A Formal Model of the Shape Expression Language

Harold Solbrig[1], Eric Prud'hommeaux[2]

[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

Using the formal definitions in *RDF 1.1 Concepts and Abstract Syntax*[1], we begin with:

"An **RDF graph** is a set of **RDF Triples**"

Formally:

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

The specification then defines the notion of **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**"

"'... **IRI**s, **literal**s and **blank nodes** are distinct and distinguishable."

The *ShEx* language treats **IRI**s and **blank nodes** as primitive types, so we can formally define them as *Z* free types:

$[IRI, BlankNode]$

The *ShEx* language can express constraints on both the type and content of **literals**, so we need to model these separately. Starting with:

"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]."

We model this by defining *String* and *LanguageTag* as free types:

$[String, LanguageTag]$

And then use them in the definition the two flavors of *RDFLiteral*, plain literal and typed literal::

$$\begin{aligned}
&TypedLiteral \mathrel{\widehat=} [lexicalForm : String;\ dataType : IRI \mid dataType \neq XSD\_String]\\
&PlainLiteral \mathrel{\widehat=} [lexicalForm : String;\ dataType : IRI;\ langTag : LanguageTag \mid\\
&\qquad\qquad dataType = XSD\_String]\\
&RDFLiteral ::= pl\langle\!\langle PlainLiteral\rangle\!\rangle \mid tl\langle\!\langle TypedLiteral\rangle\!\rangle
\end{aligned}$$

The RDF 1.1 specification the defines *RDFTerm* as:

"**IRI**s, **literal**s and **blank nodes** are collectively known as **RDF terms**

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

We are now in the position to complete the definition of *Triple* as a tuple consisting of three constrained *RDFTerm*s:

Triple

$$\begin{array}{l} s, p, o : RDFTerm \\ \hline iri \cdot s \in IRI \lor bnode \cdot s \in BlankNode \\ iri \cdot p \in IRI \\ iri \cdot o \in IRI \lor bnode \cdot o \in BlankNode \lor literal \cdot o \in RDFLiteral \end{array}$$

The *ShEx* language uses the following functions:

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

$$\begin{array}{l} subjects : RDFTerm \to Graph \to Graph \\ \hline \forall\, subj : RDFTerm;\ g : Graph \bullet subjects\ subj\ g = \{t : g \mid t.s = subj\} \end{array}$$

objects – return set of objects in *Graph*

$$\begin{array}{l} objects : Graph \to \mathbb{P}\, RDFTerm \\ \hline \forall\, g : Graph \bullet objects\ g = \{t : g \bullet t.o\} \end{array}$$

It is also useful to define a couple of well known URI's for future reference:

$$XSD\_String, RDF\_Resource : IRI$$

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 labeled rules where exactly one rule in the collection is identified as the outermost or "starting" rule. In addition, any rule that is referenced within the *Schema* is also itself a member of the *Schema* Formally:

$$\begin{array}{|l}
\textit{Schema} \\
\hline
\textit{rules} : \textit{Label} \nrightarrow \textit{Rule} \\
\textit{start} : \textit{Label} \\
\hline
\textit{start} \in \operatorname{dom} \textit{rules} \\
\forall\, r : \operatorname{ran} \textit{rules} \bullet \\
\quad (r \in \operatorname{ran} \textit{group} \Rightarrow (\textit{group} \cdot r).\textit{rule} \in \operatorname{dom} \textit{rules}) \land \\
\quad (r \in \operatorname{ran} \textit{and} \Rightarrow \operatorname{ran}(\textit{and} \cdot r) \subseteq \operatorname{dom} \textit{rules}) \land \\
\quad (r \in \operatorname{ran} \textit{xor} \Rightarrow \operatorname{ran}(\textit{xor} \cdot r) \subseteq \operatorname{dom} \textit{rules}) \land \\
\quad (r \in \operatorname{ran} \textit{arc} \land (\textit{arc} \cdot r).\textit{valueSpec} \in \operatorname{ran} \textit{valueRef} \Rightarrow \\
\qquad (\textit{valueRef} \cdot (\textit{arc} \cdot r).\textit{valueSpec}) \in \operatorname{dom} \textit{rules})
\end{array}$$

While existing *ShEx* implementations define a rule *Label* as being either an *IRI* or a *BlankNode*, the type of *Label* does not impact the evaluation semantics. For our purposes, we can simply define it as a separate free type:

$[\textit{Label}]$

The validity of a given RDF *Graph* is determined by taking the *start Rule* of a *ShEx Schema* and a subject *IRI* and evaluating the validity of the *Rule* against the subset of the triples in the graph having the supplied *subject*. It is necessary that this evaluation be done in the *context* of the entire *Schema* and *Graph*, as the *start* rule in the schema will usually reference other *rules* and it will frequently be the case that the *objects* of the subject graphs will (recursively) reference other graph subjects.

