with more than two outgoing transitions). $\mathcal{G}$'s state v, however, is able to have arbitrarily many outgoing transitions, as defined by its underlying RDF graph G.

Let t_1 be an arbitrary outgoing transition of q in $\mathcal{A}_{exp}$. By construction of $\mathcal{G}$ (Definition 15), v has only outgoing transitions t_2 of the form $axis'\texttt{::}a$ with $axis' \in \{\texttt{self\_node}, \texttt{self\_edge}, \texttt{next}, \texttt{next}^{-1}\}$ and $a \in terms(G)$.

Case t_1 is labeled with ε
This is the easiest case, since no conditions need to be checked ($\mathcal{O}(1)$) according to Equation 3.5.

Case t_1 is labeled with an ENRE of the form $axis$
According to Equation 3.1, it suffices to check if $axis' \models axis$ (which requires time $\mathcal{O}(1)$) for each outgoing transition t_2 of v.

Case t_1 is labeled with an ENRE of the form $axis\texttt{::}regexp$
Additionally to the preceding case, this case requires to match the string a against the regular expression pattern $regexp$. This matching requires time $\mathcal{O}(|a| \cdot |regexp|)$, when carried out via simulating an ε-NFA that can be built from $regexp$ in linear time (compare Section 2.5).

Case t_1 is labeled with an ENRE of the form $axis\texttt{::[}exp'\texttt{]}$ or $axis\texttt{::![}exp'\texttt{]}$
By Equations 3.3 and 3.4, this case requires to test if exp' is contained in the set $labels(a)$. This membership test can be done in time $\mathcal{O}(1)$, since $labels(\cdot)$ can be represented as an array of booleans. The size of this array is the number of nested expressions that the original ENRE exp contains (at any depth) and can be precalculated via a single-pass traversal of exp's abstract syntax tree.

We have seen that, when considering an arbitrary product state (v, q) of $\mathcal{P}$ and all outgoing transitions t_1 of q in $\mathcal{A}_{exp}$ and t_2 of v in $\mathcal{G}$, the only case where matching the label of t_2 against the label of t_1 requires a different time than $\mathcal{O}(1)$ is when t_1 is of the form $axis\texttt{::}regexp$ and t_2 is of the form $axis'\texttt{::}a$ with $axis' \models axis$ and a an arbitrary string.

The worst case is when all of $\mathcal{A}_{exp}$'s and $\mathcal{G}$'s transitions are of that form. In the worst case, matching all pairs of transitions requires time

$$\mathcal{O}\Big(\sum_{\substack{(q,l_1,q')\in\delta_{\mathcal{A}}\\(v,l_2,v')\in\delta_{\mathcal{G}}}} |l_1| \cdot |l_2|\Big) = \mathcal{O}\Big(\sum_{(q,l_1,q')\in\delta_{\mathcal{A}}} \sum_{(v,l_2,v')\in\delta_{\mathcal{G}}} |l_1| \cdot |l_2|\Big)$$
$$= \mathcal{O}\Big(\sum_{(q,l_1,q')\in\delta_{\mathcal{A}}} |l_1| \cdot \sum_{(v,l_2,v')\in\delta_{\mathcal{G}}} |l_2|\Big) = \mathcal{O}\big(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|\big)$$

□

LEMMA 5
Both the number of states and transitions of a product automaton $\mathcal{P}$ are $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$, with $\mathcal{P}$, $\mathcal{G}$, and $\mathcal{A}_{exp}$ as in Definition 18.

Proof. By construction, $\mathcal{P}$ has exactly $|terms(G)|$ (the number of $\mathcal{G}$'s states) times $|Q|$ (the number of $\mathcal{A}_{exp}$'s states) many product states, where $|Q|$ is $\mathcal{O}(|\mathcal{A}_{exp}|)$ by Lemma 3.

The number of $\mathcal{P}$'s transitions is again by construction limited by the number of $\mathcal{G}$'s transitions times the number of $\mathcal{A}_{exp}$'s transitions. The number of $\mathcal{G}$'s transitions is $\mathcal{O}(|terms(G)|)$ by Definition 15, and the number of $\mathcal{A}_{exp}$'s transitions is $\mathcal{O}(|\mathcal{A}_{exp}|)$ by Lemma 3.

However, $|terms(G)|$ is $\mathcal{O}(|\mathcal{G}|)$ (since each $t \in terms(G)$ has a transition to itself that contributes to $|\mathcal{G}|$, by Definition 15), which proves the claim. □

LEMMA 6
`LABEL`$(\mathcal{G}, exp)$ (Listing 4) runs in time $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$, with $\mathcal{G}$ as in Definition 15 and exp an arbitrary ENRE.

Proof. We prove this by induction over $\mathbf{D}_0(exp)$.

Case $\mathbf{D}_0(exp)$ does not contain any nested expressions
In this case, no recursive calls are done in lines 1 – 2 of the `LABEL` method. The construction of $\mathcal{A}_{exp}$ in line 3 is possible in linear time from exp by the rules presented in Definition 17. By Lemma 4, the construction of the product automaton $\mathcal{P}$ in line 4 is possible in time $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$. $\mathcal{P}$'s number of states and transitions is $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$ by Lemma 5. Tarjan's algorithm (Section A.2) is used to decompose $\mathcal{P}$ into its strongly connected components in line 5, when interpreting $\mathcal{P}$ as a directed graph (i.e. $\mathcal{P}$'s states and transitions are interpreted as nodes and edges, respectively). By [31], this decomposition has a time bound which is linear in the number of $\mathcal{P}$'s states and transitions, and thus is $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$. Line 6 of `LABEL` is implemented as a depth-first search on the component graph of $\mathcal{P}$. Each strongly connected component c of $\mathcal{P}$ is associated with a set of reachable final states in $\mathcal{P}$, which is represented as a boolean array over all states of $\mathcal{G}$. During the depth-first search, such an array of size $\mathcal{O}(|\mathcal{G}|)$ is calculated for each strongly connected component c, which all together results in a time bound of $\mathcal{O}(|\mathcal{G}|)$ (the size of each array) times $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$ (the number of strongly connected components), hence $\mathcal{O}(|\mathcal{G}|^2 \cdot |\mathcal{A}_{exp}|)$.

Finally, the *labels* set in line 7 is also implemented as an array of booleans. Thus, adding exp to the label set $labels(v)$ of a state v of $\mathcal{G}$ requires constant time and does not violate the time bound $\mathcal{O}(|\mathcal{G}|^2 \cdot |\mathcal{A}_{exp}|)$. As $|\mathcal{A}_{exp}|$ is $\mathcal{O}(|exp|)$ by Lemma 3, the time bound is also $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$, as desired.

Case $\mathbf{D}_0(exp)$ contains nested expressions $exp_1, \ldots, exp_n$ $(n \geq 1)$
By induction hypothesis, each recursive call $\texttt{LABEL}(\mathcal{G}, exp_i)$ $(1 \leq i \leq n)$ takes time $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp_i|)$. As $exp_1, \ldots, exp_n$ are subexpressions of exp, it follows that $\sum_{i=1}^{n} |exp_i| \leq |exp|$. Combining this with the time bound for each recursive call $\texttt{LABEL}(\mathcal{G}, exp_i)$ yields the bound $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$ for all recursive calls done in lines 1 – 2 of LABEL.

Lines 3 – 7 can be done in time $\mathcal{O}(|\mathcal{G}|^2 \cdot |\mathcal{A}_{exp}|)$, as already discussed in the base case. So, all together, the time necessary for $\texttt{LABEL}(\mathcal{G}, exp)$ is bound by $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$. □

THEOREM 3
Evaluating an ENRE exp on $\mathcal{G}$ via the EVAL method (Listing 5) is possible in time $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$, with $\mathcal{G}$ as in Definition 15.

Proof. In the first line of EVAL, a variable *result* is initialized as an empty list, which requires constant time. In lines 2 – 3, the label sets $labels(v)$ are initialized to empty sets for all states v of $\mathcal{G}$ (i.e. $v \in terms(G)$). By construction, each state v has a transition to itself in $\mathcal{G}$ having a non-empty label, hence $|terms(G)|$ is $\mathcal{O}(|\mathcal{G}|)$.

Line 4 calls $\texttt{LABEL}(\mathcal{G}, exp)$, whose time complexity is bound by $\mathcal{O}(|\mathcal{G}|^2 \cdot |exp|)$ by Lemma 6. LABEL returns a product automaton $\mathcal{P}$ of size $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{A}_{exp}|)$, which has already been decomposed into its strongly connected components in line 5 of the LABEL method. During line 6 of the LABEL method, each SCC c of $\mathcal{P}$ was associated with a boolean array, storing all of $\mathcal{G}$'s final states which can be reached from c in $\mathcal{P}$.

Reusing these arrays, the only thing left to do is the construction of the result in line 8 of the EVAL method. The variable *result* is implemented as a list, as each pair (a, b) of states of $\mathcal{G}$ is added at most once. Thus, lines 5 – 8 can be executed in time $\mathcal{O}(|\mathcal{G}|^2 \cdot |\mathcal{A}_{exp}|)$, as seen for lines 5 – 7 of the LABEL method.

All together, the time complexity of $\texttt{EVAL}(\mathcal{G}, exp)$ is bound by $\mathcal{O}(|\mathcal{G}^2| \cdot |exp|)$, which arises from the call to LABEL by Lemma 6. All remaining operations within EVAL can be done in time $\mathcal{O}(|\mathcal{G}^2| \cdot |\mathcal{A}_{exp}|)$, which however is $\mathcal{O}(|\mathcal{G}^2| \cdot |exp|)$ as $|\mathcal{A}_{exp}|$ is $\mathcal{O}(|exp|)$ by Lemma 3. □

3.4 PATH-BASED EVALUATION

In this section, we present a second approach for evaluating RPL, which is based on path matrices. Similar matrices are also used by algorithms to find the shortest path between all pairs of nodes, e.g. the Floyd-Warshall algorithm [12].

The underlying idea is to enrich each subexpression of a RPL query (more precisely, each node of the query's abstract syntax tree) with a *partial path matrix* M reaching over all nodes of an RDF graph G. Each entry $m_{i,j}$ of M can either be 1 (indicating that there is a path from node i to node j) or 0 (indicating that there is no path from node i to node j). In addition, whole rows may be undefined (indicating that no path information is available on the node described by the row).

After the notion of partial path matrices is formally given in Definition 19, three operations are introduced: the concatenation (Definition 20), the union (Definition 21), and the intersection (Definition 22) of two partial path matrices. Finally, Lemma 7 shows that in the special case of two diagonal partial path matrices (which is the only case that is used within the evaluation algorithm), their concatenation and intersection yield the same result.

We present two variants of the Path-based evaluation algorithm.

Section 3.4.2 gives an outline of its first variant, which applies a fixpoint-based approach for the evaluation of an `AdornedRPE` *adorned* and is shown to have exponential data time complexity in Section 3.4.3.

Section 3.4.4 presents the second variant, which instead computes the transitive closure of *adorned*'s child and is shown to have linear query and cubic data time complexity (Theorem 5).

However, the experimental evaluation in Chapter 5 indicates that both variants seem to have cubic data time complexity.

3.4.1 *Partial Path Matrices*

Definition 19 (Path Matrix, Partial Path Matrix)
A *path matrix* over an RDF graph G is a quadratic matrix of size $|nodes(G)| \times |nodes(G)|$ over $\mathbb{B} := \{0, 1\}$. A *partial path matrix* is a path matrix, which leaves zero or more of its rows undefined (graphically indicated by [redacted]). Hence, every path matrix is also a partial path matrix.

Example 14

Let G be an RDF graph with $nodes(G) = \{a, b, c, d\}$ and P a partial path matrix. P should contain a path from a to b (i.e. $P_{a,b} := 1$), and two paths from c to b and d (i.e. $P_{c,b} := 1, P_{c,d} := 1$). Additionally, no information is available over paths starting at node b, hence row b of P is undefined. There are no more paths in P (i.e. all other entries of P are set to zero and omitted below).

$$P := \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} & 1 & & \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ & 1 & & 1 \\ & & & \end{bmatrix} \end{array}$$

In the following, we present three operations on partial path matrices: the concatenation (Definition 20), union (Definition 21), and intersection (Definition 22) of two partial path matrices.

Definition 20 (Concatenation of Partial Path Matrices)

Concatenating (denoted by $\circ$) two partial path matrices means concatenating their paths. For instance, if the first matrix P contains a path from a to b ($P_{a,b} = 1$), and the second matrix Q contains a path from b to c ($Q_{b,c} = 1$), the resulting matrix R will contain a path from a to c ($R_{a,c} = 1$).

More precisely, for calculating a row x of R, one has to "combine" all rows in Q which are reached from x in P. To "combine" two (defined) rows means to apply the boolean OR operation, interpreting their values 0 and 1 as $false$ and $true$, respectively. In case row x is undefined in P, or (at least) one of the to be combined rows in undefined in Q, row x is set to undefined in R.

The time complexity of this operation is $\mathcal{O}(n^3)$, when n denotes the number of rows. This is because for constructing one of R's result rows, one might have to consider all rows of the second matrix Q in the worst case.

Example 15

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & & & 1 \\ & & 1 & 1 \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ 1 & & & \end{bmatrix} \end{array} \circ \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} & & & \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ & 1 & & 1 \\ & & 1 & \end{bmatrix} \end{array} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} & & 1 & \\ & 1 & 1 & 1 \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ & & & \end{bmatrix} \end{array}$$

Definition 21 (Union of Partial Path Matrices)

Calculating the union (denoted by $\cup$) of two partial path matrices P and Q means to union their contained paths. For instance, if P contains a path from a to b ($P_{a,b} = 1$) and Q from c to d ($Q_{c,d} = 1$), then the resulting matrix R will contain a path from a to b ($R_{a,b} = 1$), as well as from c to d ($R_{c,d} = 1$).

More precisely, for calculating a row x of R, one has to "union" row x in P with row x in Q. Like in Definition 20, to "union" two (defined) rows means to apply the boolean OR operation, interpreting 0 and 1 as $false$ and $true$, respectively.

If row x is undefined in either P or Q, it will be treated as being empty (but defined). If row x is undefined in both P and Q, it will also be undefined in R.

The time complexity of this operation is $\mathcal{O}(n^2)$, as the construction of a result row of R only requires to consider its correspondig rows in P and in Q; n is assumed to denote the number of rows.

Example 16

$$
\begin{array}{c|cccc} & a & b & c & d \\ a & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ b & & & 1 & 1 \\ c & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ d & 1 & & & \end{array}
\cup
\begin{array}{c|cccc} & a & b & c & d \\ a & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ b & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ c & & 1 & & 1 \\ d & 1 & & 1 & \end{array}
=
\begin{array}{c|cccc} & a & b & c & d \\ a & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ b & & & 1 & 1 \\ c & & 1 & & 1 \\ d & 1 & & 1 & \end{array}
$$

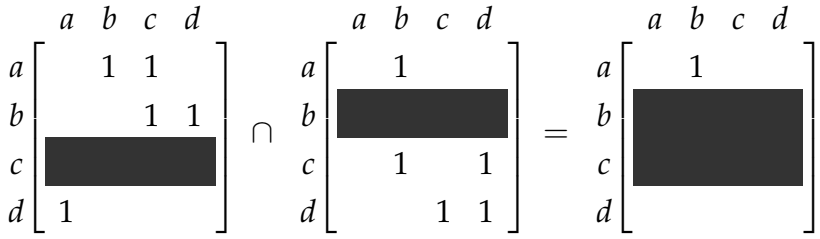
Definition 22 (Intersection of Partial Path Matrices)

The intersection (denoted by $\cap$) of two partial path matrices P and Q means to intersect their contained paths. For instance, if P contains a path from a to b ($P_{a,b} = 1$) and from a to c ($P_{a,c} = 1$) and Q just from a to b ($Q_{a,b} = 1$, $Q_{a,c} = 0$), then the resulting matrix R will contain just a path from a to b ($R_{a,b} = 1$), but not from a to c ($R_{a,c} = 0$).

More precisely, for calculating a row x of R, one has to "intersect" row x in P with row x in Q. Similar to Definition 20, to "intersect" two (defined) rows means to apply the boolean AND operation, interpreting 0 and 1 as $false$ and $true$, respectively. If row x is undefined in P or Q (not exclusively), it will also be undefined in R.

The time complexity of this operation is $\mathcal{O}(n^2)$. As for the union operation, constructing a result row of R only requires to consider its correspondig rows in P and in Q; n again denotes the number of rows.

Example 17



The following Lemma 7 shows that the intersection of two diagonal partial path matrices can also be expressed via their concatenation. For this reason, only the concatenation operation needs to be implemented.

Lemma 7

A *diagonal partial path matrix* P is a partial path matrix where each row r is either undefined or $P_{r,s} = 0$ if $r \neq s$. The concatenation of two diagonal partial path matrices P and Q is equal to the intersection of P and Q.

Proof. Case Concatenation
Consider row x. If x is undefined in P, x will by Definition 20 also be undefined in R. If x is all empty (all entries set to 0), x will also be empty in R, as no row can be reached from x in Q. If x is non-empty (i.e. $P_{x,x} = 1$ and all other entries $P_{x,\cdot} = 0$), it only reaches row x in Q. In this case, row x of R is just a copy of row x in Q. Hence, the time complexity for this concatenation is just $\mathcal{O}(n^2)$, when n denotes the number of rows.

Case Intersection
Consider row x. If x is undefined in P, x will by Definition 22 also be undefined in R. If x is all empty, the intersection with row x from Q will by definition be all empty as well. If x is non-empty, row x of R is again by definition just a copy of row x from Q. □

3.4.2 *Algorithm Outline*

This section gives an outline of the first variant of the Path-based evaluation algorithm, which applies a fixpoint-based approach when evaluating `AdornedRPEs`.

