Towards an RDF validation language based on Regular Expression derivatives

Jose Emilio Labra Gayo [*]
University of Oviedo
Spain

Eric Prud'hommeaux
W3c
Stata Center, MIT

Sławek Staworko
LINKS, INRIA & CNRS
University of Lille, France

Harold Solbrig
Mayo Clinic, College of Medicine
Rochester, MN, USA

ABSTRACT
There is a growing interest in the validation of RDF based solutions where one can express the topology of an RDF graph using some schema language that can check if RDF documents comply with it.

Shape Expressions have been proposed as a simple, intuitive language that can be used to describe expected graph patterns and to validate RDF graphs against those patterns. The syntax and semantics of Shape Expressions are designed to be familiar to users of regular expressions.

In this paper, we propose an implementation of Shape Expressions inspired by the regular expression derivatives but adapted to RDF graphs.

1. INTRODUCTION
The industry need to describe and validate conformance of RDF instance data with some *schema* has motivated a W3C Workshop [24] and the chartering of W3C RDF Data Shapes Working Group.[1] Here, a *schema* defines an RDF graph structure where a node has expected properties with defined cardinalities, connecting to literal values or other described nodes.

As currently defined, RDF Schema [2] and OWL [22] are widely recognized as being insufficient to fulfil this task, leading to proposals like the RDF vocabulary *Resource Shapes*[2] and the *Shape Expressions*[3] language.

The operational semantics of Shape Expressions has been presented at [23] and the complexity and expressiveness of the language has been studied at [1]. A Shape Expression is a labelled pattern that describes RDF nodes using a syntax inspired by regular expressions.

Example 1. The following shape expression describes `Person` shapes as nodes that have one property `foaf:age` with values of type `xsd:int`, one or more properties `foaf:name` with values of type `xsd:string` and zero or more properties `foaf:knows` with values of shape `Person`.

```
<Person> {
  foaf:age   xsd:integer
, foaf:name  xsd:string+
, foaf:knows @<Person>*
}
```

It is possible to automatically check which nodes comply with the declared shapes in an RDF Graph.

Example 2. The nodes `:john` and `:bob` in the following graph have shape `Person` while the node `:mary` does not have that shape.

```
:john foaf:age 23;
      foaf:name "John";
      foaf:knows :bob .

:bob  foaf:age 34;
      foaf:name "Bob", "Robert" .

:mary foaf:age 50, 65 .
```

Shape expressions can be used to describe and validate the contents of linked data portals [16] and there are several implementations and online validation tools like ShEx Workbench[4] and RDFShape[5].

[*]Corresponding author
[1]`http://www.w3.org/2014/data-shapes/`
[2]`http://www.w3.org/Submission/shapes/`
[3]`http://www.w3.org/Submission/shex-defn/`
[4]`http://www.w3.org/2013/ShEx/FancyShExDemo`
[5]`http://rdfshape.weso.es`

Regular expressions are a well-known formalism to describe the shape of sequences of characters. They have also been employed to describe the shape of XML trees and form the theoretical basis of RelaxNG. In 1964, Janusz Brzozowski proposed a method for directly implementing a regular expression recognizer based on regular expression derivatives [3]. In this paper, we adapt the derivatives approach to RDF Graph based recognizers. We define regular shape expressions, which form the basis of the Shape Expressions language, and present the algorithm that can be used to check if an RDF node has a given Shape. The algorithm has been implemented and the performance results are better than a backtracking implementation.

2. PRELIMINARIES

Given a set S, we denote S^* as the powerset of S, $\{\}$ denotes the empty set and $\{a_1, \ldots, a_n\}$ denotes a set with elements $a_1, \ldots, a_n$. The singleton set $\{a\}$ will be simplified as a.

Let V_s = vocabulary of subjects, V_p = vocabulary of predicates and V_o = vocabulary of objects. In RDF, if we define $\mathcal{I}$ as the set of IRIs, $\mathcal{B}$ as the set of blank nodes and $\mathcal{L}$ as the set of literals, we have $V_s = \mathcal{I} \cup \mathcal{B}$, $V_p = \mathcal{I}$ and $V_o = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$.

A graph Σ is defined as a set of triples $\langle s, p, o\rangle$ such that $s \in V_s$, $p \in V_p$ and $o \in V_o$. Σ^* denotes all possible graphs. The expression $t \rtimes t_s$ represents the addition of triple t to a graph t_s. Given two graphs g_1 and g_2, $g_1 \oplus g_2$ denotes the union of g_1 and g_2. Notice that we are using union of RDF graphs instead of merging. Union of two RDF graphs preserves the identity of blank nodes shared between graphs while merging does not [11].

The decomposition of a graph g is defined as the set $\{(g_1, g_2)|g_1 \oplus g_2 = g\}$. The decomposition of a graph with n triples is an exponential operation that generates a graph with 2^n pairs of graphs that can be obtained by calculating the powerset of g and pairing each element with its complement.

Example 3. Let $g = \{\langle n, a, 1\rangle, \langle n, b, 1\rangle, \langle n, b, 2\rangle\}$, the decomposition of g is:

$$
\begin{array}{ll}
\{ & (\{\},\{\langle n,a,1\rangle,\langle n,b,1\rangle,\langle n,b,2\rangle\}),\\
 & (\{\langle n,a,1\rangle\},\{\langle n,b,1\rangle,\langle n,b,2\rangle\}),\\
 & (\{\langle n,b,1\rangle\},\{\langle n,a,1\rangle,\langle n,b,2\rangle\}),\\
 & (\{\langle n,b,2\rangle\},\{\langle n,a,1\rangle,\langle n,b,1\rangle\}),\\
 & (\{\langle n,a,1\rangle,\langle n,b,1\rangle\},\{\langle n,b,2\rangle\}),\\
 & (\{\langle n,a,1\rangle,\langle n,b,2\rangle\},\{\langle n,b,1\rangle\}),\\
 & (\{\langle n,b,1\rangle,\langle n,b,2\rangle\},\{\langle n,a,1\rangle\}),\\
 & (\{\langle n,a,1\rangle,\langle n,b,1\rangle,\langle n,b,2\rangle\},\{\}),\\
\} &
\end{array}
$$

We define the shape of a node n in a graph g, Σ^g_n as the set of triples related to n in graph g. It is formed by all the triples of the form $\langle n, p, o\rangle \in g$. We define Σ^* as all possible shapes that a node n can have.