Formally, the *evaluate* function takes a *Schema*, a *Graph* and a subject *IRI* and, if the *start Rule* in the *Schema*, when evaluated against the subject graph in the context of the starting *Schema* and *graph*, returns either *nomatch* ($\boldsymbol{z}$) or *pass* ($\boldsymbol{p}$) then the function returns *pass*. In all other cases, the function returns *fail* ($\boldsymbol{f}$).

$$\begin{array}{|l}
\hline
\textit{evaluate} : \textit{Schema} \to \textit{Graph} \to \textit{IRI} \to \textit{OptValidity} \\
\hline
\forall\, s : \textit{Schema};\ g : \textit{Graph};\ i : \textit{IRI};\ v : \textit{OptValidity};\ ec : \textit{EvalContext} \mid \\
\quad ec.\textit{graph} = g \land ec.\textit{schema} = s \bullet \\
\textit{evaluate}\ s\ g\ i = \\
\quad \textbf{if}\ \textit{evalRule}\ ec\ (\textit{subjects}\ (\textit{iri}\ i)\ g)\ (s.\textit{rules}\ s.\textit{start}) \in \{\boldsymbol{z}, \boldsymbol{p}\} \\
\quad \textbf{then}\ \boldsymbol{p}\ \textbf{else}\ \boldsymbol{f}
\end{array}$$

4 Rule Evaluation

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

$\textit{EvalContext} \mathrel{\widehat{=}} [\textit{schema} : \textit{Schema};\ \textit{graph} : \textit{Graph}]$

The *Graph* being evaluated does not necessarily have to be a subset of the *Graph* in *EvalContext*, and may also have more than one *subject*. While the *evaluate* function described earlier in this section actually ensures that both of these conditions are true, it is envisioned that the *eval* function may have other applications beyond the scope of the *evaluate* function itself.

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

– **Pass** $(\boldsymbol{p})$ - the supplied *Graph* satisfied the evaluation *Rule*
– **Fail** $(\boldsymbol{f})$ - the supplied *Graph* did not satisfy the evaluation *Rule*
– **Nomatch** $(\boldsymbol{z})$ - an *optional GroupRule* was encountered and there were no matching triples
– **Dunno** $(\varnothing)$ - 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.

$$OptValidity ::= \boldsymbol{p} \mid \boldsymbol{f} \mid \boldsymbol{z} \mid \varnothing \mid \varepsilon$$

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

– ***ArcRule*** – selects a subset of the graph having predicates that match criteria described in the rule and and validates the objects of this subset
– ***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*

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

```
$evalRule : EvalContext \rightarrow Graph \rightarrow Rule \rightarrow OptValidity$
----------
$\forall\, ec : EvalContext;\ g : Graph;\ r : Rule \bullet evalRule\ ec\ g\ r =$
    **if** $r \in \operatorname{ran} arc$
        **then** $evalArcRule\ ec\ g\ (arc \cdot r)$
    **else if** $r \in \operatorname{ran} group$
        **then** $evalGroupRule\ ec\ g\ (group \cdot r)$
    **else if** $r \in \operatorname{ran} and$
        **then** $evalAndRule\ ec\ g\ (and \cdot r)$
    **else**
        $evalXorRule\ ec\ g\ (xor \cdot r)$
```

The function below is used to execute the rules that are referenced by *GroupRule*s, *AndRules* and *OrRules*. Its purpose is to de-reference the supplied *Label* and invoke *evalRule* with the result. This is not explicitly represented because the *Z* specification language does not allow cyclic dependencies. Note that this function is undefined if *Label* is not in *EvalContext*

$$evalRuleLabel : EvalContext \rightarrow Graph \rightarrow Label \rightarrow OptValidity$$

$$\forall\, ec : EvalContext;\ l : Label \bullet l \in \operatorname{dom} ec.schema.rules$$

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*:

ArcRule

$$filter : PredicateFilter$$
$$valueSpec : ObjectSpecification$$
$$min, max : \mathbb{N}[0\,..\,1]$$
$$actions : \mathbb{P}\, Action$$

$$(\#min = 1 \wedge \#max = 1) \Rightarrow value\ min \leq value\ max$$

ArcRule evaluation consists of:

1. Selecting all of the triples in *Graph* having predicates that match *Predicate-Filter*
2. If the number of triples from the previous step is 0, return either *nomatch* ($\boldsymbol{z}$) if *min* is 0 otherwise *dunno*($\varnothing$).
3. If *min* is specified and the number of triples from the previous step is $<$ *min* then return *fail* ($\boldsymbol{f}$)
4. If *max* is specified and the number of triples from the previous step is $>$ *max* then return *fail* ($\boldsymbol{f}$)
5. Evaluate the object of each of the triples in the set against *ObjectSpecification*. If any of the evaluations fail, return *fail* ($\boldsymbol{f}$).
6. Return the result of evaluating *actions* against the matching triples.

```
$evalArcRule : EvalContext \rightarrow Graph \rightarrow ArcRule \rightarrow OptValidity$
---
$\forall ec : EvalContext;\ g, sg : Graph;\ ar : ArcRule \mid$
    $sg = evalPredicateFilter\ ar.filter\ ec.graph \bullet$
$evalArcRule\ ec\ g\ ar =$
    $\textbf{if}\ \#ar.min = 1 \land \#sg = 0 \land value\ ar.min = 0$
        $\textbf{then}\ \boldsymbol{z}$
    $\textbf{else if}\ \#sg = 0$
        $\textbf{then}\ \varnothing$
    $\textbf{else if}\ (\#ar.min = 1 \land \#sg < value\ ar.min) \lor$
            $(\#ar.max = 1 \land \#sg > value\ ar.max)$
        $\textbf{then}\ \boldsymbol{f}$
    $\textbf{else if}\ evalObjectSpecification\ ec\ ar.valueSpec\ sg = \boldsymbol{p}$
        $\textbf{then}\ dispatch\ ar.actions\ sg\ ec$
    $\textbf{else}$
        $evalObjectSpecification\ ec\ ar.valueSpec\ sg$
```

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

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

$$IRIorStem ::= iosi\langle\langle IRI\rangle\rangle \mid ioss\langle\langle IRIstem\rangle\rangle$$
$$PredicateFilter ::= pfIRI\langle\langle IRIorStem\rangle\rangle \mid pfWild\langle\langle \mathbb{P}\ IRIorStem\rangle\rangle$$

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*:

```
$evalPredicateFilter : PredicateFilter \rightarrow\!\!\!\!\!\!+\ Graph \rightarrow\!\!\!\!\!\!+\ Graph$
---
$\forall f : PredicateFilter;\ g : Graph \bullet evalPredicateFilter\ f\ g =$
    $\textbf{if}\ f \in \operatorname{ran} pfIRI\ \textbf{then}\ evalIRIorStem\ (pfIRI \sim f)\ g$
    $\textbf{else}\ evalWild\ (pfWild \sim f)\ g$
```

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

$$\begin{array}{|l}
evalIRIorStem : IRIorStem \rightarrow\!\!\!\!\!+ Graph \rightarrow\!\!\!\!\!+ Graph \\
\hline
\forall e : IRIorStem;\ g : Graph \bullet evalIRIorStem\ e\ g = \\
\quad \textbf{if}\ e \in \operatorname{ran} iosi\ \textbf{then}\ \{t : g \mid iri \cdot t.p = iosi \cdot e\} \\
\quad \textbf{else}\ \{t : g \mid iri \cdot t.p \in IRIstemRange\,(ioss \cdot e)\}
\end{array}$$

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

$$\begin{array}{|l}
evalWild : \mathbb{P}\,IRIorStem \rightarrow\!\!\!\!\!+ Graph \rightarrow\!\!\!\!\!+ Graph \\
\hline
\forall es : \mathbb{P}\,IRIorStem;\ g : Graph \bullet evalWild\ es\ g = \\
\quad \{t : g \mid t \notin \bigcup\{e : es \bullet evalIRIorStem\ e\ g\}\}
\end{array}$$