The basic idea of the Path-based evaluation is memoization: each subexpression e of a RPL query q is able to memoize a partial path matrix holding all paths (in the sense of Definition 4) starting from nodes in G on which e has already been evaluated on. At initialization, the partial path matrix of e

contains only undefined rows. This means that e has not yet been evaluated on any node of the RDF graph G.

When evaluating a RPL query q, it is evaluated from left to right (concerning the query string). The evaluation is started with an initial diagonal path matrix I with $I_{r,r} := 1$ for each row r. This means, that in the beginning, each path of length zero (from each node to itself) is allowed. So, the very first subexpression of q is evaluated on each node of G (as each node in G is reached from I), but all following subexpressions are probably just evaluated on a subset of $nodes(G)$.

Evaluating `ConcatenatedRPE`s

Listing 6 shows a code snippet that deals with `ConcatenatedRPE`s. This snippet is part of the `EvalVisitor` class, which implements the first variant of RPL's Path-based evaluation approach.

```
1  public void visit(ConcatenatedRPE concatenated) throws RPEException {
2    if (concatenated.getAdorneds().size() == 1)
3      concatenated.getAdorneds().getFirst().accept(this);
4    else {
5      PathMatrix matrix = matrices.pop();
6      if (concatenated.getMatrix() == null)
7        concatenated.setMatrix(new PathMatrix(nodes.length));
8      PathMatrix memo = concatenated.getMatrix();
9      PathMatrix todo = matrix.todo(memo);
10     if (!todo.isEmpty()) {
11       for (AdornedRPE adorned : concatenated) {
12         matrices.push(todo);
13         adorned.accept(this);
14         todo = matrices.pop();
15       }
16       memo.union(todo);
17     }
18     matrices.push(matrix.concatenate(memo));
19   }
20 }
```

Listing 6: Evaluating `ConcatenatedRPE`s

In the following, we give a concrete description of how this algorithm evaluates `ConcatenatedRPE`s based on Listing 6.

Lines 2 – 3 do not express the general idea of the algorithm, but contain an optimization: in case a `ConcatenatedRPE` *concatenated* has only one adorned child, it does not need to maintain its own partial path matrix since it can just reuse the one belonging to its child. So, the interesting case starts at line 5, where we first get the partial path matrix *matrix* from the stack of matrices (the parent of *concatenated* has pushed it before visiting its child/children).

In lines 6 – 8, we either retrieve or create (if it has not been created yet) the *memo* matrix of *concatenated*. After creation, this *memo* matrix consists of undefined rows only (which means that *concatenated* has not yet been evaluated on any node of G). If *concatenated* has already been visited (for at least one time), *memo* contains a defined (but still possibly empty) row for each node of G, on which *concatenated* has already been evaluated.

In line 9, the given matrix *matrix* is compared with the memoized matrix *memo* in order to retrieve the nodes on which *concatenated* still has to be evaluated. These nodes are stored in the partial path matrix *todo*, which therefore is in diagonal form (Lemma 7).

If each row of *todo* is undefined (line 10), the memoized matrix *memo* already contains all required information – so one only has to compute $matrix \circ memo$ (Definition 20) and push the result on the stack of matrices (line 18).

Otherwise (if *todo* contains at least one non-zero entry), the adorned children of *concatenated* have to be evaluated on the nodes contained in the *todo* matrix. By the semantics of a `ConcatenatedRPE` (Section 3.2), the paths defined by its adorned children have to be concatenated – so the matrix that gets passed to the second adorned child is the concatenation of *todo* with the result of the first adorned child and so forth (lines 11 – 15). Finally, the *memo* matrix of *concatenated* is merged (Definition 21) with the *todo* matrix, which now holds all paths that resulted from concatenating the paths of *concatenated*s adorned children (line 16). Again, *matrix* is concatenated with the updated *memo* matrix and the result is pushed on the stack of matrices (line 18).

Evaluating `AdornedRPE`*s*

The semantics of *ad*+ over an RDF graph G and an `AdornableRPE` *ad* is defined as $[\![ad+]\!]_G = [\![ad]\!]_G \cup [\![ad\ ad]\!]_G \cup [\![ad\ ad\ ad]\!]_G \cup \ldots$ (compare Definition 9). Since this definition is not suitable for evaluation purposes, `AdornedRPE`s are evaluated using a fixpoint-based approach. First, *ad* is evaluated on the initial set of nodes. Then, *ad* is evaluated on the resulting set of nodes until a fixpoint is found (i.e. no more additional nodes can be reached). Such a fixpoint always exists as the set of reachable nodes is bound by $nodes(G)$.

Example 18

We demonstrate this fixpoint-based approach by evaluating the query `PATH :a (>_ _)+`, denoted by q, on the RDF graph G from Figure 12. Note that q has the same semantics as the ENRE `self_node:::a/(next/self_node)+`, which together with G already served as an example throughout the ENRE-based evaluation.

Before going into detail, we try to intuitively determine the semantics of q.

q defines a path that starts at node `:a`, and then traverses an arbitrary outgoing edge from `:a` leading into an arbitrary node. This step, taking an arbitrary outgoing edge from the current node leading to another arbitrary node, can be repeated any non-zero number of times (as indicated by `(...)+`).

Applied to G, this means that we start at node `:a`, take its outgoing edge `:e1` and arrive at node `:b`. So, the first pair in the result will be (`:a`, `:b`). Now, let us repeat this step: we traverse `:b`'s outgoing edge `:e2` and arrive at node `:c` (hence, the pair (`:a`, `:c`) will also be part of the result set). We continue by traversing the edge `:e3`, which brings us to the node `:d` and again add the pair (`:a`, `:d`) to the result set. The outgoing edge of `:d` is `:e4` and brings us to node `:b`, but the pair (`:a`, `:b`) is already part of the result! As `:d` does not have any other outgoing edges, we will always arrive at node `:b`, and continuing at node `:b`, we would again arrive at node `:c` via the edge `:e2` and again at node `:d` via the edge `:e3`, but none of these moves would add a new pair to our result set — so, we are obviously finished.

This idea of a fixpoint-based evaluation also forms the basis for the implementation of this first variant of the Path-based evaluation algorithm. As the query q is evaluated from left to right, the `AdornedRPE (>_ _)+`, denoted by ad, just needs to be evaluated on the node `:a`. ad maintains a partial path matrix $hull$, which stores all paths that have been found so far. More precisely, by $hull_i$, we denote this partial path matrix after the i-th visit of its child `ConcatenatedRPE >_ _`, denoted by c.

The evolution of the matrix $hull$ during the evaluation of ad is shown in Table 2. This perfectly fits our intuition, as $hull_1$ contains the path (`:a`, `:b`), $hull_2$ additionally contains the path (`:a`, `:c`), and so on — and this exactly reflects the chronological order of how we intuitively discovered these paths.

Table 2 also shows, that after a finite number of steps, a fixpoint concerning these $hull$ matrices is reached, i.e. there exists an $i \geq 1$ (in this case $i = 3$) such that $hull_{i+1} = hull_i$, and this implies that $hull_{i+n} = hull_i$ for all $n \in \mathbb{N}$.

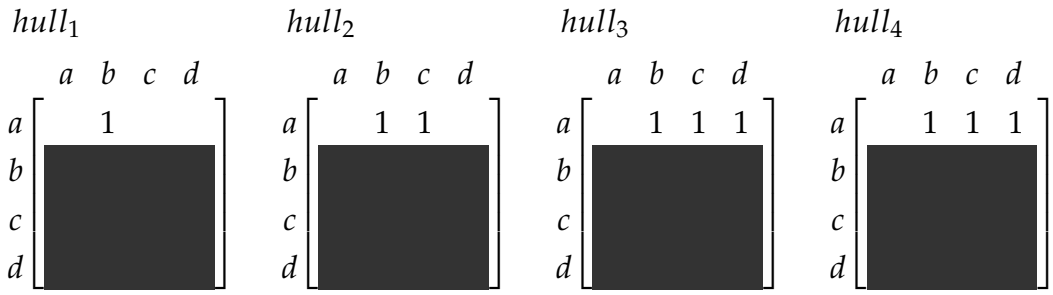
AdornedRPE (>_ _)+


Table 2: Evaluating the `AdornedRPE (>_ _)+` within the RPL query **`PATH`** `:a (>_ _)+`

Let us have a closer look at what happens at each time, when the `ConcatenatedRPE` (`>_ _`), denoted as c, is visited. Table 3 shows the single steps during the evaluation of c.

Within Table 3, $matrix_1$ corresponds to the *matrix* variable in line 5 of Listing 6, which gets passed from *ad* on a stack of matrices before c is visited for the first time. As previously noted, *ad* just needs to be evaluated on the node `:a`, hence $matrix_1$ just contains the path $(\texttt{:a}, \texttt{:a})$.

Similarly, $memo_1$ corresponds to the *memo* matrix in line 8 of Listing 6, and consists of only undefined rows after initialization. $todo_1$ corresponds to the *todo* matrix in line 9 and expresses that c needs to be evaluated on node `:a` (this is because $matrix_1$ contains a path $(\cdot, \texttt{:a})$ and $memo_1$ is undefined on row `:a`). Finally, $result_1$ corresponds to the result matrix built in line 18 of Listing 6. It contains the pair $(\texttt{:a}, \texttt{:b})$, as `:b` is the only node that can be reached via an arbitrary edge (only the edge `:e1` in this case).

$hull_1$ (Table 2) is set to $result_1$, and c is visited for a second time. The stack of matrices is not modified in between, so $matrix_2$ is equal to $result_1$. After the second visit of c, $result_2$ is used to compute $hull_2 := hull_1 \cup result_2$.

As $hull_2$ contains an additional path (the path $(\texttt{:a}, \texttt{:c})$ in this case) compared to $hull_1$, c is visited for a third time and so on. If $hull_{i+1}$ contains no additional path compared to $hull_i$, the smallest fixpoint has been found and the evaluation of the `AdornedRPE` *ad* is finished.

Finally, Figure 15 shows the abstract syntax tree and its partial path patrices of our sample RPL query **`PATH`** `:a (>_ _)+`, after its evaluation on the RDF graph from Figure 12 has been completed.

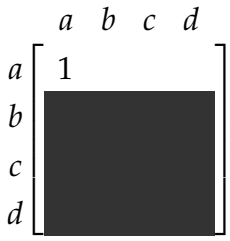
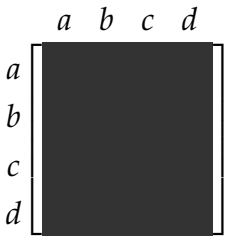
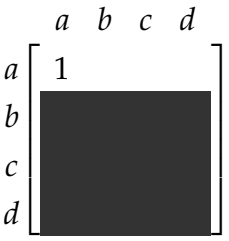
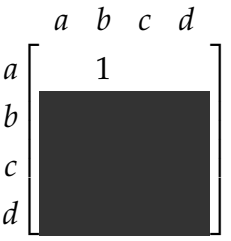
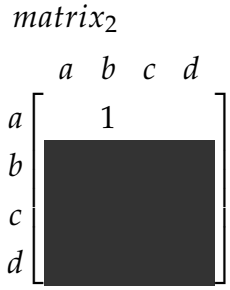
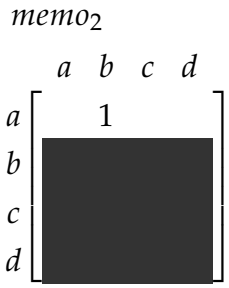
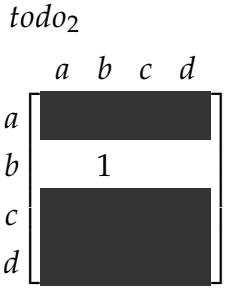
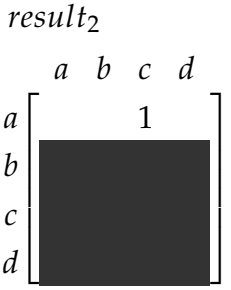
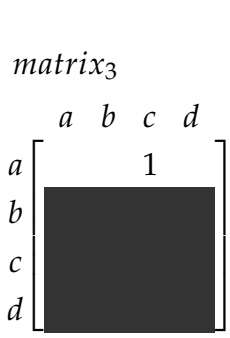
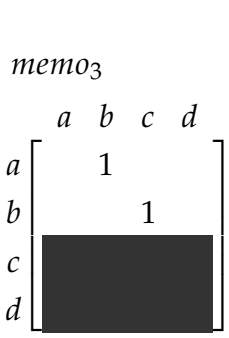
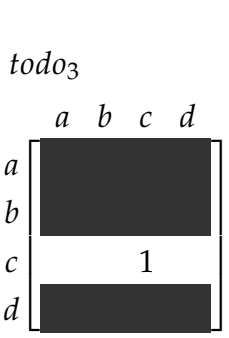
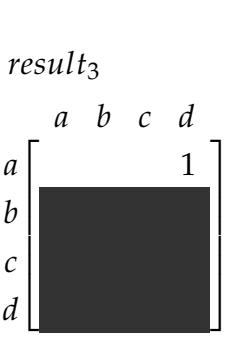
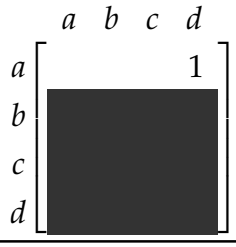
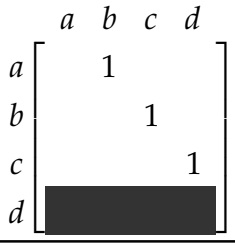
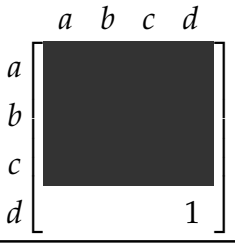
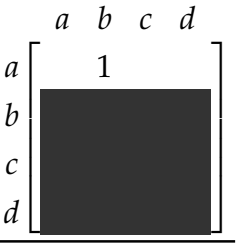
`ConcatenatedRPE >_ _`			
$matrix_1$	$memo_1$	$todo_1$	$result_1$
			
$matrix_2$	$memo_2$	$todo_2$	$result_2$
			
$matrix_3$	$memo_3$	$todo_3$	$result_3$
			
$matrix_4$	$memo_4$	$todo_4$	$result_4$
			

Table 3: Evaluating the `ConcatenatedRPE >_ _` within the `AdornedRPE (>_ _)+`

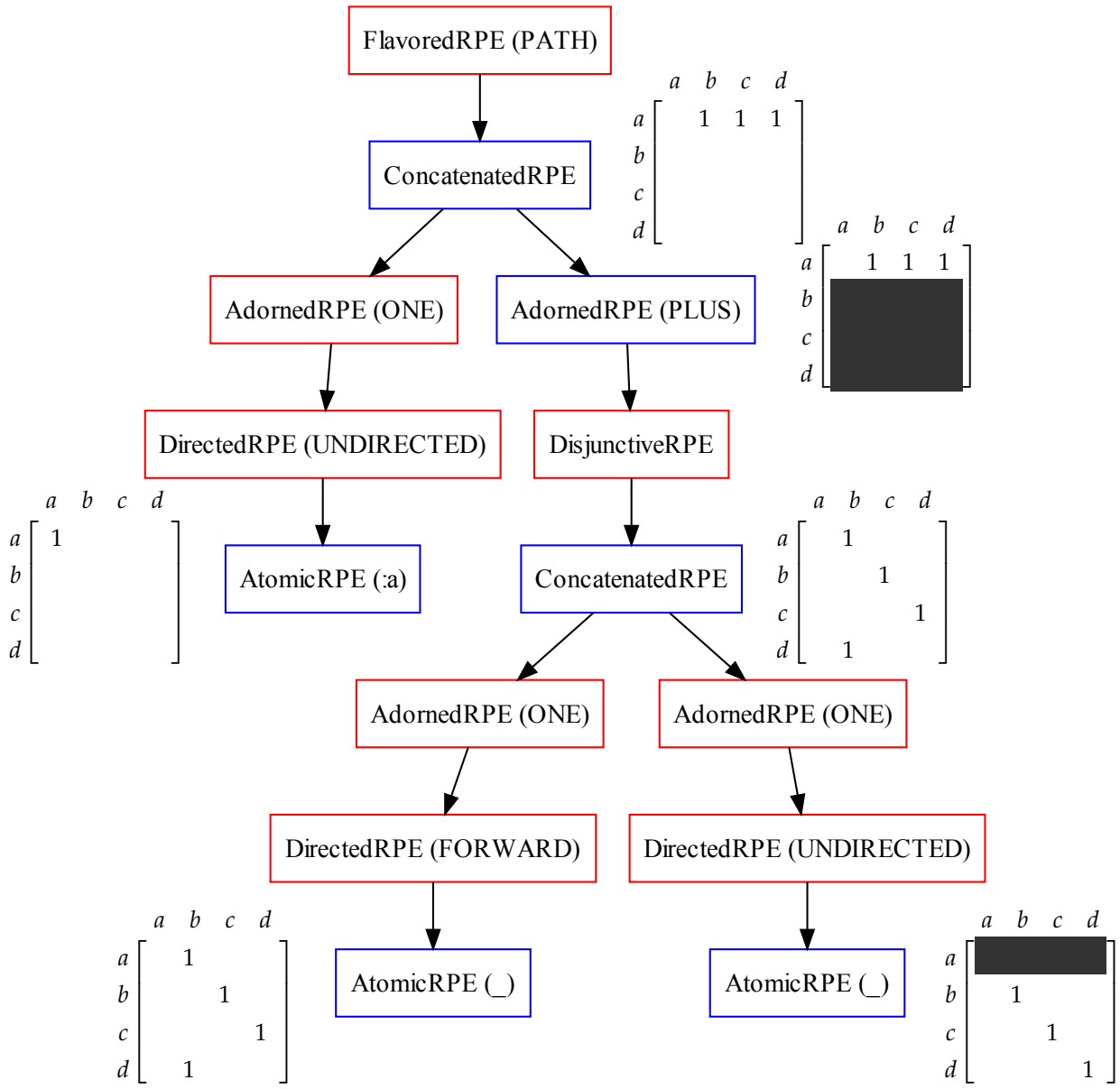


Figure 15: The abstract syntax tree of the RPL query `PATH :a (>_ _)+`, after evaluating it on the RDF graph from Figure 12. Only the nodes with a blue border are linked with partial path matrices, shown next to them; nodes with a red border do not have a matrix, as the matrix of its (direct or indirect) child node already contains all of its paths.

