Recursion in RDF Data Shape Languages

Arthur Ryman, arthur.ryman@gmail.com

June 21, 2015

Abstract

An RDF data shape is a description of the expected contents of an RDF document (aka graph) or dataset. A major part of this description is the set of constraints that the document or dataset is required to satisfy. W3C recently (2014) chartered the RDF Data Shapes Working Group to define SHACL, a standard RDF data shape language. We refer to the ability to name and reference shape language elements as recursion. This article provides a precise definition of the meaning of recursion as used in Resource Shape 2.0. The definition of recursion presented in this article is largely independent of language-specific details. We speculate that it also applies to ShEx and to all three of the current proposals for SHACL. In particular, recursion is not permitted in the SHACL-SPARQL proposal, but we conjecture that recursion could be added by using the definition proposed here as a top-level control structure.

1 Introduction

An RDF *data shape* is a description of the expected contents of an RDF document (aka graph) or dataset. A major part of this description is the set of constraints that the document or dataset is required to satisfy. In this respect, data shapes do for RDF what XML Schema[6] does for XML. The term *shape* is used instead of *schema* to avoid confusion with RDF Schema[3] which, like OWL[9], describes inference rules, not constraints.

W3C recently (2014) chartered the RDF Data Shapes Working Group to define SHACL, a standard RDF data shape language[8]. Both of the member submissions to this working group, Resource Shape 2.0[12] and Shape Expressions (ShEx) [13] allow shapes to refer to each other. For example, in Resource Shape 2.0 the property `oslc:valueShape` lets one resource shape refer to another. ShEx has a similar feature. In these languages, a shape may refer directly or indirectly to itself.

We refer to the ability to name and reference shape language elements as *recursion* in analogy with that ubiquitous feature of programming languages which allows a function to call other functions, including itself. Of course, when writing a recursive function care must be taken to ensure that recursion terminates. Similarly, when defining a shape language care must be taken to spell out the precise meaning of recursion. Neither of the member submissions included a precise definition of recursion.

This article provides a precise definition of the meaning of recursion as used in Resource Shape 2.0. Precision is achieved through the use of Z Notation [14], a formal specification language based on typed set theory. The LaTeX source for this article has been type-checked using the *f*UZZ type-checker [15].

The definition of recursion presented in this article is largely independent of language-specific details. We speculate that it also applies to ShEx and to all three of the current proposals for SHACL. In particular, recursion is not permitted in the SHACL-SPARQL proposal [10], but we conjecture that recursion could be added by using the definition proposed here as a top-level control structure.

1.1 Organization of this Article

The remainder of this article is organized as follows.

- Section 2 introduces examples in order to ground the following definitions.
- Section 3 defines a few basic RDF concepts.
- Section 4 defines neighbour functions and graphs which form the basis for the following definition of recursion.
- Section 5 defines constraints.
- Section 6 defines recursive shapes.
- Section 7 discusses how the proposed definition of recursion relates to the existing and proposed shape languages.
- Section 8 concludes the article.

2 Examples

This section defines two examples. The first is the Personal Information Management application, which is highly simplified, but illustrates the concepts and is easy to understand. The second is the ....

2.1 The Personal Information Management Application

We use a highly simplified running example to illustrate the concepts defined in the following sections. Each formal definition is instantiated with data drawn from the running example in order to help the reader understand the formalism and relate it to RDF. Although the inclusion of examples lengthens the presentation, we hope that it will make the formalism more tangible and accessible to readers who are unfamiliar with Z Notation.

Consider a Linked Data [2] application for Personal Information Management (PIM). The application manages documents that contain information about a *contact* person and their *associates*. As a Linked Data application, the PIM application provides a REST API for creating, retrieving, updating, and deleting constant information over HTTP using RDF representations of the data. Shapes are useful in this context for two main reasons. First, the PIM application may publish shapes that describe the contact information so that application developers who want to use the REST API understand the API contract. Second, the PIM application may internally use a shape engine that automatically validates the data, especially incoming creation and update requests.

The application maintains the following integrity constraints where the prefixes `rdf:`[5] and `foaf:`[4] have the usual meanings.

- Each document contains information about exactly one contact person and zero or more of their associates. A contact person is never an associate of themself.
- Each contact has type `foaf:Person` and has exactly one name given by the property `foaf:name`.
- The contact's associates are given by the property `foaf:knows` which may have zero or more values.
- Each associate has type `foaf:Person` and has exactly one name given by `foaf:name`.
- Each associate is known by exactly one contact given by following the property `foaf:knows` in the backward direction, i.e. the associate is the object of the property and the contact is the subject.

Note that these constraints are circular since the definition of contact refers to the definition of associate, and conversely. We have an obligation to give this circularity a precise meaning.

These constraints are illustrated by a valid document for Alice (Listing 1) and and invalid document for Bob (Listing 2). All RDF source code examples are written in Turtle format [1].

The following document about Alice satisfies all the constraints of the application.

```
# http://example.org/contacts/alice
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@base <http://example.org/contacts/> .

<alice#me> a foaf:Person ;
    foaf:name "Alice" ;
    foaf:knows
        <bob#me> ,
        <charlie#me> .
```

```

<bob#me> a foaf:Person ;
    foaf:name "Bob" .

<charlie#me> a foaf:Person ;
    foaf:name "Charlie" .
```

Listing 1: Contact document for Alice

The following document about Bob violates some of the constraints of the application.

```
# http://example.org/contacts/bob
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@base <http://example.org/contacts/> .

<bob#me> a foaf:Person ;
    foaf:name "Bob" ;
    foaf:knows
        <alice#me> ,
        <charlie#me> .

<alice#me> foaf:name "Alice" .

<charlie#me> a foaf:Person .
```

Listing 2: Contact document for Bob

It is clear that the document about Bob is invalid, since Alice has no type and Charlie has no name.

It is also intuitively clear that the document about Alice is valid. However, if we naively translate the PIM constraints into logical conditions on the document about Alice, then we run into a problem. All the constraints about types, names, and who knows who are satisfied and unproblematic, but the constraints about what it means to be a contact or an associate are circular. A naive translation of these constraints on the Alice document is as follows.

- If Bob is an associate and Charlie is an associate then Alice is a contact.
- If Alice is a contact then Bob is an associate.
- If Alice is a contact the Charlie is an associate.

Table 1 introduces propositional variables to stand for statements about being a contact or associate in the Alice document.

