

A Formal Model of the Shape Expression Language

Harold Solbrig¹, Eric Prud'hommeaux²

¹ Mayo Clinic, Rochester MN 55095, USA,
solbrig.harold@mayo.edu

² World Wide Web Consortium, Cambridge MA 02139, USA,
eric@w3.org

Abstract. Shape Expressions express formal constraints on the content of RDF graphs and are intended to be used to validate RDF documents, communicate expected graph patterns for interfaces and to generate forms and validation code. This document describes the formal semantics of the Shape Expressions language through the use of the *Z* Specification Language.

Keywords: RDF, RDF Graph, RDF Dataset, validation, formal schema, ShEx, RDF Validation, Z Specification Language

1 Introduction

The Shape Expressions Language (*ShEx*) is used to specify formal constraints on the content of RDF graphs and are intended to be used to validate RDF documents, communicate expected graph patterns for interfaces and to generate forms and validation code. *ShEx* can be used to:

- Describe the contents of an RDF graph
- Express invariants about an RDF triple store
- Define a predicate that can be tested against an RDF graph instance
- Define a set of rules that can be used to generate forms, validation code and other constructs in specific target languages

Information about the use, grammar and syntax of *ShEx* can be found at <http://www.w3.org/2013/ShEx>. The purpose of this document is to describe the formal *semantics* of the *ShEx* language using the *Z* specification language. We begin with a formal specification of the characteristics of an *RDF Graph* that are referenced by *ShEx*.

2 The RDF Data Model

Starting with the definitions in *RDF 1.1 Concepts and Abstract Syntax*[1]:

“An **RDF graph** is a set of **RDF Triples**”

Formally:

$Graph == \mathbb{P} Triple$

- “An **RDF triple** consists of three components:
- the *subject*, which is an **IRI** or a **blank node**
 - the *predicate*, which is an **IRI**
 - the *object*, which is an **IRI**, a **literal** or a **blank node**”

“... **IRIs**, **literals** and **blank nodes** are distinct and distinguishable.”

The *ShEx* language treats **IRIs** and **blank nodes** as primitive types, which are reflected as the *Z* basic types:

$[IRI, BlankNode]$

The *ShEx* language can express constraints on both the type and content of RDF literals:

- “A **literal** in an RDF graph consists of two or three elements:
- a **lexical form**, being a Unicode string...
 - a **datatype IRI**, being an IRI
 - if and only if the datatype IRI is `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`, a non-empty language tag as defined in [BCP47][3]. The language tag MUST be well-formed according to section 2.2.9 of [BCP47][3].”

This can be modelled by *String* and *LanguageTag* as basic types:

$[String, LanguageTag]$

and using them in the definition the two flavours of *RDFLiteral*, plain literal and typed literal:

$TypedLiteral \hat{=} [lexicalForm : String; dataType : IRI \mid dataType \neq XSD_String]$
 $PlainLiteral \hat{=} [lexicalForm : String; dataType : IRI; langTag : LanguageTag \mid dataType = XSD_String]$
 $RDFLiteral ::= pl\langle\langle PlainLiteral \rangle\rangle \mid tl\langle\langle TypedLiteral \rangle\rangle$

RDFTerm is defined as:

“**IRIs**, **literals** and **blank nodes** are collectively known as **RDF terms**

$RDFTerm ::= iri\langle\langle IRI \rangle\rangle \mid literal\langle\langle RDFLiteral \rangle\rangle \mid bnode\langle\langle BlankNode \rangle\rangle$

The definition of RDF *Triple* is modelled as a tuple consisting of three constrained *RDFTerms*:

<i>Triple</i>
$s, p, o : RDFTerm$
$iri \cdot s \in IRI \vee bnode \cdot s \in BlankNode$
$iri \cdot p \in IRI$
$iri \cdot o \in IRI \vee bnode \cdot o \in BlankNode \vee literal \cdot o \in RDFLiteral$

2.1 RDF Access Functions

The *ShEx* language requires the following basic functions:

triplesForSubject – return set of triples in a graph triples whose subject is a given *RDFTerm*

$triplesForSubject : RDFTerm \rightarrow Graph \rightarrow Graph$
$\forall subj : RDFTerm; g : Graph \bullet triplesForSubject\ subj\ g = \{t : g \mid t.s = subj\}$

triplesForObject – return set of triples in a graph triples whose object is a given *RDFTerm*

$triplesForObject : RDFTerm \rightarrow Graph \rightarrow Graph$
$\forall obj : RDFTerm; g : Graph \bullet triplesForObject\ obj\ g = \{t : g \mid t.o = obj\}$

2.2 Well Known URIs

The following URI's are referenced explicitly in the *ShEx* implementation:

| *XSD_String*, *RDF_Resource* : *IRI*

This completes the formal definition of *Graph*, *Triple*, *RDFTerm* and their components, which we can now use to describe the relationship between an *ShEx Schema* and an RDF graph.