3. WHY NOT SPARQL?

Shape Expressions have been proposed as a high level, intuitive language to validate RDF. This problem can also be partially solved using SPARQL queries [14] which leverage on the whole expressiveness of the SPARQL query language.

The main issue of SPARQL queries is that they can become unwieldy and difficult to generate, manage and debug by hand.

Example 4. A SPARQL query that can express part of example 1 is:

```
ASK {
{ SELECT ?Person {
  ?Person foaf:age ?o .
} GROUP BY ?Person HAVING (COUNT(*)=1)}
{ SELECT ?Person {
  ?Person foaf:age ?o .
FILTER ( isLiteral(?o) &&
         datatype(?o) = xsd:integer )
} GROUP BY ?Person HAVING (COUNT(*)=1)}
{ SELECT ?Person (COUNT(*) AS ?Person_c0) {
  ?Person foaf:name ?o .
} GROUP BY ?Person HAVING (COUNT(*)>=1)}
{ SELECT ?Person (COUNT(*) AS ?Person_c1) {
  ?Person foaf:name ?o .
  FILTER (isLiteral(?o) &&
  datatype(?o) = xsd:string)
} GROUP BY ?Person HAVING (COUNT(*)>=1)}
  FILTER (?Person_c0 = ?Person_c1)
{ {
{ SELECT ?Person (COUNT(*) AS ?Person_c2){
  ?Person foaf:knows ?o .
} GROUP BY ?Person}
{ SELECT ?Person (COUNT(*) AS ?Person_c3){
  ?Person foaf:knows ?o .
  FILTER ((isIRI(?o) || isBlank(?o)))
  }
 GROUP BY ?Person HAVING (COUNT(*) >= 1)}
 FILTER (?Person_c2 = ?Person_c3)
} UNION  { SELECT ?Person {
   OPTIONAL { ?Person foaf:knows ?o }
   FILTER (!bound(?o))
}}}}
```

Representing RDF validation constraints as SPARQL queries is not practical for large data portals and there is a need for a higher level, declarative language with a more intuitive semantics.

Apart from that, the previous example is not completely right as it has omitted the recursive definition where it should validate that the values of `foaf:knows` all have the shape of `Person`. Trying to represent recursive definitions in SPARQL is not possible in general.[6] From our point of view SPARQL can be used as a lower level language for constraint validation in the sense that Shape Expressions can be mapped to SPARQL queries. In fact, one of our implementation of Shape Expressions is already able to generate those SPARQL queries from Shape Expressions.

4. INTRODUCING REGULAR SHAPE EXPRESSIONS

In this section we define Regular Shape Expressions as a simplified language based on the whole Shape Expressions language. This

[6]This particular query could be represented using zero-length paths as proposed by Joshua Taylor in StackOverflow `http://goo.gl/uMoXBQ`

language will be used as the basis for our implementations. A regular shape expression E defines the triples related with a given node in a graph. Although the concept presented in this paper is focused on RDF graphs, we consider that these definitions can be applied to describe the topology of other graph structures.

Given three non-empty sets V_s, V_p, V_o and $v_s \subseteq V_s$, $v_p \subseteq V_p$ and $v_o \subseteq V_o$, the abstract syntax of regular shape expressions (E) over V_s, V_p, V_o is:

E, F	::=	$\emptyset$	empty, no shape
	|	ε	empty set of triples
	|	$\_ \xrightarrow{v_p} v_o$	arc with predicate $p \in v_p$ and object $o \in v_o$
	|	$E*$	Kleene closure (0 or more E)
	|	$E \parallel F$	And (unordered concatenation)
	|	$E \mid F$	Alternative

We do not provide the concatenation operator from string based regular expressions because the arcs in a graph are not ordered. The And operator (||) for unordered concatenation appears in [1] and is similar to interleave or shuffle [6, 10] although in the case of graphs and regular shape expressions there is no ordered concatenation operator.

The operators $E+$ (one or more) and $E?$ (optional) can be defined as:

$$\begin{aligned} E+ &= E \parallel E* \\ E? &= E \mid \varepsilon \end{aligned}$$

The Shape Expressions language also contains a range operator $E\{m, n\}$ which represents between m and n repetitions of E. It can be defined as:

$$E\{m,n\} = \begin{cases} E\{m, n-1\} | E & \text{if } m < n \\ E\{m-1, n-1\} \parallel E & \text{if } m = n > 0 \\ \varepsilon & \text{if } m = n = 0 \end{cases}$$

Example 5. The regular shape expression

$$\_ \xrightarrow{a} 1 \parallel \_ \xrightarrow{b} \{1, 2\}*$$

declares a shape that contains one arc with predicate a and value 1, and one or more arcs with predicate b and values 1 or 2.