The PIM constraints on the Alice document translate to the following consistency condition.

$$(B \land C \Rightarrow A) \land (A \Rightarrow B) \land (A \Rightarrow C)$$

Unfortunately, this consistency condition does not uniquely determine the values of the propositional variables. In fact, this consistency condition has

Variable	Meaning
A	Alice is a contact.
B	Bob is an associate.
C	Charlie is an associate.

Table 1: Meaning of propositional variables in the Alice document

A	B	C
true	true	true
false	true	false
false	false	true
false	false	false

Table 2: Solutions to PIM constraints in the Alice document

several solutions as shown in Table 2. Only the solution in which all the propositional variable are true agrees with our intuition.

This analysis shows that the naive translation of the constraints about contacts and associates produces a necessary, but not sufficient, consistency condition on the meaning of these constraints. A precise definition for this type of constraint is given in Section 6. A brief overview of this definition follows.

The correct interpretation of the constraints is based on the observation that they specify two essentially different kinds of information. One kind defines rules for labelling nodes with names. The other kind defines a set of conditions associated with each name and asserts that these conditions must hold at each node labelled with that name.

In the PIM application, the names are *contact* and *associate.* The rules for labelling the nodes in a document are as follows.

1. Initially, no node has any labels.
2. Start with the node that corresponds to the person that the document is about, and label it as a contact.
3. For each node labelled as a contact, find all the nodes they know, and add an associate label to each of them.
4. For each node labelled as an associate, find all the nodes that they are known by and add a contact label each of them.
5. Repeat the previous two steps until no new labels are added.
6. Note that this procedure always terminates because the number of nodes is finite and the number of names is finite (2 in this case).

In the Alice document, the labelling procedure results in the nodes being labelled as follows.

- Alice is labelled as a contact because the document is about Alice and Alice is known by Bob and Charlie.
- Bob is labelled as an associate because Alice knows Bob.
- Charlie is labelled as an associate because Alice knows Charlie.

Whenever a node gets labelled with a name, the conditions associated with the name must hold. No recursion is involved in this step.

The conditions that must hold for nodes labelled with contact are as follows.

- A contact must be a person.
- A contact must have exactly one name.
- A contact must not know itself.

The conditions that must hold for nodes labelled with associate are as follows.

- An associate must be a person.
- An associate must have exactly one name.
- An associate must be known by exactly one node.

Although the statement of the PIM constraints uses recursion, the properties of the data in this case allow us to write an equivalent non-recursive statement[11]. Specifically, since a node is an associate only if it is known by a contact, and an associate must be known by exactly one contact, nothing more is gained by requiring that all nodes that know an associate must be contacts. Dropping this condition removes the recursion. However, in general we cannot convert a recursive constraint into an equivalent non-recursive constraint. The following example illustrates an essentially recursive constraint.

3 Basic RDF Concepts

This section introduces some basic RDF concepts. For full definitions consult the RDF specification[5].

3.1 Terms

Let *TERM* be the set of all RDF *terms*.

$$[\mathit{TERM}]$$

The set of all RDF terms is partitioned into *IRIs*, *blank nodes*, and *literals*.

$$\begin{array}{|l}
\mathit{IRI}, \mathit{BNode}, \mathit{Literal} : \mathbb{P}\ \mathit{TERM} \\
\hline
\langle \mathit{IRI}, \mathit{BNode}, \mathit{Literal} \rangle \ \mathsf{partition}\ \mathit{TERM}
\end{array}$$

For example, the documents for Alice and Bob contain the following distinct literals where *Alice* denotes `"Alice"`, etc.

$$\begin{array}{|l} \mathit{Alice}, \mathit{Bob}, \mathit{Charlie} : \mathit{Literal} \\ \hline \mathsf{disjoint}\ \langle \{\mathit{Alice}\}, \{\mathit{Bob}\}, \{\mathit{Charlie}\} \rangle \end{array}$$

and the following distinct IRIs where *alice* denotes `http://example.org/contacts/alice#me`, etc., *rdf_type* denotes `rdf:type`, and *foaf_Person* denotes `foaf:Person`, etc.

$$\begin{array}{|l} \mathit{alice}, \mathit{bob}, \mathit{charlie} : \mathit{IRI} \\ \mathit{rdf\_type} : \mathit{IRI} \\ \mathit{foaf\_Person}, \mathit{foaf\_name}, \mathit{foaf\_knows} : \mathit{IRI} \\ \hline \mathsf{disjoint}\ \langle \{\mathit{alice}\}, \{\mathit{bob}\}, \{\mathit{charlie}\}, \{\mathit{rdf\_type}\}, \\ \qquad \{\mathit{foaf\_Person}\}, \{\mathit{foaf\_name}\}, \{\mathit{foaf\_knows}\} \rangle \end{array}$$

3.2 Triples

An RDF *triple* is a statement that consists of three terms referred to as *subject*, *predicate*, and *object*.

$$\mathit{Triple} == \{\, s, p, o : \mathit{TERM} \mid s \notin \mathit{Literal} \land p \in \mathit{IRI} \,\}$$

- The subject must not be a literal.
- The predicate must be an IRI.

For example, the statement that Alice is a person is represented by the following triple.

$$\vdash (\mathit{alice}, \mathit{rdf\_type}, \mathit{foaf\_Person}) \in \mathit{Triple}$$

3.3 Graphs

It is common to visualize a triple as a directed arc from the subject to the object, labelled by the predicate. A set of triples may therefore may visualized as a directed graph (technically, a directed, labelled, multigraph). We are only concerned with finite graphs here.

An RDF *graph* is a finite set of triples.

$$\mathit{Graph} == \mathbb{F}\ \mathit{Triple}$$

For example, the following graph contains the triples in the document about Alice.

$$\begin{array}{|l}
alice\_graph : Graph \\
\hline
alice\_graph = \\
\qquad \{(alice, rdf\_type, foaf\_Person), \\
\qquad (alice, foaf\_name, Alice), \\
\qquad (alice, foaf\_knows, bob), \\
\qquad (alice, foaf\_knows, charlie), \\
\qquad (bob, rdf\_type, foaf\_Person), \\
\qquad (bob, foaf\_name, Bob), \\
\qquad (charlie, rdf\_type, foaf\_Person), \\
\qquad (charlie, foaf\_name, Charlie)\}
\end{array}$$

It is convenient to define the sets of subjects, predicates, and objects that appear in a graph.

$$subjects == (\lambda\, g : Graph \bullet \{\, s, p, o : TERM \mid (s, p, o) \in g \bullet s \,\})$$

$$predicates == (\lambda\, g : Graph \bullet \{\, s, p, o : TERM \mid (s, p, o) \in g \bullet p \,\})$$

$$objects == (\lambda\, g : Graph \bullet \{\, s, p, o : TERM \mid (s, p, o) \in g \bullet o \,\})$$