3 Shape Expression Evaluation

A Shape Expression *Schema* is a collection of labelled shapes where exactly one shape in the collection is identified as the outermost or “starting” shape. In addition, any shape that is referenced within the *Schema* is also itself a member of the *Schema* Formally:

<i>Schema</i>
$shapes : Label \leftrightarrow Rule$
$start : Label$
$start \in \text{dom } shapes$
$\forall r : \text{ran } shapes \bullet$
$(r \in \text{ran } group \Rightarrow (group \cdot r).rule \in \text{dom } shapes) \wedge$
$(r \in \text{ran } and \Rightarrow \text{ran}(and \cdot r) \subseteq \text{dom } shapes) \wedge$
$(r \in \text{ran } xor \Rightarrow \text{ran}(xor \cdot r) \subseteq \text{dom } shapes) \wedge$
$(r \in \text{ran } arc \wedge (arc \cdot r).o \in \text{ran } osShape \Rightarrow$
$(osShape \cdot (arc \cdot r).o) \in \text{dom } shapes) \wedge$
$(r \in \text{ran } rarc \wedge (rarc \cdot r).s \in \text{ran } ssShape \Rightarrow$
$(ssShape \cdot (rarc \cdot r).s) \in \text{dom } shapes)$

While *ShEx* implementations define a rule *Label* as being either an *IRI* or a *BlankNode*, the type of *Label* does not impact the evaluation semantics, allowing it to be treated as a basic type:

[*Label*]

The validity of a given RDF *Graph* is determined by taking the *start Label* of a *ShEx Schema* and a reference *IRI* and evaluating the validity of the corresponding *shape* against the supplied graph.

Formally, the *evaluate* function takes a *Schema*, a *Graph* and a reference *IRI* and, if the *Rule* associated with the *start Label* in the *Schema*, in the context of the starting *Schema* and *graph*, returns either *nomatch* (**z**) or *pass* (**p**) then the function returns *pass*. In all other cases, the function returns *fail* (**f**).

$evaluate : Schema \rightarrow Graph \rightarrow IRI \rightarrow OptValidity$
$\forall s : Schema; g : Graph; i : IRI; v : OptValidity; ec : EvalContext \mid$ $ec.graph = g \wedge ec.schema = s \bullet$ $evaluate\ s\ g\ i =$ if $evalRule\ ec\ (iri\ i)\ (s.shapes\ s.start) \in \{z, p\}$ then p else f

4 Rule Evaluation

A *ShEx Rule* is a set of constraints that can be evaluated against a reference *RDFTerm* in the context of a given *Schema* and RDF *Graph*:

$EvalContext \hat{=} [schema : Schema; graph : Graph]$

Formally, the *evalRule* function takes an *EvalContext*, a reference *RDFTerm* and a *Rule* and returns one of the following:

- **Pass** (**p**) - the supplied *Graph* satisfied the evaluation *Rule*
- **Fail** (**f**) - the supplied *Graph* did not satisfy the evaluation *Rule*
- **Nomatch** (**z**) - an *optional GroupRule* was encountered and there were no matching triples
- **Dunno** (\emptyset) - an *ArcRule* was encountered with a minimum cardinality of 0 and there were no matching triples
- **Error**(ε) - an *XorRule* was evaluated and two or more components passed the evaluation.

$OptValidity ::= p \mid f \mid z \mid \emptyset \mid \varepsilon$

A *Rule* can take one of five forms. Each will be formally described later in this document, but informally they are:

