Z Specification for the W3C Editor's Draft Core SHACL Semantics

Arthur Ryman, arthur.ryman@gmail.com

November 10, 2015

Abstract

This article provides a formalization of the W3C Draft Core SHACL Semantics specification using Z notation. This formalization exercise has identified a number of quality issues in the draft. It has also established that the recursive definitions in the draft are well-founded. Further formal validation of the draft will require the use of an executable specification technology.

1 Introduction

The W3C RDF Data Shapes Working Group [3] is developing SHACL, a new language for describing constraints on RDF graphs. A semantics for Core SHACL has been proposed [2], hereafter referred to as the *semantics draft*. The proposed semantics includes an abstract syntax, inference rules, and a definition of typing which allows for certain kinds of recursion. The semantics draft uses precise mathematical language, but is informal in the sense that it is not written in a formal specification language and therefore cannot benefit from tools such as type-checkers.

This document provides a formal translation of the semantics draft into Z Notation [6]. The LaTeX source for this article has been type-checked using the *f*UZZ type-checker [7] and is available in the GitHub repository [4] `agryman/z-core-shacl-semantics`.

Our motive for formalizing and type-checking the semantics draft is to help to improve its quality and the ultimate design of SHACL.

1.1 Organization of this Article

The remainder of this article is organized as follows.

- Section 2 formalizes some basic RDF concepts.
- Section 3 translates the abstract syntax of SHACL into Z notation.
- Section 4 formalizes the evaluation semantics of SHACL.

- Section 5 formalizes the declarative semantics of shape expression schemas.
- Section 6 summarizes the quality issues found in the draft.
- Section 7 concludes with some remarks about the benefits of the formalization exercise and possible next steps.

2 Basic RDF Concepts

This section formalizes some basic RDF concepts. We reuse some formal definitions given in [5], modifying the identifiers to match those used in the semantics draft.

2.1 *TERM*

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

$$[\mathit{TERM}]$$

2.2 *Iri*, *Blank*, and *Lit*

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

$$\begin{array}{|l} \mathit{Iri}, \mathit{Blank}, \mathit{Lit} : \mathbb{P}\ \mathit{TERM} \\ \hline \langle \mathit{Iri}, \mathit{Blank}, \mathit{Lit} \rangle\ \mathsf{partition}\ \mathit{TERM} \end{array}$$

2.3 *IRI*

The semantics draft introduces the term *Iri*, but it uses the term *IRI* in the definitions of the abstract syntax. We treat *IRI* as a synonym for *Iri*.

$$\mathit{IRI} == \mathit{Iri}$$

2.4 *Triple*

An RDF triple is an ordered triple of RDF terms referred to as the subject, predicate, and object.

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

- The subject is not a literal.
- The predicate is an IRI.

2.5 *subject*, *predicate*, and *object*

It is convenient to define generic functions that select the first, second, or third component of a Cartesian product of three sets.

$$fst[X, Y, Z] == (\lambda\, x : X;\ y : Y;\ z : Z \bullet x\,)$$

$$snd[X, Y, Z] == (\lambda\, x : X;\ y : Y;\ z : Z \bullet y\,)$$

$$trd[X, Y, Z] == (\lambda\, x : X;\ y : Y;\ z : Z \bullet z\,)$$

The subject, predicate, and object of an RDF triple are the terms that appear in the corresponding positions.

$$subject == (\lambda\, t : Triple \bullet fst(t)\,)$$

$$predicate == (\lambda\, t : Triple \bullet snd(t)\,)$$

$$object == (\lambda\, t : Triple \bullet trd(t)\,)$$

2.6 *Graph*

An RDF graph is a finite set of RDF triples.

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

2.7 *subjects*, *predicates*, and *objects*

The subjects, predicates, and objects of a graph are the sets of RDF terms that appear in the corresponding positions of its triples.

$$subjects == (\lambda\, g : Graph \bullet \{\, t : g \bullet subject(t)\,\}\,)$$

$$predicates == (\lambda\, g : Graph \bullet \{\, t : g \bullet predicate(t)\,\}\,)$$

$$objects == (\lambda\, g : Graph \bullet \{\, t : g \bullet object(t)\,\}\,)$$

2.8 *nodes*

The nodes of an RDF are its subjects and objects.

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

2.9 *PointedGraph*

A pointed graph is a graph and a distinguished node in the graph. The distinguished node is variously referred to as the start, base, or focus node of the pointed graph, depending on the context.

$$PointedGraph == \{\, g : Graph;\ n : TERM \mid n \in nodes(g)\,\}$$

3 Abstract Syntax

This section contains a translation of the abstract syntax of SHACL into Z. The semantics draft defines the abstract syntax using an informal Extended Backus-Naur Form (EBNF).

The approach used here is to interpret each term or expression that appears in the abstract syntax as a mathematical set that is isomorphic to the set of abstract syntax tree fragments denoted by the corresponding term or expression. Care has been taken to preserve the exact spelling and case of each abstract syntax term so that there is a direct correspondence between the abstract syntax and Z. For example, the term `Schema` is interpreted as the set *Schema*.

We give a Z definition for each abstract syntax term that appears on the left-hand side of the EBNF definition operator (`::=`). The order in which these terms appear in the semantics draft has been preserved in this document. If a Z term has a corresponding EBNF rule, we include it here for easy reference. Refer to [2] for the complete definition of the abstract syntax.

A sequence of two or more abstract syntax terms is interpreted as the Cartesian product of the corresponding sets, i.e. `A B` is interpreted as $A \times B$.

The abstract syntax Kleene star (`*`) and plus (`+`) operators are interpreted as sequence (seq) and non-empty sequence (seq_1) operators on the corresponding sets, i.e. `A+` is interpreted as $\mathrm{seq}_1 A$.

The abstract syntax optional operator (`?`) is interpreted as taking the union of the set of singletons and the empty set of the corresponding set using the generic function *OPTIONAL* (defined below), i.e. `A?` is interpreted as $OPTIONAL[A]$.

Abstract syntax terms that are defined as alternations (`|`) of two or more expressions are translated into either free types or unions of sets. A side effect of this process is that constructors may be required for each branch of the alternation. In some cases the name of the constructors can be derived from a corresponding element of the abstract syntax. For example, in `ShapeDefinition`, `open` and `close` are mapped to the constructors *open* and *close*. In the cases where there is no convenient element of the abstract syntax, we mint new constructor names.

We also introduce new Z identifiers when an element of the abstract syntax does not map to a valid alphanumeric Z identifier. For example the the shape label negation operator (`!`) is mapped to *negate*.