3.4.3 *Complexity Analysis*

After having presented its major parts, we investigate the time complexity of this first variant of the Path-based evaluation algorithm.

Similar as in the ENRE-based approach (compare Definition 15), we assume that an RDF graph G is stored as an adjacency list, i.e. each term t of G ($t \in terms(G)$) is associated with a list of incoming and outgoing *transitions*. A transition can in turn be seen as an RDF triple, as it is made up of three terms: its source, its edge label and its destination (corresponding to the subject, predicate, and object of an RDF triple, respectively).

Theorem 4
Let q be a RPL query and G be an RDF graph that is already stored as an adjacency list as described above. The evaluation of q over G, $[\![q]\!]_G$ can be calculated in time $\mathcal{O}(|G|^{2 \cdot |q|} + |G| \cdot |q|)$.

Proof. First, it is ensured via memoization, that each regular expression pattern from an `AtomicRPE` *atomic* is matched at most once to each term t of G. Each matching is done by constructing an ε-NFA from *atomic*, as described in Section 2.5. All together, this leads to a worst case time complexity of $\mathcal{O}(|G| \cdot |q|)$, where $|G|$ is the size of G (Definition 5) and $|q|$ is the size of q in characters, after all namespace prefixes have been resolved.

Each node of q's abstract syntax tree is associated with a partial path matrix M of size $|nodes(G)|^2$, hence M has $\mathcal{O}(|G|^2)$ entries. When evaluating q, its abstract syntax tree is visited in a top-down manner. When evaluating (i.e. visiting) an `AdornedRPE` *adorned* with multiplicity $*$ or $+$ according to the fixpoint-based approach as described above, its *adornable* child has to be visited $|nodes(G)|^2$ times in the worst case. This is because each single expansion of *adorned* must at least contribute one new path (in other words: at least one entry must be shifted from 0 or an undefined row into 1) in its partial path matrix M, and M is of size $|nodes(G)|^2$, as seen above. Otherwise, a fixpoint would have already been found.

This leads to the recursive equation

$$T(n) = T(n-1) \cdot \mathcal{O}(|G|^2) + \mathcal{O}(|G|^4) \tag{3.6}$$

where $n := |adorned|$ and thus $|adornable| = n - 1$ (as $|+| = |*| = 1$). The summand $\mathcal{O}(|G|^4)$ in Equation 3.6 is caused by the time effort of calculating the union of *adorned*'s partial path matrix and the resulting matrix after each visit of *adornable* (*adornable* is visited $\mathcal{O}(|G|^2)$ times, and calculating the union

of two partial path matrices requires time linear in their size). Obviously, $T(n) = \mathcal{O}(|G|^{2 \cdot n})$ is a solution for Equation 3.6 for $n \geq 2$. In the worst case, an essential part of the whole RPL query q might be made up of `AdornedRPE`s that are nested into one another, yielding a time complexity of $\mathcal{O}(|G|^{2 \cdot |q|})$.

Until now, we just considered `AtomicRPE`s (leading to the bound $\mathcal{O}(|G| \cdot |q|)$) and `AdornedRPE`s with multiplicity `*` and `+` (leading to the bound $\mathcal{O}(|G|^{2 \cdot |q|})$). Combining their bounds yields the final bound $\mathcal{O}(|G|^{2 \cdot |q|} + |G| \cdot |q|)$, as claimed.

In each other case, all children of a e.g. `ConcatenatedRPE` or `DisjunctiveRPE` are visited only once, so these visits do not violate this final complexity bound, as can be easily checked (compare Theorem 5). □

3.4.4 *An Efficient Variant of the Algorithm*

This section presents the second variant of the Path-based evaluation approach. Compared to the first variant, only `AdornedRPE`s are evaluated differently: this variant visits the adornable child of an `AdornedRPE` only once and calculates the transitive closure of its path matrix. In contrast to the first variant, this redesign allowed to prove a polynomial time complexity bound (Theorem 5).

Let *adorned* be an `AdornedRPE` with multiplicity `+` and *adornable* its child. The basic idea is to compute all paths that *adornable* satisfies by passing it an initial diagonal path matrix I with $I_{r,r} := 1$ for each row r. The resulting matrix R of evaluating *adornable* on I is a path matrix without any undefined rows.

In a second step, *adorned*'s path matrix is set to the transitive closure of R, which is computed as shown in Listing 7: If there is no path from node i to node j (line 5), a path can be obtained by first taking the path from i to k and then from k to j, if these paths both exist (line 6).

The correctness of this algorithm has been shown in [12]; its time complexity is obviously cubic in the number of R's rows.

```
1 public void transitiveClosure() {
2   for (int k = 0; k < matrix.length; k++)
3     for (int i = 0; i < matrix.length; i++)
4       for (int j = 0; j < matrix.length; j++)
5         if (matrix[i][j] == false)
6           matrix[i][j] = matrix[i][k] && matrix[k][j];
7 }
```

Listing 7: Calculating the transitive closure of a path matrix

Example 19

The `ConcatenatedRPE` `>_ _` evaluates to the matrix R on the RDF graph G from Figure 12, when I is used as incoming matrix. Listing 7 computes the transitive closure of R, denoted as C. Hence, the `AdornedRPE` `(>_ _)+` is evaluated to C, which coincides with the first line of the matrix $hull_4$ in Table 2.

$$I := \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \end{array} \quad R := \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & & & \end{bmatrix} \end{array} \quad C := \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} & 1 & 1 & 1 \\ & 1 & 1 & 1 \\ & 1 & 1 & 1 \\ & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Theorem 5

Let q be a RPL query and G be an RDF graph that is already stored as an adjacency list as described in Section 3.4.3. The evaluation of q over G, $[\![q]\!]_G$ can be calculated in time $\mathcal{O}(|G|^3 \cdot |q|)$.

Proof. We prove this theorem by inductively analysing the time complexity T for each abstract syntax tree class.

Case `AtomicRPE`

Let *atomic* be an `AtomicRPE` and *regexp* its associated regular expression (Section 3.1.5). A time of $\mathcal{O}(|G| \cdot |atomic|)$ is needed to evaluate *atomic* on all terms of G, when constructing an ε-NFA from *regexp* as described in Section 2.5. It is ensured via memoization that each term of G is matched at most once to each `AtomicRPE` of the RPL query q. Hence, the accumulated effort for this matching is $\mathcal{O}(|G| \cdot |q|)$.

To construct the partial path matrix of *atomic*, a time of $\mathcal{O}(|G|^2)$ is needed, hence $T(atomic) = \mathcal{O}(|G| \cdot |atomic|) + \mathcal{O}(|G|^2)$.

Case `PredicateRPE`

Let *predicate* be a `PredicateRPE` and $flavored$ its child. All operations within a visit of *predicate* can be done in time $\mathcal{O}(|G|^2)$, and $flavored$ is only visited once. These operations include e.g. "inverting" (in the case that *predicate* has a `NEGATIVE` sign) and "diagonalizing" $flavored$s partial path matrix, as well as concatenating *predicate*s matrix with its incoming matrix (as the latter is in diagonal form, this can be done in $\mathcal{O}(|G|^2)$ as well, see Lemma 7). All together, this yields the equation $T(predicate) = T(flavored) + \mathcal{O}(|G|^2)$.

Case `PredicatesRPE`
Let *predicates* be a `PredicatesRPE` and $p_1, \ldots, p_n$ its children. *predicates* visits each of its children once. Each child p_i $(1 \leq i \leq n)$ requires time $T(p_i)$ and a time of $\mathcal{O}(|G|^2)$ for concatenating p_i to the matrix that was returned from p_{i-1} $(2 \leq i \leq n)$ (this is because the matrix that gets passed to p_1 is in diagonal form, and each p_i's matrix is as well in diagonal form, so their concatenation in turn is in diagonal form, Lemma 7). Finally, *predicates* concatenates its matrix with its incoming matrix from its parent, which requires time $\mathcal{O}(|G|^3)$. All together, this yields the equation $T(predicates) = \sum_{i=1}^{n}(T(p_i) + \mathcal{O}(|G|^2)) + \mathcal{O}(|G|^3)$.

Case `DirectedRPE`
A `DirectedRPE` *directed* does not maintain a partial path matrix, but just passes its `DIRECTION` to its directable child: $T(directed) = T(directable)$.

Case `DisjunctiveRPE`
A `DisjunctiveRPE` *disjunctive* visits each child $concatenated_i$ $(1 \leq i \leq n)$ once and unions their matrices. Finally, *disjunctive* concatenates its matrix with its incoming matrix from its parent, which requires time $\mathcal{O}(|G|^3)$. All together, we get $T(disjunctive) = \sum_{i=1}^{n}(T(concatenated_i) + \mathcal{O}(|G|^2)) + \mathcal{O}(|G|^3)$.

Case `AdornedRPE`
Let *adorned* be an `AdornedRPE` and *adornable* its child. Independent from its multiplicity, *adornable* is visited only once. If the multiplicity of *adorned* is `*` or `+`, the transitive closure of *adornable*s matrix has to be calculated as shown in Listing 7, and this requires time $\mathcal{O}(|G|^3)$. As in the previous cases, concatenating *adorned*s matrix with its incoming matrix requires time $\mathcal{O}(|G|^3)$. So, $T(adorned) = T(adornable) + \mathcal{O}(|G|^3)$.

Case `ConcatenatedRPE`
Let *concatenated* be a `ConcatenatedRPE` and $adorned_1, \ldots, adorned_n$ its children. As can be seen from Listing 6, all children are visited only once (and each child takes care of concatenating its matrix with its incoming matrix). Concatenating *concatenated*s matrix with its incoming matrix requires time $\mathcal{O}(|G|^3)$, so $T(concatenated) = \sum_{i=1}^{n} T(adorned_i) + \mathcal{O}(|G|^3)$.

Case `FlavoredRPE`
As in the case of a `DirectedRPE`, a `FlavoredRPE` *flavored* does not maintain a partial path matrix, and just calls its concatenated child: $T(flavored) = T(concatenated)$.

In the cases above, we can see the three following results.

1. The time effort of matching all terms of G against all `AtomicRPE`s of the RPL query q is $\mathcal{O}(|G| \cdot |q|)$, as already mentioned.

2. In the case of `PredicatesRPE`s and `DisjunctiveRPE`s, an additional time of $\mathcal{O}(|G|^2)$ is needed for each of their children. When having the abstract syntax tree of q in mind, we can image every edge leading from e.g. a `DisjunctiveRPE` to one of its children to be labeled with $\mathcal{O}(|G|^2)$. So, in the worst case, traversing each single edge of q's abstract syntax tree requires time $\mathcal{O}(|G|^2)$, which sums up to $\mathcal{O}(|G|^2 \cdot |q|)$ in total (and each single edge is only traversed once, since in each of the above cases, each child node is only visited once).

   Please note that we do not have to further consider the $\mathcal{O}(|G|^2)$ terms in $T(predicates)$ and $T(disjunctive)$ from now on.

3. Further sticking to our picture of q's abstract syntax tree, we can image each node labeled with $\mathcal{O}(|G|^3)$, which is an upper bound for the remaining additional time complexity in each case. This finally leads to an overall time complexity of $\mathcal{O}(|G|^3 \cdot |q|)$, as claimed.

□

Example 20

Let us consider the RPL query `PATH (:a >:b :c | :d)`, denoted by q. Figure 16 illustrates the proof of Theorem 5 by showing q's abstract syntax tree, annotated with the complexity bounds given in the theorem. The abstract syntax tree of q is only traversed once, and it can be seen that $\mathcal{O}(|G|^3)$ is an upper bound for all its node and edge labels in Figure 16. Hence, it is obvious that $\mathcal{O}(|G|^3 \cdot |q|)$ is an upper bound for evaluating the whole RPL query q on an RDF graph G, in accordance with Theorem 5.

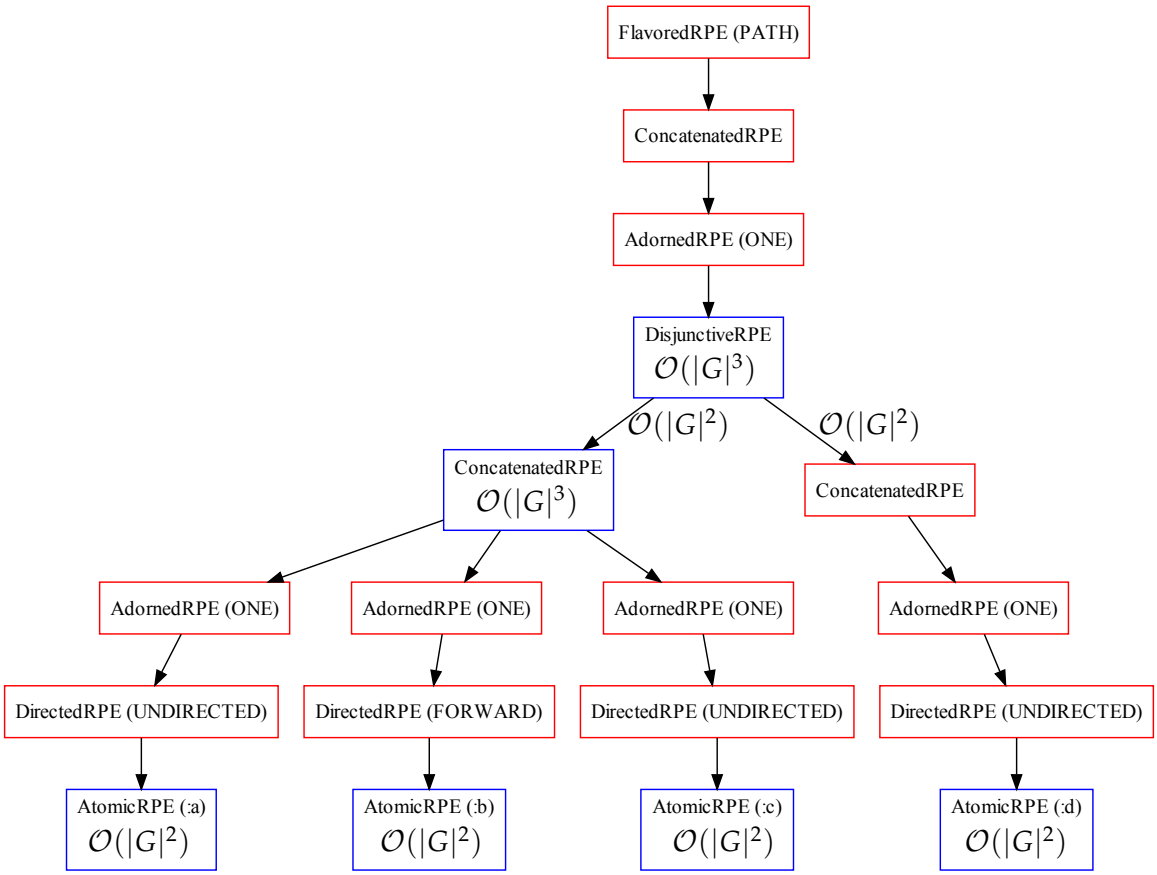


Figure 16: Abstract syntax tree of `PATH (:a >:b :c | :d)`, annotated with complexity results from Theorem 5

4

RPL AS EMBEDDED LANGUAGE

SPARQL is so far the only standardized RDF query language. Surprisingly, it is designed in the way of relational languages like SQL, and pays only little attention to the graph-based data model of RDF.

Though a SPARQL query already appeared in the motivating example within the introduction of this thesis (Section 1.2), the following snippet again shows how we intend to embed RPL into SPARQL.

```
1 SELECT ?a ?b WHERE
2   {
3      ?a [q] ?b
4   }
```

Listing 8: Basic idea how to embed a RPL query q into SPARQL

We thus want to allow a RPL query q to appear at predicate position within a SPARQL triple pattern.

Regarding the syntactic view (Section 4.1), we have to ensure that the square brackets we intend to use for separating q from its enclosing SPARQL query do not interfere with (parts of) the SPARQL grammar.

The intended semantics of Listing 8 is quite obvious: we want the SPARQL variables `?a` and `?b` to be bound to the first and second component of each of q's result pairs, respectively. In Section 4.2, we formally define this intended semantics by extending SPARQL with RPL triple patterns (line 3).

In Section 4.3, we cover the implementation of this extension and give several examples. In this context, Sesame has been chosen due to its clean and modular architecture [8] among several SPARQL implementations that have been examined.

Finally, we show that it is possible to imitate (an essential core of) the RDFS semantics via RPL in Section 4.4.

4.1 SYNTACTIC EMBEDDING

Although square brackets are already used by the SPARQL grammar [29] for the ⟨*BlankNodePropertyList*⟩ and ⟨*ANON*⟩ rules, the $LL(1)$ condition is not violated when extending the ⟨*Verb*⟩ rule as highlighted in blue within Figure 17. This is because there is no rule within the SPARQL grammar, from which (over arbitrary many intermediate steps) ⟨*Verb*⟩ and ⟨*BlankNodePropertyList*⟩ or ⟨*ANON*⟩ can be derived. In other words, using square brackets to embed RPL expressions at predicate position into SPARQL is possible, since RDF and hence SPARQL do not allow blank nodes at this position.