Example 6. We can consider `xsd:int` and `xsd:string` as subsets of $\mathcal{L}$ (the set of Literals) in RDF, so we can define the shape:

$$\_ \xrightarrow{\texttt{foaf:age}} \texttt{xsd:integer} \parallel (\_ \xrightarrow{\texttt{foaf:name}} \texttt{xsd:string})+$$

that declares nodes that must have an arc with predicate `foaf:age` and value in `xsd:int` and one or more arcs with predicate `foaf:name` and value in `xsd:string`. In ShEx notation it can be represented as:

```
<Example> {
  foaf:age  xsd:integer
, foaf:name xsd:string+
}
```

Given a node n, the shape of a regular shape expression e with respect to n, denoted as $\mathcal{S}_n[\![e]\!]$ is the set of graphs $\mathcal{S}_n[\![e]\!] \subseteq \Sigma^*$ generated by the following rules:

$$\begin{aligned} \mathcal{S}_n[\![\emptyset]\!] &= \emptyset \\ \mathcal{S}_n[\![\varepsilon]\!] &= \{\} \\ \mathcal{S}_n[\![\_ \xrightarrow{v_p} v_o]\!] &= \{\langle n, p, o\rangle | p \in v_p \text{ and } o \in v_o\} \\ \mathcal{S}_n[\![e*]\!] &= \{\} \cup \mathcal{S}_n[\![e \parallel e*]\!] \\ \mathcal{S}_n[\![e_1 \parallel e_2]\!] &= \{t_1 \cup t_2 | \, t_1 \in \mathcal{S}_n[\![e_1]\!] \text{ and } t_2 \in \mathcal{S}_n[\![e_2]\!]\} \\ \mathcal{S}_n[\![e_1 \mid e_2]\!] &= \mathcal{S}_n[\![e_1]\!] \cup \mathcal{S}_n[\![e_2]\!] \end{aligned}$$

Example 7. Let $e = \_ \xrightarrow{a} 1 \parallel \_ \xrightarrow{b} \{1, 2\}*$, then

$$\begin{aligned} \mathcal{S}_n[\![e]\!] = \{ & \{\langle n, a, 1\rangle\}, \\ & \{\langle n, a, 1\rangle, \langle n, b, 1\rangle\}, \\ & \{\langle n, a, 1\rangle, \langle n, b, 2\rangle\}, \\ & \{\langle n, a, 1\rangle, \langle n, b, 1\rangle, \langle n, b, 2\rangle\}\} \end{aligned}$$

For any expression x, the operators ||, |, ε and $\emptyset$ obey the following simplification rules:

$$\begin{aligned} \emptyset \mid x &= x \\ x \mid \emptyset &= x \\ \emptyset \parallel x &= \emptyset \\ x \parallel \emptyset &= \emptyset \\ \varepsilon \parallel x &= x \\ x \parallel \varepsilon &= x \end{aligned}$$

5. MATCHING REGULAR SHAPE EXPRESSIONS

Given a regular shape expression e and a node n in a graph g, we want to determine if Σ^g_n (the subgraph formed by the triples related with n) matches the regular shape expression $\mathcal{S}_n[\![e]\!]$, i.e. we want to determine if $\Sigma^g_n \in \mathcal{S}_n[\![e]\!]$.

The semantics of Regular Shape Expressions is defined by a relation $e \simeq \Sigma^g_n$ (e matches Σ^g_n) which can be expressed using axioms and inference rules [23]. Figure 1 presents the operational semantics of Regular Shape Expressions. Those rules can be directly implemented using backtracking.

Example 8. Let $e = \_ \xrightarrow{a} 1 \parallel \_ \xrightarrow{b} \{1, 2\}*$ and a graph g where $\Sigma^g_n = \{\langle n, a, 1\rangle, \langle n, b, 1\rangle, \langle n, b, 2\rangle\}$, a trace of the matching algorithm is represented in figure 2. Notice that we have to decompose the matching graph g in all the pairs of graphs g_1 and g_2 whose union give g. In this case, the decomposition returns all the pairs depicted in example 3.

As can be seen, a naïve implementation of Regular Shape expression matching using backtracking leads to exponential growth and has poor performance.

6. REGULAR SHAPE EXPRESSION DERIVATIVES

The derivative of a shape $\mathcal{S}_n(E) \subseteq \Sigma^*$ with respect to a triple $t \in \Sigma$ is a shape that includes only the remaining triples that when appended to t will become $\mathcal{S}_n(E)$.

$$Or_1\frac{r_1 \simeq g}{r_1|r_2 \simeq g} \qquad Or_2\frac{r_2 \simeq g}{r_1|r_2 \simeq g}$$

$$And\frac{r_1 \simeq g_1 \qquad r_2 \simeq g_2}{r_1 \parallel r_2 \simeq g_1 \oplus g_2}$$

$$Empty\frac{}{\varepsilon \simeq \{\}}$$

$$Star_1\frac{}{r* \simeq \{\}} \qquad Star_2\frac{r \simeq g_1 \qquad r* \simeq g_2}{r* \simeq g_1 \oplus g_2}$$

$$Arc\frac{p \in v_p \qquad o \in v_o}{\textvisiblespace \xrightarrow{v_p} v_o \simeq \langle s,p,o\rangle}$$