3.1 *OPTIONAL*

An optional value is represented by a singleton set, if the value is present, or the empty set, if the value is absent.

$$OPTIONAL[X] == \{ v : X \bullet \{v\} \} \cup \{\varnothing\}$$

3.2 *Schema*

```
Schema ::= Rule+
```

A schema is a sequence of one or more rules.

$$Schema == \mathrm{seq}_1\ Rule$$

3.3 *Rule*

```
Rule ::= ShapeLabel ShapeDefinition ExtensionCondition*
```

A rule consists of a shape label, a shape definition, and a sequence of zero or more extension conditions.

$$Rule == ShapeLabel \times ShapeDefinition \times \mathrm{seq}\ ExtensionCondition$$

It is convenient to introduce functions that select the components of a rule.

$$\begin{aligned} &shapeLabel == (\lambda\, r : Rule \bullet fst(r)\,) \\ &shapeDef == (\lambda\, r : Rule \bullet snd(r)\,) \\ &extConds == (\lambda\, r : Rule \bullet trd(r)\,) \end{aligned}$$

3.4 *ShapeLabel*

```
ShapeLabel ::= an identifier
```

A shape label is an identifier drawn from some given set.

$$[ShapeLabel]$$

3.5 *ShapeDefinition*

```
ShapeDefinition ::= ClosedShape | OpenShape
```

A shape definition is either a closed shape or an open shape.

$$\begin{aligned} &ShapeDefinition ::= \\ &\quad close\langle\!\langle ShapeExpr \rangle\!\rangle \mid \\ &\quad open\langle\!\langle OPTIONAL[InclPropSet] \times ShapeExpr \rangle\!\rangle \end{aligned}$$

Note that abstract syntax terms that are defined using alternation are naturally represented as free types in Z Notation.

- *close* is the constructor for closed shapes.A closed shape consists of a shape expression.
- *open* is the constructor for open shapes. An open shape consists of an optional included properties set and a shape expression.

Given a shape definition d, let $shapeExpr(d)$ be its shape expression.

$$\begin{array}{|l} shapeExpr : ShapeDefinition \longrightarrow ShapeExpr \\ \hline \forall\, x : ShapeExpr \bullet \\ \quad shapeExpr(close(x)) = x \\ \forall\, o : OPTIONAL[InclPropSet];\ x : ShapeExpr \bullet \\ \quad shapeExpr(open(o, x)) = x \end{array}$$

3.6 *ClosedShape*

```
ClosedShape ::= 'close' ShapeExpr
```

The set of closed shapes is the range of the *close* shape definition constructor.

$$ClosedShape == \operatorname{ran} close$$

3.7 *OpenShape*

```
OpenShape ::= 'open' InclPropSet? ShapeExpr
```

The set of open shapes is the range of the *open* shape definition constructor.

$$OpenShape == \operatorname{ran} open$$

3.8 *InclPropSet*

```
InclPropSet ::= PropertiesSet
```

An included properties set is a properties set.

$$InclPropSet == PropertiesSet$$

Note that there seems little motivation to introduce the term *InclPropSet* since it is identical to *PropertiesSet*.

3.9 *PropertiesSet*

```
PropertiesSet ::= set of IRI
```

A properties set is a set of IRIs.

$$PropertiesSet == \mathbb{P}\ IRI$$

3.10 *ShapeExpr*

```
ShapeExpr ::= EmptyShape
| TripleConstraint Cardinality
| InverseTripleConstraint Cardinality
| NegatedTripleConstraint
| NegatedInverseTripleConstraint
| SomeOfShape
| OneOfShape
| GroupShape
| RepetitionShape
```

A shape expression defines constraints on RDF graphs.

$$
\begin{aligned}
&\mathit{ShapeExpr} ::= \\
&\quad \mathit{emptyshape} \mid \\
&\quad \mathit{triple}\langle\!\langle \mathit{DirectedTripleConstraint} \times \mathit{Cardinality} \rangle\!\rangle \mid \\
&\quad \mathit{someOf}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeExpr} \rangle\!\rangle \mid \\
&\quad \mathit{oneOf}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeExpr} \rangle\!\rangle \mid \\
&\quad \mathit{group}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeExpr} \rangle\!\rangle \mid \\
&\quad \mathit{repetition}\langle\!\langle \mathit{ShapeExpr} \times \mathit{Cardinality} \rangle\!\rangle
\end{aligned}
$$

- *emptyshape* is the empty shape expression.
- *triple* is the constructor for triple constraint shape expressions. A triple constraint shape expression consists of a directed triple constraint and a cardinality.
- *someOf* is the constructor for some-of shape expressions. A some-of shape expression consists of a sequence of one or more shape expressions.
- *oneOf* is the constructor for one-of shape expressions. A one-of shape expression consists of a sequence of one or more shape expressions.
- *group* is the constructor for grouping shape expressions. A grouping shape expression consists of a sequence of one or more shape expressions.
- *repetition* is the constructor for repetition shape expressions. A repetition shape expression consists of a shape expression and a cardinality.

3.11 *EmptyShape*

```
EmptyShape ::= 'emptyshape'
```

The set of empty shape expressions is the singleton set that contains the empty shape.

$$\mathit{EmptyShape} == \{\mathit{emptyshape}\}$$

3.12 *DirectedPredicate*

A directed predicate is an IRI with a direction that indicates its usage in a triple. *nop* indicates the normal direction, namely the predicate relates the subject node to the object node. *inv* indicates the inverse direction, namely the predicate relates the object node to the subject node.

$$
\begin{aligned}
&\mathit{DirectedPredicate} ::= \\
&\quad \mathit{nop}\langle\!\langle \mathit{IRI} \rangle\!\rangle \mid \\
&\quad \mathit{inv}\langle\!\langle \mathit{IRI} \rangle\!\rangle
\end{aligned}
$$

The semantics draft uses the notation `^p` for $\mathit{inv}(p)$.

Let $predDF(dp)$ denote the predicate of a directed predicate dp.

$$
\begin{array}{|l}
predDP : DirectedPredicate \longrightarrow IRI \\
\hline
\forall\, p : IRI \bullet \\
\qquad predDP(nop(p)) = predDP(inv(p)) = p
\end{array}
$$

3.13 *DirectedTripleConstraint*

A directed triple constraint consists of a directed predicate and a constraint. The constraint is a value or shape constraint on the object of a triple if the direction is normal, or a shape constraint on the subject of a triple if the direction is inverted.

$$
\begin{array}{l}
DirectedTripleConstraint == \\
\quad \{\, dp : DirectedPredicate;\ C : Constraint \mid \\
\qquad dp \in \operatorname{ran} inv \Rightarrow C \in ShapeConstr \,\}
\end{array}
$$

The semantics draft uses the notation `p::C` for $(nop(p), C)$ and `^p::C` for $(inv(p), C)$.