- **ArcRule** – selects the subset of the graph having the reference *RDFTerm* as the subject and matching predicates and and validates the resulting objects
- **RevArcRule** – selects the subset of the graph having the reference *RDFTerm* as the object and matching predicates and and validates the resulting subjects
- **GroupRule** – identifies a *Rule* and declares it as optional and/or describes a set of external *Actions* to be evaluated if the inner *Rule* passes.
- **AndRule** – identifies a set of *Rules*, all of which must pass when evaluated against the supplied *Graph* and *EvalContext*
- **XorRule** – identifies a set of *Rules*, exactly one of which must pass when evaluated against the supplied *Graph* and *EvalContext*

$$\begin{aligned}
\text{Rule} ::= & \text{arc}\langle\langle \text{ArcRule} \rangle\rangle \mid \\
& \text{rarc}\langle\langle \text{RevArcRule} \rangle\rangle \mid \\
& \text{group}\langle\langle \text{GroupRule} \rangle\rangle \mid \\
& \text{and}\langle\langle \text{AndRule} \rangle\rangle \mid \\
& \text{xor}\langle\langle \text{XorRule} \rangle\rangle
\end{aligned}$$

$ \begin{aligned} & \text{evalRule} : \text{EvalContext} \rightarrow \text{RDFTerm} \rightarrow \text{Rule} \rightarrow \text{OptValidity} \\ & \forall ec : \text{EvalContext}; i : \text{RDFTerm}; r : \text{Rule} \bullet \text{evalRule } ec \ i \ r = \\ & \quad \text{if } r \in \text{ran arc} \\ & \quad \quad \text{then evalArcRule } ec \ i \ (arc \cdot r) \\ & \quad \text{else if } r \in \text{ran rarc} \\ & \quad \quad \text{then evalRevArcRule } ec \ i \ (rarc \cdot r) \\ & \quad \text{else if } r \in \text{ran group} \\ & \quad \quad \text{then evalGroupRule } ec \ i \ (group \cdot r) \\ & \quad \text{else if } r \in \text{ran and} \\ & \quad \quad \text{then evalAndRule } ec \ i \ (and \cdot r) \\ & \quad \text{else} \\ & \quad \quad \text{evalXorRule } ec \ i \ (xor \cdot r) \end{aligned} $

The *evalRule'* function de-references the supplied *Label* and invokes *evalRule* with the result. This is not explicitly represented because the *Z* specification language does not allow cyclic dependencies. This function is undefined if *Label* is not in *EvalContext*

$ \begin{aligned} & \text{evalRule}' : \text{EvalContext} \rightarrow \text{RDFTerm} \rightarrow \text{Label} \rightarrow \text{OptValidity} \\ & \forall ec : \text{EvalContext}; l : \text{Label} \bullet l \in \text{dom } ec.\text{schema.shapes} \end{aligned} $

4.1 ArcRule evaluation

The *ArcRule* is used to select the subset of the graph having a given predicate or predicates and to determine whether the cardinality and/or “type” of this

subset matches a supplied criteria. The rule itself consists of a *PredicateFilter* to select the triples, an *ObjectSpecification* to evaluate the result, a optional *min* and *max* cardinality and a (possibly empty) set of *Actions*:

$ \begin{array}{l} \textit{ArcRule} \\ \hline p : \textit{PredicateFilter} \\ o : \textit{ObjectSpecification} \\ min, max : \mathbb{N}[0 \dots 1] \\ actions : \mathbb{P} \textit{Action} \\ \hline (\#min = 1 \wedge \#max = 1) \Rightarrow value\ min \leq value\ max \end{array} $
--

ArcRule evaluation consists of:

1. Select the subset of the *EvalContext Graph* with the supplied subject and predicates matching *PredicateFilter*
2. Evaluate the cardinality and return the result if it doesn't pass
3. Evaluate the object of each of the triples in the set against *ObjectSpecification*. If any of the evaluations fail, return *fail* (*f*).
4. Return the result of evaluating *actions* against the matching triples.

$ \begin{array}{l} evalArcRule : EvalContext \rightarrow RDFTerm \rightarrow ArcRule \rightarrow OptValidity \\ \hline \forall ec : EvalContext; s : RDFTerm; ar : ArcRule; sg : Graph \mid \\ sg = evalPredicateFilter\ ar.p\ (triplesForSubject\ s\ ec.graph) \bullet \\ evalArcRule\ ec\ s\ ar = \\ \quad \textbf{if}\ evalCardinality\ sg\ ar.min\ ar.max \neq p \\ \quad \quad \textbf{then}\ evalCardinality\ sg\ ar.min\ ar.max \\ \quad \textbf{else if}\ evalObjectSpecification\ ec\ ar.o\ sg \neq p \\ \quad \quad \textbf{then}\ evalObjectSpecification\ ec\ ar.o\ sg \\ \quad \textbf{else} \\ \quad \quad dispatch\ ar.actions\ sg\ ec \end{array} $