⟨*BlankNodePropertyList*⟩ ::= '[' ⟨*PropertyListNotEmpty*⟩ ']'

⟨*ANON*⟩ ::= '[' ⟨*WS*⟩* ']'

⟨*Verb*⟩ ::= ⟨*VarOrIRIref*⟩ | 'a' | '[' ⟨*RPE*⟩ ']'

Figure 17: Syntactic Embedding of RPL: Relevant part of the SPARQL grammar [29]

The ⟨*RPE*⟩ rule in Figure 17 corresponds to the ⟨*flavored*⟩ rule in Definition 1. Applying the extension shown in Figure 17, SPARQL however only serves as host language: an embedded RPL expression can not access any information from its enclosing SPARQL query. So, in a second step, it seems natural to allow RPL access to SPARQL variables. Concerning the syntax, this can be achieved by extending RPL as shown (highlighted in blue) in Figure 18.

⟨*atomic*⟩ ::= ⟨*STRING_LITERAL1*⟩ | ⟨*STRING_LITERAL2*⟩ | ⟨*REGEXP*⟩ | '_' | ⟨*IRI_REF*⟩ | ⟨*PNAME_NS*⟩ | ⟨*PNAME_LN*⟩ | ⟨*PNAME_RX*⟩ | ⟨*VAR1*⟩ | ⟨*VAR2*⟩

Figure 18: Extending RPL with variables

⟨*VAR1*⟩ and ⟨*VAR2*⟩ are defined in the SPARQL grammar [29] and allow for variables starting with a ? and $ character, respectively. As no RPL token starts with any of these characters, the $LL(1)$ condition of the extended RPL grammar is still satisfied.

4.2 SEMANTIC EMBEDDING

Although a set-based semantics of SPARQL has been given in [26], we directly refer to the official multiset-based semantics given in the W3C SPARQL Recommendation [29].

Semantic embedding into SPARQL is done by introducing RPL triple patterns (Definition 24), a new type of triple patterns. SPARQL triple patterns are given in Definition 23, which, as most other definitions, corresponds to the one given in [29]. Hence, basic graph patterns (Definition 25) now allow for SPARQL triple patterns as well as RPL triple patterns.

The centerpiece of this section consists of Definitions 27 and 31, in which the semantics of a basic graph pattern containing at least one RPL triple pattern is settled.

As stated by [3], an arbitrary SPARQL query Q can be simulated by a SPARQL query Q' without blank nodes in its graph pattern. If P is the graph pattern of Q, then Q' is the same SPARQL query where each blank node b in P has been replaced by a fresh variable v_b. In case Q is of the query form **SELECT** or **DESCRIBE** together with the parameter $*$, then Q' should explicitly list all variables of the original pattern P (otherwise, also the fresh variables replacing the blank nodes of P would be returned).

Definition 23 (SPARQL Triple Pattern [29])
A *SPARQL triple pattern* is a member of the set $(T \cup V) \times (I \cup V) \times (T \cup V)$, where

- I is the set of all IRIs, which is a subset of the RDF URI references that omits spaces
- L is the set of all RDF literals
- B is the set of all RDF blank nodes
- T is the set of RDF terms, $T := I \cup L \cup B$
- V is the infinite set of all query variables, with $V \cap T = \emptyset$

Definition 24 (RPL Triple Pattern)
A *RPL triple pattern* is a member of the set $(T \cup V) \times R \times (T \cup V)$, where T and V are as in Definition 23 and R is the infinite set of all valid RPL queries.

Definition 25 (Basic Graph Pattern, Solution Mapping, Multiset)

A *basic graph pattern* is a finite set where each element is a triple pattern (i.e. a SPARQL or RPL triple pattern). By $var(B)$, we denote the set of all variables occurring in B.

A *solution mapping* μ is a partial function $\mu : T \cup V \to T$. μ is the identity on T, and the *domain* of μ, denoted as $dom(\mu)$, is the subset of V where μ is defined.

A *multiset* is an unordered collection of elements in which each element may appear more than once. It is described by a set of elements Ω and a cardinality function $card_\Omega : \Omega \to \mathbb{N}$ giving the number of occurrences of each element from Ω.

Definition 26 (RDF Dataset, Active, Default and Named Graph [29])

An *RDF dataset* is a set $\{G, (u_1, G_1), (u_2, G_2), \ldots, (u_n, G_n)\}$, where G and each G_i are graphs, and each u_i (pairwise distinct) is an IRI. G is called the *default graph*, and each (u_i, G_i) is called a *named graph*.

The *active graph* is the graph from the dataset which is used for basic graph pattern matching.

Definition 27

Let B be a basic graph pattern consisting of exactly one RPL triple pattern (s, rpe, o) where all blank nodes have been replaced by fresh variables and let $D(G)$ denote a dataset D having G as active graph. The semantics $eval(D(G), B)$ of B with respect to $D(G)$ is defined as the multiset of solution mappings $\Omega := \{\mu \mid dom(\mu) = var(B) \wedge (\mu(s), \mu(o)) \in [\![rpe]\!]_G\}$, with $card_\Omega(\mu) := 1$ for all $\mu \in \Omega$.

Definition 28

Let B be a basic graph pattern consisting of SPARQL triple patterns only, where each blank node has been replaced by a fresh variable. Let $D(G)$ denote a dataset D having G as active graph. The semantics of $eval(D(G), B)$ is defined as $\Omega := \{\mu \mid \mu(B)$ is a subgraph of G and $dom(\mu) = var(B)\}$ where $\mu(B)$ is the set of triples that is obtained by replacing the variables in the triple patterns of B according to the solution mapping μ, with $card_\Omega(\mu) := 1$ for all $\mu \in \Omega$.

Definition 29 (Compatible Solution Mappings [29])

Two solution mappings μ_1 and μ_2 are *compatible* iff $\mu_1(v) = \mu_2(v)$ for all $v \in dom(\mu_1) \cap dom(\mu_2)$. Two compatible solution mappings μ_1 and μ_2 can be

merged into a new solution mapping $\mu = merge(\mu_1, \mu_2)$ with

$$\mu(v) := \begin{cases} \mu_1(v) & \text{if } v \in dom(\mu_1) \\ \mu_2(v) & \text{if } v \in dom(\mu_2) \end{cases} \text{ and } dom(\mu) = dom(\mu_1) \cup dom(\mu_2).$$

Definition 30 (SPARQL Join Operator [29])
Let Ω_1 and Ω_2 be multisets of solution mappings. We define $Join(\Omega_1, \Omega_2)$ as

$$Join(\Omega_1, \Omega_2) := \{merge(\mu_1, \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ compatible}\}$$

and $$card_{Join(\Omega_1,\Omega_2)}(\mu) := \sum_{\substack{\mu = merge(\mu_1,\mu_2) \\ \mu_1 \in \Omega_1, \mu_2 \in \Omega_2}} card_{\Omega_1}(\mu_1) \cdot card_{\Omega_2}(\mu_2).$$

Definition 31 (Semantics of a Basic Graph Pattern)
Let B be a basic graph pattern, $B = \{r_1, \ldots, r_n, s_1, \ldots, s_m\}$, $(n, m \geq 0)$, each r_i $(1 \leq i \leq n)$ a RPL triple pattern, and each s_j $(1 \leq j \leq m)$ a SPARQL triple pattern. Each element of B should have its blank nodes replaced by fresh variables. Let $D(G)$ denote a dataset D having G as active graph. The semantics $eval(D(G), B)$ of B with respect to $D(G)$ is inductively defined as

Case $n = 0$
$eval(D(G), B)$ is given by Definition 28.

Case $n \geq 1$
$eval(D(G), B) := Join\Big(eval\big(D(G), \{r_1\}\big), eval\big(D(G), \{r_2, \ldots, r_n, s_1, \ldots, s_m\}\big)\Big)$
where $eval(D(G), \{r_1\})$ is given by Definition 27.

4.3 IMPLEMENTATION & EXAMPLES

After having discussed the syntactic and semantic dimension of embedding RPL into SPARQL, the necessary extensions are to be implemented in one of the available SPARQL implementations. Sesame[1] has been chosen due to its clean and modular architecture [8] among several SPARQL implementations that have been examined.

Like for RPL (Section 2.3), JavaCC is also used within Sesame for parsing SPARQL queries. However, it is used in conjunction with JJTree, which — as a preprocessor — works on top of JavaCC and augments it by automatically generating an abstract syntax tree skeleton. As a consequence, the abstract syntax generated by JJTree for RPL expressions first has to be translated into the abstract syntax presented in Figure 2.

1 see http://www.openrdf.org/

In order to access SPARQL variables within embedded RPL expressions, the enum `ATOMIC` (Figure 2) is extended by the literal `VARIABLE`, which is used when a ⟨*VAR1*⟩ or ⟨*VAR2*⟩ token is found in the query input stream. Each `AtomicRPE` of type `VARIABLE` is also able to hold the variable's value, if available.

The semantic analysis phase (Section 3.1) now includes *binding verification* as its first pass, which cares about the assignment of values to variables. If no value is available, the variable cannot be bound and the corresponding `BindingVisitor` will throw an exception as shown in Example 21.

Additionally, the namespace resolution pass (Section 3.1.5) now relies on the namespace prefixes defined in the enclosing SPARQL query rather than those from an RDF graph.

Example 21

As the variable `?t` is not bound within the SPARQL query below, binding verification fails and leads to the following error message.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?a ?b WHERE
  { ?a [PATH _ (>[PATH (_ >rdfs:subPropertyOf)* ?t] _)+] ?b }
```

```
Variable "?t" is not bound:

PATH _ (>[PATH (_ >rdfs:subPropertyOf)* ?t] _)+
                                       ^^
```

Let us demonstrate the interaction between SPARQL and RPL.

Example 22

Consider the SPARQL query shown below. Variable `?t` will get bound to subproperties of `:transport` in line 4, and the RPL triple pattern in line 5 will bind the endpoints of all paths satisfying its RPL query to the variables `?a` and `?b`. This means, we are querying all paths of non-zero length whose edges are subproperties of the same means of transport `?t`.

```
1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX : <http://example.org/>
3 SELECT ?a ?b ?t WHERE
4  { ?t rdfs:subPropertyOf :transport .
5    ?a [PATH _ (>[PATH (_ >rdfs:subPropertyOf)* ?t] _)+] ?b }
```

Applied to our sample RDF graph within this thesis (Figure 6), the above query yields the following result.

```
[b=http://example.org/Calais;   t=http://example.org/train; a=http://example.org/Paris]
[b=http://example.org/Dijon;    t=http://example.org/train; a=http://example.org/Paris]
[b=http://example.org/Dover;    t=http://example.org/ferry; a=http://example.org/Calais]
[b=http://example.org/Hastings; t=http://example.org/bus;   a=http://example.org/Dover]
[b=http://example.org/London;   t=http://example.org/bus;   a=http://example.org/Dover]
```

When evaluating the single basic graph pattern in lines 4 – 5, no reordering of its triple patterns is implemented, i.e. the order of evaluation is solely determined by Sesame.

It is also possible to embed multiple RPL triple patterns (and thus multiple RPL queries) into a single SPARQL query, as the following Example 23 illustrates.

EXAMPLE 23

We modify the SPARQL query from Example 22 by replacing the SPARQL triple pattern `?t rdfs:subPropertyOf :transport` with an equivalent RPL triple pattern as shown below. Please be aware that `[]` in line 4 appears at object position and describes a blank node — it is not part of any RPL query.

```
1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX : <http://example.org/>
3 SELECT ?a ?b ?t WHERE
4  { ?t [PATH _ >rdfs:subPropertyOf :transport] [] .
5    ?a [PATH _ (>[PATH (_ >rdfs:subPropertyOf)* ?t] _)+] ?b }
```

The result, of course, will stay unchanged (compare Example 22).

In general, both evaluation approaches (ENRE-based and Path-based evaluation, see Sections 3.3 and 3.4) are convenient when evaluating RPL queries inside of SPARQL queries. However, the latter might be advantageous due to its memoization technique, if the RPL query references variables from its enclosing SPARQL query and if these variables get bound to several RDF terms during evaluation.

When the value of a variable v changes during multiple evaluations of the same RPL query q from which v is referenced, not all partial path matrices (Definition 19) linked to q's abstract syntax tree have to be reset, since at least some of them might contain paths that are still relevant to q after binding v to a different value as before. It suffices to reset only those partial path matrices which are located on the path starting at the `AtomicRPE` of the variable v and ending at the root node of q's abstract syntax tree.

The following Example 24 shows how memoization can not only be used within one single evaluation, but across multiple evaluations of the same RPL query, just with different bindings of its referenced variables.

Example 24

The RPL query q shown below is evaluated three times in total, with the variable `?t` being bound to `:train`, `:ferry`, and `:bus` (in this order). q describes all paths of non-zero length that start with the node `:Paris`, which is followed by arbitrary nodes. All intermediate edges represent an arbitrary means of transport, except the one that `?t` is bound to (indicated by the negated predicate `[!PATH (_ >rdfs:subPropertyOf)* ?t]`).

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://example.org/>
SELECT ?a ?b ?t WHERE {
  ?t rdfs:subPropertyOf :transport .
  ?a [PATH :Paris
          ( >[!PATH (_ >rdfs:subPropertyOf)* ?t][
               PATH (_ >rdfs:subPropertyOf)* :transport]
            _ )+
     ] ?b }
```

Figure 19 shows q's abstract syntax tree T, after q has been evaluated with `?t` bound to `:ferry`. A blue (red) border line around an arbitrary node n of T means that the partial path matrix of n has (not) been initialized during evaluation.

After `?t` has been bound to `:bus`, the partial path matrices of all nodes highlighted in gray have to be reset, as their paths are no longer valid for the new binding of `?t`.

According to the semantics of RPL expressions (Definition 9) and RPL triple patterns (Definition 24), it is also possible to bind the subject position of a RPL triple pattern to an RDF literal, which is not allowed in RDF but in SPARQL[2]. In contrast, an arbitrary SPARQL triple pattern with an RDF literal as subject will fail to match on any RDF graph.

2 The W3C SPARQL Recommendation [29] justifies this design decision by citing the following note of the RDF core Working Group:
"[The RDF core Working Group] noted that it is aware of no reason why literals should not be subjects and a future WG with a less restrictive charter may extend the syntaxes to allow literals as the subjects of statements."

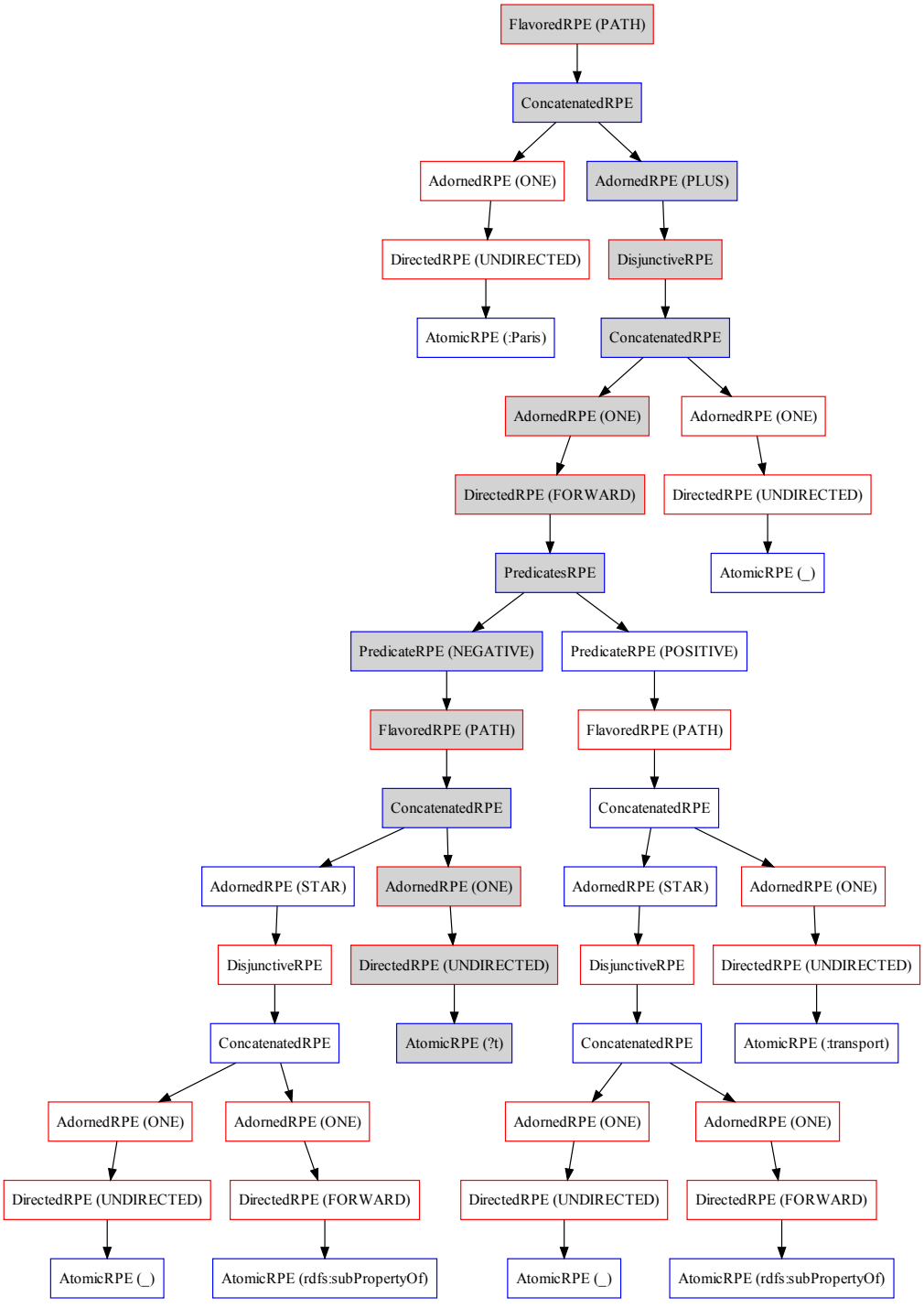


Figure 19: Memoization across multiple evaluations of the same RPL query

Example 25

The following SPARQL query makes use of the FOAF vocabulary[3] in order to retrieve the names (which are represented as RDF literals) of all magicians.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://example.org/>
SELECT ?a ?b WHERE {
  ?a [PATH _ <foaf:name [PATH _ >rdf:type :magician]] ?b }
```

The result looks like follows, when evaluated on the RDF graph belonging to the second application scenario of RPL's web-based interface (Section 6.1). This illustrates that RPL triple patterns can indeed return RDF literals in subject position.