Let $predDTC(dtc)$ denote the predicate of the directed triple constraint dtc.

$$
\begin{array}{|l}
predDTC : DirectedTripleConstraint \longrightarrow IRI \\
\hline
\forall\, dp : DirectedPredicate;\ C : Constraint \mid \\
\qquad (dp, C) \in DirectedTripleConstraint \bullet \\
\qquad\qquad predDTC(dp, C) = predDP(dp)
\end{array}
$$

Let $constrDTC(dtc)$ denote the constraint of the directed triple constraint dtc.

$$
\begin{array}{|l}
constrDTC : DirectedTripleConstraint \longrightarrow Constraint \\
\hline
\forall\, dp : DirectedPredicate;\ C : Constraint \mid \\
\qquad (dp, C) \in DirectedTripleConstraint \bullet \\
\qquad\qquad constrDTC(dp, C) = C
\end{array}
$$

3.14 *TripleConstraint*

```
TripleConstraint ::= IRI ValueConstr | IRI ShapeConstr
```

A triple constraint places conditions on triples whose subject is a given focus node and whose predicate is a given IRI.

$$
\begin{array}{|l}
TripleConstraint : \mathbb{P}\, DirectedTripleConstraint \\
\hline
TripleConstraint = \\
\qquad \{\, p : IRI;\ C : Constraint \bullet (nop(p), C) \,\}
\end{array}
$$

3.15 *InverseTripleConstraint*

```
InverseTripleConstraint ::= '^' IRI ShapeConstr
```

An inverse triple constraint places conditions on triples whose object is a given focus node and whose predicate is a given IRI.

$$
\begin{array}{|l}
\mathit{InverseTripleConstraint} : \mathbb{P}\ \mathit{DirectedTripleConstraint} \\
\hline
\mathit{InverseTripleConstraint} = \\
\quad \{\, p : \mathit{IRI};\ C : \mathit{ShapeConstr} \bullet (\mathit{inv}(p), C) \,\}
\end{array}
$$

3.16 *Constraint*

A constraint is a condition on the object node of a triple for normal predicates or the subject node of a triple for inverse predicates.

$$
\begin{aligned}
&\mathit{Constraint} ::= \\
&\quad \mathit{valueSet}\langle\!\langle \mathbb{P}(\mathit{Lit} \cup \mathit{IRI}) \rangle\!\rangle \mid \\
&\quad \mathit{datatype}\langle\!\langle \mathit{LiteralDatatype} \times \mathit{OPTIONAL}[\mathit{XSFacet}] \rangle\!\rangle \mid \\
&\quad \mathit{kind}\langle\!\langle \mathit{NodeKind} \rangle\!\rangle \mid \\
&\quad \mathit{or}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeLabel} \rangle\!\rangle \mid \\
&\quad \mathit{and}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeLabel} \rangle\!\rangle \mid \\
&\quad \mathit{nor}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeLabel} \rangle\!\rangle \mid \\
&\quad \mathit{nand}\langle\!\langle \mathrm{seq}_1\ \mathit{ShapeLabel} \rangle\!\rangle
\end{aligned}
$$

- *valueSet* is the constructor for value set value constraints. A value set value constraint consists of a set of literals and IRIs.
- *datatype* is the constructor for literal datatype value constraints. A literal datatype value constraint consists of a literal datatype and an optional XML Schema facet.
- *kind* is the constructor for node kind value constraints. A node kind value constraint consists of a specification for a subset of RDF terms.
- *or* is the constructor for disjunction shape constraints. A node must satisfy at least one of the shapes.
- *and* is the constructor for conjunction shape constraints. A node must satisfy all of the shapes.
- *nor* is the constructor for negated disjunction shape constraints. A node must not satisfy any of the shapes.
- *nand* is the constructor for negated conjunction shape constraints. A node must not satisfy all of the shapes.

3.17 *Cardinality*

```
Cardinality ::= '[' MinCardinality ';' MaxCardinality ']'
```

Cardinality defines a range for the number of elements in a set.

$$Cardinality == MinCardinality \times MaxCardinality$$

- A cardinality consists of a minimum cardinality and a maximum cardinality.

3.18 *MinCardinality*

```
MinCardinality ::= a natural number
```

Minimum cardinality is the minimum number of elements required to be in a set.

$$MinCardinality == \mathbb{N}$$

3.19 *MaxCardinality*

```
MaxCardinality ::= a natural number | 'unbound'
```

Maximum cardinality is the maximum number of elements required to be in a set.

$$MaxCardinality ::= maxCard\langle\langle \mathbb{N} \rangle\rangle \mid unbound$$

- *maxCard* is the constructor for finite maximum cardinalities. A finite maximum cardinality is a natural number. Note that a maximum cardinality of 0 means that the set must be empty.
- *unbound* indicates that the maximum number of elements in a set is unbounded.

3.20 *inBounds*

A natural number k is said to be in bounds of a cardinality when k is between the minimum and maximum limits of the cardinality.

$$inBounds : \mathbb{N} \leftrightarrow Cardinality$$

$$\forall\, k, n : \mathbb{N} \bullet$$
$$\quad k\ \underline{inBounds}\ (n, unbound) \Leftrightarrow n \leq k$$

$$\forall\, k, n, m : \mathbb{N} \bullet$$
$$\quad k\ \underline{inBounds}\ (n, maxCard(m)) \Leftrightarrow n \leq k \leq m$$

3.21 Notation

Let a be an IRI, let C be a value or shape constraint, let n and m be non-negative integers. The semantics draft uses the notation listed in Table 1 for some shape expressions.

Notation	Meaning
`a::C[n;m]`	$triple(nop(a, C), (n, maxCard(m)))$
`^a::C[n;m]`	$triple(inv(a, C), (n, maxCard(m)))$
`a::C`	`a::C[1;1]`
`^a::C`	`^a::C[1;1]`
`!a::C`	`a::C[0;0]`
`!^a:C`	`^a::C[0;0]`

Table 1: Meaning of shape expression notation

- If the cardinality is `[1;1]` it may be omitted.
- The negated shape expressions are semantically equivalent to the corresponding non-negated shape expressions with cardinality `[0;0]`.