***PredicateFilter* Validation** A *PredicateFilter* can be one of:

- *inclusion* - the IRI of a specific predicate or the *IRIstem* that defines a set of predicates
- *exclusion* - an expression that matches any predicate *except* those matching the (possibly empty) set of *IRIorStems*

$$\begin{array}{l}
IRIorStem ::= iosIRI \langle \langle IRI \rangle \rangle \mid stem \langle \langle IRIstem \rangle \rangle \\
PredicateFilter ::= inclusion \langle \langle IRIorStem \rangle \rangle \mid exclusion \langle \langle \mathbb{P} IRIorStem \rangle \rangle
\end{array}$$

An *IRIstem* matches any *IRI* whose stringified representation begins with the stringified representation of *IRIstem* according to standard IRI matching rules [2]. This is represented by the function:

$[IRIstem]$

$$IRIstemRange : IRIstem \rightarrow \mathbb{P} IRI$$

evalPredicateFilter returns all of the triples in a *Graph* whose predicate matches the supplied *PredicateFilter*:

$$evalPredicateFilter : PredicateFilter \rightarrow Graph \rightarrow Graph$$

$$\forall f : PredicateFilter; g : Graph \bullet evalPredicateFilter f g =$$

$$\quad \text{if } f \in \text{ran } inclusion \text{ then } evalIRIorStem (inclusion \cdot f) g$$

$$\quad \text{else } evalWild (exclusion \cdot f) g$$

evalIRIorStem returns all of the triples in a *Graph* matching the supplied *IRIorStem*

$$evalIRIorStem : IRIorStem \rightarrow Graph \rightarrow Graph$$

$$\forall e : IRIorStem; g : Graph \bullet evalIRIorStem e g =$$

$$\quad \text{if } e \in \text{ran } iosIRI \text{ then } \{t : g \mid iri \cdot t.p = iosIRI \cdot e\}$$

$$\quad \text{else } \{t : g \mid iri \cdot t.p \in IRIstemRange (stem \cdot e)\}$$

evalWild returns all of the triples in a *Graph* that do *not* match an entry in the set of *IRIorStems*.

$$evalWild : \mathbb{P} IRIorStem \rightarrow Graph \rightarrow Graph$$

$$\forall es : \mathbb{P} IRIorStem; g : Graph \bullet evalWild es g =$$

$$\quad \{t : g \mid t \notin \bigcup \{e : es \bullet evalIRIorStem e g\}\}$$

evalCardinality – evaluates the cardinality the supplied graph. Rules:

# elements	<i>min</i>	<i>max</i>	result
0	0		<i>z</i>
0			\emptyset
> 0 and < <i>min</i>	present		<i>f</i>
> 0 and > <i>max</i>		present	<i>f</i>
			<i>p</i>

```

evalCardinality : Graph  $\rightarrow \mathbb{N}[0 \dots 1] \rightarrow \mathbb{N}[0 \dots 1] \rightarrow \textit{OptValidity}$ 


---


 $\forall g : \textit{Graph}; \min, \max : \mathbb{N}[0 \dots 1] \bullet \textit{evalCardinality } g \min \max =$ 
  if  $\# \min = 1 \wedge \# g = 0 \wedge \textit{value } \min = 0$ 
    then  $z$ 
  else if  $\# g = 0$ 
    then  $\emptyset$ 
  else if  $(\# \min = 1 \wedge \# g < \textit{value } \min) \vee$ 
     $(\# \max = 1 \wedge \# g > \textit{value } \max)$ 
    then  $f$ 
  else  $p$ 

```

ObjectSpecification evaluation *ObjectSpecification* specifies a set of possible values for an *RDFTerm* and takes one of the following forms:

- *valueType* - matches *Literals* having a specified data type
- *osInclusion* - matches *IRIs* or *Literals* that match one or more of the expressions in the specified set
- *osExclusion* - matches any target *except* those matching the (possibly empty) set of *IRIstems*
- *osShape* - matches any target that is considered valid according the *Rule* identified by *Label*.

$$MatchValue ::= mviri \langle\langle IRI \rangle\rangle \mid mviris \langle\langle IR Istem \rangle\rangle \mid \\ mvlit \langle\langle RDFLiteral \rangle\rangle$$
$$\begin{aligned} \text{ObjectSpecification} ::= & \text{valueType}\langle\langle\text{IRI}\rangle\rangle \mid \\ & \text{osInclusion}\langle\langle\text{P Match Value}\rangle\rangle \mid \\ & \text{osExclusion}\langle\langle\text{P Match Value}\rangle\rangle \mid \\ & \text{osShape}\langle\langle\text{Label}\rangle\rangle \end{aligned}$$

evalObjectSpecification – returns *pass* (**p**) if all of the triples in a *Graph* match the supplied *ObjectSpecification*, otherwise *fail* (**f**)

