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`. This document describes the formal *semantics* of the *ShEx* language using the *Z* specification language, beginning with a *Z* specification of the characteristics of an *RDF Graph* that are referenced by *ShEx*.

2 The RDF Data Model in Z

Using the formal definitions in *RDF 1.1 Concepts and Abstract Syntax*[**?**]:

"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**"

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

The *ShEx* language treats **IRI**s and **blank nodes** as primitive types, which are defined as *Z* free types:

$$[IRI, BlankNode]$$

The *ShEx* language can express constraints on both the type and content of **literals**, which are modeled separately:

"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][**?**]. The language tag MUST be well-formed according to section 2.2.9 of [BCP47][**?**]."

This is modelled by *String* and *LanguageTag* as free types:

$$[String, LanguageTag]$$

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

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

RDFTerm is defined as:

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

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

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

$$\begin{array}{|l}
\textit{Triple} \\
\hline
s, p, o : RDFTerm \\
\hline
iri^{\sim} s \in IRI \lor bnode^{\sim} s \in BlankNode \\
iri^{\sim} p \in IRI \\
iri^{\sim} o \in IRI \lor bnode^{\sim} o \in BlankNode \lor literal^{\sim} o \in RDFLiteral \\
\hline
\end{array}$$

2.1 RDF Access Functions

The *ShEx* language uses the following 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:

ID	URI
RDF_langString	`http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`
SHEX_IRI	`http://www.w3.org/ns/shacl#IRI`
SHEX_BNODE	`http://www.w3.org/ns/shacl#BNode`
SHEX_LITERAL	`http://www.w3.org/ns/shacl#Literal`
SHEX_NONLITERAL	?

$$SHEX\_IRI, SHEX\_BNODE, SHEX\_LITERAL, SHEX\_NONLITERAL,$$
$$RDF\_langString : IRI$$
$$\mathsf{disjoint}\ \langle\{SHEX\_IRI\}, \{SHEX\_BNODE\}, \{SHEX\_LITERAL\},$$
$$\{SHEX\_NONLITERAL\}, \{RDF\_langString\}\rangle$$

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:

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

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 reference *IRI* and evaluating the validity of the *Rule* against the supplied graph.

Formally, the *evaluate* function takes a *Schema*, a *Graph* and a reference *IRI* and, if the *start Rule* in the *Schema*, 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* ($\boldsymbol{f}$).

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

4 Rule Evaluation

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

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

Formally, the *evalRule* function takes an *EvalContext*, a reference *RDFTerm* 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

– **None ($\varnothing$) -** 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 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*

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

$$evalRule : EvalContext \to RDFTerm \to Rule \to OptValidity$$

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

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*

$$evalRule' : EvalContext \to RDFTerm \to Label \to OptValidity$$

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

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, an optional *min* and *max* cardinality and a (possibly empty) set of *Actions*:

$$
\begin{array}{|l}
\textit{ArcRule} \\
\hline
\textit{filter} : \textit{PredicateFilter} \\
\textit{valueSpec} : \textit{ObjectSpecification} \\
\textit{min}, \textit{max} : \textit{Optional}[\mathbb{N}] \\
\textit{actions} : \mathbb{P}\ \textit{Action} \\
\hline
(\#\textit{min} = 1 \land \#\textit{max} = 1) \Rightarrow \textit{value min} \leq \textit{value max} \\
\hline
\end{array}
$$

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* ($\boldsymbol{f}$).
4. Return the result of evaluating *actions* against the matching triples.

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

***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*

$$
\begin{array}{l}
\textit{IRIorStem} ::= \textit{iosi}\langle\!\langle \textit{IRI} \rangle\!\rangle \mid \textit{ioss}\langle\!\langle \textit{IRIstem} \rangle\!\rangle \\
\textit{PredicateFilter} ::= \textit{pfIRI}\langle\!\langle \textit{IRIorStem} \rangle\!\rangle \mid \textit{pfWild}\langle\!\langle \mathbb{P}\ \textit{IRIorStem} \rangle\!\rangle
\end{array}
$$

An *IRIstem* matches any *IRI* whose stringified representation begins with the stringified representation of *IRIstem* according to standard IRI matching rules [**?**]. 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*:

$$\begin{array}{l} evalPredicateFilter : PredicateFilter \rightarrow Graph \rightarrow Graph \\ \hline \forall f : PredicateFilter;\ g : Graph \bullet evalPredicateFilter\ f\ g = \\ \quad \textbf{if}\ f \in \operatorname{ran} pfIRI\ \textbf{then}\ evalIRIorStem\,(pfIRI^{\sim} f)\ g \\ \quad \textbf{else}\ evalWild\,(pfWild^{\sim} f)\ g \end{array}$$

evalIRIorStem returns all of the triples in a *Graph* matching the supplied *IRI-orStem*

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

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*.