3.22 *none*, *one*

It is convenient to define some common cardinalities.

$$none == (0, maxCard(0))$$

$$one == (1, maxCard(1))$$

- A cardinality of $none$ = `[0;0]` is used to indicate a negated triple or inverse triple constraint.
- A cardinality of one = `[1;1]` is the default cardinality of a triple or inverse triple constraint when no cardinality is explicitly given in the notations `a::C` and `^a::C`.

3.23 *NegatedTripleConstraint*

```
NegatedTripleConstraint ::= '!' TripleConstraint
```

A negated triple constraint shape expression is a triple constraint shape expression that has a cardinality of *none*.

$$\begin{aligned}&NegatedTripleConstraint == \\ &\quad \{\, tc : TripleConstraint \bullet triple(tc, none) \,\}\end{aligned}$$

3.24 *NegatedInverseTripleConstraint*

```
NegatedInverseTripleConstraint ::= '!' InverseTripleConstraint
```

A negated inverse triple constraint shape expression is an inverse triple constraint shape expression that has a cardinality of *none*.

$$NegatedInverseTripleConstraint == \\ \{ itc : InverseTripleConstraint \bullet triple(itc, none) \}$$

3.25 *ValueConstr*

```
ValueConstr ::= ValueSet | LiteralDatatype XSFacet? | NodeKind
```

A value constraint places conditions on the object nodes of triples for normal predicates and on the subject nodes of triples for inverse predicates.

$$ValueConstr == \operatorname{ran} valueSet \cup \operatorname{ran} datatype \cup \operatorname{ran} kind$$

3.26 *ValueSet*

```
ValueSet ::= set of literals and IRI
```

The set of value set value constraints is the range of the *valueSet* constructor.

$$ValueSet == \operatorname{ran} valueSet$$

3.27 *LiteralDatatype*

```
LiteralDatatype ::= an RDF literal datatype
```

A literal datatype is an IRI that identifies a set of literal RDF terms. We assume that this subset of IRIs is given.

$$LiteralDatatype : \mathbb{P}\, IRI$$

We also assume that we are given an interpretation of each literal datatype as a set of literals.

$$literalsOfDatatype : LiteralDatatype \longrightarrow \mathbb{P}\, Lit$$

3.28 *NodeKind*

```
NodeKind ::= 'iri' | 'blank' | 'literal' | 'nonliteral'
```

A node kind identifies a subset of RDF terms.

$$NodeKind ::= iri \mid blank \mid literal \mid nonliteral$$

- *iri* identifies the set of IRIs.

- *blank* identifies the set of blank nodes.
- *literal* identifies the set of literals.
- *nonliteral* identifies the complement of the set of literals, i.e. the union of IRIs and blank nodes.

Each node kind corresponds to a set of RDF terms.

$$
\begin{array}{|l}
termsOfKind : NodeKind \longrightarrow \mathbb{P}\ TERM \\
\hline
termsOfKind(iri) = IRI \\
termsOfKind(blank) = Blank \\
termsOfKind(literal) = Lit \\
termsOfKind(nonliteral) = TERM \setminus Lit
\end{array}
$$

3.29 *XSFacet*

`XSFacet ::= an XSD restriction`

An XML Schema facet places restrictions on literals. We assume this is a given set.

$[XSFacet]$

We also assume that we are given an interpretation of facets as sets of literals.

$$
\begin{array}{|l}
literalsOfFacet : LiteralDatatype \times XSFacet \longrightarrow \mathbb{P}\ Lit \\
\hline
\forall\, d : LiteralDatatype; f : XSFacet \bullet \\
\qquad literalsOfFacet(d, f) \subseteq literalsOfDatatype(d)
\end{array}
$$

- The literals that correspond to a facet of a datatype are a subset of the literals that correspond to the datatype.

3.30 *ShapeConstr*

`ShapeConstr ::= ('!')? DisjShapeConstr | ConjShapeConstraint`

A shape constraint requires that a node satisfy logical combinations of one or more other shapes which are identified by their shape labels.

$$ShapeConstr == \operatorname{ran} or \cup \operatorname{ran} and \cup \operatorname{ran} nor \cup \operatorname{ran} nand$$

3.31 *DisjShapeConstr*

`DisjShapeConstr ::= ShapeLabel ('or' ShapeLabel)*`

The set of all disjunctive shape constraints is the range of the *or* constructor.

$$DisjShapeConstr == \operatorname{ran} or$$

3.32 *ConjShapeConstraint*

```
ConjShapeConstraint ::= ShapeLabel ('and' ShapeLabel)*
```

The set of all conjunctive shape constraints is the range of the *and* constructor.

$$ConjShapeConstraint == \operatorname{ran} and$$

3.33 *SomeOfShape*

```
SomeOfShape ::= ShapeExpr ('|' ShapeExpr)*
```

The set of some-of shape expressions is the range of *someOf*.

$$SomeOfShape == \operatorname{ran} someOf$$

3.34 *OneOfShape*

```
OneOfShape ::= ShapeExpr ('@' ShapeExpr)*
```

The set of one-of shape expressions is the range of *oneOf*.

$$OneOfShape == \operatorname{ran} oneOf$$

3.35 *GroupShape*

```
GroupShape ::= ShapeExpr (',' ShapeExpr)*
```

The set of grouping shape expressions is the range of *group*.

$$GroupShape == \operatorname{ran} group$$

3.36 *RepetitionShape*

```
RepetitionShape ::= ShapeExpr Cardinality
```

The set of repetition shape expressions is the range of *repetition*.

$$RepetitionShape == \operatorname{ran} repetition$$

3.37 *ExtensionCondition*

```
ExtensionCondition ::= ExtLangName ExtDefinition
```

An extension condition is the definition of a constraint written in an extension language

$$ExtensionCondition == ExtLangName \times ExtDefinition$$

3.38 *ExtLangName*

`ExtLangName ::= an identifier`

An extension language name is an identifier for an extension language, such as JavaScript. We assume this is a given set.

$$[\mathit{ExtLangName}]$$

3.39 *ExtDefinition*

`ExtDefinition ::= a string`

An extension definition is a program written in some extension language that implements a constraint check. We assume this is a given set.

$$[\mathit{ExtDefinition}]$$

An extension condition represents a function that takes as input a pointed graph, and returns as output a boolean with the value `true` if the constraint is violated and `false` if satisfied. We assume we are given a mapping that associates each extension condition with the set of pointed graphs that violate it.

$$\mathit{violatedBy} : \mathit{ExtensionCondition} \longrightarrow \mathbb{P}\ \mathit{PointedGraph}$$

3.40 *ShapeLabel* Definitions

Given a schema S, let $\mathit{defs}(S)$ be the set of all shape labels defined in S.

$$\begin{array}{l} \mathit{defs} == (\lambda\, S : \mathit{Schema} \bullet \\ \qquad \{\, r : \operatorname{ran} S \bullet \mathit{shapeLabel}(r) \,\}\,) \end{array}$$

Each rule in a schema must be identified by a unique shape label.

$$\mathit{SchemaUL} == \{\, S : \mathit{Schema} \mid \#S = \#(\mathit{defs}(S)) \,\}$$

- In a schema with unique rule labels there are as many rules as labels.