```
[a="Saruman"; b=http://example.org/Person8]
[a="Gandalf"; b=http://example.org/Person9]
```

4.4 RDFS AND RPL

RDFS (RDF Schema) [20] is a semantic extension of RDF. It consists of a set of reserved IRIs (the RDFS vocabulary) that has a predefined semantics and can e.g. be used to express inheritance of classes and properties, domain and range restrictions, and typing (like in object-oriented programming).

Within this section, we just discuss the subset of the ρdf fragment [25] that has been considered in [27]. This subset consists of the vocabulary {`rdfs:sc`, `rdfs:sp`, `rdfs:range`, `rdfs:domain`, `rdf:type`} along with the inference rules presented in Table 4, which capture the essence of RDFS. The namespace prefixes `rdf:` and `rdfs:` are defined as usual, and `rdfs:sc` and `rdfs:sp` are short for `rdfs:subClassOf` and `rdfs:subPropertyOf`, respectively.

SUBPROPERTY	$\frac{(?a,\texttt{rdfs:sp},?b) \quad (?b,\texttt{rdfs:sp},?c)}{(?a,\texttt{rdfs:sp},?c)}$	$\frac{(?a,\texttt{rdfs:sp},?b) \quad (?x,?a,?y)}{(?x,?b,?y)}$
SUBCLASS	$\frac{(?a,\texttt{rdfs:sc},?b) \quad (?b,\texttt{rdfs:sc},?c)}{(?a,\texttt{rdfs:sc},?c)}$	$\frac{(?a,\texttt{rdfs:sc},?b) \quad (?x,\texttt{rdf:type},?a)}{(?x,\texttt{rdf:type},?b)}$
TYPING	$\frac{(?a,\texttt{rdfs:domain},?b) \quad (?x,?a,?y)}{(?x,\texttt{rdf:type},?b)}$	$\frac{(?a,\texttt{rdfs:range},?b) \quad (?x,?a,?y)}{(?y,\texttt{rdf:type},?b)}$

Table 4: RDFS inference rules

3 see http://www.foaf-project.org/

The inference rules shown in Table 4 can be grouped into rules that deal with subproperty and subclass relations, and typing. `?a`, `?b`, `?c`, `?x`, and `?y` stand for *variables*. Each rule consists of a set of *premises* (all triples above the horizontal line) and a *conclusion* (the triple below the horizontal line). Once a rule is *instantiated*, all variables that are contained within its triples get replaced by RDF terms (Definition 23). A rule r can be *applied* to an RDF graph G, if it can be instantiated to $\frac{P_1 \ldots P_n}{C}$ such that all of its premises are satisfied, i.e. $P_i \subseteq G$ $(1 \leq i \leq n)$. The result of this application is an RDF graph G' where the conclusion C holds, i.e. $G' := G \cup C$.

As already formulated for nSPARQL in [27] and indicated for RPL in [9], RPL is able to imitate these inference rules without previously computing the RDFS closure of G (this means applying the rules from Table 4 until G does not change any more).

Definition 32

Let $t = (s, p, o)$ be a SPARQL triple pattern (Definition 23) with $s, o \in T \cup V$ and $p \in I$. The translation function *mimic* maps a SPARQL triple pattern that does not contain a variable in predicate position to a SPARQL graph pattern [29] as follows.

Case $p =$ `rdf:type`
$mimic((s, \texttt{rdf:type}, o)) :=$

```
{
    {s [EDGES >rdf:type >rdfs:sc*] o}
    UNION
    {s ?p [] . ?p [EDGES >rdfs:sp* >rdfs:domain >rdfs:sc*] o}
    UNION
    {[] ?p s . ?p [EDGES >rdfs:sp* >rdfs:range >rdfs:sc*] o}
}
```

Case $p \neq$ `rdf:type`

$$mimic(t) := \begin{cases} \{s \texttt{ [EDGES >rdfs:sc+] } o\} & \text{if } p = \texttt{rdfs:sc} \\ \{s \texttt{ [EDGES >rdfs:sp+] } o\} & \text{if } p = \texttt{rdfs:sp} \\ \{s \texttt{ [EDGES >rdfs:domain] } o\} & \text{if } p = \texttt{rdfs:domain} \\ \{s \texttt{ [EDGES >rdfs:range] } o\} & \text{if } p = \texttt{rdfs:range} \\ \{s \texttt{ [EDGES >[PATH (_ >rdfs:sp)*} \ p\texttt{]] } o\} & \text{otherwise} \end{cases}$$

Theorem 6

Let $t = (s, p, o)$ be a SPARQL triple pattern (Definition 23) with $s, o \in T \cup V$ and $p \in I$. Let $eval(D(closure(G)), \{t\})$ denote the evaluation of the basic graph pattern $\{t\}$ over a dataset D having $closure(G)$ (by $closure(G)$, we denote the RDFS closure of G) as active graph.

Then, $eval(D(closure(G)), \{t\}) = eval(D(G), mimic(t))$.

Proof. The proof is omitted here for the sake brevity, as it is already given for NREs (Nested Regular Expressions) in [27] and easily adapts to RPL expressions. □

Example 26

The following SPARQL query retrieves all coastal cities, according to RDFS semantics. It has been constructed via applying the *mimic* function (Definition 32) to the SPARQL triple pattern `?c rdf:type :coastal_city`.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX : <http://example.org/>
4  SELECT ?c WHERE {
5    {?c [EDGES >rdf:type >rdfs:subClassOf*] :coastal_city}
6    UNION
7    {?c ?p [] . ?p [EDGES >rdfs:subPropertyOf* >rdfs:domain >rdfs:subClassOf*] :coastal_city}
8    UNION
9    {[] ?p ?c . ?p [EDGES >rdfs:subPropertyOf* >rdfs:range >rdfs:subClassOf*] :coastal_city}
10 }
```

`:Hastings` has an `rdf:type` edge to `:coastal_city` and is found via the first graph pattern (line 5), while `:Calais` and `:Dover` can only be retrieved via the second (line 7) and third (line 9) graph pattern, respectively. The result is shown below.

```
[c=http://example.org/Hastings]
[c=http://example.org/Calais]
[c=http://example.org/Dover]
```

5

EXPERIMENTAL EVALUATION

This chapter presents *RPLgen* (Section 5.1), an RDF data generator based on RPL queries. It is able to create richly structured, path-based RDF data, and has been applied to generate the datasets (Section 5.2) for experimentally verifying the complexity bounds of all presented evaluation algorithms (Section 5.3).

5.1 RPLGEN – GENERATING RDF DATA FROM RPL QUERIES

Each RPL query describes a certain set of paths within an RDF graph. The new view that *RPLgen* brings into play is to interpret these paths in a generative way, i.e. materialize some of them into RDF data.

For instance, the RPL query q

```
1 PATH :Paris
2      ( >[PATH (:bus | :ferry | :train) >rdfs:subPropertyOf :transport]
3        :/city[0-9]+/
4      )+
```

generates an RDF graph similar to the one from Figure 6, as it describes a path p of non-zero length. The first node of p is `:Paris` (line 1), while all following nodes are labeled with `:city` followed by an arbitrary, non-empty sequence of digits (indicated by `:/city[0-9]+/` in line 3). All edges of p have to satisfy paths p', which are described by the predicate in line 2. However, each p' is a path of length one, starting with a node either labeled `:bus`, `:ferry`, or `:train`, which is connected to a `:transport` node via an `rdfs:subPropertyOf` edge.

The RPL query q presented above can indeed be used as an input for RPLgen. Figure 20 shows the resulting RDF graph, when calling RPLgen with one instance of q, a success probability of 0.3 and an initial seed of 98765 (these parameters are described in the following).

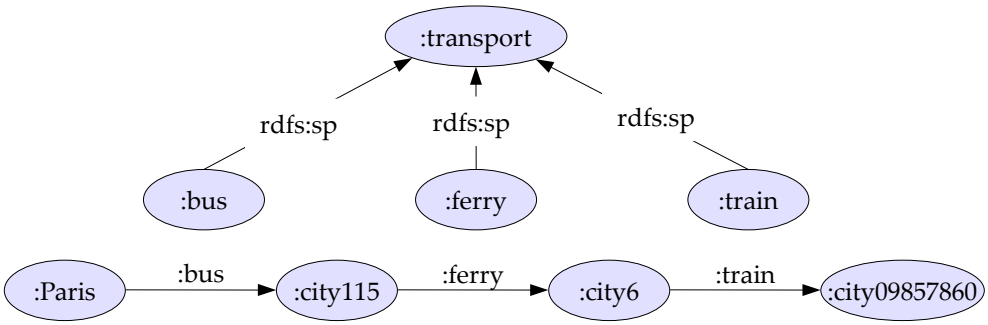


Figure 20: An example RDF graph created by RPLgen

RPLgen is an RDF data generator that is based on using RPL queries in a generative way. An important idea has been adopted from the SP^2 Bench Generator[1], an RDF data generator for DBLP-like data: repeatability, i.e. when called with the same parameters, the same data will be generated.

Hence, the first of RPLgen's parameters is a *seed* value to initialize the random number generator. When RPLgen is run several times with identical seed, each single "randomized" decision is deterministic and will lead to the same outcome — thus, the generated data will be identical as well.

The second parameter that RPLgen requires is a *success probability* p, which controls how often Kleene star and plus operators are expanded. This applies to regular expressions over strings (used in `AtomicRPEs`), as well as for the multiplicities within RPL expressions (used in `AdornedRPEs`).

More precisely, the probability that a Kleene star operator is expanded k times is given by $\mathcal{G}_p(\{k\})$ in the probability space $(\mathbb{N}, 2^{\mathbb{N}}, \mathcal{G}_p)$, where $2^{\mathbb{N}}$ denotes the power set of $\mathbb{N}$. $\mathcal{G}_p$ is called the *geometric distribution* [18] with parameter p and is defined as $\mathcal{G}_p(\{k\}) := p(1-p)^k$.

In case of a Kleene plus operator, we just need to shift $\mathcal{G}_p$ one unit to the right. So, the probability that a Kleene plus operator is expanded k times is given by $\mathcal{G}_p(\{k-1\}) = p(1-p)^{k-1}$.

However, the application of the geometric distribution is limited to Kleene star and plus operators – all remaining random decisions range over a finite set of alternatives, and each of these alternatives should obtain the same probability (leading to a *uniform distribution*).

RPLgen's third parameter is a list of RPL queries, where each query q is equipped with a number stating how many instances of q are to be generated.

1 see http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/data.php

EXAMPLE 27
This example gives a lookup table for the geometric distribution with parameters $p = 0.1$ and $p = 0.2$.

k	0	1	2	3	4	5
$\mathcal{G}_{0.1}(\{k\})$	0.1	0.09	0.081	0.0729	0.06561	0.05905
$\mathcal{G}_{0.2}(\{k\})$	0.2	0.16	0.128	0.1024	0.08192	0.06554

As already mentioned, regular expressions occur at two levels: a RPL expression itself can be interpreted as a regular expression, as well as all of its atomic subexpressions. The implementation of these two levels is discussed in the following Sections 5.1.1 and 5.1.2.

5.1.1 *Regular Expressions over Strings*

RPLgen has to do some preprocessing concerning regular expressions over strings, the lower of these two levels. The `dk.brics.automaton` implementation [24] represents regular expressions as a binary tree (i.e. each node must not have more than two children). If a regular expression contains more than two alternatives (at any depth), not of all of these will be at the same level within its abstract syntax tree. This representation is quite uncomfortable since each of these alternatives should be chosen with the same probability. So, before generating strings from regular expressions, the latter have to be flattened by translating them into a tree structure where each node can have arbitrary many children.

The `AtomicRPE` `_` (the wildcard) matches each node or edge label and thus can be seen as the regular expression `.*`, where `.` matches any character from the character classes `[a-z]`, `[A-Z]`, and `[0-9]`.

5.1.2 *RPL Queries as Regular Expressions*

In contrast, no further preprocessing of RPL queries is required, as these are already "flattened" after the simplification pass (Section 3.1.1).

RPL predicates with a negative sign (`[!...]`) are, when using them in a generative way like RPLgen does, treated as if they carried a positive sign. Otherwise, it would be unclear which part of the predicate the user has intended to fail.

When a `PredicatesRPE` is made up of multiple `PredicateRPEs` $p_1, \ldots, p_n$ ($n \geq 2$), only p_1 is guaranteed to be satisfied when evaluating its parent RPL query. The semantics of RPL actually defines that all predicates $p_1, \ldots, p_n$ should be satisfied, but this would have meant a lot of additional programming effort. The intersection of two regular expressions r_1, r_2 (which could represent the first atomic expressions of p_1 and p_2) is usually computed by converting r_1 and r_2 into NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, which in turn are determinised into DFAs $\mathcal{D}_1$ and $\mathcal{D}_2$, and combined into a product automaton $\mathcal{P}$ that accepts the language $\mathcal{L}(r_1) \cap \mathcal{L}(r_2)$. Finally, $\mathcal{P}$ would have to be converted back into a regular expression r, which allows for usage in a generative way.

5.2 EVALUATION SETUP

Before presenting its results in Section 5.3, we need to concisely describe the setup of the experimental evaluation.

Therefore, we present the query that is used both as an input for RPLgen and later for querying the generated datasets (Section 5.2.1).

Moreover, a little pseudo code fragment illustrates how time measurement is conducted (Section 5.2.2).

5.2.1 *Data Generation*

The generated data has a similar structure as the introductory example from Figure 20. More precisely, RPLgen is initialized with the following RPL query.

```
1 PATH :/c[0-9]+/
2      ( >[PATH (:/t[0-9]+/ >rdfs:subPropertyOf)* :transport]
3        :/c[0-9]+/
4      )+
```

Listing 9: RPL query serving as input for RPLgen

This query q describes a path of non-zero length, where each node should represent a city and is labeled with `:/c[0-9]+/` (`c` followed by an arbitrary, non-empty sequence of digits, see line 3). Each edge of the path represents a transportation service and has to satisfy a predicate q', which describes a path consisting of zero or more `>rdfs:subPropertyOf` edges that finally reach a node labeled `:transport` (line 2). Each node of q' (except the last one) is labeled with `:/t[0-9]+/` (`t` followed by an arbitrary, non-empty sequence of digits, see line 2).

Different datasets resulted by varying the number of instances of q that were to be created. Each dataset, however, uses the same success probability of 0.1 and the same seed of 987654321. The following Table 5 gives a mapping from the number of q's instances that were created to the number of terms $|terms(G)|$ of the resulting RDF graph G. $|terms(G)|$ serves as an estimation for the size of G, as given by Definition 5.

INSTANCES OF q	4	8	12	16	20	24	28	32	36
$\|terms(G)\|$	382	494	788	1068	1425	1688	1890	2050	2170

INSTANCES OF q	40	44	48	52	56	60	64	68
$\|terms(G)\|$	2262	2654	3098	3287	3476	3666	3970	4557

INSTANCES OF q	72	76	80	84	88	92	96	100
$\|terms(G)\|$	4647	4811	5164	5604	6111	6338	6534	6907

Table 5: Relation between the number of q's instances and the number of terms of the resulting RDF graph G

The resulting datasets from Table 5 are queried with the same query q that has already been used for their generation, see Listing 9. Hence, the evaluation of q on each dataset involves the entire dataset, i.e. all of its nodes and edges need to be considered.

5.2.2 *Time Measurement*

The time measurement is done according to the pseudo code presented in Listing 10. A library called JETM[2] (Java Execution Time Measurement Library) implements the stopwatch functionality in lines 7 and 9. JETM internally relies on the `System.nanoTime()` method, which returns "the current value of the most precise available system timer, in nanoseconds"[3].

2 see http://jetm.void.fm/index.html
3 see http://java.sun.com/javase/6/docs/api/java/lang/System.html#nanoTime%28%29

The `MEASUREMENT` procedure from Listing 10 is called with all datasets from Table 5, the RPL query string from Listing 9, and 25 iterations. During these iterations, JETM collects all stopwatch times and calculates the minimum, average, and maximum run times for each dataset and each evaluation approach: the ENRE-based evaluation (Section 3.3) and the two variants of the Path-based evaluation (Section 3.4).

```
   procedure MEASUREMENT(File[] dataset, String query, int iterations)
1  run once each evaluation approach of query on dataset[0]
2  for each dataset[i]
3      for each evaluation approach approach
4          for each iteration
5              create graph G from dataset[i]
6              parse q from the query string query
7              start stopwatch
8                  evaluate q on G via approach
9              stop stopwatch
```

Listing 10: Pseudo code for time measurement

The first line of Listing 10 ensures that the very first measurement for each evaluation approach is not affected by any classloading times. To prevent memoization from taking place beyond different iterations (line 4), a new instance of an RDF graph is generated from the current dataset and the query string is parsed into a new RPL query at the beginning of each iteration (compare lines 5 – 6).

The time measurement has been carried out on an Intel Pentium Dual Core CPU T2330, with each core clocked at 1.60 GHz. A 64-bit edition of the Java Development Kit, version 1.6.0_17, was run with an initial and maximum heap size of 1.5 GB (by the command line switches `-Xms1536m` and `-Xmx1536m`, respectively).

5.3 RESULTS

The (average) resulting run times measured by JETM under the setup described in the previous section are shown in Figure 21.