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

evalCardinality – evaluates the cardinality the supplied graph.

- If the graph has no elements and:
  - *min* value is 0 – *nomatch* ($\boldsymbol{z}$)
  - *min* value isn't specified or is > 0 – *none* ($\varnothing$)
- Otherwise:
  - If number of elements in graph $<$ *min* or $>$ *max* – *fail* ($\boldsymbol{f}$)
  - Otherwise – *pass* ($\boldsymbol{p}$)

$$evalCardinality : Graph \rightarrow Optional[\mathbb{N}] \rightarrow Optional[\mathbb{N}] \rightarrow OptValidity$$

$$\begin{array}{l}
\forall\, g : Graph;\ min, max : Optional[\mathbb{N}] \bullet evalCardinality\ g\ min\ max = \\
\quad \textbf{if}\ \#min = 1 \land \#g = 0 \land value\ min = 0 \\
\quad\quad \textbf{then}\ \boldsymbol{z} \\
\quad \textbf{else if}\ \#g = 0 \\
\quad\quad \textbf{then}\ \varnothing \\
\quad \textbf{else if}\ (\#min = 1 \land \#g < value\ min) \lor \\
\quad\quad\quad (\#max = 1 \land \#g > value\ max) \\
\quad\quad \textbf{then}\ \boldsymbol{f} \\
\quad \textbf{else}\ \boldsymbol{p}
\end{array}$$

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

$$\begin{array}{l}
evalObjectSpecification : EvalContext \rightarrow ObjectSpecification \rightarrow Graph \rightarrow \\
\quad OptValidity
\end{array}$$

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

$$\begin{array}{l}
evalObjectSpecificationTriple : EvalContext \rightarrow ObjectSpecification \rightarrow \\
\quad RDFTerm \rightarrow OptValidity
\end{array}$$

$$\begin{array}{l}
\forall\, ec : EvalContext;\ os : ObjectSpecification;\ n : RDFTerm \bullet \\
evalObjectSpecificationTriple\ ec\ os\ n = \\
\quad \textbf{if}\ os \in \operatorname{ran} valueType\ \textbf{then} \\
\quad\quad evalValueType\ (valueType^{\sim}\, os)\ n \\
\quad \textbf{else if}\ os \in \operatorname{ran} valueSet\ \textbf{then} \\
\quad\quad evalTermSet\ (valueSet^{\sim}\, os)\ n \\
\quad \textbf{else if}\ os \in \operatorname{ran} osWild\ \textbf{then} \\
\quad\quad evalTermWild\ (osWild^{\sim}\, os)\ n \\
\quad \textbf{else} \\
\quad\quad evalTermReference\ ec\ (valueRef^{\sim}\, os)\ n
\end{array}$$

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

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

$$
\begin{array}{|l}
evalValueType : IRI \rightarrowtail\!\!\!\!\!+ RDFTerm \rightarrowtail\!\!\!\!\!+ OptValidity \\
\hline
\forall vt : IRI;\ n : RDFTerm;\ l : RDFLiteral \bullet evalValueType\ vt\ n = \\
\quad \textbf{if } vt = SHEX\_IRI \land n \in \operatorname{ran} iri \textbf{ then } \boldsymbol{p} \\
\quad \textbf{else if } vt = SHEX\_BNODE \land n \in \operatorname{ran} bnode \textbf{ then } \boldsymbol{p} \\
\quad \textbf{else if } vt = SHEX\_NONLITERAL \land (n \in \operatorname{ran} iri \lor n \in \operatorname{ran} bnode) \textbf{ then } \boldsymbol{p} \\
\quad \textbf{else if } vt = SHEX\_LITERAL \land n \in \operatorname{ran} literal \textbf{ then } \boldsymbol{p} \\
\quad \textbf{else if } n \in \operatorname{ran} literal \land l = (literal^{\sim} n) \land \\
\qquad ((l \in \operatorname{ran} pl \land (pl^{\sim} l).dataType = vt) \lor \\
\qquad (l \in \operatorname{ran} tl \land (tl^{\sim} l).dataType = vt)) \textbf{ then } \boldsymbol{p} \\
\quad \textbf{else } \boldsymbol{f}
\end{array}
$$