3.41 *rule*

Given a schema S with unique rule labels, and a label T defined in S, let $\mathit{rule}(T, S)$ be the corresponding rule.

$$\begin{array}{l} \mathit{rule} : \mathit{ShapeLabel} \times \mathit{SchemaUL} \rightarrowtail\!\!\!\!\!\!\to \mathit{Rule} \\ \hline \operatorname{dom} \mathit{rule} = \{\, T : \mathit{ShapeLabel};\ S : \mathit{SchemaUL} \mid T \in \mathit{defs}(S) \,\} \\ \\ \forall\, S : \mathit{SchemaUL} \bullet \\ \qquad \forall\, r : \operatorname{ran}(S) \bullet \\ \qquad\qquad \mathbf{let}\ T == \mathit{shapeLabel}(r) \bullet \\ \qquad\qquad\qquad \mathit{rule}(T, S) = r \end{array}$$

3.42 *ShapeLabel* References

Given a schema S, let $refs(S)$ be the set of shape labels referenced in S.

$$refs == (\lambda\, S : Schema \bullet \bigcup \{\, r : \operatorname{ran} S \bullet refsRule(r) \,\}\,)$$

- The set of references in a schema is the union of the sets of references in its rules.

Given a rule r, let $refsRule(r)$ be the set of shape labels referenced in r.

$$refsRule == (\lambda\, r : Rule \bullet refsShapeDefinition(shapeDef(r))\,)$$

- The set of references in a rule is the set of references in its shape definition.

Given a shape definition d, let $refsShapeDefinition(d)$ be the set of shape labels referenced in d.

$$\begin{array}{|l}
refsShapeDefinition : ShapeDefinition \longrightarrow \mathbb{F}\, ShapeLabel \\
\hline
\forall\, d : ShapeDefinition \bullet \\
\quad refsShapeDefinition(d) = refsShapeExpr(shapeExpr(d))
\end{array}$$

- The set of references in a shape definition is the set of references in its shape expression.

Given a shape expression x, let $refsShapeExpr(x)$ be the set of shape labels referenced in x.

$$\begin{array}{|l}
refsShapeExpr : ShapeExpr \longrightarrow \mathbb{F}\, ShapeLabel \\
\hline
refsShapeExpr(emptyshape) = \varnothing \\
\forall\, dtc : DirectedTripleConstraint;\ c : Cardinality \bullet \\
\quad refsShapeExpr(triple(dtc, c)) = \\
\qquad refsDirectedTripleConstraint(dtc) \\
\forall\, xs : \operatorname{seq}_1 ShapeExpr \bullet \\
\quad refsShapeExpr(someOf(xs)) = \\
\quad refsShapeExpr(oneOf(xs)) = \\
\quad refsShapeExpr(group(xs)) = \\
\qquad \bigcup \{\, x : \operatorname{ran} xs \bullet refsShapeExpr(x) \,\} \\
\forall\, x : ShapeExpr;\ c : Cardinality \bullet \\
\quad refsShapeExpr(repetition(x, c)) = \\
\qquad refsShapeExpr(x)
\end{array}$$

- The empty shape expression references no labels.
- A directed triple constraint shape expression references the labels referenced in the directed triple constraint.

- A some-of or one-of or group shape expression references the union of the labels referenced in each component shape expression.
- A repetition shape expression references the labels referenced in its unrepeated shape expression.

Given a directed triple constraint dtc, let $refsDirectedTripleConstraint(dtc)$ be the set of shape labels referenced in dtc.

$$
\begin{array}{|l}
refsDirectedTripleConstraint : \\
\quad DirectedTripleConstraint \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall\, a : IRI;\ C : ValueConstr \bullet \\
\quad refsDirectedTripleConstraint((nop(a), C)) = \varnothing \\
\\
\forall\, a : IRI;\ C : ShapeConstr \bullet \\
\quad refsDirectedTripleConstraint((nop(a), C)) = \\
\quad refsDirectedTripleConstraint((inv(a), C)) = \\
\qquad refsShapeConstr(C)
\end{array}
$$

- A value triple constraint references no labels.
- A shape triple constraint references the labels in its shape constraint.

Given a shape constraint C, let $refsShapeConstr(C)$ be the set of shape labels referenced in C.

$$
\begin{array}{|l}
refsShapeConstr : ShapeConstr \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall\, ls : \mathrm{seq}_1\ ShapeLabel \bullet \\
\quad refsShapeConstr(or(ls)) = \\
\quad refsShapeConstr(and(ls)) = \\
\quad refsShapeConstr(nor(ls)) = \\
\quad refsShapeConstr(nand(ls)) = \\
\qquad \mathrm{ran}\ ls
\end{array}
$$

- A shape constraint references the range of its sequence of shape labels.

Every shape label referenced in a schema must be defined in the schema.

$$SchemaRD == \{ s : Schema \mid refs(s) \subseteq defs(s) \}$$

A schema is well-formed if its rules have unique labels and all referenced shape labels are defined.

$$SchemaWF == SchemaUL \cap SchemaRD$$

4 Evaluation

This section defines the interpretation of shapes as constraints on RDF graphs. All functions that are defined in the semantics draft are given formal definitions here. We assume that from this point on whenever the semantics draft refers to schemas they are well-formed.

4.1 *shapes*

Given a well-formed schema S, let $shapes(S)$ be the set of shape labels that appear in S.

$$shapes == (\lambda\, S : SchemaWF \bullet defs(S)\,)$$

4.2 *expr*

Given a shape label T and a well-formed schema S, let $expr(T, S)$ be the shape expression in the rule with label T in S.

$$
\begin{array}{|l}
expr : ShapeLabel \times SchemaWF \rightarrowtail\!\!\!\!\!+ \; ShapeExpr \\
\hline
\mathrm{dom}\, expr = \{\, T : ShapeLabel;\ S : SchemaWF \mid T \in shapes(S)\,\} \\
\forall\, T : ShapeLabel;\ S : SchemaWF \mid T \in shapes(S) \bullet \\
\quad \mathbf{let}\ r == rule(T, S) \bullet \\
\quad\quad expr(T, S) = shapeExpr(shapeDef(r))
\end{array}
$$

- The shape expression for a shape label T is the shape expression in the shape definition of the rule r that has shape label T.