The upper diagram contains all values that were captured during the ENRE-based evaluation of the RPL query from Listing 9 on all datasets from Table 5. As the ENRE-based approach is shown to have a time complexity quadratic

in the size of the data (Section 3.3.4), a quadratic polynomial is fitted to the acquired values.

Similarly, the lower diagram contains all values that were captured for both variants of the Path-based evaluation. The first series of measurements refers to the algorithm from Section 3.4.2, the first variant of the Path-based evaluation, for which an exponential upper bound has been shown in Section 3.4.3. The second series of measurements refers to the second variant of the Path-based evaluation (Section 3.4.4), for which cubic data time complexity has been shown. Similar to the upper diagram, a cubic polynomial is fitted to each series of measurements.

Figure 21 hence experimentally verifies the shown time complexity bounds for all of the presented evaluation approaches. Furthermore, it depicts that the first variant of the Path-based evaluation (Section 3.4.2) seems to also have a time complexity that is cubic in the size of the data, although only an exponential upper bound has been formulated in this thesis (Theorem 4).

When comparing the upper and lower diagrams of Figure 21 with each other, keep in mind that the unit of the vertical axis in the upper diagram is *milliseconds*, while it is *seconds* in the lower diagram. Even though ENRE-based evaluation outperforms Path-based evaluation on large enough datasets, the latter approach might be favorable if (1) it benefits from its memoization approach, e.g. when a RPL query is evaluated more than once in embedded mode, and the bindings of imported SPARQL variables change in between (compare Section 4.3), and (2) the selectivity of a RPL query is higher than on the generated datasets.

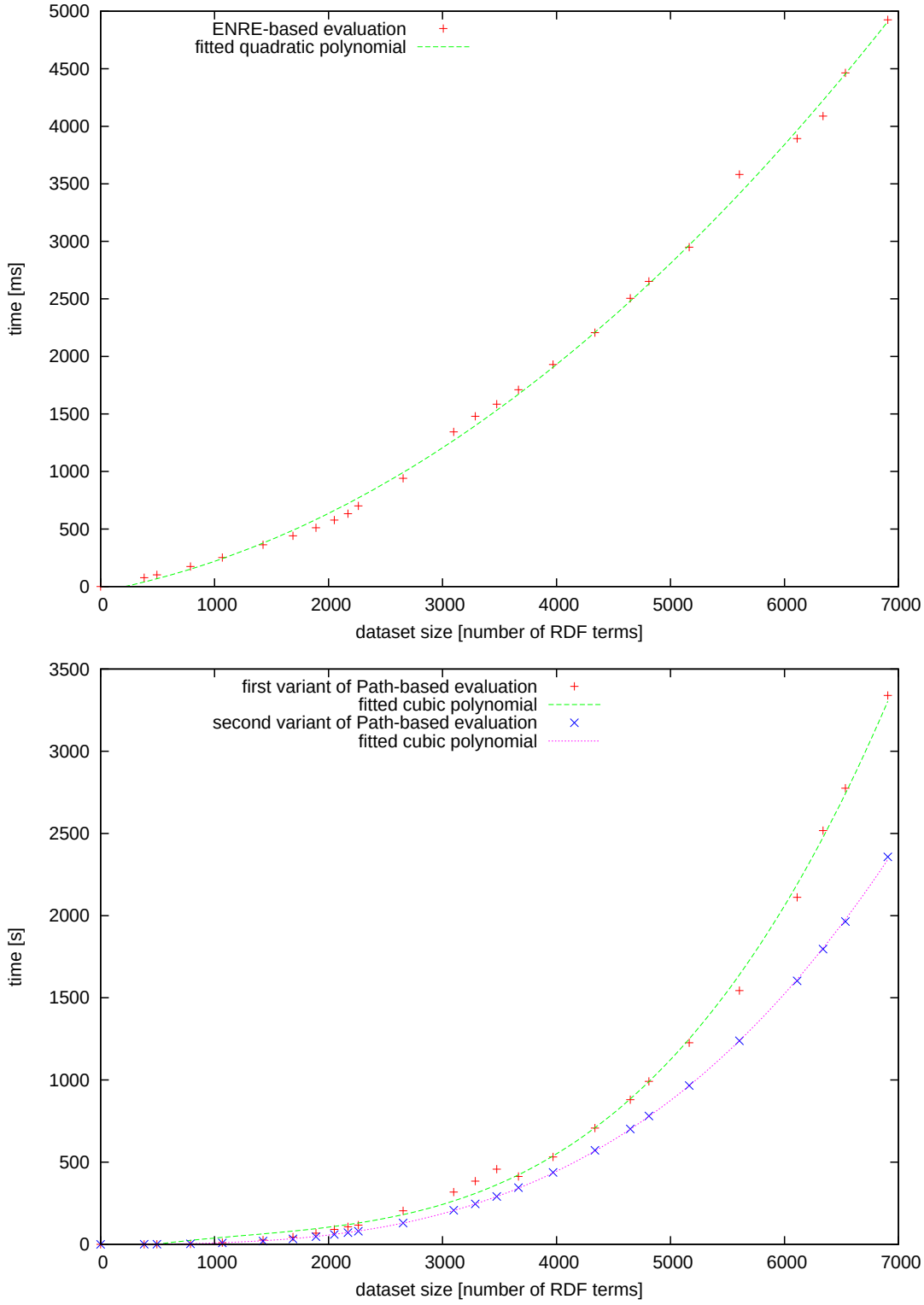


Figure 21: The two diagrams show the resulting run times of the RPL query from Listing 9 on all datasets from Table 5. The ENRE-based run times are shown in the upper, the Path-based run times (for both variants) in the lower diagram.

6

LEARNING RPL

This chapter briefly presents three tools that have been created for learning RPL and to ease the authoring of RPL expressions. All tools (the web-based interface presented in Section 6.1, and the two Eclipse plugins presented in Sections 6.2 and 6.3) are available online at `http://rpl.pms.ifi.lmu.de/`.

6.1 WEB-BASED INTERFACE

The web-based interface of RPL is targeted at users willing to cast an eye over RPL. It includes two application scenarios (named *Transport* and *FOAF*, like the FOAF vocabulary[1]), each consisting of an RDF dataset along with several predefined RPL queries, as shown in Figure 22.

Clicking the *Submit Query* button submits the blue form in the upper half of Figure 22 via an AJAX POST request. At the server side, a single servlet processes the incoming request, and sends back a response, encoded in XML. In the absence of errors, this response consists of the simplified and normalized RPL query, its corresponding ENRE (along with the time required for translation), and the result of the evaluation (again along with its required time), as shown in Figure 23. Otherwise, an error dialog containing a message like those shown in Section 3.1 appears.

This web-based interface can be easily deployed on any Java servlet engine (e.g. Apache Tomcat[2]) by installing a single web archive file, which in turn can be built via Apache Ant[3]. User interaction and AJAX communication on client side are built on top of a javascript framework called jQuery[4]. RPL's web-based interface has been tested to work with any modern browser.

1 see `http://www.foaf-project.org/`
2 see `http://tomcat.apache.org/`
3 see `http://ant.apache.org/`
4 see `http://jquery.com/`

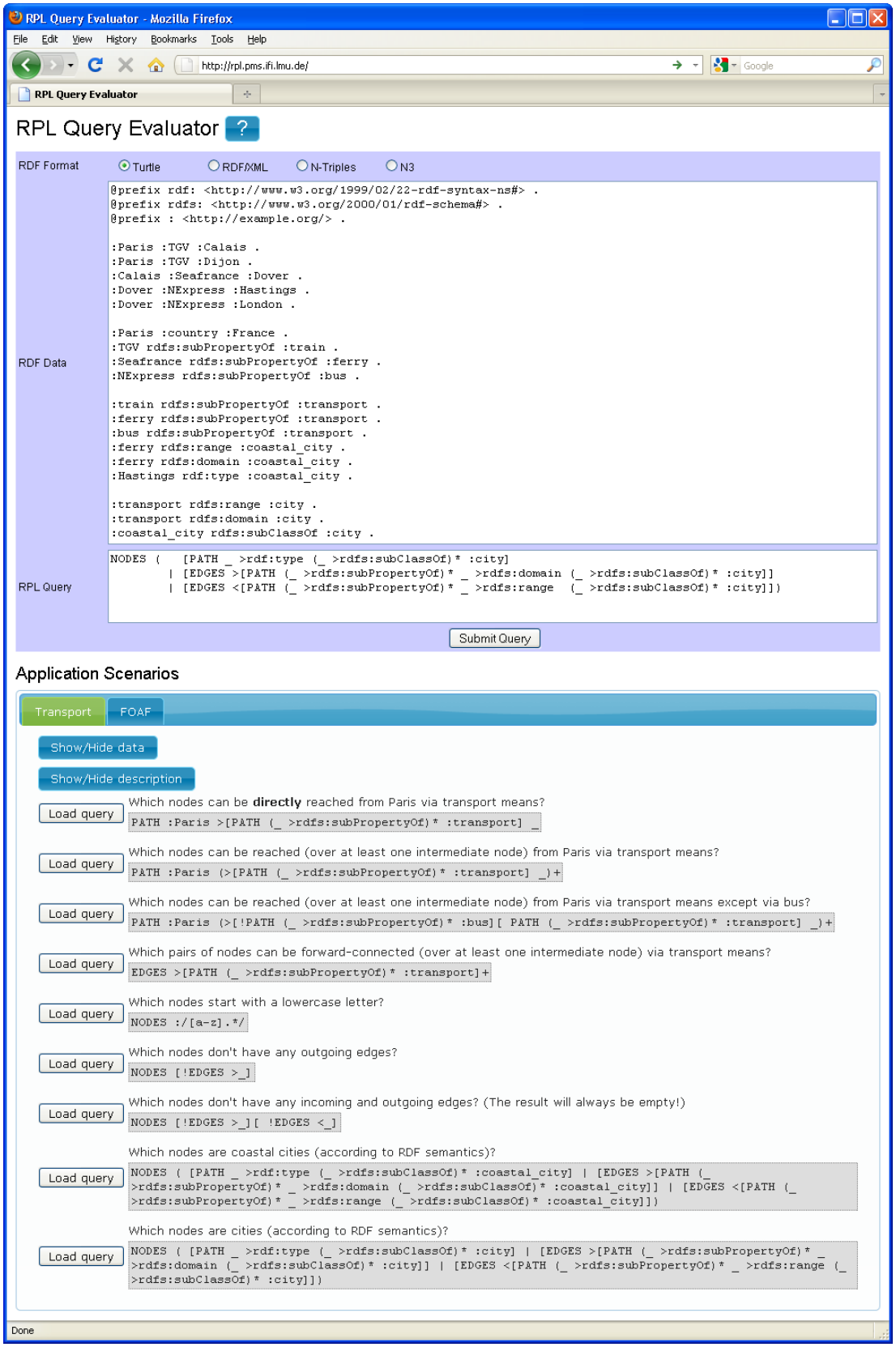


Figure 22: Screenshot of RPL's web-based interface

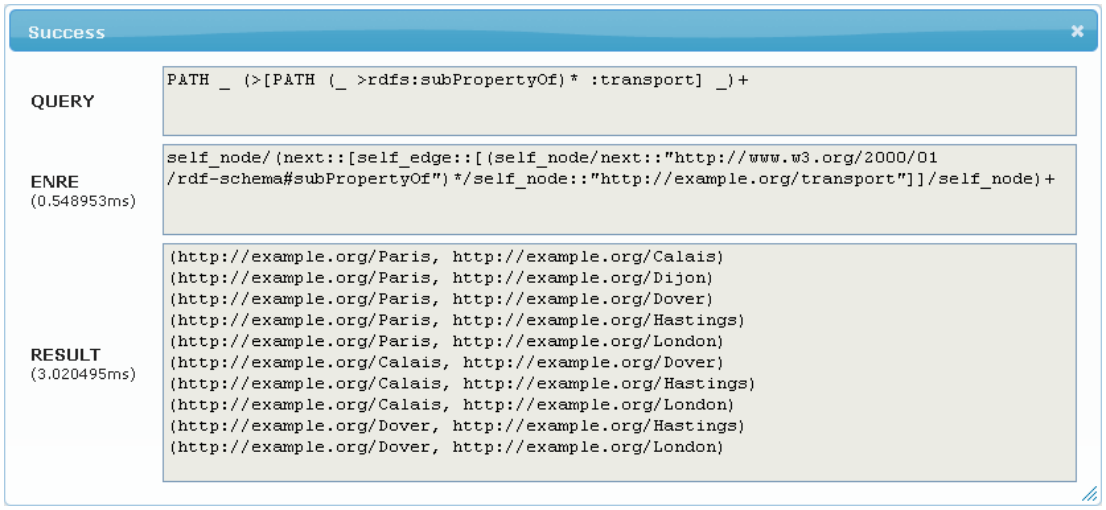


Figure 23: Response of the web-based interface, when evaluating the RPL query `EDGES >[PATH (_ >rdfs:subPropertyOf)* :transport]+` on the *Transport* dataset shown in Figures 6 and 22

6.2 VISRPL – A VISUAL EDITOR FOR RPL

The idea of creating *visRPL*, a visual editor for RPL, was born during the final presentation of visKWQL [19].

When designing a visual editor for an arbitrary (textual) programming or query language, one crucial challenge is to find a visual model for that language that is as expressive as the textual language itself (i.e. each textual query should be expressible by a visual query and the other way round), and still convenient to work with for the user. Simply building a visual editor by have the user "draw" an abstract syntax tree of a textual query would certainly not lead to any gain over a textual editor.

Concerning visRPL, this visual model consists of boxes that can be nested into one another, as shown in Figure 24. A *Composite* box, for instance, combines the expressivity of an `AdornedRPE` (by a multiplicity value located in its upper right corner), a `DisjunctiveRPE` (by allowing several alternate paths within itself) and a `ConcatenatedRPE` (by allowing several child boxes on each path).

The connectors (the red lines in Figure 24) run from left to right within each box, and thus visually describe its paths. For instance, the outermost *Flavored* box in Figure 24 defines only one path, consisting of an *Atomic* box (the one

containing the string ":Paris") followed by a *Composite* box (the one whose multiplicity in the upper right corner is about to be changed via the shown drop-down menu). This *Composite* box in turn defines a path, now consisting of a *Predicates* box (highlighted in green) that is connected to another *Atomic* box. However, the multiplicity + of this *Composite* box tells us that we are allowed to run through its inner path(s) for an arbitrary, non-zero number of times.

The lower part of Figure 24 shows the *RPL Query View*, which contains a textual query that corresponds to the visual query shown in the editor above. As textual and visual RPL queries have the same expressivity, roundtripping [19] is possible in both directions. Roundtripping can either be manually initiated via the *Generate Diagram* and *Generate Query* buttons, or automatically be run each time when the textual or visual query is changed by the user (this can be enabled via the *Synchronize* checkbox).

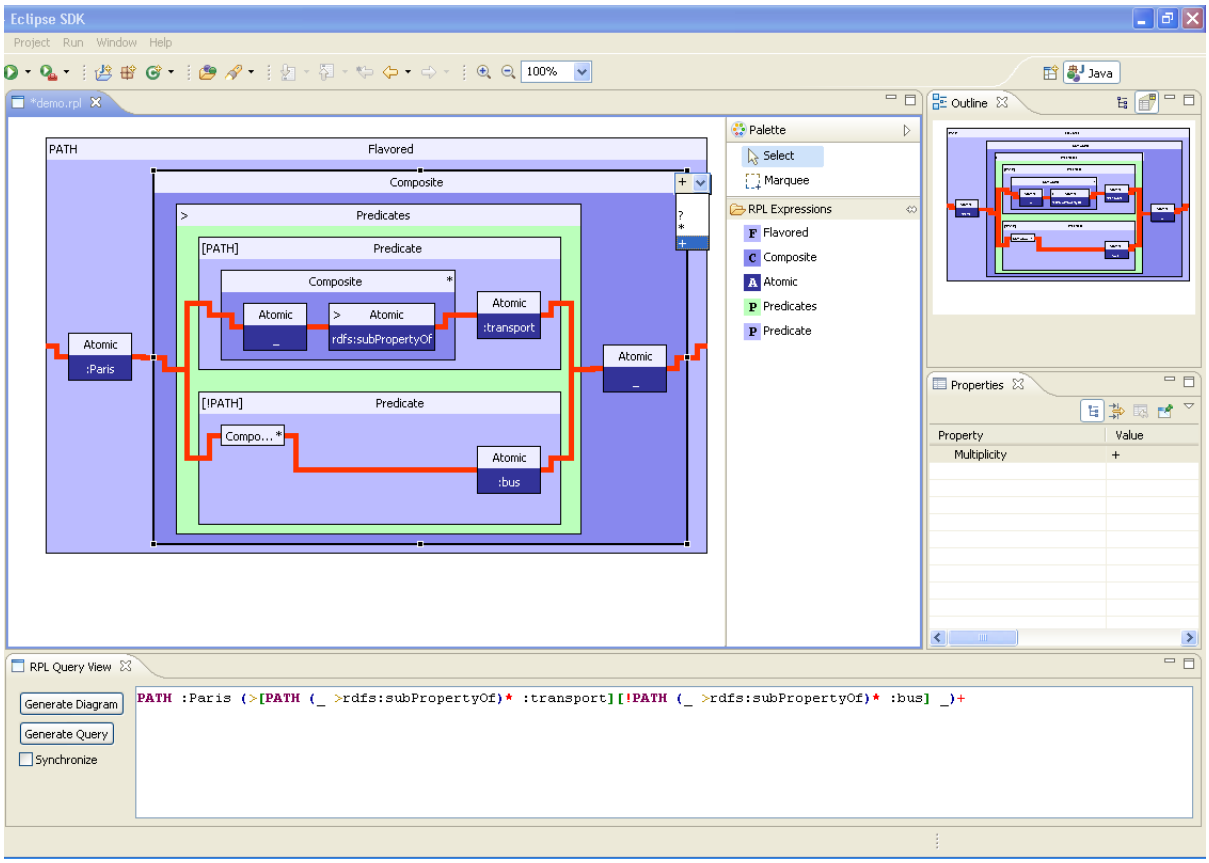


Figure 24: Screenshot of visRPL

visRPL is implemented as an Eclipse plugin, on top of Eclipse GEF[5]. Besides the already mentioned conceptual ideas and features, visRPL offers all amenities expected from today's editors, which include

- undoing and redoing commands, copy and paste support
- dragging and dropping, as well as collapsing and expanding elements (for example, the *Composite* box within the lower *Predicate* box in Figure 24 has been collapsed for the sake of clarity)
- a context menu offering all editing options for the selected element
- zooming in and out (see the magnifier icons in the uppermost toolbar within Figure 24)
- syntax highlighting and visual feedback on errors and warnings
- an *Outline* and *Properties* view (see the right part of Figure 24)
- a *Palette* containing several selection and creation tools (located right next to the visual query in Figure 24)

Challenges during the implementation of visRPL arose from enabling drag and drop with connectors as drop targets, and from clipping connectors to the bounds of their parent boxes. Furthermore, a bunch of visitors over the the visual model was necessary, for example to implement roundtripping and to display warnings in case a visual query is not complete yet or would fail semantic analysis (Section 3.1).