For example, the graph for Alice contains the following predicates.

$$\begin{array}{l}
\vdash predicates(alice\_graph) = \\
\qquad \{rdf\_type, foaf\_name, foaf\_knows\}
\end{array}$$

The *nodes* of a graph are its subjects and objects.

$$nodes == (\lambda\, g : Graph \bullet subjects(g) \cup objects(g))$$

For example, the graph for Alice contains the following nodes.

$$\begin{array}{l}
\vdash nodes(alice\_graph) = \\
\qquad \{alice, bob, charlie, Alice, Bob, Charlie, foaf\_Person\}
\end{array}$$

A *pointed graph* consists of a graph and a *base node* in the graph.

$$\begin{array}{|l|}
\hline
PointedGraph \\
graph : Graph \\
baseNode : TERM \\
\hline
baseNode \in nodes(graph) \\
\hline
\end{array}$$

- The base node is a node in the graph.

The base node of a pointed graph is also referred to as the *start node* or *focus node* of the graph, depending on the context.

For example, *alice* is the natural base node of the graph for Alice.

$$\begin{array}{|l}
alice\_pg : PointedGraph \\
\hline
alice\_pg.graph = alice\_graph \\
alice\_pg.baseNode = alice
\end{array}$$

- The graph is *alice_graph*.
- The base node is *alice*.

4 Neighbour Functions

RDF applications often impose conditions on nodes, and related conditions on their *neighbours*, where a neighbour is some node that bears a specified relation to the given node. When the neighbour relation between nodes is specified by traversing triples, we say that the nodes are connected by a *path*. SPARQL 1.1[7] defines a *property path* syntax for specifying paths.

More generally, applications may use neighbour relations that cannot be specified by property paths. Many such relations might be specified by SPARQL queries that bind pairs of variables to nodes. For maximum generality, we do not place restrictions on how neighbour relations are specified.

A *neighbour function* is any mapping from graphs to pair of nodes.

$$\begin{array}{|l}
\mathit{Neighbour} : \mathbb{P}(\mathit{Graph} \longrightarrow (\mathit{TERM} \leftrightarrow \mathit{TERM})) \\
\hline
\mathit{Neighbour} = \\
\quad \{\, q : \mathit{Graph} \longrightarrow (\mathit{TERM} \leftrightarrow \mathit{TERM}) \mid \\
\qquad (\forall\, g : \mathit{Graph} \bullet q(g) \subseteq \{\, x, y : \mathit{nodes}(g) \,\}) \,\}
\end{array}$$

- A neighbour function is a mapping that maps a graph g to a binary relation on the nodes of g.

We say that the pair of nodes (x, y) *matches* the neighbour function q in the graph g when $(x, y) \in q(g)$.

4.1 Simple Path Expressions

Simple path expressions define a very commonly used type of neighbour function.

A predicate p defines a simple path expression $\mathit{forward}(p)$ by traversing triples in the forward direction. Forward path expressions are referred to as `PredicatePath` expressions in SPARQL 1.1.

$$\begin{array}{|l}
\mathit{forward} : \mathit{IRI} \longrightarrow \mathit{Neighbour} \\
\hline
\forall\, p : \mathit{IRI};\ g : \mathit{Graph} \bullet \\
\quad \mathit{forward}(p)(g) = \\
\qquad \{\, s, o : \mathit{nodes}(g) \mid (s, p, o) \in g \,\}
\end{array}$$

- The simple path expression $\mathit{forward}(p)$ matches all pairs (s, o) such that (s, p, o) is a triple in g.

For example, the following are forward path expressions.

$$\begin{array}{l}
\mathit{has\_type} == \mathit{forward}(\mathit{rdf\_type}) \\
\mathit{has\_name} == \mathit{forward}(\mathit{foaf\_name}) \\
\mathit{knows} == \mathit{forward}(\mathit{foaf\_knows})
\end{array}$$

The forward path expression $\mathit{has\_type}$ matches the following pairs of nodes in the graph for Alice.

$$\begin{array}{l}
\vdash \mathit{has\_type}(\mathit{alice\_graph}) = \\
\qquad \{(\mathit{alice}, \mathit{foaf\_Person}), \\
\qquad (\mathit{bob}, \mathit{foaf\_Person}), \\
\qquad (\mathit{charlie}, \mathit{foaf\_Person})\}
\end{array}$$

Similarly, a predicate p defines a simple path expression $\mathit{backward}(p)$ by traversing triples in the backward direction. Backward path expressions are referred to as `InversePath` expressions in SPARQL 1.1.

$$\begin{array}{|l}
\mathit{backward} : \mathit{IRI} \longrightarrow \mathit{Neighbour} \\ \hline
\forall\, p : \mathit{IRI};\ g : \mathit{Graph} \bullet \\
\quad \mathit{backward}(p)(g) = \\
\qquad \{\, o, s : \mathit{nodes}(g) \mid (s, p, o) \in g \,\}
\end{array}$$

- The simple path expression $\mathit{backward}(p)$ matches all pairs (o, s) such that (s, p, o) is a triple in g.

For example, the following is a backward path expression.

$$\mathit{is\_known\_by} == \mathit{backward}(\mathit{foaf\_knows})$$

The backward path expression $\mathit{is\_known\_by}$ matches the following pairs of nodes in the graph for Alice.

$$\begin{array}{l}
\vdash \mathit{is\_known\_by}(\mathit{alice\_graph}) = \\
\qquad \{(\mathit{bob}, \mathit{alice}), \\
\qquad (\mathit{charlie}, \mathit{alice})\}
\end{array}$$

4.2 Values

Given a graph g and a node $x \in \mathit{nodes}(g)$, the set of all nodes that can be reached from x by matching the neighbour function q is $\mathit{values}(g, x, q)$.

$$\begin{array}{|l}
\mathit{values} : \mathit{Graph} \times \mathit{TERM} \times \mathit{Neighbour} \longrightarrow \mathbb{F}\ \mathit{TERM} \\ \hline
\forall\, g : \mathit{Graph};\ x : \mathit{TERM};\ q : \mathit{Neighbour} \bullet \\
\quad \mathit{values}(g, x, q) = \{\, y : \mathit{nodes}(g) \mid (x, y) \in q(g) \,\}
\end{array}$$

- The node y is in $\mathit{values}(g, x, q)$ when (x, y) matches q in g.