Figure 1: Inference rules for Shape expression rules

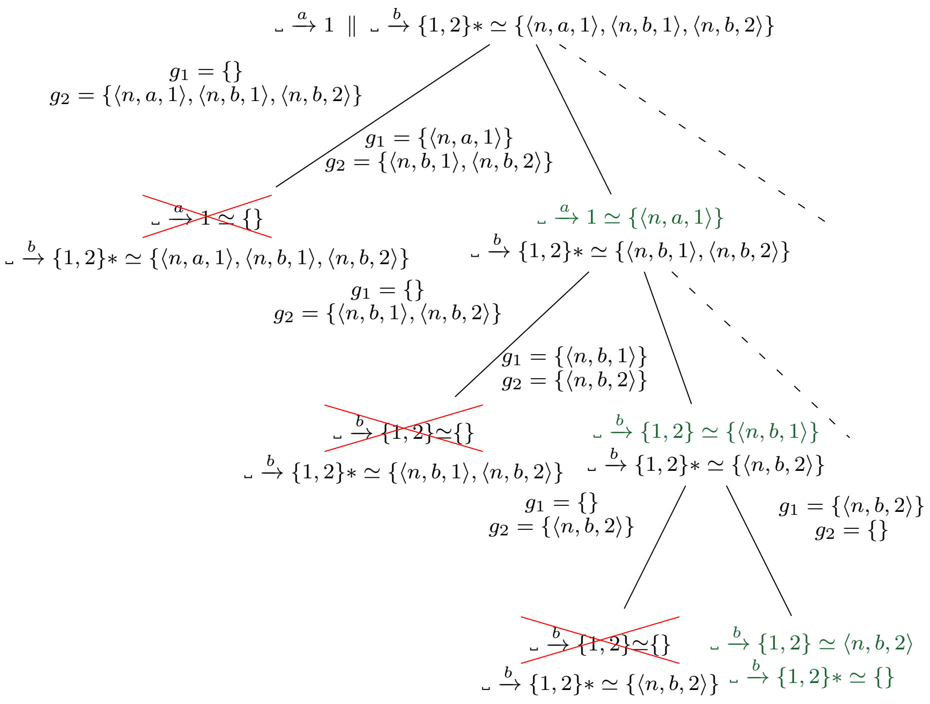
$\textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,a,1\rangle, \langle n,b,1\rangle, \langle n,b,2\rangle\}$
$g_1 = \{\}$
$g_2 = \{\langle n,a,1\rangle, \langle n,b,1\rangle, \langle n,b,2\rangle\}$
$g_1 = \{\langle n,a,1\rangle\}$
$g_2 = \{\langle n,b,1\rangle, \langle n,b,2\rangle\}$
$\cancel{\textvisiblespace \xrightarrow{a} 1 \simeq \{\}}$
$\textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,a,1\rangle, \langle n,b,1\rangle, \langle n,b,2\rangle\}$
$\textvisiblespace \xrightarrow{a} 1 \simeq \{\langle n,a,1\rangle\}$
$\textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,b,1\rangle, \langle n,b,2\rangle\}$
$g_1 = \{\}$
$g_2 = \{\langle n,b,1\rangle, \langle n,b,2\rangle\}$
$g_1 = \{\langle n,b,1\rangle\}$
$g_2 = \{\langle n,b,2\rangle\}$
$\cancel{\textvisiblespace \xrightarrow{b} \{1,2\} \simeq \{\}}$
$\textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,b,1\rangle, \langle n,b,2\rangle\}$
$\textvisiblespace \xrightarrow{b} \{1,2\} \simeq \{\langle n,b,1\rangle\}$
$\textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,b,2\rangle\}$
$g_1 = \{\}$
$g_2 = \{\langle n,b,2\rangle\}$
$g_1 = \{\langle n,b,2\rangle\}$
$g_2 = \{\}$
$\cancel{\textvisiblespace \xrightarrow{b} \{1,2\} \simeq \{\}}$
$\textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,b,2\rangle\}$
$\textvisiblespace \xrightarrow{b} \{1,2\} \simeq \langle n,b,2\rangle$
$\textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\}$

Figure 2: Regular Shape Expression matching using backtracking

Definition 1. The derivative of a Shape $\mathcal{S}_n(E) \subseteq \Sigma^*$ with respect to a triple $t \in \Sigma$ is defined as $\partial_t(\mathcal{S}_n(E)) = \{t_s | t \rtimes t_s \in \mathcal{S}_n(E)\}$

We need a helper function $\nu : E \to Bool$ (also called nullable) that checks if a regular shape expression can match the empty graph.

$$\nu(E) = \begin{cases} \texttt{true} & \text{if } E \text{ matches the empty graph} \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \nu(\emptyset) &= \texttt{false} \\ \nu(\varepsilon) &= \texttt{true} \\ \nu(\textvisiblespace \xrightarrow{v_p} v_o) &= \texttt{false} \\ \nu(e*) &= \texttt{true} \\ \nu(e_1 \parallel e_2) &= \nu(e_1) \wedge \nu(e_2) \\ \nu(e_1 \mid e_2) &= \nu(e_1) \vee \nu(e_2) \end{aligned}$$

The following rules, inspired from Brzozowski [3], compute the derivative of a regular shape expression with respect to a triple t.

$$\begin{aligned} \partial_t(\emptyset) &= \emptyset \\ \partial_t(\varepsilon) &= \emptyset \\ \partial_{\langle s,p,o\rangle}(\textvisiblespace \xrightarrow{v_p} v_o) &= \begin{cases} \varepsilon & \text{if } p \in v_p \text{ and } o \in v_o \\ \emptyset & \text{otherwise} \end{cases} \\ \partial_t(e*) &= \partial_t(e) \parallel e* \\ \partial_t(e_1 \parallel e_2) &= \partial_t(e_1) \parallel e_2 \mid \partial_t(e_2) \parallel e_1 \\ \partial_t(e_1 \mid e_2) &= \partial_t(e_1) \mid \partial_t(e_2) \end{aligned}$$