$evalObjectSpecification : EvalContext \rightarrow ObjectSpecification \rightarrow Graph \rightarrow OptValidity$
$\forall ec : EvalContext; os : ObjectSpecification; g : Graph \bullet$ $evalObjectSpecification\ ec\ os\ g =$ $\quad \text{if } \forall t : g \bullet evalObjectSpecificationTriple\ ec\ os\ t.o = p \text{ then } p$ $\quad \text{else } f$

$\text{evalObjectSpecificationTriple} : \text{EvalContext} \rightarrow \text{ObjectSpecification} \rightarrow \text{RDFTerm} \rightarrow \text{OptValidity}$
$\forall ec : \text{EvalContext}; os : \text{ObjectSpecification}; n : \text{RDFTerm} \bullet$ $\text{evalObjectSpecificationTriple } ec \ os \ n =$ $\text{if } os \in \text{ran valueType} \text{ then}$ $\quad \text{evalValueType } (valueType \cdot os) \ n$ $\text{else if } os \in \text{ran osInclusion} \text{ then}$ $\quad \text{evalInclusion } (osInclusion \cdot os) \ n$ $\text{else if } os \in \text{ran osExclusion} \text{ then}$ $\quad \text{evalExclusion } (osExclusion \cdot os) \ n$ else $\quad \text{evalShape } ec \ (osShape \cdot os) \ n$

evalValueType – returns pass if the supplied *RDFTerm* is:

- type *literal* and whose *dataType* matches *ValueType*
- type *IRI* and *ValueType* is type *RDF_Literal*

$\text{evalValueType} : \text{IRI} \rightarrow \text{RDFTerm} \rightarrow \text{OptValidity}$
$\forall vt : \text{IRI}; n : \text{RDFTerm}; l : \text{RDFLiteral} \bullet \text{evalValueType } vt \ n =$ $\text{if } vt = \text{RDF_Resource} \wedge n \in \text{ran iri} \text{ then } p$ $\text{else if } n \in \text{ran literal} \wedge l = (literal \cdot n) \wedge$ $\quad ((l \in \text{ran pl} \wedge (pl \cdot l).dataType = vt) \vee$ $\quad (l \in \text{ran tl} \wedge (tl \cdot l).dataType = vt)) \text{ then } p$ $\text{else } f$

evalInclusion – return *p* if the supplied *RDFTerm* is a member of *MatchValue*

$\text{evalInclusion} : \mathbb{P} \text{ MatchValue} \rightarrow \text{RDFTerm} \rightarrow \text{OptValidity}$
$\forall mvs : \mathbb{P} \text{ MatchValue}; n : \text{RDFTerm} \bullet \text{evalInclusion } mvs \ n =$ $\text{if } \exists mv : mvs \bullet$ $\quad ((mv \in \text{ran mviri} \wedge n \in \text{ran iri} \wedge (iri \cdot n) = mviri \cdot mv) \vee$ $\quad (mv \in \text{ran mviris} \wedge n \in \text{ran iri} \wedge$ $\quad \quad (iri \cdot n) \in \text{IRIstemRange } (mviris \cdot mv)) \vee$ $\quad (n \in \text{ran literal} \wedge mvlit \cdot mv = literal \cdot n))$ $\text{then } p$ $\text{else } f$

evalExclusion – return pass (*p*) if the supplied *RDFTerm* is *not* a member of *MatchValue*.

$evalExclusion : \mathbb{P} MatchValue \rightarrow RDFTerm \rightarrow OptValidity$
$\forall mvs : \mathbb{P} MatchValue; n : RDFTerm \bullet evalExclusion mvs n =$ $\quad \mathbf{if} \quad evalInclusion mvs n = \mathbf{p} \mathbf{then} \mathbf{f} \mathbf{else} \mathbf{p}$

evalShape – return **p** if the subgraph of the *EvalContext* graph whose subjects match the supplied *RDFTerm* satisfies the *ValueReference* rule.