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

$$
\begin{array}{|l}
evalTermSet : \mathbb{P}\, MatchValue \rightarrowtail\!\!\!\!\!+ RDFTerm \rightarrowtail\!\!\!\!\!+ OptValidity \\
\hline
\forall mvs : \mathbb{P}\, MatchValue;\ n : RDFTerm \bullet evalTermSet\ mvs\ n = \\
\quad \textbf{if } \exists\, mv : mvs \bullet \\
\qquad ((mv \in \operatorname{ran} mviri \land n \in \operatorname{ran} iri \land (iri^{\sim} n) = mviri^{\sim} mv) \lor \\
\qquad (mv \in \operatorname{ran} mviris \land n \in \operatorname{ran} iri \land \\
\qquad\quad (iri^{\sim} n) \in IRIstemRange\,(mviris^{\sim} mv)) \lor \\
\qquad (n \in \operatorname{ran} literal \land mvlit^{\sim} mv = literal^{\sim} n)) \\
\quad \textbf{then } \boldsymbol{p} \\
\quad \textbf{else } \boldsymbol{f}
\end{array}
$$

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

$$
\begin{array}{|l}
evalTermWild : \mathbb{P}\, MatchValue \rightarrow RDFTerm \rightarrow OptValidity \\
\hline
\forall mvs : \mathbb{P}\, MatchValue;\ n : RDFTerm \bullet evalTermWild\ mvs\ n = \\
\quad \textbf{if } evalTermSet\ mvs\ n = \boldsymbol{p} \textbf{ then } \boldsymbol{f} \textbf{ else } \boldsymbol{p}
\end{array}
$$

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

$$
\begin{array}{|l}
evalTermReference : EvalContext \rightarrowtail\!\!\!\!\!+ Label \rightarrowtail\!\!\!\!\!+ RDFTerm \rightarrowtail\!\!\!\!\!+ OptValidity \\
\hline
\forall ec : EvalContext;\ vr : Label;\ n : RDFTerm \bullet \\
evalTermReference\ ec\ vr\ n = \\
\quad \textbf{if } n \notin \operatorname{ran} literal \textbf{ then } evalRule'\ ec\ n\ vr \\
\quad \textbf{else } \boldsymbol{f}
\end{array}
$$

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*:

$$
\begin{array}{|l}
\textit{RevArcRule} \\
\hline
\textit{filter} : \textit{PredicateFilter} \\
\textit{valueSpec} : \textit{SubjectSpecification} \\
\textit{min}, \textit{max} : \textit{Optional}[\mathbb{N}] \\
\textit{actions} : \mathbb{P}\ \textit{Action} \\
\hline
(\#\textit{min} = 1 \land \#\textit{max} = 1) \Rightarrow \textit{value min} \leq \textit{value max} \\
\hline
\end{array}
$$

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* ($\boldsymbol{f}$).
4. Return the result of evaluating *actions* against the matching triples.

$$
\begin{array}{|l}
\textit{evalRevArcRule} : \textit{EvalContext} \rightarrow \textit{RDFTerm} \rightarrow \textit{RevArcRule} \rightarrow \textit{OptValidity} \\
\hline
\forall\, \textit{ec} : \textit{EvalContext};\ o : \textit{RDFTerm};\ \textit{rar} : \textit{RevArcRule};\ \textit{og} : \textit{Graph} \mid \\
\quad \textit{og} = \textit{evalPredicateFilter rar.filter}\ (\textit{triplesForObject o ec.graph}) \bullet \\
\textit{evalRevArcRule ec o rar} = \\
\quad \textbf{if}\ \textit{evalCardinality og rar.min rar.max} \neq \boldsymbol{p} \\
\quad\quad \textbf{then}\ \textit{evalCardinality og rar.min rar.max} \\
\quad \textbf{else if}\ \textit{evalSubjectSpecification ec rar.valueSpec og} \neq \boldsymbol{p} \\
\quad\quad \textbf{then}\ \textit{evalSubjectSpecification ec rar.valueSpec og} \\
\quad \textbf{else} \\
\quad\quad \textit{dispatch rar.actions og ec}
\end{array}
$$