For example, in the graph for Alice the node *alice* and forward path expression *knows* have the following values.

$$\vdash \mathit{values}(\mathit{alice\_graph}, \mathit{alice}, \mathit{knows}) = \{\mathit{bob}, \mathit{charlie}\}$$

5 Constraints

RDF applications often impose constraints on the data graphs they process. A given graph either *satisfies* or *violates* the constraint. Thus a constraint partitions the set of all graphs into two disjoint subsets, namely the set of all graphs that satisfy the constraint and the set of all graphs that violate the constraint. We therefore equate a constraint to the set of graphs that satisfy it.

A *constraint* is a, possibly infinite, set of graphs.

$$Constraint == \mathbb{P}\ Graph$$

For example, suppose we define a *small graph* to be a graph that has at most 10 triples. The set of all small graphs is a constraint.

$$\begin{array}{|l}
small\_graphs : Constraint \\
\hline
small\_graphs = \{ g : Graph \mid \#g \leq 10 \}
\end{array}$$

The Alice graph satisfies this constraint.

$$\vdash alice\_graph \in small\_graphs$$

5.1 Node Constraints

A *parameterized constraint* is a mapping from some parameter set X to constraints.

$$ParameterizedConstraint[X] == X \longrightarrow Constraint$$

A *term constraint* is a constraint that is parameterized by terms.

$$TermConstraint == ParameterizedConstraint[TERM]$$

For example, given a term $x \in TERM$, the constraint $hasSubject(x)$ is the set of all graphs that have x as a subject.

$$\begin{array}{|l}
hasSubject : TermConstraint \\
\hline
\forall x : TERM \bullet \\
\qquad hasSubject(x) = \{ g : Graph \mid x \in subjects(g) \}
\end{array}$$

Similarly, $hasPredicate(x)$, $hasObject(x)$, and $hasNode(x)$ are constraints with the analogous definitions.

$$hasPredicate == (\lambda x : TERM \bullet \{ g : Graph \mid x \in predicates(g) \})$$

$$hasObject == (\lambda x : TERM \bullet \{ g : Graph \mid x \in objects(g) \})$$

$$hasNode == (\lambda x : TERM \bullet \{ g : Graph \mid x \in nodes(g) \})$$

Note that $hasNode(x)$ is the union of $hasSubject(x)$ and $hasObject(x)$.

$$\vdash \forall x : TERM \bullet \\ \quad hasNode(x) = hasSubject(x) \cup hasObject(x)$$

A *node constraint* is a term constraint in which the term is a node in each graph that satisfies the constraint.

$$\begin{array}{|l} NodeConstraint : \mathbb{P}\ TermConstraint \\ \hline NodeConstraint = \\ \quad \{\, c : TermConstraint \mid \forall x : TERM \bullet \forall g : c(x) \bullet x \in nodes(g) \,\} \end{array}$$

For example, *hasNode* is a node constraint.

$$\vdash hasNode \in NodeConstraint$$

The PIM application enforces the following node constraints.
Both contact and associate nodes must be people.

$$\begin{array}{|l} is\_a\_person : NodeConstraint \\ \hline \forall x : TERM \bullet \\ \quad is\_a\_person(x) = \\ \qquad \{\, g : Graph \mid (x, rdf\_type, foaf\_Person) \in g \,\} \end{array}$$

- A node *is a person* when it has a `foaf:Person` as its RDF type.

For example, the Alice graph satisfies this constraint at the *alice*, *bob*, and *charlie* nodes.

$$\vdash alice\_graph \in is\_a\_person(alice) \land \\ alice\_graph \in is\_a\_person(bob) \land \\ alice\_graph \in is\_a\_person(charlie)$$

Both contact and associate nodes must have exactly one name.

$$\begin{array}{|l} has\_one\_name : NodeConstraint \\ \hline \forall x : TERM \bullet \\ \quad has\_one\_name(x) = \\ \qquad \{\, g : Graph \mid \exists_1\, y : TERM \bullet (x, foaf\_name, y) \in g \,\} \end{array}$$

- A node *has one name* when it is the subject of exactly one `foaf:name` triple.

For example, the Alice graph satisfies this constraint at the *alice*, *bob*, and *charlie* nodes.

$$\vdash alice\_graph \in has\_one\_name(alice) \land \\ alice\_graph \in has\_one\_name(bob) \land \\ alice\_graph \in has\_one\_name(charlie)$$

Associate nodes must be known by exactly one node.

$$\begin{array}{|l} \mathit{is\_known\_by\_one} : \mathit{NodeConstraint} \\ \hline \forall x : \mathit{TERM} \bullet \\ \quad \mathit{is\_known\_by\_one}(x) = \\ \qquad \{ g : \mathit{Graph} \mid \exists_1 y : \mathit{TERM} \bullet (y, \mathit{foaf\_knows}, x) \in g \} \end{array}$$

- A node *is known by one* node when it is the object of exactly one `foaf:knows` triple.

For example, the Alice graph satisfies this constraint at the *bob* and *charlie* nodes.

$$\begin{array}{l} \vdash \mathit{alice\_graph} \in \mathit{is\_known\_by\_one}(\mathit{bob}) \land \\ \mathit{alice\_graph} \in \mathit{is\_known\_by\_one}(\mathit{charlie}) \end{array}$$

A contact node must satisfy the following constraint.

$$\begin{array}{|l} \mathit{contact\_nc} : \mathit{NodeConstraint} \\ \hline \forall x : \mathit{TERM} \bullet \\ \quad \mathit{contact\_nc}(x) = \\ \qquad \mathit{is\_a\_person}(x) \cap \\ \qquad \mathit{has\_one\_name}(x) \end{array}$$

- A contact is a person and has one name.

The Alice graph satisfies this constraint at the *alice*, *bob*, and *charlie* nodes.

$$\begin{array}{l} \vdash \mathit{alice\_graph} \in \mathit{contact\_nc}(\mathit{alice}) \land \\ \mathit{alice\_graph} \in \mathit{contact\_nc}(\mathit{bob}) \land \\ \mathit{alice\_graph} \in \mathit{contact\_nc}(\mathit{charlie}) \end{array}$$

An associate node must satisfy the following constraint.

$$\begin{array}{|l} \mathit{associate\_nc} : \mathit{NodeConstraint} \\ \hline \forall x : \mathit{TERM} \bullet \\ \quad \mathit{associate\_nc}(x) = \\ \qquad \mathit{is\_a\_person}(x) \cap \\ \qquad \mathit{has\_one\_name}(x) \cap \\ \qquad \mathit{is\_known\_by\_one}(x) \end{array}$$

- An associate is a person, has one name, and is known by one node.