Example 9. Let $e = \textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}*$, the derivative of e with respect to $\langle n,a,1\rangle$ is $\textvisiblespace \xrightarrow{b} \{1,2\}*$. A trace of the derivatives calculation can be:

$$\begin{aligned} & \partial_{\langle n,a,1\rangle}(\textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}*) \\ = \; & \partial_{\langle n,a,1\rangle}(\textvisiblespace \xrightarrow{a} 1) \parallel \textvisiblespace \xrightarrow{b} \{1,2\}* \\ \mid \; & \partial_{\langle n,a,1\rangle}(\textvisiblespace \xrightarrow{b} \{1,2\}*) \parallel \textvisiblespace \xrightarrow{a} 1 \\ = \; & \varepsilon \parallel \textvisiblespace \xrightarrow{b} \{1,2\}* \\ \mid \; & \partial_{\langle n,a,1\rangle}(\textvisiblespace \xrightarrow{b} \{1,2\}) \parallel \textvisiblespace \xrightarrow{b} \{1,2\}* \parallel \textvisiblespace \xrightarrow{a} 1 \\ = \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \\ \mid \; & \emptyset \parallel \textvisiblespace \xrightarrow{b} \{1,2\}* \parallel \textvisiblespace \xrightarrow{a} 1 \\ = \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \\ \mid \; & \emptyset \parallel \textvisiblespace \xrightarrow{a} 1 \\ = \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \mid \emptyset \\ = \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \end{aligned}$$

Notice that the derivative of a Regular Shape Expression can grow in its size.

Example 10. The regular shape expression $e = (\textvisiblespace \xrightarrow{a} \{1,2\} | \textvisiblespace \xrightarrow{b} \{1,2\})*$,checks that there are the number of arcs with predicate a and values in $\{1,2\}$ and arcs with predicate b and values in $\{1,2\}$ is the same. The derivative of e with respect to $\langle n,a,1\rangle$ is $\textvisiblespace \xrightarrow{b} \{1,2\} \parallel (\textvisiblespace \xrightarrow{a} \{1,2\} | \textvisiblespace \xrightarrow{b} \{1,2\})*$. Notice that it grows because once it finds an arc with predicate a, it needs to find another arc with predicate b and continue with the rest of the graph.

The rules can be extended to graphs (sets of triples) as follows:

$$\begin{aligned} \partial_{\{\}}(e) &= e \\ \partial_{t \rtimes t_s}(e) &= \partial_{t_s}(\partial_t(e)) \end{aligned}$$

7. MATCHING USING DERIVATIVES

For any graph g, we have that $\Sigma_n^g \in \mathcal{S}_n[\![e]\!]$ if, and only if, $\varepsilon \in \mathcal{S}_n[\![\partial_{\Sigma_n^g}(e)]\!]$ which is true when $\nu(\partial_{\Sigma_n^g}(e)) = \texttt{true}$. We can express the algorithm in terms of the relation $e \simeq \Sigma_n^g$ defined as the smallest relation satisfying:

$$\begin{aligned} e \simeq \{\} &\Leftrightarrow \nu(e) \\ e \simeq t \rtimes t_s &\Leftrightarrow \partial_t(e) \simeq t_s \end{aligned}$$

It is straightforward to show that $e \simeq \Sigma_n^g$ if, and only if, $\Sigma_n^g \in \mathcal{S}_n[\![e]\!]$.

Notice that when a regular shape expression matches a set of triples, we compute the derivative for each of the triples in the set.

Example 11. Let $e = \textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}*$ and $\Sigma_n^g = \{\langle n,a,1\rangle, \langle n,b,1\rangle, \langle n,b,2\rangle\}$, the matching algorithm proceeds as:

$$\begin{aligned} & \textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,a,1\rangle, \langle n,b,1\rangle, \langle n,b,2\rangle\} \\ \Leftrightarrow \; & \partial_{\langle n,a,1\rangle}(\textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}*) \simeq \{\langle n,b,1\rangle, \langle n,b,2\rangle\} \\ \Leftrightarrow \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,b,1\rangle, \langle n,b,2\rangle\} \\ \Leftrightarrow \; & \partial_{\langle n,b,1\rangle}(\textvisiblespace \xrightarrow{b} \{1,2\}*) \simeq \{\langle n,b,2\rangle\} \\ \Leftrightarrow \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\langle n,b,2\rangle\} \\ \Leftrightarrow \; & \partial_{\langle n,b,2\rangle}(\textvisiblespace \xrightarrow{b} \{1,2\}*) \simeq \{\} \\ \Leftrightarrow \; & \textvisiblespace \xrightarrow{b} \{1,2\}* \simeq \{\} \\ \Leftrightarrow \; & \nu(\textvisiblespace \xrightarrow{b} \{1,2\}*) \\ \Leftrightarrow \; & \texttt{true} \end{aligned}$$

As can be seen the derivatives algorithm takes a linear approach where it is consuming a triple in each step and calculating the corresponding derivative of the regular shape expression. The algorithm does not need to decompose the graph or to do backtracking. The main complexity of the algorithm comes from the process of calculating and representing derivatives of shape expressions.

Example 12. Let $e = \textvisiblespace \xrightarrow{a} 1 \parallel \textvisiblespace \xrightarrow{b} \{1,2\}*$ and $\Sigma_n^g = \{\langle n,a,1\rangle, \langle n,a,2\rangle, \langle n,b,1\rangle\}$, the matching algorithm proceeds as:

$$
\begin{aligned}
& \text{\textvisiblespace} \xrightarrow{a} 1 \;\|\; \text{\textvisiblespace} \xrightarrow{b} \{1,2\}* \simeq \{\langle n,a,1\rangle, \langle n,a,2\rangle, \langle n,b,1\rangle\} \\
\Leftrightarrow\; & \partial_{\langle n,a,1\rangle}(\text{\textvisiblespace} \xrightarrow{a} 1 \;\|\; \text{\textvisiblespace} \xrightarrow{b} \{1,2\}*) \simeq \{\langle n,a,2\rangle, \langle n,b,1\rangle\} \\
\Leftrightarrow\; & \text{\textvisiblespace} \xrightarrow{b} \{1,2\}* \simeq \{\langle n,a,2\rangle, \langle n,b,1\rangle\} \\
\Leftrightarrow\; & \partial_{\langle n,a,2\rangle}(\text{\textvisiblespace} \xrightarrow{b} \{1,2\}*) \simeq \{\langle n,b,1\rangle\} \\
\Leftrightarrow\; & \emptyset \simeq \{\langle n,b,1\rangle\} \\
\Leftrightarrow\; & \texttt{false}
\end{aligned}
$$

8. SHAPE EXPRESSION SCHEMAS

In this section, we extend the regular shape expressions language to include labels for shape expressions. We assume a finite set of labels Λ.