$evalShape : EvalContext \rightarrow Label \rightarrow RDFTerm \rightarrow OptValidity$
$\forall ec : EvalContext; vr : Label; n : RDFTerm \bullet$ $evalShape ec vr n =$ $\quad \mathbf{if} \quad n \notin \text{ran } literal \mathbf{then} \quad evalRule' ec n vr$ $\quad \mathbf{else} \mathbf{f}$

4.2 RevArcRule evaluation

The *RevArcRule* is used to select the subset of the graph having a given predicate or predicates and to determine whether the cardinality and/or “type” of this subset matches a supplied criteria. The rule itself consists of a *PredicateFilter* to select the triples, an *SubjectSpecification* to evaluate the result, a optional *min* and *max* cardinality and a (possibly empty) set of *Actions*:

<i>RevArcRule</i>
$p : PredicateFilter$ $s : SubjectSpecification$ $min, max : \mathbb{N}[0 \dots 1]$ $actions : \mathbb{P} Action$
$(\#min = 1 \wedge \#max = 1) \Rightarrow value min \leq value max$

RevArcRule evaluation consists of:

1. Select the subset of the *EvalContext Graph* with the supplied object and predicates matching *PredicateFilter*
2. Evaluate the cardinality and return the result if it doesn't pass
3. Evaluate the object of each of the triples in the set against *SubjectSpecification*. If any of the evaluations fail, return *fail* (**f**).
4. Return the result of evaluating *actions* against the matching triples.

$evalRevArcRule : EvalContext \rightarrow RDFTerm \rightarrow RevArcRule \rightarrow OptValidity$
$\forall ec : EvalContext; o : RDFTerm; rar : RevArcRule; og : Graph \mid$ $og = evalPredicateFilter rar.p (triplesForObject o ec.graph) \bullet$ $evalRevArcRule ec o rar =$ $\text{if } evalCardinality og rar.min rar.max \neq p$ $\quad \text{then } evalCardinality og rar.min rar.max$ $\text{else if } evalSubjectSpecification ec rar.s og \neq p$ $\quad \text{then } evalSubjectSpecification ec rar.s og$ else $\quad dispatch rar.actions og ec$

SubjectSpecification evaluation *SubjectSpecification* specifies a set of possible values for an *RDFTerm* and takes one of the following forms:

- *SubjectSet* - matches *IRIs* or *IRIstems* that match one or more of the expressions in the specified set
- *SubjectWild* - matches any target *except* those matching the (possibly empty) set of *IRIstems*
- *ssShape* - matches any target that is considered valid according the *Rule* identified by *Label*.

$SubjectSpecification ::= ssInclusion\langle\langle P Match Value \rangle\rangle \mid$
 $ssExclusion\langle\langle P Match Value \rangle\rangle \mid$
 $ssShape\langle\langle Label \rangle\rangle$

paragraphevalSubjectSpecification – returns *pass* (**p**) if all of the triples in a *Graph* match the supplied *SubjectSpecification*, otherwise *fail* (**f**)

$evalSubjectSpecification : EvalContext \rightarrow SubjectSpecification \rightarrow Graph \rightarrow OptValidity$
$\forall ec : EvalContext; ss : SubjectSpecification; g : Graph \bullet$ $evalSubjectSpecification ec ss g =$ $\text{if } \forall t : g \bullet evalSubjectSpecificationTriple ec ss t.s = p \text{ then } p$ $\text{else } f$

$evalSubjectSpecificationTriple : EvalContext \rightarrow SubjectSpecification \rightarrow RDFTerm \rightarrow OptValidity$
$\forall ec : EvalContext; ss : SubjectSpecification; n : RDFTerm \bullet$ $evalSubjectSpecificationTriple ec ss n =$ $\text{if } ss \in \text{ran } ssInclusion \text{ then}$ $evalInclusion(ssInclusion \cdot ss) n$ $\text{else if } ss \in \text{ran } ssExclusion \text{ then}$ $evalExclusion(ssExclusion \cdot ss) n$ else $evalShape ec (ssShape \cdot ss) n$

4.3 GroupRule evaluation

A *GroupRule* serves two purposes. The first is to declare that a referenced rule is to be treated as “optional”, which, in this case means that if (a) the referenced rule returned *dunno* (\emptyset), (meaning an *ArcRule* was encountered that had no matching predicates and a non-zero minimum cardinality) the group rule returns *nomatch* (\mathbf{z}). An optional *GroupRule* also treats an error situation as a *fail* (\mathbf{f}).

The second purpose of a group rule is to allow a set of external *actions* to be evaluated whenever the referenced *rule* returns *pass* (\mathbf{p}).