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

$$\begin{array}{l}
MatchValue ::= mviri\langle\!\langle IRI\rangle\!\rangle \mid mviris\langle\!\langle IRIstem\rangle\!\rangle \mid \\
\qquad\qquad mvlit\langle\!\langle RDFLiteral\rangle\!\rangle \\
ObjectSpecification ::= valueType\langle\!\langle IRI\rangle\!\rangle \mid \\
\qquad valueSet\langle\!\langle \mathbb{P}\,MatchValue\rangle\!\rangle \mid \\
\qquad osWild\langle\!\langle \mathbb{P}\,MatchValue\rangle\!\rangle \mid \\
\qquad valueRef\langle\!\langle Label\rangle\!\rangle
\end{array}$$

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

$$\begin{array}{|l}
evalObjectSpecification : EvalContext \rightarrow ObjectSpecification \rightarrow Graph \rightarrow \\
\quad OptValidity \\
\hline
\forall ec : EvalContext;\ os : ObjectSpecification;\ g : Graph \bullet \\
evalObjectSpecification\ ec\ os\ g = \\
\quad \textbf{if}\ \forall t : g \bullet evalObjectSpecificationTriple\ ec\ os\ t.o = \boldsymbol{p}\ \textbf{then}\ \boldsymbol{p} \\
\quad \textbf{else}\ \boldsymbol{f}
\end{array}$$

```
$evalObjectSpecificationTriple : EvalContext \rightarrow ObjectSpecification \rightarrow$
        $RDFTerm \rightarrow OptValidity$
---
$\forall ec : EvalContext;\ os : ObjectSpecification;\ t : RDFTerm \bullet$
$evalObjectSpecificationTriple\ ec\ os\ t =$
        $\textbf{if}\ os \in \operatorname{ran} valueType\ \textbf{then}$
                $evalValueType\ (valueType \cdot os)\ t$
        $\textbf{else if}\ os \in \operatorname{ran} valueSet\ \textbf{then}$
                $evalValueSet\ (valueSet \cdot os)\ t$
        $\textbf{else if}\ os \in \operatorname{ran} osWild\ \textbf{then}$
                $evalValueWild\ (osWild \cdot os)\ t$
        $\textbf{else}$
                $evalValueReference\ ec\ (valueRef \cdot os)\ t$
```

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

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

```
$evalValueType : IRI \rightarrow RDFTerm \rightarrow OptValidity$
---
$\forall vt : IRI;\ t : RDFTerm;\ l : RDFLiteral \bullet evalValueType\ vt\ t =$
        $\textbf{if}\ vt = RDF\_Resource \land t \in \operatorname{ran} iri\ \textbf{then}\ \boldsymbol{p}$
        $\textbf{else if}\ t \in \operatorname{ran} literal \land l = (literal \cdot t) \land$
                $((l \in \operatorname{ran} pl \land (pl \cdot l).dataType = vt) \lor$
                $(l \in \operatorname{ran} tl \land (tl \cdot l).dataType = vt))\ \textbf{then}\ \boldsymbol{p}$
        $\textbf{else}\ \boldsymbol{f}$
```

evalValueSet – return $\boldsymbol{p}$ if the supplied *RDFTerm* is a member of *MatchValue*

```
$evalValueSet : \mathbb{P}\,MatchValue \rightarrow RDFTerm \rightarrow OptValidity$
---
$\forall mvs : \mathbb{P}\,MatchValue;\ t : RDFTerm \bullet evalValueSet\ mvs\ t =$
        $\textbf{if}\ \exists\, mv : mvs \bullet$
                $((mv \in \operatorname{ran} mviri \land (iri \cdot t) = mviri \cdot mv) \lor$
                $(mv \in \operatorname{ran} mviris \land (iri \cdot t) \in IRIstemRange\ (mviris \cdot mv)) \lor$
                $(mvlit \cdot mv = literal \cdot t))$
        $\textbf{then}\ \boldsymbol{p}$
        $\textbf{else}\ \boldsymbol{f}$
```

evalValueWild – return pass ($\boldsymbol{p}$) if the supplied *RDFTerm* is *not* a member of *MatchValue*.

$$\mathit{evalValueWild} : \mathbb{P}\,\mathit{MatchValue} \rightarrow \mathit{RDFTerm} \rightarrow \mathit{OptValidity}$$

$$\begin{aligned}&\forall\, \mathit{mvs} : \mathbb{P}\,\mathit{MatchValue};\ t : \mathit{RDFTerm} \bullet \mathit{evalValueWild}\ \mathit{mvs}\ t = \\ &\quad \textbf{if}\ \mathit{evalValueSet}\ \mathit{mvs}\ t = \boldsymbol{p}\ \textbf{then}\ \boldsymbol{f}\ \textbf{else}\ \boldsymbol{p}\end{aligned}$$

evalValueReference – return $\boldsymbol{p}$ if the subgraph of the *EvalContext* graph whose subjects match the supplied *RDFTerm* satisfies the *ValueReference* rule.