6.3 A TEXTUAL EDITOR

While visRPL is mainly intended for visual editing and getting familiar with RPL's syntax, the textual editor presented in this section assists the authoring of RPL expressions for intermediate to advanced users.

Figure 25 shows a screenshot of this textual editor. It consists of an editor that supports syntax highlighting and syntax completion, along with an *Outline* view displaying the queries' abstract syntax tree. Like visRPL, this textual editor is realized as Eclipse plugin. It is built on top of Xtext[6], a framework for the "development of programming languages and domain specific languages".

5 see http://www.eclipse.org/gef/
6 see http://www.eclipse.org/Xtext/

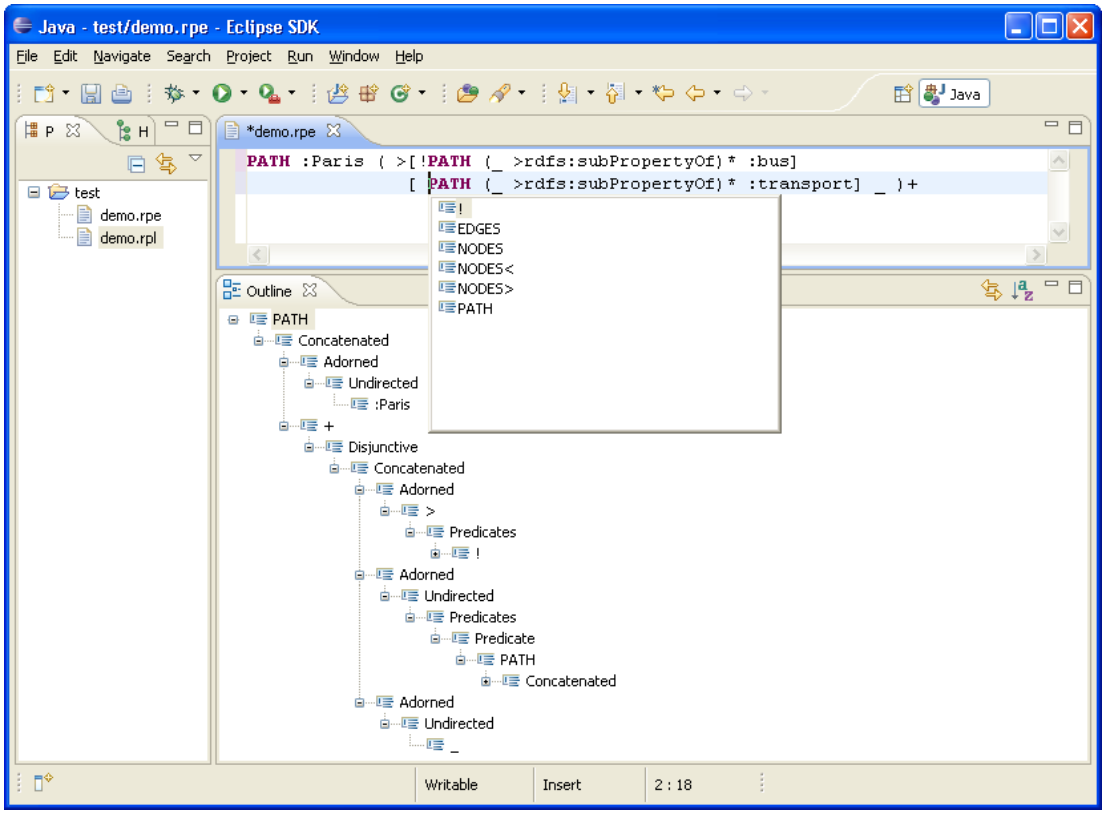


Figure 25: Screenshot of RPL's textual editor

Unfortunately, Xtext internally applies ANTLRv3, whose lexer component is unable to decide whether a < character in the input stream marks the beginning of an ⟨*IRI_REF*⟩ token (e.g. `<http://example.org>`, see Definition 1), or is used as a direction modifier. This drawback of ANTLRv3 is already discussed in Section 2.3, and leads to an error message like shown in Figure 26 within the textual editor.

As a workaround, the generated Java lexer classes have been manually modified such that whenever an ⟨*IRI_REF*⟩ token is to be matched, it is early checked if this match will succeed — without consuming any characters from the input stream yet. If this check is successful, an ⟨*IRI_REF*⟩ token will be emitted; otherwise, < will be interpreted as direction modifier. Figure 27 demonstrates the success of this workaround.

However, this workaround has to be reapplied, if the RPL syntax is about to change in the future, as the concerning Java lexer files will have to be regenerated from an Xtext grammar file.

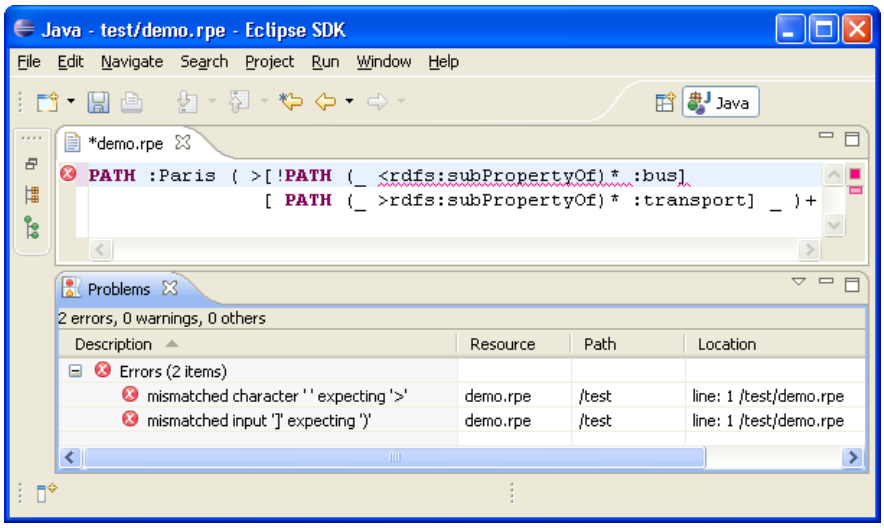


Figure 26: Screenshot of the textual editor for RPL, demonstrating ANTLRv3's lexer weakness. As can be seen in the *Problems* view, the < character was interpreted as the beginning of an ⟨*IRI_REF*⟩ token (which is expected to end with a > character)

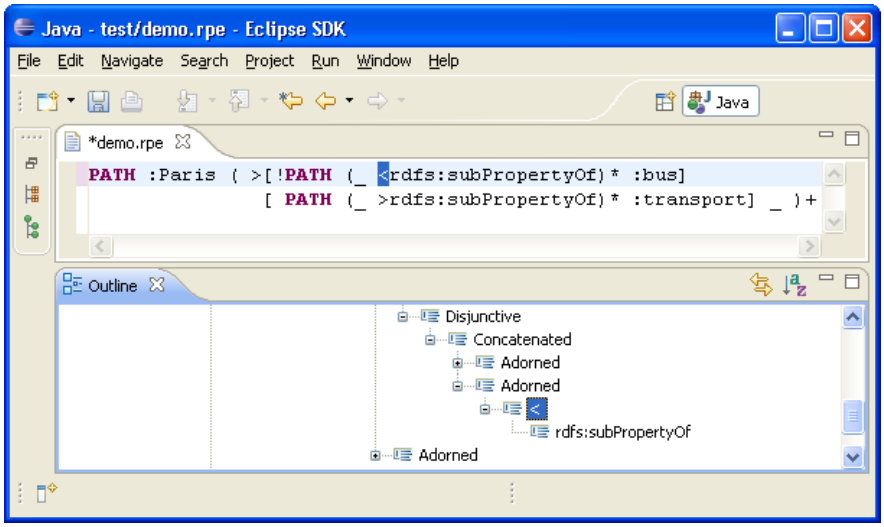


Figure 27: Same screenshot as in Figure 26, after the workaround has been applied. It is shown in the *Outline* view, that the < character is now correctly recognized as direction modifier

7

RELATED WORK

SPARQL's lack of navigational capabilities has led to the development of a plethora of SPARQL extensions like SPARQLeR [22], (C)PSPARQL [2], and nSPARQL [27].

Since RPL has already been compared to all of these extensions in [9], only one new aspect shall be contributed here. In Chapter 3, ENREs have been introduced as an extension of NREs (Nested Regular Expressions), which form the basis of nSPARQL. However, ENREs have been tailored to fit the peculiarities of RPL, and extend (by allowing regular expressions as label tests, some new navigation axes, and the negation of nested expressions) but at the same time limit NREs (by abandoning their navigation axes **node** and **edge**, together with their inverse axes $\mathbf{node}^{-1}$ and $\mathbf{edge}^{-1}$, respectively).

The concept of navigation axes arguably helped a lot to turn XPath [11] into its huge success, and has been borrowed for NREs, whose navigation axes are shown in Figure 28.

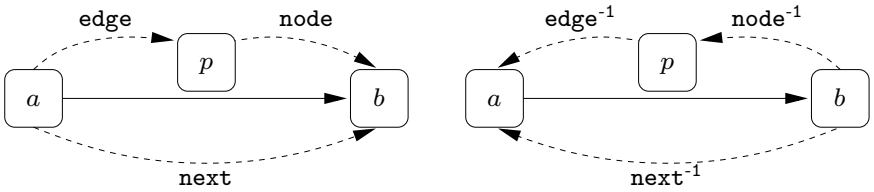


Figure 28: Navigational axes for an RDF triple (a, p, b), as presented in [27]

In the following, we demonstrate that the additional axes that NREs offer can be expressed via query rewriting, when RPL is used in embedded mode.

In Figure 29, the path from *?a* to *?b* (highlighted in maroon) and the path from *?d* to *?e* (highlighted in blue) should be in the evaluation of the NREs α and β, respectively. α and β should furthermore not contain any navigation axis from the set $\{\mathbf{node}, \mathbf{node}^{-1}, \mathbf{edge}, \mathbf{edge}^{-1}\}$.

The NRE α/**edge**::c/β (denoted by γ) defines a path from *?a* to *?e* (its endpoints are highlighted with green background). Note that this "path" is not a path in the sense of Definition 4, so it can not be queried by an ENRE nor a RPL query[1].

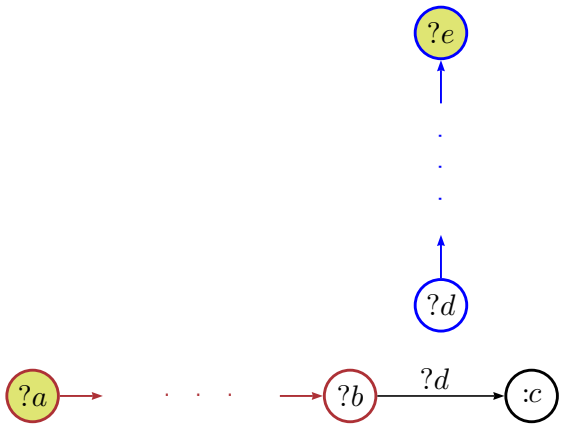


Figure 29: Expressivity of NREs and ENREs

However, when using RPL inside of SPARQL, this lack of expressivity can be compensated by two RPL triple patterns expressing the NREs α and β, and one SPARQL triple pattern expressing the **edge**::c move, as shown in the snippet below.

```
SELECT ?a ?e WHERE { ?a [q1] ?b . ?b ?d :c . ?d [q2] ?e }
```

This shows that RPL indeed avoids reinventing the wheel, and reuses the functionality already provided by SPARQL. Although similar for the remaining axes **edge**$^{-1}$, **node**, and **node**$^{-1}$, this rewriting can only be done a finite number of times; the result, however, is an acyclic SPARQL basic graph pattern and can be efficiently evaluated [16]. If, for instance, an **edge** axis is enclosed in an outer Kleene star or Kleene plus operator, this would lead to an infinite number of SPARQL triple patterns (one is required for each possible **edge** move) and thus to a SPARQL query of infinite length. So, in this case, the gain of expressivity from SPARQL is not big enough to fully incorporate the additional axes available for NREs.

1 When using a RPL predicate for the path from *?d* to *?e*, only the pair $(?a, :c)$ would be in the evaluation of the resulting query, in contrast to the evaluation of γ, which contains pairs of the form $(?a, ?e)$.

8

CONCLUSION AND FURTHER WORK

Concluding, this work presents a refined version of RPL's syntax and semantics. The graph labeling algorithm for the evaluation of nSPARQL's nested regular expressions [27] has been extended to fit the peculiarities of RPL, without increasing its time complexity of $\mathcal{O}(|G|^2 \cdot |q|)$. Furthermore, a novel evaluation approach is introduced: it stores a matrix along with each subexpression e of a RPL query, holding (the endpoints of) all paths satisfying e. Though this approach has time complexity $\mathcal{O}(|G|^3 \cdot |q|)$, it can be beneficial on small datasets and when using RPL in embedded mode, as part of these matrices might be reused. It has been shown that RPL is able to imitate the RDFS semantics when using SPARQL as host language. An entire tool suite for RPL is developed in this thesis: (1) a RPL-based RDF data generator named RPLgen, which has already been applied during the experimental evaluation, (2) a web-based interface for seeing RPL in action, and (3) visRPL, an Eclipse plugin allowing to graphically compose RPL queries.

In the following, we point out some of the remaining open issues, grouped into four categories.

RPL Language

Compared to the NREs (Nested Regular Expressions) of nSPARQL [27], RPL is not able to express **node** and **edge** moves (and their inverse moves $\mathbf{node}^{-1}$ and $\mathbf{edge}^{-1}$, respectively), as illustrated in Chapter 7. It could be checked, if RPL (and its evaluation algorithms) can be extended to fully incorporate the expressivity of NREs.

Another open question concerning the expressivity of RPL is the following: Is RPL's expressivity, after having it extended to incorporate the full expressivity of NREs as described above, already at the limit concerning the expressivity of

RDF path languages that can be evaluated in time $\mathcal{O}(|G|^2 \cdot |q|)$, or might there be any further interesting language extensions whose time complexity stays within that bound?

RPL Evaluation

As shown in Section 3.4.4, the time complexity of the second variant of the Path-based evaluation is cubic in the size of the queried RDF graph. However, its memoization technique is a beneficial approach, in particular when RPL is used in embedded mode and imports variables from its host language. It could be analysed, if this memoization technique can also be applied to the ENRE-based evaluation, ideally without increasing its time complexity.

Both evaluation approaches (ENRE-based and Path-based evaluation, see Sections 3.3 and 3.4) presented in this thesis are based on a RAM model. It can be further investigated, if indexing techniques allow for an efficient disk-based evaluation of RPL, and how these algorithms would have to be adapted.

The recent spreading of multi-core and networked environments gives rise to another interesting research approach: the parallelization of RPL's evaluation. Google's MapReduce [14] (together with its open source implementation Apache Hadoop[1]) might serve as a starting point here.

RPLgen

RPLgen, the RPL-based RDF data generator presented in Section 5.1, can be further extended such that each predicate within a `PredicatesRPE` is satisfied on the generated data (currently, this is only the case for the first predicate).

visRPL

visRPL, the visual editor for RPL queries presented in Section 6.2, can be extended to not only cover the construction, but also the evaluation of RPL queries — either directly against RDF data in standalone mode or against the storage interfaces provided by a SPARQL framework when using RPL in embedded mode. This, at the same time, would be visRPL's first step from a standalone visual editor towards a comprehensive IDE (Integrated Development Environment) for RPL.

1 see http://hadoop.apache.org/

A

APPENDIX

A.1 PROOF OF THEOREM 2

Proof. We use structural induction over RPEs and start with the base case, which is AtomicRPE. The notation used within the following boxes is explained in Section 2.2.2.

Let a be an AtomicRPE, $pos := a.getPosition()$, $rx := a.getRegExp()$.