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*
- *subjectRef* - matches any target that is considered valid according the *Rule* identified by *Label*.

$$
\begin{array}{l}
\textit{SubjectSpecification} ::= \textit{subjectSet}\langle\!\langle \mathbb{P}\ \textit{MatchValue} \rangle\!\rangle \mid \\
\quad \textit{ssWild}\langle\!\langle \mathbb{P}\ \textit{MatchValue} \rangle\!\rangle \mid \\
\quad \textit{subjectRef}\langle\!\langle \textit{Label} \rangle\!\rangle
\end{array}
$$

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

$$evalSubjectSpecification : EvalContext \rightarrow SubjectSpecification \rightarrow Graph \rightarrow OptValidity$$

$$\begin{array}{l} \forall\, ec : EvalContext;\ ss : SubjectSpecification;\ g : Graph \bullet \\ evalSubjectSpecification\ ec\ ss\ g = \\ \quad \textbf{if}\ \forall\, t : g \bullet evalSubjectSpecificationTriple\ ec\ ss\ t.o = \boldsymbol{p}\ \textbf{then}\ \boldsymbol{p} \\ \quad \textbf{else}\ \boldsymbol{f} \end{array}$$

$$evalSubjectSpecificationTriple : EvalContext \rightarrow SubjectSpecification \rightarrow RDFTerm \rightarrow OptValidity$$

$$\begin{array}{l} \forall\, ec : EvalContext;\ ss : SubjectSpecification;\ n : RDFTerm \bullet \\ evalSubjectSpecificationTriple\ ec\ ss\ n = \\ \quad \textbf{if}\ ss \in \operatorname{ran} subjectSet\ \textbf{then} \\ \quad\quad evalTermSet\ (subjectSet^{\sim}\, ss)\ n \\ \quad \textbf{else if}\ ss \in \operatorname{ran} ssWild\ \textbf{then} \\ \quad\quad evalTermWild\ (ssWild^{\sim}\, ss)\ n \\ \quad \textbf{else} \\ \quad\quad evalTermReference\ ec\ (subjectRef^{\sim}\, ss)\ n \end{array}$$

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 *none* ($\varnothing$), (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* ($\boldsymbol{p}$).

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

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

$$evalGroupRule : EvalContext \rightarrow RDFTerm \rightarrow GroupRule \rightarrow OptValidity$$

$$\begin{array}{l} \forall\, ec : EvalContext;\ i : RDFTerm;\ gr : GroupRule \bullet evalGroupRule\ ec\ i\ gr = \\ \quad \textbf{if}\ evalRule'\ ec\ i\ gr.rule = \varnothing \wedge gr.opt = OPTIONAL \\ \quad\quad \textbf{then}\ \boldsymbol{z} \\ \quad \textbf{else if}\ evalRule'\ ec\ i\ gr.rule = \varepsilon \wedge gr.opt = OPTIONAL \\ \quad\quad \textbf{then}\ \boldsymbol{f} \\ \quad \textbf{else if}\ evalRule'\ ec\ i\ gr.rule = \boldsymbol{p} \\ \quad\quad \textbf{then}\ dispatch\ gr.actions\ \emptyset\ ec \\ \quad \textbf{else}\ evalRule'\ ec\ i\ gr.rule \end{array}$$

4.4 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.