The Alice graph satisfies this constraint at the *bob* and *charlie* nodes.

$$\begin{array}{l} \vdash \mathit{alice\_graph} \in \mathit{associate\_nc}(\mathit{bob}) \land \\ \mathit{alice\_graph} \in \mathit{associate\_nc}(\mathit{charlie}) \end{array}$$

6 Shapes

In general, a shape is any description of the expected contents of a graph. In this article we deal only with shapes that describe graphs using the following structure. A shape is a structure that defines how to associate a set of node constraints with each node of a data graph in two steps.

1. Label each node of the graph with a set of node constraint names using a set of neighbour functions.

2. Map each name to a node constraint.

These steps are described in detail below.

Note that this definition of shape is very prescriptive about the labelling process but is completely independent of the details of both the neighbour functions and the node constraints. We speculate that the labelling process can be used to handle the recursive aspects of a wide variety of shape languages that differ only in their expressiveness for defining neighbour functions and node constraints. For example, Resource Shape 2.0 uses forward and backward path expressions as neighbour functions and has a small, fixed set of simple node constraints. SHACL-SPARQL allows node constraints to be expressed by arbitrary SPARQL 1.1 queries, but does not allow explicit recursion.

6.1 Labelling Data Graph Nodes with Constraint Names

A shape contains a set of named constraints. A constraint may refer to other constraints by name. This means that a constraint may refer directly or indirectly to itself, in which case the constraint is recursive.

Shapes themselves may be represented as RDF graphs, so it is tempting to use IRIs to name node constraints. However, we introduce a new given set of names to emphasize that this set is logically independent of how we represent shapes.

$$[NAME]$$

For example, there are two distinct kinds of node in the PIM application, namely contact and associate.

$$\begin{array}{|l}
contact, associate : NAME \\
\hline
contact \neq associate
\end{array}$$

- *contact* and *associate* are distinct names.

Since graphs appear in several roles, there is scope for confusion. To clarify its role, the graph to which constraints are being applied will be referred to as the *data graph*.

The part of a shape that defines how data graph nodes are labelled is a *neighbour graph*. A neighbour graph is a directed, labelled, multigraph whose nodes are names and whose arcs are labelled by neighbour functions.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NeighbourGraph}} \\ \hline
\mathit{names} : \mathbb{F}\, \mathit{NAME} \\
\mathit{arcs} : \mathbb{F}(\mathit{NAME} \times \mathit{Neighbour} \times \mathit{NAME}) \\ \hline
\mathit{arcs} \subseteq \mathit{names} \times \mathit{Neighbour} \times \mathit{names} \\ \hline
\end{array}$$

- The nodes are names and the arcs are labelled by neighbour functions.

For example, in the PIM application, contacts are related to associates by the *knows* forward path expression, and associates are related to contacts by the *is_known_by* backward path expression.

$$\begin{array}{|l}
\mathit{pim\_ng} : \mathit{NeighbourGraph} \\ \hline
\mathit{pim\_ng.names} = \{\mathit{contact}, \mathit{associate}\} \\[1ex]
\mathit{pim\_ng.arcs} = \\
\qquad \{(\mathit{contact}, \mathit{knows}, \mathit{associate}), \\
\qquad (\mathit{associate}, \mathit{is\_known\_by}, \mathit{contact})\}
\end{array}$$

A *pointed neighbour graph* consists of a neighbour graph and a *base name* in the graph.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PointedNeighbourGraph}} \\ \hline
\mathit{NeighbourGraph} \\
\mathit{baseName} : \mathit{NAME} \\ \hline
\mathit{baseName} \in \mathit{names} \\ \hline
\end{array}$$

- The base name belongs to the graph.

The base name of a pointed neighbour graph is also referred to as the *start name* or *focus name*, depending on the context.

For example, *contact* is the natural base name in the PIM application.

$$\begin{array}{|l}
\mathit{pim\_png} : \mathit{PointedNeighbourGraph} \\ \hline
\mathit{pim\_png.names} = \mathit{pim\_ng.names} \\[1ex]
\mathit{pim\_png.arcs} = \mathit{pim\_ng.arcs} \\[1ex]
\mathit{pim\_png.baseName} = \mathit{contact}
\end{array}$$

A *named node* is pair of the form (x, a) where x is a data graph node and a is a node constraint name.

$$\mathit{NamedNode} == \mathit{TERM} \times \mathit{NAME}$$

For example, $(\mathit{alice}, \mathit{contact})$ is named node.

$$\vdash (\mathit{alice}, \mathit{contact}) \in \mathit{NamedNode}$$

A data graph g and a neighbour graph ng define a *requires* binary relation $requires(g, ng)$ on the set of named nodes. The meaning of this relation is that if $(x, a) \mathrel{\underline{requires}} (y, b)$ then whenever x must satisfy the constraints named by a then y must satisfy the constraints named by b.

$$
\begin{array}{|l}
requires : Graph \times NeighbourGraph \longrightarrow (NamedNode \leftrightarrow NamedNode) \\
\hline
\forall\, g : Graph;\ ng : NeighbourGraph \bullet \\
\quad requires(g, ng) = \\
\qquad \{\, x, y : nodes(g);\ a, b : NAME;\ q : Neighbour \mid \\
\qquad\quad (a, q, b) \in ng.arcs \land \\
\qquad\quad (x, y) \in q(g) \bullet \\
\qquad\qquad (x, a) \mapsto (y, b) \,\}
\end{array}
$$

- The named node (x, a) requires (y, b) when the neighbour graph includes an arc (a, q, b) and the node y can be reached from x by matching the neighbour function q in g.

For example, the requires relation for the Alice graph in the PIM application is as follows.

$$
\begin{array}{l}
\vdash requires(alice\_graph, pim\_ng) = \\
\qquad \{(alice, contact) \mapsto (bob, associate), \\
\qquad (alice, contact) \mapsto (charlie, associate), \\
\qquad (bob, associate) \mapsto (alice, contact), \\
\qquad (charlie, associate) \mapsto (alice, contact)\}
\end{array}
$$

A *labelled graph* is a data graph whose nodes are each labelled by a, possibly empty, set of names.

$$
\begin{array}{|l}
\hline
LabelledGraph \\
graph : Graph \\
names : \mathbb{F}\, NAME \\
label : TERM \mathrel{\mkern5mu\mapstochar\mkern-5mu\rightarrow} \mathbb{F}\, NAME \\
\hline
label \in nodes(graph) \longrightarrow \mathbb{P}\, names \\
\hline
\end{array}
$$

- Each node in the graph is labelled by a set of names.