$OPT ::= OPTIONAL \mid REQUIRED$
 $GroupRule \hat{=} [rule : Label; opt : OPT; actions : \mathbb{P} Action]$

evalGroupRule evaluates *Rule*, applies *opt* and, if the result is *pass* (\mathbf{p}) evaluates the actions with respect the passing *Graph*

$evalGroupRule : EvalContext \rightarrow RDFTerm \rightarrow GroupRule \rightarrow OptValidity$
$\forall ec : EvalContext; i : RDFTerm; gr : GroupRule \bullet evalGroupRule ec i gr =$ $\text{if } evalRule' ec i gr.rule = \emptyset \wedge gr.opt = OPTIONAL$ $\text{then } \mathbf{z}$ $\text{else if } evalRule' ec i gr.rule = \mathbf{p}$ $\text{then } dispatch\ gr.actions\ \emptyset\ ec$ $\text{else } evalRule' ec i gr.rule$

4.4 AndRule evaluation

An *AndRule* consists of a set of one or more *Rules*, whose evaluation is determined by the following table:

And	\emptyset	\mathbf{z}	\mathbf{f}	\mathbf{p}	ε
\emptyset	\emptyset	\emptyset	\mathbf{f}	\mathbf{f}	ε
\mathbf{z}	\emptyset	\mathbf{z}	\mathbf{f}	\mathbf{p}	ε
\mathbf{f}	\mathbf{f}	\mathbf{f}	\mathbf{f}	\mathbf{f}	ε
\mathbf{p}	\mathbf{f}	\mathbf{p}	\mathbf{f}	\mathbf{p}	ε
ε	ε	ε	ε	ε	ε

The formal implementation of which will be realized in the corresponding function:

$$\text{And} : \text{OptValidity} \rightarrow \text{OptValidity} \rightarrow \text{OptValidity}$$

Observing that the above table is a monoid with *nomatch* (\mathbf{z}) as an identity element, *evalAndRule* can be applied using the standard functional pattern:

$$\text{AndRule} == \text{seq}_1 \text{ Label}$$

$$\text{evalAndRule} : \text{EvalContext} \rightarrow \text{RDFTerm} \rightarrow \text{AndRule} \rightarrow \text{OptValidity}$$

$$\begin{aligned} \forall ec : \text{EvalContext}; i : \text{RDFTerm}; r : \text{AndRule} \bullet \\ \text{evalAndRule } ec \ i \ r = \\ \text{foldr And } \mathbf{z} \ (\text{map } (\text{evalRule}' \ ec \ i) \ r) \end{aligned}$$

4.5 XorRule evaluation

An *XorRule* consists of a set of one or more *Rules*, whose evaluation is determined by the following table:

Xor	\emptyset	\mathbf{z}	\mathbf{f}	\mathbf{p}	ε
\emptyset	\emptyset	\mathbf{z}	\emptyset	\mathbf{p}	ε
\mathbf{z}	\mathbf{z}	\mathbf{z}	\mathbf{z}	\mathbf{p}	ε
\mathbf{f}	\emptyset	\mathbf{z}	\mathbf{f}	\mathbf{p}	ε
\mathbf{p}	\mathbf{p}	\mathbf{p}	\mathbf{p}	\mathbf{p}	ε
ε	ε	ε	ε	ε	ε

The formal implementation of which will be realized in the corresponding function:

$$\text{Xor} : \text{OptValidity} \rightarrow \text{OptValidity} \rightarrow \text{OptValidity}$$

As with the *And* function above, *Xor* is a monoid whose identity is *fail* (\mathbf{f}) resulting in the following definition for *evalXorRule*

$$\text{XorRule} == \text{seq}_1 \text{ Label}$$

$evalXorRule : EvalContext \rightarrow RDFTerm \rightarrow XorRule \rightarrow OptValidity$
$\forall ec : EvalContext; i : RDFTerm; r : XorRule \bullet$ $evalXorRule\ ec\ i\ r =$ $foldr\ Xor\ \mathbf{f}\ (map\ (evalRule'\ ec\ i)\ r)$

5 Action evaluation

The *dispatch* function allows the evaluation / execution of arbitrary external “Actions”. While the evaluation of an *Action* can (obviously) have side effects outside the context of the *ShEx* environment, it must be side effect free within the execution context. In particular, an *Action* may not change anything in the *EvalContext*. The action dispatcher exists to allow external events to happen. Parameters:

- *as* - the set of *Actions* associated with the *GroupRule*, *ArcRule* or *RevArcRule*
- *g* - the *Graph* that passed the *ArcRule* or *RevArcRule*. Empty in the case of *GroupRule*
- *ec* - the *EvalContext* containing the *Schema* and *Graph*

The dispatch function usually returns *pass* (**p**) or *fail* (**f**), although there may also be cases for other *OptValidity* values in certain circumstances. The dispatch function always returns *pass* (**p**) if the set of actions is empty.