A Shape Expression Schema is a tuple (Λ, δ) where δ is a shape definition function that maps labels to regular shape expressions over $V_s \cup \Lambda, V_p \cup \Lambda, V_o \cup \Lambda$. Typically, we present a schema as a collection of rules of the form $\lambda \longmapsto e$ where $\lambda \in \Lambda$ and $e \in E$

Example 13. Let Λ={p}, we can define the following Shape Expression Schema: The regular shape expression

$$
\begin{aligned}
p \;\longmapsto\; & \text{\textvisiblespace} \xrightarrow{a} 1 \\
\| \;\; & \text{\textvisiblespace} \xrightarrow{b} \{1,2\}+ \\
\| \;\; & \text{\textvisiblespace} \xrightarrow{c} p*
\end{aligned}
$$

declares a schema where nodes of shape p contain an arc with predicate a and value 1, one or more arcs with predicate b and values 1 or 2, and zero or more arcs with predicate c and values of shape p. Notice that shape expression schemas can contain recursive references.

Example 14. Let Λ={person}, we can define the following Shape Expression Schema which corresponds to example 1

$$
\begin{aligned}
\texttt{person} \longmapsto\; & \text{\textvisiblespace} \xrightarrow{\texttt{foaf:age}} \texttt{xsd:int} \\
\| \; & \text{\textvisiblespace} \xrightarrow{\texttt{foaf:name}} \texttt{xsd:string+} \\
\| \; & \text{\textvisiblespace} \xrightarrow{\texttt{foaf:knows}} \texttt{person*}
\end{aligned}
$$

A shape typing is a mapping from nodes in a graph to labels. Given a graph and a regular shape schema, we define a type inference algorithm which assigns a shape typing to the nodes in the graph. The expression $\Gamma \vdash n \simeq_s s$ represents the shape typings generated when matching a node n with a shape s in the context Γ.

The context contains the current typing which can be accessed through $\Gamma.typing$. The expression $\Gamma\{n \rightarrow t\}$ means the addition of type t to n in context Γ. The semantic definition of $\simeq_s$ is depicted in Figure 3.

We define the following definitions on shape typings:

$\odot$	=	Empty typing
$n \rightarrow s : \tau$	=	Add shape type s to node n in typing t
$\tau_1 \uplus \tau_2$	=	Combine typings τ_1 and τ_2

The operational semantics presented in figure 1 can be extended to handle shape typings. The definitions are presented in figure 4. As can be seen the definitions are straightforward. The main novelty is the semantics of arcs which have been divided in two cases. Arc_{type} handles the case where the shape expression contains a value set, while Arc_{ref} handles the case where the shape expression contains a reference to a label. In that case, the object is matched against the shape expression associated with that label.

In order to adapt the inference rules to employ the derivatives algorithm, we modify the derivative function $\partial_t(e, \Gamma)$ to take a new parameter Γ that represents the typing context and to return a pair (e', τ) where $e \in E$ represents the derivative and τ represents the resulting typing. The new definition is:

$$
\begin{aligned}
\partial_t(\emptyset, \Gamma) &= (\emptyset, \odot) \\
\partial_t(\varepsilon, \Gamma) &= (\emptyset, \odot) \\
\partial_{\langle s,p,o\rangle}(\text{\textvisiblespace} \xrightarrow{v_p} v_o, \Gamma) &= \begin{cases} (\varepsilon, \Gamma.\tau) & \text{if } p \in v_p \text{ and } o \in v_o \\ (\emptyset, \odot) & \text{otherwise} \end{cases} \\
\partial_{\langle s,p,o\rangle}(\text{\textvisiblespace} \xrightarrow{v_p} l, \Gamma) &= \begin{cases} (\varepsilon, \tau) & \text{if } \Gamma\{o \rightarrow l\} \vdash \delta(l) \simeq_s \Sigma^g_o \rightsquigarrow \tau \\ (\emptyset, \odot) & \text{otherwise} \end{cases} \\
\partial_t(e*) &= \textbf{let } (e', \tau) = \partial_t(e, \Gamma) \\
&\quad\; \textbf{in } e' \;\|\; e* \\
\partial_t(e_1 \;\|\; e_2) &= \textbf{let } (e_1', \tau_1) = \partial_t(e_1, \Gamma) \\
&\qquad\quad\; (e_2', \tau_2) = \partial_t(e_2, \Gamma) \\
&\quad\; \textbf{in } (e_1' \,\|\, e_2 \mid e_2' \,\|\, e_1, \tau_1 \uplus \tau_2) \\
\partial_t(e_1 \mid e_2) &= \textbf{let } (e_1', \tau_1) = \partial_t(e_1, \Gamma) \\
&\qquad\quad\; (e_2', \tau_2) = \partial_t(e_2, \Gamma) \\
&\quad\; \textbf{in } (e_1' \mid e_2', \tau_1 \uplus \tau_2)
\end{aligned}
$$