For example, the following is a labelled graph based on the Alice graph.

$$
\begin{array}{|l}
alice\_lg : LabelledGraph \\
\hline
alice\_lg.graph = alice\_graph \\
alice\_lg.names = \{contact, associate\} \\
alice\_lg.label = \\
\quad \{alice \mapsto \{contact\}, \\
\quad bob \mapsto \{associate\}, \\
\quad charlie \mapsto \{associate\}, \\
\quad Alice \mapsto \varnothing, \\
\quad Bob \mapsto \varnothing, \\
\quad Charlie \mapsto \varnothing, \\
\quad foaf\_Person \mapsto \varnothing\}
\end{array}
$$

A pointed graph and a pointed neighbour graph determine a unique labelled graph. Intuitively, the labelling process starts by labelling the base node with the base name. Next, the neighbour graph is checked for arcs that begin at *baseName*, e.g. $(baseName, q, b)$. For each such arc compute the set $values(g, q, baseNode)$ and for each node y in this set, label y with b. Now repeat these steps taking y as the new base node and b as the new base name, but only do this once for each named node (y, b). Since there are a finite number of nodes and a finite number of names, this process always terminates.

$$
\begin{array}{|l}
\textit{LabelGraph} \\
\hline
PointedGraph \\
PointedNeighbourGraph \\
LabelledGraph \\
\hline
\textbf{let } ng == \theta NeighbourGraph \bullet \\
\quad \textbf{let } R == (requires(graph, ng))^* \bullet \\
\quad\quad label = (\lambda\, y : nodes(graph) \bullet \\
\quad\quad\quad \{\, b : names \mid (baseNode, baseName)\ \underline{R}\ (y, b)\,\})
\end{array}
$$

- The label of a node y is the set of names b such that the named node (y, b) is related to the base named node $(baseNode, baseName)$ by R the reflexive-transitive closure of the requires relation $requires(graph, shape)$.
- Note that this labelling process makes use of R, the reflexive-transitive closure of the requires relation. The use of R avoids difficulties associated with explicitly recursive definitions. We have, in effect, eliminated explicit recursion by computing a transitive closure of a finite binary relation.
- Note that the components of *LabelledGraph* are uniquely determined by the components of *PointedGraph* and *PointedNeighbourGraph*.

For example, the pointed graph *alice_pg* and the pointed neighbour graph

pim_png uniquely determine the labelled graph $alice\_lg$.

$$\begin{array}{l}
\forall\, LabelGraph \mid \\
\quad \theta PointedGraph = alice\_pg \land \\
\quad \theta PointedNeighbourGraph = pim\_png \bullet \\
\qquad \theta LabelledGraph = alice\_lg
\end{array}$$

6.2 Mapping Constraint Names to Node Constraints

The association of node constraints to graph nodes is given by a mapping.

$$\begin{array}{|l}
\multicolumn{1}{l}{NodeConstraints} \\
\hline
names : \mathbb{F}\, NAME \\
constraint : NAME \mathrel{+\!\!\!\to} NodeConstraint \\
\hline
\mathrm{dom}\, constraint = names \\
\hline
\end{array}$$

- Each name maps to a node constraint.

For example, the PIM application associates the following node constraints with names.

$$\begin{array}{|l}
pim\_ncs : NodeConstraints \\
\hline
pim\_ncs.names = \{contact, associate\} \\
pim\_ncs.constraint = \\
\quad \{contact \mapsto contact\_nc, \\
\quad associate \mapsto associate\_nc\}
\end{array}$$

- The PIM application has two kinds of nodes, named *contact* and *associate*.
- *contact* nodes must satisfy the *contact_nc* node constraint.
- *associate* nodes must satisfy the *associate_nc* node constraint.

A *constrained graph* is an assignment of a, possibly empty, set of node constraints to each node of the graph.

$$\begin{array}{|l}
\multicolumn{1}{l}{ConstrainedGraph} \\
\hline
graph : Graph \\
constraints : TERM \mathrel{+\!\!\!\to} \mathbb{F}\, NodeConstraint \\
\hline
\mathrm{dom}\, constraints = nodes(graph) \\
\hline
\end{array}$$

- Each node of the data graph has a set of node constraints.

For example, the PIM application enforces the following constraints on the Alice graph.

$$\begin{array}{|l}
alice\_cg : ConstrainedGraph \\
\hline
alice\_cg.graph = alice\_graph \\
\\
alice\_cg.constraints = \\
\qquad \{alice \mapsto \{contact\_nc\}, \\
\qquad bob \mapsto \{associate\_nc\}, \\
\qquad charlie \mapsto \{associate\_nc\}, \\
\qquad Alice \mapsto \varnothing, \\
\qquad Bob \mapsto \varnothing, \\
\qquad Charlie \mapsto \varnothing, \\
\qquad foaf\_Person \mapsto \varnothing\}
\end{array}$$

- *alice* must satisfy the contact node constraint.
- *bob* and *charlie* must satisfy the associate node constraint.
- There are no node constraints on the remaining nodes.

A constrained graph is *valid* if it satisfies all the constraints at each node.

$$\begin{array}{|l}
\hline
ValidGraph \\
ConstrainedGraph \\
\hline
\forall x : nodes(graph) \bullet \\
\qquad \forall c : constraints(x) \bullet \\
\qquad\qquad graph \in c(x) \\
\hline
\end{array}$$

- A valid data graph satisfies each node constraint at each node.

For example, the constrained graph *alice_cg* is valid.

$$\vdash alice\_cg \in ValidGraph$$

A mapping from nodes to names (*LabelledGraph*) and a mapping from names to node constraints (*NodeConstraints*) uniquely determines a mapping from nodes to node constraints (*ConstrainedGraph*).

$$\begin{array}{|l}
\hline
ConstrainGraph \\
LabelledGraph \\
NodeConstraints \\
ConstrainedGraph \\
\hline
\forall x : nodes(graph) \bullet \\
\qquad constraints(x) = \\
\qquad\qquad \{\, a : label(x) \bullet constraint(a) \,\} \\
\hline
\end{array}$$

- The set of node constraints at each node x of a labelled data graph is equal to the to the set node constraints named by the labels a at x.

- Note that the components of *ConstrainedGraph* are uniquely determined by the components of *LabelledGraph* and *NodeConstraints*.