[*Action*]

$dispatch : \mathbb{P}\ Action \rightarrow Graph \rightarrow EvalContext \rightarrow OptValidity$
$\forall as : \mathbb{P}\ Action; g : Graph; ec : EvalContext \bullet$ $as = \emptyset \Rightarrow dispatch\ as\ g\ ec = \mathbf{p}$

A Appendix

A.1 Foldr

The *foldr* function is the standard functional pattern, which takes a binary function of type *T*, an identity function for type *T*, a sequence of type *T* and returns the result of applying the function to the right to left pairs of the sequence.

$[T]$
$foldr : (T \rightarrow T \rightarrow T) \rightarrow T \rightarrow seq\ T \rightarrow T$
$\forall f : T \rightarrow T \rightarrow T; id : T; xs : seq\ T \bullet foldr\ f\ id\ xs =$ $\mathbf{if}\ xs = \langle \rangle\ \mathbf{then}\ id$ $\mathbf{else}\ f\ (head\ xs)\ (foldr\ f\ id\ (tail\ xs))$

A.2 Map

The *map* function takes a function from type A to type B and applies it to all members in the supplied sequence

$$\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} [A, B] \text{---} \\ \text{---} \text{map} : (A \rightarrow B) \rightarrow \text{seq } A \rightarrow \text{seq } B \\ \text{---} \\ \forall f : A \rightarrow B; \text{ xs} : \text{seq } A \bullet \text{map } f \text{ xs} = \\ \quad \text{if } \text{xs} = \langle \rangle \text{ then } \langle \rangle \\ \quad \text{else } \langle f(\text{head } \text{xs}) \rangle \frown \text{map } f(\text{tail } \text{xs}) \end{array}$$

A.3 Helper Functions

Z uses the notion of *Free type definitions* in the form:

$$\text{FreeType} ::= \text{constructor} \langle\langle \text{source} \rangle\rangle$$

which introduces a collection of constants of type *FreeType*, one for each element of the set *source*. *constructor* is an injective function from *source* to *FreeType*:

$$\text{constructor} ::= \text{source} \rightarrow \text{FreeType}$$

In the models that follow, there is a need to reverse this – to find the *source* for a given *FreeType* instance. The \cdot function exists for this purpose. As an example, if one were to define:

$$\text{TravelDirections} ::= \text{bus} \langle\langle \text{BusDirections} \rangle\rangle \mid \text{walking} \langle\langle \text{WalkingDirections} \rangle\rangle$$

If one is supplied with an instance of *Travel*, one can convert it to the appropriate type by:

$$\begin{array}{c} x : \text{TravelDirections} \\ \text{---} \\ \text{if } x \in \text{ran bus then bus} \cdot x \text{ else walking} \cdot x \end{array}$$

$$\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} [X, Y] \text{---} \\ \text{---} - \cdot - : (X \rightarrow Y) \times Y \rightarrow X \\ \text{---} \\ \forall y : Y; f : X \rightarrow Y \bullet f \cdot y = (\mu x : \text{dom } f \mid f x = y) \end{array}$$

One way to represent optional values is a set with one member. We take that route here and introduce a bit of syntactic sugar to show our intent:

$$T[0 \dots 1] == \{s : \mathbb{P} T \mid \#s \leq 1\}$$

And a shorthand for addressing the content:

$$\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} [T] \text{---} \\ \text{---} \text{value} : \mathbb{P} T \rightarrow T \\ \text{---} \\ \forall s : \mathbb{P} T \bullet \text{value } s = (\mu e : T \mid e \in s) \end{array}$$

References

1. Richard Cyganiak and David Wood. RDF 1.1 concepts and abstract syntax - generalized rdf triples, graphs, and datasets. World Wide Web Consortium, Working Draft WD-rdf11-concepts-20130723, August 2013.
2. M. Dürst and M. Suignard. Internationalized resource identifiers (iris). RFC, January 2005.
3. RA Phillips and M. Davis. Tags for identifying language. IETF Best Current Practices, September 2009.