Case pos = NODE

$$\begin{aligned} [\![a]\!]_G &= \{(n,n) \mid \exists n \in nodes(G) \wedge n \in \mathcal{L}(rx)\} && \text{(Def 9)} \\ &= [\![\textbf{self\_node::}rx]\!]_G = [\![trans(a)]\!]_G && \text{(Def 11, 13)} \end{aligned}$$

Case pos = EDGE

$$\begin{aligned} [\![a]\!]_G^{\text{FORWARD}} &= \{(s,o) \mid \exists p : (s,p,o) \in G \wedge p \in \mathcal{L}(rx)\} && \text{(Def 9)} \\ &= [\![\textbf{next::}rx]\!]_G = [\![trans^{\text{FORWARD}}(a)]\!]_G && \text{(Def 11, 13)} \\ [\![a]\!]_G^{\text{BACKWARD}} &= \{(o,s) \mid (s,o) \in [\![a]\!]_G^{\text{FORWARD}}\} && \text{(Def 9)} \\ &= \{(o,s) \mid (s,o) \in [\![\textbf{next::}rx]\!]_G\} && \text{(I.H.)} \\ &= [\![\textbf{next}^{-1}\textbf{::}rx]\!]_G = [\![trans^{\text{BACKWARD}}(a)]\!]_G && \text{(Def 11, 13)} \\ [\![a]\!]_G^{\text{UNDIRECTED}} &= [\![a]\!]_G^{\text{FORWARD}} \cup [\![a]\!]_G^{\text{BACKWARD}} && \text{(Def 9)} \\ &= [\![\textbf{next::}rx]\!]_G \cup [\![\textbf{next}^{-1}\textbf{::}rx]\!]_G && \text{(I.H.)} \\ &= [\![\textbf{next\_or\_next}^{-1}\textbf{::}rx]\!]_G && \text{(Def 11)} \\ &= [\![trans^{\text{UNDIRECTED}}(a)]\!]_G && \text{(Def 13)} \end{aligned}$$

Let p be a `PredicateRPE`, $f := p.getFlavored()$, $pos := p.getPosition()$, and $sign := p.getSign()$.
Case pos = `NODE`, $sign$ = `POSITIVE`

$$\begin{aligned}
[\![p]\!]_G &= \{(n,n) \mid n \in nodes(G) \wedge \exists q : (n,q) \in [\![f]\!]_G\} && \text{(Def 9)}\\
&= \{(n,n) \mid n \in nodes(G) \wedge \exists q : (n,q) \in [\![trans(f)]\!]_G\} && \text{(I.H.)}\\
&= [\![\texttt{self_node::[}trans(f)\texttt{]}]\!]_G = [\![trans(p)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

Case pos = `NODE`, $sign$ = `NEGATIVE`
Like the `POSITIVE` case, just replace $\exists$ with $\nexists$ and **`self_node::[`**$trans(f)$**`]`** with **`self_node::![`**$trans(f)$**`]`**.

Case pos = `EDGE`, $sign$ = `POSITIVE`

$$\begin{aligned}
[\![p]\!]_G &= \{(e,e) \mid e \in edges(G) \wedge \exists q : (e,q) \in [\![f]\!]_G\} && \text{(Def 9)}\\
&= \{(e,e) \mid e \in edges(G) \wedge \exists q : (e,q) \in [\![trans(f)]\!]_G\} && \text{(I.H.)}\\
&= [\![\texttt{self_edge::[}trans(f)\texttt{]}]\!]_G = [\![trans(p)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

Case pos = `EDGE`, $sign$ = `NEGATIVE`
Like the `POSITIVE` case, just replace $\exists$ with $\nexists$ and **`self_edge::[`**$trans(f)$**`]`** with **`self_edge::![`**$trans(f)$**`]`**.

Let p be a `PredicatesRPE`, $p_1, p_2, \ldots, p_n$ $(n \geq 1)$ its children, $pos := p.getPosition()$.
Case pos = `NODE`

$$\begin{aligned}
&[\![p]\!]_G\\
&= \{(n,n) \mid n \in nodes(G) \wedge (n,n) \in [\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G\} && \text{(Def 9)}\\
&= \{(n,n) \mid n \in nodes(G) \wedge \exists q : (n,q) \in [\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G\} && \text{(*)}\\
&= \{(n,n) \mid n \in nodes(G) \wedge \exists q : (n,q) \in [\![trans(p_1)]\!]_G \circ \ldots \circ [\![trans(p_n)]\!]_G\} && \text{(I.H.)}\\
&= \{(n,n) \mid n \in nodes(G) \wedge \exists q : (n,q) \in [\![trans(p_1)\texttt{/}\ldots\texttt{/}trans(p_n)]\!]_G\} && \text{(Def 11)}\\
&= [\![\texttt{self_node::[}trans(p_1)\texttt{/}\ldots\texttt{/}trans(p_n)\texttt{]}]\!]_G = [\![trans(p)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

Case pos = `EDGE`

$$\begin{aligned}
&[\![p]\!]_G^{\texttt{FORWARD}}\\
&= \{(s,o) \mid \exists (s,e,o) \in G \wedge (e,e) \in [\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G\} && \text{(Def 9)}\\
&= \{(s,o) \mid \exists (s,e,o) \in G \wedge \exists q : (e,q) \in [\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G\} && \text{(*)}\\
&= \{(s,o) \mid \exists (s,e,o) \in G \wedge \exists q : (e,q) \in [\![trans(p_1)]\!]_G \circ \ldots \circ [\![trans(p_n)]\!]_G\} && \text{(I.H.)}\\
&= \{(s,o) \mid \exists (s,e,o) \in G \wedge \exists q : (e,q) \in [\![trans(p_1)\texttt{/}\ldots\texttt{/}trans(p_n)]\!]_G\} && \text{(Def 11)}\\
&= [\![\texttt{next::[}trans(p_1)\texttt{/}\ldots\texttt{/}trans(p_n)\texttt{]}]\!]_G = [\![trans^{\texttt{FORWARD}}(p)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

The remaining cases $[\![p]\!]_G^{\texttt{BACKWARD}}$ and $[\![p]\!]_G^{\texttt{UNDIRECTED}}$ can be done similar as for `AtomicRPE`s.
(*): As each $[\![p_i]\!]_G$ $(1 \leq i \leq n)$ just contains symmetric pairs (x,y) where $x = y$, this also applies for $[\![p_1]\!]_G \circ \ldots \circ [\![p_n]\!]_G$.

Let d be a `DirectedRPE` and $direction := d.getDirection()$.

$$\begin{aligned}
[\![d]\!]_G &= [\![d.getDirectable()]\!]_G^{direction} && \text{(Def 9)}\\
&= [\![trans^{direction}(d.getDirectable())]\!]_G && \text{(I.H.)}\\
&= [\![trans(d)]\!]_G && \text{(Def 13)}
\end{aligned}$$

Let d be a `DisjunctiveRPE` and $c_1, c_2, \ldots, c_n (n \geq 1)$ its children.

$$\begin{aligned}
[\![d]\!]_G &= [\![c_1]\!]_G \cup [\![c_2]\!]_G \cup \ldots \cup [\![c_n]\!]_G && \text{(Def 9)}\\
&= [\![trans(c_1)]\!]_G \cup [\![trans(c_2)]\!]_G \cup \ldots \cup [\![trans(c_n)]\!]_G && \text{(I.H.)}\\
&= [\![trans(c_1)\,|\,trans(c_2)\,|\ldots|\,trans(c_n)]\!]_G && \text{(Def 11)}\\
&= [\![trans(d)]\!]_G && \text{(Def 13)}
\end{aligned}$$

Let a be an `AdornedRPE`, $ad := a.getAdornable()$, and $mult := a.getMultiplicity()$.

Case $mult$ = `ONE`

$$[\![a]\!]_G = [\![ad]\!]_G = [\![trans(ad)]\!]_G = [\![trans(a)]\!]_G \qquad \text{(Def 9, I.H., Def 13)}$$

Case $mult$ = `OPT`

$$\begin{aligned}
[\![a]\!]_G &= \{(v,v) \mid v \in terms(G)\} \cup [\![ad]\!]_G && \text{(Def 9)}\\
&= \{(n,n) \mid n \in nodes(G)\} \cup \{(e,e) \mid e \in edges(G)\} \cup [\![ad]\!]_G && \text{(Def 3)}\\
&= [\![\texttt{self_node}]\!]_G \cup [\![\texttt{self_edge}]\!]_G \cup [\![trans(ad)]\!]_G && \text{(Def 11, I.H.)}\\
&= [\![trans(ad)\texttt{?}]\!]_G = [\![trans(ad\texttt{?})]\!]_G = [\![trans(a)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

Case $mult$ = `STAR`

$$\begin{aligned}
[\![a]\!]_G &= \{(v,v) \mid v \in terms(G)\} \cup [\![ad\texttt{+}]\!]_G && \text{(Def 9)}\\
&= \{(n,n) \mid n \in nodes(G)\} \cup \{(e,e) \mid e \in edges(G)\} \cup [\![ad\texttt{+}]\!]_G && \text{(Def 3)}\\
&= [\![\texttt{self_node}]\!]_G \cup [\![\texttt{self_edge}]\!]_G \cup [\![trans(ad\texttt{+})]\!]_G && \text{(Def 11, I.H.)}\\
&= [\![\texttt{self_node}]\!]_G \cup [\![\texttt{self_edge}]\!]_G \cup [\![trans(ad)\texttt{+}]\!]_G && \text{(Def 13)}\\
&= [\![trans(ad)\texttt{*}]\!]_G = [\![trans(ad\texttt{*})]\!]_G = [\![trans(a)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

Case $mult$ = `PLUS`

$$\begin{aligned}
[\![a]\!]_G &= [\![ad]\!]_G \cup [\![ad]\!]_G \circ [\![ad]\!]_G \cup \ldots && \text{(Def 9)}\\
&= [\![trans(ad)]\!]_G \cup [\![trans(ad)]\!]_G \circ [\![trans(ad)]\!]_G \cup \ldots && \text{(I.H.)}\\
&= [\![trans(ad)]\!]_G \cup [\![trans(ad)\texttt{/}trans(ad)]\!]_G \cup \ldots && \text{(Def 11)}\\
&= [\![trans(ad)\texttt{+}]\!]_G = [\![trans(ad\texttt{+})]\!]_G = [\![trans(a)]\!]_G && \text{(Def 11, 13)}
\end{aligned}$$

Let c be a `ConcatenatedRPE` and $a_1, a_2, \ldots, a_n (n \geq 1)$ its children.

$$\begin{aligned}
[\![c]\!]_G &= [\![a_1]\!]_G \circ [\![a_2]\!]_G \circ \ldots \circ [\![a_n]\!]_G && \text{(Def 9)} \\
&= [\![trans(a_1)]\!]_G \circ [\![trans(a_2)]\!]_G \circ \ldots \circ [\![trans(a_n)]\!]_G && \text{(I.H.)} \\
&= [\![trans(a_1)/trans(a_2)/\ldots/trans(a_n)]\!]_G && \text{(Def 11)} \\
&= [\![trans(c)]\!]_G && \text{(Def 13)}
\end{aligned}$$

Let f be a `FlavoredRPE` and $c := f.getConcatenated()$.

$$[\![f]\!]_G = [\![c]\!]_G = [\![trans(c)]\!]_G = [\![trans(f)]\!]_G \qquad \text{(Def 9, I.H., Def 13)}$$

□

A.2 TARJAN'S ALGORITHM

Tarjan's algorithm identifies strongly connected components in directed graphs $G = (V, E)$. The pseudo code presented here consists of several global variables, a `TARJAN` method to be called by the user after G has been set, and a `STRONGCONNECT` method that is called once on every node $v \in V$ of G. This presentation is based on [31] as well as [23].

The correctness of this algorithm is shown in [31]. Furthermore, it is shown to have a time bound of $\mathcal{O}(|V| + |E|)$, if the *open* stack (a global variable) is additionally represented as a boolean array of size $|V|$, such that it can be tested in constant time whether a node is on the *open* stack or not.

```
index                an integer
G = (V, E)           a directed graph
open                 a stack over V
number, lowlink      arrays of integers over V
```

```
procedure TARJAN()
index := 0
for each v ∈ V
  number[v] := 0
open := empty stack
for each v ∈ V
  if number[v] = 0
    STRONGCONNECT(v)
```

```
procedure STRONGCONNECT(v)
lowlink[v] := number[v] := ++index
open.push(v)
for each w ∈ V with (v,w) ∈ E
  if number[w] = 0
    STRONGCONNECT(w)
    lowlink[v] := min(lowlink[v], lowlink[w])
  else if w ∈ open                          // can be done in O(1)
    lowlink[v] := min(lowlink[v], number[w])
  if lowlink[v] = number[v]
    start new strongly connected component
    repeat
      u := open.pop()
      add u to the current component
    until u = v
```

BIBLIOGRAPHY

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0. (Cited on pages 35 and 52.)

[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2):57 – 73, 2009. (Cited on page 101.)

[3] Renzo Angles and Claudio Gutierrez. The Expressive Power of SPARQL. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 114–129. Springer, 2008. http://www.dcc.uchile.cl/TR/2006/TR_DCC-2006-009.pdf. (Cited on page 75.)

[4] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, 2008. ISBN 978-0-262-01242-3. (Cited on page 1.)

[5] Jean-François Baget. RDF Entailment as a Graph Homomorphism. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 82–96. Springer, 2005. http://www.lirmm.fr/~baget/publications/ISWC05.pdf. (Cited on page 17.)

[6] Dave Beckett. RDF/XML Syntax Specification (Revised). Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. (Cited on page 17.)

[7] Dave Beckett and Tim Berners-Lee. Turtle - Terse RDF Triple Language, W3C Team Submission, 2008. Online only. http://www.w3.org/TeamSubmission/turtle/, retrieved at 2010/01/01. (Cited on page 17.)

[8] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 54–68. Springer, 2002. http://www.openrdf.org/doc/papers/Sesame-ISWC2002.pdf. (Cited on pages 73 and 77.)

[9] François Bry, Tim Furche, and Benedikt Linse. The Perfect Match: RPL and RDF Rule Languages. In *Web Reasoning and Rule Systems*, Lecture Notes in Computer Science, pages 227–241. Springer, 2009. http://www.pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2009-9/PMS-FB-2009-9-paper.pdf. (Cited on pages 4, 5, 9, 14, 33, 49, 83, and 101.)

[10] Jeremy J. Carroll and Graham Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. (Cited on page 16.)

[11] James Clark and Steven DeRose. XML Path Language (XPath) Version 1.0. Recommendation, W3C, November 1999. http://www.w3.org/TR/xpath/. (Cited on pages 7, 14, 21, and 101.)

[12] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Algorithmen - Eine Einführung*. Oldenbourg, 2004. ISBN 3-486-27515-1. (Cited on pages 57 and 68.)

[13] Russ Cox. Regular Expression Matching Can Be Simple And Fast, January 2007. Online only. http://swtch.com/~rsc/regexp/regexp1.html, retrieved at 2010/01/01. (Cited on pages 19, 20, and 45.)

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating Systems Design & Implementation*, pages 137–150. USENIX Association, 2004. http://www.usenix.org/events/osdi04/tech/dean.html. (Cited on page 104.)

[15] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Reasoning Web*, Lecture Notes in Computer Science, pages 1–52. Springer, 2006. (Cited on page 3.)

[16] Tim Furche, Antonius Weinzierl, and François Bry. Labeling RDF Graphs for Linear Time and Space Querying. In *Semantic Web Information Management*, pages 309–339. Springer, 2010. http://furche.net/content/publications/Furche2009CIGLabeling.pdf. (Cited on page 102.)

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. ISBN 0-201-63361-2. (Cited on page 11.)

[18] Hans-Otto Georgii. *Stochastik. Einführung in die Wahrscheinlichkeitstheorie und Statistik.* Gruyter, 2004. ISBN 3-1101-8282-3. (Cited on page 86.)

[19] Andreas Hartl. A Visual Rendering of a Semantic Wiki Query Language. Diploma thesis, Institute of Computer Science, LMU, Munich, 2009. http://www.runicsoft.com/Andreas%20Hartl%20-%20A%20Visual%20Rendering%20of%20a%20Semantic%20Wiki%20Query%20Language.pdf. (Cited on pages 6, 95, and 96.)

[20] Patrick Hayes. RDF Semantics. Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/. (Cited on pages 6, 17, and 82.)

[21] John Hopcroft and Jeffrey Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1990. ISBN 3-89319-181-X. (Cited on pages 43 and 45.)

[22] Krys J. Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *The Semantic Web: Research and Applications*, Lecture Notes in Computer Science, pages 145–159. Springer, 2007. http://lsdis.cs.uga.edu/~mjanik/KJ07-ESWC2007.pdf. (Cited on pages 3 and 101.)

[23] Kurt Mehlhorn, Stefan Näher, and Peter Sanders. Engineering DFS-Based Graph Algorithms, 2007. Online only. http://www.mpi-inf.mpg.de/~mehlhorn/ftp/EngineeringDFS.pdf, retrieved at 2010/01/01. (Cited on page 108.)

[24] Anders Møller. dk.brics.automaton - finite-state automata and regular expressions for Java. Online only. http://www.brics.dk/automaton/, retrieved at 2010/01/01. (Cited on pages 20 and 87.)

[25] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Minimal Deductive Systems for RDF. In *The Semantic Web: Research and Applications*, Lecture Notes in Computer Science, pages 53–67. Springer, 2007. http://www.dcc.uchile.cl/~cgutierr/papers/minimal.pdf. (Cited on page 82.)

[26] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 30–43. Springer, 2006. http://www.dcc.uchile.cl/~cgutierr/papers/sparql.pdf. (Cited on page 75.)

[27] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A Navigational Language for RDF. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 66–81. Springer, 2008. http://web.ing.puc.cl/~jperez/papers/nsparql_ext.pdf. (Cited on pages 3, 4, 5, 7, 8, 18, 34, 35, 36, 42, 50, 51, 52, 82, 83, 84, 101, and 103.)

[28] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum Akademischer Verlag, 2003. ISBN 3-8274-1099-1. (Cited on pages 20 and 43.)

[29] Andy Seaborne and Eric Prud'hommeaux. SPARQL Query Language for RDF. Recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. (Cited on pages 5, 10, 21, 74, 75, 76, 77, 80, and 83.)

[30] Giancarlo Succi and Raymond W. Wong. The application of JavaCC to develop a C/C++ preprocessor. *SIGAPP Appl. Comput. Rev.*, 7(3):11–18, 1999. (Cited on page 16.)

[31] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. (Cited on pages 55 and 108.)

[32] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. (Cited on pages 20, 44, and 45.)