For example, the Alice labelled graph *alice_lg* and the PIM node constraints *pim_ncs* uniquely determine the Alice constrained graph *alice_cg*.

$$\begin{array}{l}
\vdash \forall\, ConstrainGraph \mid \\
\quad \theta LabelledGraph = alice\_lg \land \\
\quad \theta NodeConstraints = pim\_ncs \bullet \\
\qquad \theta ConstrainedGraph = alice\_cg
\end{array}$$

6.3 Shapes as Constraints

A *shape* consists of a neighbour graph and node constraints.

$$\begin{array}{|l}
\hline
Shape \\
NeighbourGraph \\
NodeConstraints \\
\hline
\end{array}$$

For example, the neighbour graph *pim_ng* and the node constraints *pim_nc* define a shape for the PIM application.

$$\begin{array}{|l}
pim\_shape : Shape \\
\hline
pim\_shape.names = \{contact, associate\} \\
pim\_shape.arcs = pim\_ng.arcs \\
pim\_shape.constraint = pim\_ncs.constraint
\end{array}$$

A *pointed shape* consists of a pointed neighbour graph and node constraints.

$$\begin{array}{|l}
\hline
PointedShape \\
PointedNeighbourGraph \\
NodeConstraints \\
\hline
\end{array}$$

For example, the pointed neighbour graph *pim_png*, which has base name *contact*, and the node constraints *pim_ncs* define a pointed shape for the PIM application.

$$\begin{array}{|l}
pim\_ps : PointedShape \\
\hline
pim\_ps.names = \{contact, associate\} \\
pim\_ps.baseName = contact \\
pim\_ps.arcs = pim\_ng.arcs \\
pim\_ps.constraint = pim\_ncs.constraint
\end{array}$$

A pointed data graph satisfies a pointed shape if the constrained graph produced by the composition of the labelling and constraining processes is valid.

$$\begin{array}{|l}
\hline \textit{SatisfiesShape} \\
\textit{PointedShape} \\
\textit{PointedGraph} \\
\textit{LabelGraph} \\
\textit{ConstrainGraph} \\
\hline
\textit{ValidGraph} \\
\hline
\end{array}$$

- The constrained graph that results from the labelling and constraining processes must be valid.

- Note that the components of *LabelGraph* and *ConstrainGraph* are uniquely determined by the components of *PointedShape* and *PointedGraph*. The validity condition (*ValidGraph*) therefore determines a relation between *PointedShape* and *PointedGraph*. The pointed graph is said to *satisfy* the pointed shape.

For example, the pointed Alice graph satisfies the pointed PIM shape.

$$\begin{array}{l}
\vdash \exists_1 \textit{SatisfiesShape} \bullet \\
\qquad \theta \textit{PointedShape} = \textit{pim\_ps} \land \\
\qquad \theta \textit{PointedGraph} = \textit{alice\_pg}
\end{array}$$

A pointed shape determines a node constraint.

$$\begin{array}{|l}
\textit{shapeConstraint} : \textit{PointedShape} \longrightarrow \textit{NodeConstraint} \\
\hline
\forall \textit{ps} : \textit{PointedShape}; x : \textit{TERM} \bullet \\
\qquad \textit{shapeConstraint}(\textit{ps})(x) = \\
\qquad\qquad \{ \textit{SatisfiesShape} \mid \textit{ps} = \theta \textit{PointedShape} \land x = \textit{baseNode} \bullet \textit{graph} \}
\end{array}$$

- Given a pointed shape *ps*, *shapeConstraint*(*ps*) is a node constraint. Given a node x, *shapeConstraint*(*ps*)(x) is the set of all graphs *graph* such that the pointed graph formed by using x as the base node satisfies *ps*.

For example, the pointed PIM shape defines a node constraint.

$$\begin{array}{|l}
\textit{pim\_nc} : \textit{NodeConstraint} \\
\hline
\textit{pim\_nc} = \textit{shapeConstraint}(\textit{pim\_ps})
\end{array}$$

The *alice* node of the Alice graph satisfies this node constraint.

$$\vdash \textit{alice\_graph} \in \textit{pim\_nc}(\textit{alice})$$

7 Relation to Shape Languages

This section discusses how the preceding formalism relates to Resource Shape 2.0, ShEx, and SHACL.

7.1 Relation to Resource Shape 2.0

Resource Shape 2.0 provides a small vocabulary for defining simple, commonly occurring node constraints such as property occurrence, range, and allowed values. These are uncontentious and will not be discussed further.

As mentioned above, Resource Shape 2.0 also allows recursive shapes via the property `oslc:valueShape`. The preceding formalism was motivated by Resource Shape 2.0 and, not surprisingly, provides a precise description of the meaning of recursive shapes in that language. The following listings illustrate the use of `oslc:valueShape` for the running example.

Listing 3 contains the resource shape for contacts.

```
# http://example.org/shapes/contact
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix oslc: <http://open-services.net/ns/core#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@base <http://example.org/shapes/> .

<contact> a oslc:ResourceShape ;
    oslc:describes foaf:Person ;
    oslc:property
        <contact#name> ,
        <contact#knows> .

<contact#name> a oslc:Property ;
    oslc:name "name" ;
    oslc:occurs oslc:Exactly-one ;
    oslc:propertyDefinition foaf:name ;
    oslc:valueType xsd:string .

<contact#knows> a oslc:Property ;
    oslc:name "knows" ;
    oslc:occurs oslc:Zero-or-many ;
    oslc:propertyDefinition foaf:knows ;
    oslc:range foaf:Person ;
    oslc:valueShape <associate> .
```

Listing 3: Resource shape for contacts

Listing 4 contains the resource shape for associates.

```
# http://example.org/shapes/associate
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix oslc: <http://open-services.net/ns/core#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@base <http://example.org/shapes/> .

<associate> a oslc:ResourceShape ;
    oslc:describes foaf:Person ;
    oslc:property
        <associate#name> ,
        <associate#isKnownBy> .

<associate#name> a oslc:Property ;
    oslc:name "name" ;
    oslc:occurs oslc:Exactly-one ;
    oslc:propertyDefinition foaf:name ;
    oslc:valueType xsd:string .

<associate#isKnownBy> a oslc:Property ;
    oslc:name "isKnownBy" ;
    oslc:occurs oslc:Exactly-one ;
    oslc:propertyDefinition foaf:knows ;
    oslc:isInverseProperty true ;
    oslc:range foaf:Person ;
    oslc:valueShape <contact> .
```

Listing 4: Resource shape for associates

This example illustrates recursive shapes since the contact shape refers to the associate shape and the associate shape refers to the contact shape. This apparent circularity would cause difficulty if contact and associate were each described as a constraint. However, using the preceding formalism, the composite shape consisting of both the contact and associate resource shapes is given a well-defined meaning.