The algorithm to match a pointed node n in a graph g against a label l from schema s and context typing Γ is represented as:

$$
\Gamma \vdash l \simeq_s n \rightsquigarrow \Gamma\{n \rightarrow l\} \vdash \delta^s(l) \simeq \Sigma^g_n
$$

where

$$
\begin{aligned}
\Gamma \vdash e \simeq t \rtimes t_s &\rightsquigarrow \Gamma \vdash \partial_t(e) \simeq t_s \\
\Gamma \vdash e \simeq \{\} &\rightsquigarrow \begin{cases} \Gamma & \text{if } \nu(e) \\ \odot & \text{otherwise} \end{cases}
\end{aligned}
$$

The algorithm presented in this paper has been implemented in Scala[7] and Haskell[8]. The Scala implementation contains several extensions like reverse arcs, relations, negations, etc. that have been omitted in this paper for brevity while the Haskell prototype follows the simplified definitions presented here. Comparing the per-

[7] `http://labra.github.io/ShExcala/`
[8] `http://labra.github.io/Haws/`

$$MatchShape\frac{\Gamma\{n \to l\} \vdash \delta(l) \simeq \Sigma_n^g \rightsquigarrow \tau}{\Gamma \vdash l \simeq_s n \rightsquigarrow \tau}$$

Figure 3: Inference rule to match shapes

$$Or_1\frac{\Gamma \vdash r_1 \simeq g \rightsquigarrow \tau}{\Gamma \vdash r_1|r_2 \simeq g \rightsquigarrow \tau} \qquad Or_2\frac{\Gamma \vdash r_2 \simeq g \rightsquigarrow \tau}{\Gamma \vdash r_1|r_2 \simeq g \rightsquigarrow \tau}$$

$$And\frac{\Gamma \vdash r_1 \simeq g_1 \rightsquigarrow \tau_1 \qquad \Gamma \vdash r_2 \simeq g_2 \rightsquigarrow \tau_2}{\Gamma \vdash r_1 \parallel r_2 \simeq g_1 \oplus g_2 \rightsquigarrow \tau_1 \uplus \tau_2}$$

$$Empty\frac{}{\Gamma \vdash \varepsilon \simeq \{\} \rightsquigarrow \odot}$$

$$Star_1\frac{}{\Gamma \vdash r* \simeq \{\} \rightsquigarrow \odot} \qquad Star_2\frac{\Gamma \vdash r \simeq g_1 \rightsquigarrow \tau_1 \qquad \Gamma \vdash r* \simeq g_2 \rightsquigarrow \tau_2}{\Gamma \vdash r* \simeq g_1 \oplus g_2 \rightsquigarrow \tau_1 \uplus \tau_2}$$

$$Arc_{type}\frac{p \in v_p \qquad o \in v_o}{\Gamma \vdash \textvisiblespace \xrightarrow{v_p} v_o \simeq \langle s,p,o\rangle \rightsquigarrow \odot} \qquad Arc_{ref}\frac{\Gamma \vdash l \simeq_s o \rightsquigarrow \tau}{\Gamma \vdash \textvisiblespace \xrightarrow{v_p} l \simeq \langle s,p,o\rangle \rightsquigarrow \tau}$$

Figure 4: Inference rules for Shape expression schemas

formance between the backtracking and the derivatives approach, we noticed that the latter obtains better results than the former.

Although the theoretical complexity of Shape Expression validation, which has been characterized in [1], remains the same, the derivatives algorithm behaves much better than the backtracking one. Further work needs to be done to check if we can identify a subset of the language with better complexity results while being expressive enough. In particular, the *Single Occurrence Regular Bag Expressions* subset defined in that paper offers a tractable language which could be expressive enough. In the future we are planning to adapt our implementation to that subset and study its performance behaviour in practice.

9. RELATED WORK

Regular expression derivatives where introduced by Brzozowski in 1964[3] and were used for string based recognizers of regular expressions. In 1999, Joe English proposed the use of derivatives for XML validation [9]. That idea was taken by James Clark to implement RelaxNG [4]. An updated presentation of regular expression derivatives is presented in [21] where the authors describe how to handle large character sets (Unicode). Our presentation follows the notations used in that paper adapted to regular shape expressions. With regards to the implementation, we took some inspiration by the Haskell implementation of a W3C XML Schema regular expression matcher maintained by Uwe Schmidt [26] which contains a definition for the interleave operator. There has also been some recent work applying regular expression derivatives to submatching [28] and parsing [17] and comparing it with the more traditional approach to regular expression matching based on NFA [7].

The main inspiration for Shape Expressions has been RelaxNG [30], a Schema language for XML that offers a good trade-off between expressiveness and validation efficiency. The semantics of RelaxNG has also been expressed using inference rules in the specification document [20] and is based on tree grammars [19]. Inspired by that specification, we presented the semantics of Shape Expressions using type inference rules in [23]. Our first prototype implementation of Shape Expressions in Haskell[9] employed a direct translation of the inference rules using a backtracking monad transformer. We consider that the equational reasoning presentation of the algorithm can be used to proof its correctness using an inductive representation of RDF graphs [15].

Besides Shape Expressions, there are several approaches that have been proposed to validate RDF Graphs which can be roughly classified as: inference based, SPARQL-based and grammar-based approaches.

OWL based approaches try to adapt RDF Schema or OWL to express validation semantics. However, using Open World and Non-unique name assumption limits validation possibilities. [5, 29, 18] propose the use of OWL expressions with a Closed World Assumption to express integrity constraints.

SPARQL-based approaches use the SPARQL Query Langugage to express the validation constraints. SPARQL has much more expressiveness than Shape Expressions and can even be used to validate numerical and statistical computations [14]. SPARQL Inferencing Notation (SPIN)[12] constraints associate RDF types or nodes with validation rules. These rules are expressed as SPARQL queries. There have been other proposals using SPARQL combined with other technologies, Simister and Brickley[27] propose a combination between SPARQL queries and property paths which is used in Google and Kontokostas et al [13] proposed *RDFUnit* a Test-driven framework which employs SPARQL query templates that are instantiated into concrete quality test queries.