$$\mathit{evalValueReference} : \mathit{EvalContext} \nrightarrow \mathit{Label} \nrightarrow \mathit{RDFTerm} \nrightarrow \mathit{OptValidity}$$

$$\begin{aligned}&\forall\, \mathit{ec} : \mathit{EvalContext};\ \mathit{vr} : \mathit{Label};\ t : \mathit{RDFTerm} \bullet \\ &\mathit{evalValueReference}\ \mathit{ec}\ \mathit{vr}\ t = \\ &\quad \textbf{if}\ t \notin \operatorname{ran} \mathit{literal}\ \textbf{then}\ \mathit{evalRuleLabel}\ \mathit{ec}\ (\mathit{subjects}\ t\ \mathit{ec}.\mathit{graph})\ \mathit{vr} \\ &\quad \textbf{else}\ \boldsymbol{f}\end{aligned}$$

4.2 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* ($\varnothing$), (meaning an *ArcRule* was encountered that had no matching predicates and a non-zero minimum cardinality) the group rule returns *nomatch* ($\boldsymbol{z}$). An optional *GroupRule* also treats an error situation as a *fail* ($\boldsymbol{f}$).

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

$$\begin{aligned}&\mathit{OPT} ::= \mathit{OPTIONAL} \mid \mathit{REQUIRED} \\ &\mathit{GroupRule} \mathrel{\widehat{=}} [\mathit{rule} : \mathit{Label};\ \mathit{opt} : \mathit{OPT};\ \mathit{actions} : \mathbb{P}\,\mathit{Action}]\end{aligned}$$

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

$$\mathit{evalGroupRule} : \mathit{EvalContext} \rightarrow \mathit{Graph} \rightarrow \mathit{GroupRule} \rightarrow \mathit{OptValidity}$$

$$\begin{aligned}&\forall\, \mathit{ec} : \mathit{EvalContext};\ g : \mathit{Graph};\ \mathit{gr} : \mathit{GroupRule} \bullet \mathit{evalGroupRule}\ \mathit{ec}\ g\ \mathit{gr} = \\ &\quad \textbf{if}\ \mathit{evalRuleLabel}\ \mathit{ec}\ g\ \mathit{gr}.\mathit{rule} = \varnothing \wedge \mathit{gr}.\mathit{opt} = \mathit{OPTIONAL} \\ &\qquad \textbf{then}\ \boldsymbol{z} \\ &\quad \textbf{else if}\ \mathit{evalRuleLabel}\ \mathit{ec}\ g\ \mathit{gr}.\mathit{rule} = \varepsilon \wedge \mathit{gr}.\mathit{opt} = \mathit{OPTIONAL} \\ &\qquad \textbf{then}\ \boldsymbol{f} \\ &\quad \textbf{else if}\ \mathit{evalRuleLabel}\ \mathit{ec}\ g\ \mathit{gr}.\mathit{rule} = \boldsymbol{p} \\ &\qquad \textbf{then}\ \mathit{dispatch}\ \mathit{gr}.\mathit{actions}\ g\ \mathit{ec} \\ &\quad \textbf{else}\ \mathit{evalRuleLabel}\ \mathit{ec}\ g\ \mathit{gr}.\mathit{rule}\end{aligned}$$

4.3 AndRule evaluation

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

And	$\varnothing$	$\boldsymbol{z}$	$\boldsymbol{f}$	$\boldsymbol{p}$	ε
$\varnothing$	$\varnothing$	$\varnothing$	$\boldsymbol{f}$	$\boldsymbol{f}$	ε
$\boldsymbol{z}$	$\varnothing$	$\boldsymbol{z}$	$\boldsymbol{f}$	$\boldsymbol{p}$	ε
$\boldsymbol{f}$	$\boldsymbol{f}$	$\boldsymbol{f}$	$\boldsymbol{f}$	$\boldsymbol{f}$	ε
$\boldsymbol{p}$	$\boldsymbol{f}$	$\boldsymbol{p}$	$\boldsymbol{f}$	$\boldsymbol{p}$	ε
ε	ε	ε	ε	ε	ε