- If either term is an error the result is an error
- else if either term is a fail the result is a fail
- else if both terms are the same, the result is whatever they were
- else none ($\varnothing$) and nomatch ($\boldsymbol{z}$) is nomatch ($\boldsymbol{z}$)
- nomatch ($\boldsymbol{z}$) and pass ($\boldsymbol{p}$) is fail ($\boldsymbol{f}$)
- none ($\varnothing$) and pass ($\boldsymbol{p}$) is pass($\boldsymbol{p}$)

$$\mathit{And} : \mathit{OptValidity} \to \mathit{OptValidity} \to \mathit{OptValidity}$$

$$\begin{array}{l}
\forall\, a1, a2 : \mathit{OptValidity} \bullet \mathit{And}\ a1\ a2 = \\
\quad \textbf{if}\ a1 = \varepsilon \lor a2 = \varepsilon\ \textbf{then}\ \varepsilon \\
\quad \textbf{else if}\ a1 = \boldsymbol{f} \lor a2 = \boldsymbol{f}\ \textbf{then}\ \boldsymbol{f} \\
\quad \textbf{else if}\ a1 = a2\ \textbf{then}\ a1 \\
\quad \textbf{else if}\ a1 = \varnothing\ \textbf{then} \\
\qquad \textbf{if}\ a2 = \boldsymbol{z}\ \textbf{then}\ \boldsymbol{z}\ \textbf{else}\ \boldsymbol{f} \\
\quad \textbf{else if}\ a1 = \varnothing\ \textbf{then} \\
\qquad \textbf{if}\ a2 = \boldsymbol{z}\ \textbf{then}\ \boldsymbol{z}\ \textbf{else}\ \boldsymbol{p} \\
\quad \textbf{else if}\ a2 = \boldsymbol{z}\ \textbf{then}\ \boldsymbol{f}\ \textbf{else}\ \boldsymbol{p}
\end{array}$$

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:

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

$$\mathit{evalAndRule} : \mathit{EvalContext} \to \mathit{RDFTerm} \to \mathit{AndRule} \to \mathit{OptValidity}$$

$$\begin{array}{l}
\forall\, ec : \mathit{EvalContext};\ i : \mathit{RDFTerm};\ r : \mathit{AndRule} \bullet \\
\mathit{evalAndRule}\ ec\ i\ r = \\
\quad \mathit{foldr}\ \mathit{And}\ \boldsymbol{z}\ (\mathit{map}\ (\mathit{evalRule}'\ ec\ i)\ r)
\end{array}$$

4.5 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:

- If either term is fail ($\boldsymbol{f}$) the result is the other term *Identity*
- else if either term is error (ε) the result is (ε) *unity*
- else if both terms are pass ($\boldsymbol{p}$) the result is (ε)
- else if either term is pass ($\boldsymbol{p}$) the result is ($\boldsymbol{p}$)
- else if either term is nomatch ($\boldsymbol{z}$) the result is ($\boldsymbol{z}$)
- else the result is none ($\varnothing$)

$$
\begin{array}{|l}
Xor : OptValidity \to OptValidity \to OptValidity \\
\hline
\forall\, o1, o2 : OptValidity \bullet Xor\ o1\ o2 = \\
\quad \textbf{if } o1 = \varepsilon \lor o2 = \varepsilon \lor (o1 = \boldsymbol{p} \land o2 = \boldsymbol{p}) \textbf{ then } \varepsilon \\
\quad \textbf{else if } o1 = \boldsymbol{p} \lor o2 = \boldsymbol{p} \textbf{ then } \boldsymbol{p} \\
\quad \textbf{else if } o1 = \boldsymbol{f} \textbf{ then } o2 \\
\quad \textbf{else if } o2 = \boldsymbol{f} \textbf{ then } o1 \\
\quad \textbf{else if } o1 = \boldsymbol{z} \lor o2 = \boldsymbol{z} \textbf{ then } \boldsymbol{z} \\
\quad \textbf{else } \varnothing
\end{array}
$$

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

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

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

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* 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 ($\boldsymbol{p}$) 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.

$[Action]$

$$dispatch : \mathbb{P}\, Action \rightarrow Graph \rightarrow EvalContext \rightarrow OptValidity$$

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

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.

$$[T]$$

$$foldr : (T \rightarrow T \rightarrow T) \rightarrow T \rightarrow \mathrm{seq}\, T \rightarrow T$$

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

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

$$[A, B]$$

$$map : (A \rightarrow B) \rightarrow \mathrm{seq}\, A \rightarrow \mathrm{seq}\, B$$

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

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 $\sim$ 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:

$$\begin{array}{l} x : TravelDirections \\ \hline \textbf{if } x \in \operatorname{ran} bus \textbf{ then } bus^{\sim} x \textbf{ else } walking^{\sim} x \end{array}$$

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:

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

And a shorthand for addressing the content:

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