Grammar based approaches define a domain specific language to declare the validation rules. OSLC Resource Shapes [25] have been proposed as a high level and declarative description of the expected contents of an RDF graph expressing constraints on RDF terms. Shape Expressions have been inspired by OSLC although they offer more expressive power. Dublin Core Application Profiles [8] also define a set of validation constraints using Description Templates with less expressiveness than Shape Expressions.

[9]Available at `https://github.com/labra/haws`

10. CONCLUSIONS AND FUTURE WORK

The industrial adoption of an RDF schema language will depend on rigorous analysis of efficiency of Shape Expressions and other approaches to schema.

In this paper, we propose an implementation of Shape Expressions inspired by derivatives of regular expressions.

There are two main lines of future work: On one hand, we are planning to develop a set of benchmarks that will enable us to assess the performance of the different shape expression implementations. On the other hand, we are currently working on the implementation of new features for the Shape Expression language. The evolution of the recently chartered W3c Data Shapes Working group will affect the adoption of those features. In this paper we offered a minimal set of language features which we consider representative. However, there are several extension proposals like inverse arcs, negations, predicates, etc. that could also be implemented using the proposed approach.

11. ACKNOWLEDGEMENTS

We thank Joshua Taylor for his careful review of this paper and his help in the definition of SPARQL queries. This work has been partially funded by the Spanish project MICINN-12-TIN2011-27871 ROCAS (Reasoning On the Cloud by Applying Semantics).

12. REFERENCES

[1] I. Boneva, J. E. Labra Gayo, S. Hym, E. G. Prud'hommeaux, H. Solbrig, and S. Staworko. Complexity and expressiveness of ShEx for RDF. In *International Conference on Database Theory (ICDT)*, 2015.

[2] D. Brickley and R. V. Guha. RDF Schema 1.1. `http://www.w3.org/TR/rdf-schema/`, 2014.

[3] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[4] J. Clark. An algorithm for RELAX NG validation. `http://www.thaiopensource.com/relaxng/derivative.html`, 2002.

[5] K. Clark and E. Sirin. On RDF validation, stardog ICV, and assorted remarks. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[6] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient inclusion for a class of xml types with interleaving and counting. *Inf. Syst.*, 34(7):643–656, Nov. 2009.

[7] R. Cox. Regular expression matching in the wild. `http://swtch.com/~rsc/regexp/regexp3.html`, March 2010.

[8] K. Coyle and T. Baker. Dublin core application profiles. separating validation from semantics. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[9] J. English. How to validate XML. `http://www.flightlab.com/~joe/sgml/validate.html`.

[10] W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theoretical Computer Science*, 411(31-33):2987 – 2998, 2010.

[11] P. J. Hayes and P. F. Patel-Schneider. RDF 1.1 Semantics. `http://www.w3.org/TR/rdf11-mt/`, 2014.

[12] H. Knublauch. SPIN - Modeling Vocabulary. `http://www.w3.org/Submission/spin-modeling/`, 2011.

[13] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri. Test-driven evaluation of linked data quality. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 747–758, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee.

[14] J. E. Labra Gayo and J. M. Álvarez Rodríguez. Validating statistical index data represented in RDF using SPARQL queries. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[15] J. E. Labra Gayo, J. Jeuring, and J. M. Álvarez Rodríguez. Inductive representations of RDF graphs. *Science of Computer Programming*, 95, Part 1(0):135 – 146, 2014. Special Issue on Systems Development by Means of Semantic Technologies.

[16] J. E. Labra Gayo, E. Prud'hommeaux, H. Solbrig, and J. M. Alvarez Rodríguez. Validating and describing linked data portals using RDF Shape Expressions. In *1st Workshop on Linked Data Quality*, Sept. 2014.

[17] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195, Sept. 2011.

[18] B. Motik, I. Horrocks, and U. Sattler. Adding Integrity Constraints to OWL. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWL: Experiences and Directions 2007 (OWLED 2007)*, Innsbruck, Austria, June 6–7 2007.

[19] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, Nov. 2005.

[20] OASIS Committee Specification. RELAX NG Specification:. http://relaxng.org/spec-20011203.html, 2001.

[21] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[22] W. OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 2012. Available at `http://www.w3.org/TR/owl2-overview/`.

[23] E. Prud'hommeaux, J. E. Labra, and H. Solbrig. Shape expressions: An RDF validation and transformation language. In *10th International Conference on Semantic Systems*, Sept. 2014.

[24] RDF Working Group W3c. W3c validation workshop. practical assurances for quality rdf data, September 2013.

[25] A. G. Ryman, A. L. Hors, and S. Speicher. OSLC resource shape: A language for defining constraints on linked data. In C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, editors, *Linked data on the Web*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[26] U. Schmidt. Regular expressions for XML Schema. `http://www.haskell.org/haskellwiki/Regular_expressions_for_XML_Schema`, 2010.

[27] S. Simister and D. Brickley. Simple application-specific constraints for rdf models. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c,

`http://www.w3.org/2012/12/rdf-val`.

[28] M. Sulzmann and K. Z. M. Lu. POSIX regular expression parsing with derivatives. In M. Codish and E. Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2014.

[29] J. Tao, E. Sirin, J. Bao, and D. L. McGuinness. Integrity constraints in OWL. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*. AAAI, 2010.

[30] E. van der Vlist. *Relax NG: A Simpler Schema Language for XML*. O'Reilly, Beijing, 2004.