4.3 *incl*

Given a shape label T defined in a well-formed schema S, let $incl(T, S)$ be the, possibly empty, set of included properties.

$$
\begin{array}{|l}
incl : ShapeLabel \times SchemaWF \rightarrowtail\!\!\!\!\!+ \; InclPropSet \\
\hline
\mathrm{dom}\, incl = \{\, T : ShapeLabel;\ S : SchemaWF \mid T \in shapes(S)\,\} \\
\forall\, T : ShapeLabel;\ S : SchemaWF \mid T \in shapes(S) \bullet \\
\quad \exists_1\, r : \mathrm{ran}\, S \mid T = shapeLabel(r) \bullet \\
\quad\quad incl(T, S) = inclShapeDefinition(shapeDef(r))
\end{array}
$$

- The included properties set for a shape label T is the included properties set in the shape definition of the rule r that has shape label T.

Given a shape definition d, let $inclShapeDefinition(d)$ be its included properties set.

$$
\begin{array}{|l}
inclShapeDefinition : ShapeDefinition \longrightarrow InclPropSet \\
\hline
\forall\, x : ShapeExpr \bullet \\
\quad inclShapeDefinition(close(x)) = \\
\quad inclShapeDefinition(open(\{\varnothing\}, x)) \\
\quad\quad = \varnothing \\
\forall\, ips : InclPropSet;\ x : ShapeExpr \bullet \\
\quad inclShapeDefinition(open(\{ips\}, x)) = ips
\end{array}
$$

- The included property set of a closed shape definition or an open definition with no included property set is the empty set.

- The included property set of an open shape definition with an included property set is that included property set.

4.4 *properties*

Given a shape expression x, let $properties(x)$ be the set of properties that appear in some triple constraint in x.

$$
\begin{array}{|l}
properties : ShapeExpr \longrightarrow PropertiesSet \\
\hline
properties(emptyshape) = \varnothing \\[1ex]
\forall\, tc : TripleConstraint;\ c : Cardinality \bullet \\
\quad properties(triple(tc, c)) = \\
\quad\quad propertiesTripleConstraint(tc) \\[1ex]
\forall\, itc : InverseTripleConstraint;\ c : Cardinality \bullet \\
\quad properties(triple(itc, c)) = \\
\quad\quad \varnothing \\[1ex]
\forall\, xs : \mathrm{seq}_1\ ShapeExpr \bullet \\
\quad properties(someOf(xs)) = \\
\quad properties(oneOf(xs)) = \\
\quad properties(group(xs)) = \\
\quad\quad \bigcup\{\, x : \mathrm{ran}\ xs \bullet properties(x)\,\} \\[1ex]
\forall\, x : ShapeExpr;\ c : Cardinality \bullet \\
\quad properties(repetition(x, c)) = properties(x)
\end{array}
$$

- An empty shape expression has no properties.

- The properties of a triple constraint shape expression are the properties of its triple constraint.

- Inverse triple constraint shape expressions have no properties.

- The properties of a some-of, one-of, or grouping shape expression are the union of the properties of their component shape expressions.

- The properties of a repetition shape expression are the properties of the shape expression being repeated.

Given a triple constraint tc, let $propertiesTripleConstraint(tc)$ be its set of properties.

$$
\begin{array}{|l}
propertiesTripleConstraint : TripleConstraint \longrightarrow PropertiesSet \\
\hline
\forall\, a : IRI;\ C : Constraint \bullet \\
\quad propertiesTripleConstraint((nop(a), C)) = \{a\}
\end{array}
$$

- The properties of a triple constraint is the singleton set that contains its IRI.

4.5 *invproperties*

Given a shape expression x, let $\mathit{invproperties}(x)$ be the set of properties that appear in some inverse triple constraint in x.

$$
\begin{array}{|l}
\mathit{invproperties} : \mathit{ShapeExpr} \longrightarrow \mathit{PropertiesSet} \\
\hline
\mathit{invproperties}(\mathit{emptyshape}) = \varnothing \\[1ex]
\forall\, \mathit{tc} : \mathit{TripleConstraint};\ c : \mathit{Cardinality} \bullet \\
\quad \mathit{invproperties}(\mathit{triple}(\mathit{tc}, c)) = \\
\qquad \varnothing \\[1ex]
\forall\, \mathit{itc} : \mathit{InverseTripleConstraint};\ c : \mathit{Cardinality} \bullet \\
\quad \mathit{invproperties}(\mathit{triple}(\mathit{itc}, c)) = \\
\qquad \mathit{invpropertiesInverseTripleConstraint}(\mathit{itc}) \\[1ex]
\forall\, \mathit{xs} : \mathrm{seq}_1\ \mathit{ShapeExpr} \bullet \\
\quad \mathit{invproperties}(\mathit{someOf}(\mathit{xs})) = \\
\quad \mathit{invproperties}(\mathit{oneOf}(\mathit{xs})) = \\
\quad \mathit{invproperties}(\mathit{group}(\mathit{xs})) = \\
\qquad \bigcup \{\, x : \mathrm{ran}\ \mathit{xs} \bullet \mathit{invproperties}(x) \,\} \\[1ex]
\forall\, x : \mathit{ShapeExpr};\ c : \mathit{Cardinality} \bullet \\
\quad \mathit{invproperties}(\mathit{repetition}(x, c)) = \mathit{invproperties}(x)
\end{array}
$$

- An empty shape expression has no inverse properties.
- A triple constraint shape expression has no inverse properties.
- The inverse properties of an inverse triple constraint shape expression are the inverse properties in its inverse triple constraint.
- The inverse properties of a some-of, one-of, or grouping shape expression is the union of the inverse properties of their component shape expressions.
- The inverse properties of a repetition shape expression are the inverse properties of the shape expression being repeated.

Given an inverse triple constraint itc, let $\mathit{invpropertiesInverseTripleConstraint}(\mathit{tc})$ be its set of inverse properties.

$$
\begin{array}{|l}
\mathit{invpropertiesInverseTripleConstraint} : \\
\quad \mathit{InverseTripleConstraint} \longrightarrow \mathit{PropertiesSet} \\
\hline
\forall\, a : \mathit{IRI};\ C : \mathit{ShapeConstr} \bullet \\
\quad \mathit{invpropertiesInverseTripleConstraint}((\mathit{inv}(a), C)) = \{a\}
\end{array}
$$

- The inverse properties of an inverse triple constraint is the singleton set that contains its IRI.