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

$$And : OptValidity \rightarrow OptValidity \rightarrow OptValidity$$

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

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

$$evalAndRule : EvalContext \rightarrow Graph \rightarrow AndRule \rightarrow OptValidity$$

$$\forall\, ec : EvalContext;\ g : Graph;\ r : AndRule \bullet$$
$$evalAndRule\ ec\ g\ r =$$
$$\quad foldr\ And\ \boldsymbol{z}\ (map\ (evalRuleLabel\ ec\ g)\ r)$$

4.4 XorRule evaluation

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

Xor	$\varnothing$	$\boldsymbol{z}$	$\boldsymbol{f}$	$\boldsymbol{p}$	ε
$\varnothing$	$\varnothing$	$\boldsymbol{z}$	$\varnothing$	$\boldsymbol{p}$	ε
$\boldsymbol{z}$	$\boldsymbol{z}$	$\boldsymbol{z}$	$\boldsymbol{z}$	$\boldsymbol{p}$	ε
$\boldsymbol{f}$	$\varnothing$	$\boldsymbol{z}$	$\boldsymbol{f}$	$\boldsymbol{p}$	ε
$\boldsymbol{p}$	$\boldsymbol{p}$	$\boldsymbol{p}$	$\boldsymbol{p}$	ε	ε
ε	ε	ε	ε	ε	ε

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

$$Xor : OptValidity \rightarrow OptValidity \rightarrow OptValidity$$

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

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

$$\begin{array}{|l}
evalXorRule : EvalContext \to Graph \to XorRule \to OptValidity \\
\hline
\forall\, ec : EvalContext;\ g : Graph;\ r : XorRule \bullet \\
evalXorRule\ ec\ g\ r = \\
\quad foldr\ Xor\ \boldsymbol{f}\ (map\ (evalRuleLabel\ ec\ g)\ r)
\end{array}$$

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* or passing *Graph* The action dispatcher exists to allow external events to happen. Parameters:

- *EvalContext* - the evaluation context
- *Actions* - the set of *Actions* associated with the associated *GroupRule* or *ArcRule*
- *Graph* - the *Graph* that passed the associated Rule.

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

$$[Action]$$

$$\begin{array}{|l}
dispatch : \mathbb{P}\, Action \to Graph \to EvalContext \to OptValidity \\
\hline
\forall\, as : \mathbb{P}\, Action;\ g : Graph;\ ec : EvalContext \bullet \\
\quad as = \emptyset \Rightarrow dispatch\ as\ g\ ec = \boldsymbol{p}
\end{array}$$

6 Appendix

6.1 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.

$$[T]$$
$$foldr : (T \rightarrow T \rightarrow T) \rightarrow T \rightarrow \operatorname{seq} T \rightarrow T$$
$$\begin{array}{l} \forall f : T \rightarrow T \rightarrow T;\ id : T;\ xs : \operatorname{seq} T \bullet foldr\, f\, id\, xs = \\ \quad \textbf{if } xs = \langle\rangle \textbf{ then } id \\ \quad \textbf{else } f\ (head\ xs)\ (foldr\ f\ id\ (tail\ xs)) \end{array}$$

6.2 Map

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

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

6.3 Helper Functions

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

$$FreeType ::= constructor\langle\!\langle source \rangle\!\rangle$$

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*:

$$constructor ::= source \rightarrowtail FreeType$$

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:

$$TravelDirections ::= bus\langle\!\langle BusDirections \rangle\!\rangle \mid walking\langle\!\langle WalkingDirections \rangle\!\rangle$$

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

$$x : TravelDirections$$
$$\vdash$$
$$\textbf{if } x \in \operatorname{ran} bus \textbf{ then } bus \cdot x \textbf{ else } walking \cdot x$$

$$[X, Y]$$
$$\_ \cdot \_ : (X \rightarrowtail Y) \times Y \nrightarrow X$$
$$\forall y : Y;\ f : X \rightarrowtail Y \bullet f \cdot y = (\mu\, x : \operatorname{dom} f \mid f\ x = y)$$

As Z has no notion of absence, it is convenient to add a bit of syntactic sugar.

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

It is also useful to provide a shorthand for addressing the content of singletons:

$$\begin{array}{|l}
\hline
[T] \\
\mathit{value} : \mathbb{P}\,T \pfun T \\
\hline
\forall s : \mathbb{P}\,T \bullet \mathit{value}\ s = (\mu\, e : T \mid e \in s) \\
\hline
\end{array}$$

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.