In Resource Shape 2.0, neighbour functions are limited to forward and backward path expressions. In fact, backward path expressions were missing from the original OSLC Resource Shape language and are a proposed extension in Resource Shape 2.0. The proposed syntax for backward path expressions uses the optional property `oslc:isInverseProperty` but this design could be improved to provide better compatibility with downlevel clients, i.e. a downlevel client might silently ignore this new property and produce incorrect results.

The following SPARQL query extracts the neighbour graph arcs from a set of resource shapes. Each binding of `(?a ?direction ?p ?b)` corresponds to an arc (a, q, b) where $q = \mathit{forward}(p)$ or $q = \mathit{backward}(p)$.

```
prefix oslc: <http://open-services.net/ns/core#>

select distinct ?a ?direction ?p ?b
where {
    ?a a oslc:ResourceShape ;
        oslc:property ?prop .
    ?prop a oslc:Property ;
        oslc:propertyDefinition ?p ;
        oslc:valueShape ?b .
    optional {?prop oslc:isInverseProperty ?inverse}
    bind (if(bound(?inverse) && ?inverse,
        'backward', 'forward') as ?direction)
}
```

Listing 5: Query for neighbour graph arcs

The result of running this query on the PIM resource shapes is given in Table 3.

a	direction	p	b
`<contact>`	`"forward"`	`foaf:knows`	`<associate>`
`<associate>`	`"backward"`	`foaf:knows`	`<contact>`

Table 3: Result of query on PIM shape

The query correctly extracts the neighbour graph *pim_ng* of the PIM shape.

7.2 Relation to ShEx

One major difference between Resource Shape 2.0 and ShEx is that ShEx allows the definition of much richer node constraints using regular expressions, disjunction, and other operations. ShEx also allows recursion via reference to named shapes, e.g. `@<UserShape>`, which is referred to as the `ValueReference` feature.

Most features of Resource Shape 2.0 can be expressed in ShEx. The intersection of Resource Shape 2.0 and ShEx certainly includes the recursive aspects of Resource Shape 2.0 as expressed by `oslc:valueShape`. Therefore the preceding formalism applies to ShEx provided that `ValueReference` is used in a way that maps directly to Resource Shape 2.0.

However, ShEx allows a more permissive use `ValueReference`. For example, a `ValueReference` may appear inside `GroupRule` with cardinalities. It is not clear that this usage can be expressed using suitably defined neighbour functions.

7.3 Relation to SHACL

The W3C Data Shapes Working Groups is currently developing the SHACL specification. At the time of writing, there are three competing proposals. One proposal, influenced by SPIN, appears to treat recursion similarly to Resource Shape 2.0. A second proposal is a further development of ShEx. The third proposal, SHACL-SPARQL, takes a different approach by avoiding recursion entirely.

The ability to name and refer to shapes allows for more intuitive descriptions of the constraints on graphs. The absence of recursion in SHACL-SPARQL therefore detracts from its usefulness. However, since the formalism presented here gives a clear and unobjectionable meaning to a limited form of recursion, this capability could be added to SHACL-SPARQL.

8 Conclusion

The formalism presented here gives a precise meaning to recursive shapes as defined in Resource Shape 2.0.

This formalism is applicable to a subset of ShEx in which recursion is suitably limited. More analysis is required in order to determine if the unlimited form of recursion allowed in ShEx and its SHACL follow-on can be described using suitable neighbour functions, or if some new concept is required.

Finally, the limited form of recursion presented here could be added to the SHACL-SPARQL proposal to enhance its expressiveness.

Acknowledgements

Peter Patel-Schneider reviewed `v1` of this article and provided valuable comments and criticisms. Holger Knublauch suggested the use of the Pollentini example as a better illustration of recursion.

Change History

- `v1` May 19, 2015. Initial version.
- `v2` June 17, 2015. Editorial improvements. Added Pollentini example.

References

[1] Beckett, D., Berners-Lee, T., Prud’hommeaux, E., and Carothers, G. RDF 1.1 Turtle. W3C Recommendation, World Wide Web Consortium, Feb. 2014. `http://www.w3.org/TR/2014/REC-turtle-20140225/`.

[2] Berners-Lee, T. Linked data. web page, World Wide Web Consortium, June 2009. `http://www.w3.org/DesignIssues/LinkedData.html`.

[3] Brickely, D., and Guha, R. RDF Schema 1.1. W3C Recommendation, World Wide Web Consortium, Feb. 2014. `http://www.w3.org/TR/2014/REC-rdf-schema-20140225/`.

[4] Brickley, D., and Miller, L. FOAF Vocabulary Specification 0.99. web page, The FOAF Project, Jan. 2014. `http://xmlns.com/foaf/spec/20140114.html`.

[5] Cyganiak, R., Wood, D., and Lanthaler, M. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, World Wide Web Consortium, Feb. 2014. `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`.

[6] Gao, S. S., Sperberg-McQueen, C. M., and Thompson, H. S. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C Recommendation, World Wide Web Consortium, Apr. 2012. `http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`.

[7] Harris, S., and Seaborne, A. SPARQL 1.1 Query Language. W3C Recommendation, World Wide Web Consortium, Mar. 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[8] Le Hors, A. Rdf data shapes working group. wiki page, World Wide Web Consortium, Sept. 2014. https://www.w3.org/2014/data-shapes/wiki/Main_Page.

[9] Motik, B., Patel-Schneider, P. F., and Parsia, B. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C Recommendation, World Wide Web Consortium, Dec. 2012. http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[10] Patel-Schneider, P. A SHACL Specification based on SPARQL. wiki page, World Wide Web Consortium, 2015. https://www.w3.org/2014/data-shapes/wiki/Shacl-sparql.

[11] Patel-Schneider, P. Re: Recursion in RDF Data Shape Languages. mailing list, World Wide Web Consortium, 2015. https://lists.w3.org/Archives/Public/public-data-shapes-wg/2015May/0111.html.

[12] Ryman, A. Resource Shape 2.0. W3C Member Submission, World Wide Web Consortium, Feb. 2014. http://www.w3.org/Submission/2014/SUBM-shapes-20140211/.

[13] Solbrig, H., and Prud'hommeaux, E. Shape Expressions 1.0 Definition. W3C Member Submission, World Wide Web Consortium, June 2014. http://www.w3.org/Submission/2014/SUBM-shex-defn-20140602/.

[14] Spivey, M. *The Z Notation: a reference manual.* Prentice Hall, 2001. https://spivey.oriel.ox.ac.uk/mike/zrm/index.html.

[15] Spivey, M. The fuzz type-checker for Z. web page, Oxford University, 2008. https://spivey.oriel.ox.ac.uk/mike/fuzz/.