4.6 dep_graph

4.6.1 $DiGraph$

A directed graph consists of a set of nodes and a set of directed edges that connect the nodes.

$$
\begin{array}{|l}
\hline
\textit{DiGraph}[X] \\
\textit{nodes} : \mathbb{P}\, X \\
\textit{edges} : X \leftrightarrow X \\
\hline
\textit{edges} \in \textit{nodes} \leftrightarrow \textit{nodes} \\
\hline
\end{array}
$$

- Each edge connects a pair of nodes in the graph.

4.6.2 $DepGraph$

Given a well-formed schema S, let the shapes dependency graph be the directed graph whose nodes are the shape labels in S and whose edges connect label $T1$ to label $T2$ when the shape expression that defines $T1$ refers to $T2$.

$$
\begin{array}{|l}
\hline
\textit{DepGraph} \\
S : \textit{SchemaWF} \\
\textit{DiGraph}[\textit{ShapeLabel}] \\
\hline
\textit{nodes} = \textit{shapes}(S) \\
\textit{edges} = \{\, T1, T2 : \textit{nodes} \mid T2 \in \textit{refsShapeExpr}(\textit{expr}(T1, S)) \,\} \\
\hline
\end{array}
$$

- The nodes are the shapes of the schema.
- There is an edge from $T1$ to $T2$ when the definition of $T1$ refers to $T2$.

4.6.3 dep_graph

Let $dep\_graph(S)$ be the dependency graph of S.

$$
\begin{array}{|l}
\textit{dep\_graph} : \textit{SchemaWF} \longrightarrow \textit{DiGraph}[\textit{ShapeLabel}] \\
\hline
\textit{dep\_graph} = \{\, \textit{DepGraph} \bullet S \mapsto \theta \textit{DiGraph} \,\}
\end{array}
$$

4.7 $dep\_subgraph$

4.7.1 $reachable$

Given a directed graph g and a node T in g, a node U is reachable from T if there is a directed path of one or more edges that connects T to U.

$$
\begin{array}{l}
[X] \\
\hline
reachable : DiGraph[X] \times X \longrightarrow \mathbb{P}\, X \\
\hline
\forall\, g : DiGraph[X];\; T : X \bullet \\
\quad \mathbf{let}\ edges == g.edges \bullet \\
\quad\quad reachable(g, T) = \{\, U : X \mid T \mapsto U \in edges^{+} \,\}
\end{array}
$$

4.7.2 *DepSubgraph*

Given a well-formed schema S and a shape label T in S, the shapes dependency subgraph for T is the subgraph induced by the nodes that are reachable from T.

$$
\begin{array}{l}
DepSubgraph \\
\hline
S : SchemaWF \\
T : ShapeLabel \\
DiGraph[ShapeLabel] \\
\hline
T \in shapes(S) \\
\mathbf{let}\ g == dep\_graph(S) \bullet \\
\quad nodes = reachable(g, T) \land \\
\quad edges = g.edges \cap (nodes \times nodes)
\end{array}
$$

- The nodes of the subgraph consist of all the nodes reachable from T.
- The edges of the subgraph consist of all edges of the graph whose nodes are in the subgraph.

Note that the above formal definition of the dependency subgraph is a literal translation of the text in the semantics draft. In particular, this literal translation does not explicitly include the label T as a node. Therefore T will not be in the subgraph unless it is in a directed cycle of edges.

4.7.3 *dep_subgraph*

Let $dep\_subgraph(T, S)$ be the dependency subgraph for T in S.

$$
\begin{array}{l}
dep\_subgraph : ShapeLabel \times SchemaWF \rightarrowtail\!\!\!\!\!+\; DiGraph[ShapeLabel] \\
\hline
dep\_subgraph = \{\, DepSubgraph \bullet (T, S) \mapsto \theta DiGraph \,\}
\end{array}
$$

4.8 *negshapes*

The definition of *negshapes* makes use of several auxilliary definitions. In the following we assume that S is a well-formed schema and that T is a shape label in S.

4.8.1 *inNeg*

Let $\mathit{inNeg}(S)$ be the set of labels that appear in some negated shape constraint.

$$
\begin{array}{|l}
\mathit{inNeg} : \mathit{SchemaWF} \longrightarrow \mathbb{F}\ \mathit{ShapeLabel} \\
\hline
\forall\, S : \mathit{SchemaWF} \bullet \\
\qquad \mathit{inNeg}(S) = \bigcup \{\, T : \mathit{shapes}(S) \bullet \mathit{inNegExpr}(\mathit{expr}(T, S)) \,\}
\end{array}
$$

Given a shape expression x, let $\mathit{inNegExpr}(x)$ be the set of labels that appear in some negated shape constraint in x.

$$
\begin{array}{|l}
\mathit{inNegExpr} : \mathit{ShapeExpr} \longrightarrow \mathbb{F}\ \mathit{ShapeLabel} \\
\hline
\mathit{inNegExpr}(\mathit{emptyshape}) = \varnothing \\
\\
\forall\, \mathit{tc} : \mathit{TripleConstraint};\ c : \mathit{Cardinality} \bullet \\
\qquad \mathit{inNegExpr}(\mathit{triple}(\mathit{tc}, c)) = \\
\qquad\qquad \mathit{inNegTripleConstraint}(\mathit{tc}) \\
\\
\forall\, \mathit{itc} : \mathit{InverseTripleConstraint};\ c : \mathit{Cardinality} \bullet \\
\qquad \mathit{inNegExpr}(\mathit{triple}(\mathit{itc}, c)) = \\
\qquad\qquad \mathit{inNegInverseTripleConstraint}(\mathit{itc}) \\
\\
\forall\, \mathit{xs} : \mathrm{seq}_1\ \mathit{ShapeExpr} \bullet \\
\qquad \mathit{inNegExpr}(\mathit{someOf}(\mathit{xs})) = \\
\qquad \mathit{inNegExpr}(\mathit{oneOf}(\mathit{xs})) = \\
\qquad \mathit{inNegExpr}(\mathit{group}(\mathit{xs})) = \\
\qquad\qquad \bigcup \{\, x : \mathrm{ran}\ \mathit{xs} \bullet \mathit{inNegExpr}(x) \,\} \\
\\
\forall\, x : \mathit{ShapeExpr};\ c : \mathit{Cardinality} \bullet \\
\qquad \mathit{inNegExpr}(\mathit{repetition}(x, c)) = \mathit{inNegExpr}(x)
\end{array}
$$

Given a triple constraint tc, let $\mathit{inNegTripleConstraint}(\mathit{tc})$ be the set of labels that appear in some negated shape constraint in tc.

$$
\begin{array}{|l}
\mathit{inNegTripleConstraint} : \mathit{TripleConstraint} \longrightarrow \mathbb{F}\ \mathit{ShapeLabel} \\
\hline
\forall\, a : \mathit{IRI};\ C : \mathit{ValueConstr} \bullet \\
\qquad \mathit{inNegTripleConstraint}((\mathit{nop}(a), C)) = \varnothing \\
\\
\forall\, a : \mathit{IRI};\ C : \mathit{ShapeConstr} \bullet \\
\qquad \mathit{inNegTripleConstraint}((\mathit{nop}(a), C)) = \mathit{inNegShapeConstr}(C)
\end{array}
$$

Given an inverse triple constraint itc, let $\mathit{inNegInverseTripleConstraint}(\mathit{tc})$ be the set of labels that appear in some negated shape constraint in itc.

$$
\begin{array}{|l}
\mathit{inNegInverseTripleConstraint} : \\
\qquad \mathit{InverseTripleConstraint} \longrightarrow \mathbb{F}\ \mathit{ShapeLabel} \\
\hline
\forall\, a : \mathit{IRI};\ C : \mathit{ShapeConstr} \bullet \\
\qquad \mathit{inNegInverseTripleConstraint}((\mathit{inv}(a), C)) = \mathit{inNegShapeConstr}(C)
\end{array}
$$

Given a shape constraint C, let $inNegShapeConstr(C)$ be the set of labels that appear in C when it is negated, or the empty set otherwise.

$$
\begin{array}{|l}
inNegShapeConstr : ShapeConstr \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall\ Ts : \mathrm{seq}_1\ ShapeLabel \bullet \\
\quad inNegShapeConstr(or(Ts)) = \\
\quad inNegShapeConstr(and(Ts)) = \\
\qquad \varnothing \\
\forall\ Ts : \mathrm{seq}_1\ ShapeLabel \bullet \\
\quad inNegShapeConstr(nor(Ts)) = \\
\quad inNegShapeConstr(nand(Ts)) = \\
\qquad \mathrm{ran}\ Ts
\end{array}
$$

4.8.2 *underOneOf*

Let $underOneOf(S)$ be the set of labels that appear in some triple constraint or inverse triple constraint under a one-of constraint in S.

$$
\begin{array}{|l}
underOneOf : SchemaWF \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall\ S : SchemaWF \bullet \\
\quad underOneOf(S) = \\
\qquad \bigcup\{\ T : shapes(S) \bullet underOneOfExpr(expr(T, S))\ \}
\end{array}
$$

Given a shape expression x, let $underOneOfExpr(x)$ be the set of labels that appear in some triple constraint or inverse triple constraint under a one-of constraint in x.

$$
\begin{array}{|l}
underOneOfExpr : ShapeExpr \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall\ x : ShapeExpr \bullet \\
\quad underOneOfExpr(x) = \\
\qquad \mathbf{if}\ x \in \mathrm{ran}\ someOf \\
\qquad\quad \mathbf{then}\ refsShapeExpr(x) \\
\qquad\quad \mathbf{else}\ \varnothing
\end{array}
$$

4.8.3 *inTripleConstr*

Let $inTripleConstr(S)$ be the set of labels T such that there is a shape label $T1$ and a triple constraint `p::C` or an inverse shape triple constraint `^p::C` in $expr(T1, S)$, and T appears in C.

Note that this definition looks wrong since it does not involve negation of shapes. Nevertheless, a literal translation is given here. The only difference between $inTripleConstr(S)$ and $refs(S)$ seems to be that the cardinality on the triple and inverse triple constraints is `[1,1]` since it is not explicitly included in the notations `p::C` and `^p::C`.

$$
\begin{array}{|l}
inTripleConstr : SchemaWF \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall S : SchemaWF \bullet \\
\quad inTripleConstr(S) = \\
\qquad \bigcup \{ T1 : shapes(S) \bullet inTripleConstrExpr(expr(T1, S)) \}
\end{array}
$$

Given a shape expression x, let $inTripleConstrExpr(x)$ be the set of labels T such that x contains a triple constraint `p::C` or an inverse shape triple constraint `^p::C` and T appears in x.

$$
\begin{array}{|l}
inTripleConstrExpr : ShapeExpr \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
inTripleConstrExpr(emptyshape) = \varnothing \\
\\
\forall dtc : DirectedTripleConstraint; c : Cardinality \bullet \\
\quad inTripleConstrExpr(triple(dtc, c)) = \\
\qquad \textbf{if } c = one \\
\qquad\quad \textbf{then } refsDirectedTripleConstraint(dtc) \\
\qquad\quad \textbf{else } \varnothing \\
\\
\forall xs : \mathrm{seq}_1\ ShapeExpr \bullet \\
\quad inTripleConstrExpr(someOf(xs)) = \\
\quad inTripleConstrExpr(oneOf(xs)) = \\
\quad inTripleConstrExpr(group(xs)) = \\
\qquad \bigcup \{ x : \mathrm{ran}\ xs \bullet inTripleConstrExpr(x) \} \\
\\
\forall x : ShapeExpr; c : Cardinality \bullet \\
\quad inTripleConstrExpr(repetition(x, c)) = inTripleConstrExpr(x)
\end{array}
$$

4.8.4 *negshapes*

The semantics draft makes the following statement.

> Intuitively, negshapes(S) is the set of shapes labels for which one needs to check whether some nodes in a graph do not satisfy these shapes, in order to validate the graph against the schema S.

Let $negshapes(S)$ be the set of negated shape labels that appear in S.

$$
\begin{array}{|l}
negshapes : SchemaWF \longrightarrow \mathbb{F}\ ShapeLabel \\
\hline
\forall S : SchemaWF \bullet \\
\quad negshapes(S) = inNeg(S) \cup underOneOf(S) \cup inTripleConstr(S)
\end{array}
$$

- A negated shape label is a shape label that appears in a negated shape constraint, or in a triple or inverse triple constraint under a one-of shape expression, or in a triple or inverse triple constraint that has cardinality `[1,1]`.

Note that, as remarked above, the definition of *inTripleConstr* seems wrong.

4.9 *ShapeVerdict*

The semantics draft defines the notation `!T` for shape labels T to indicate that T is negated. The semantics of a schema involves assigning sets of shape labels and negated shape labels to the nodes of a graph, which indicates which shapes must be satisfied or violated at each node.

A shape verdict indicates if a shape must be satisfied or violated. An asserted label must be satisfied. A negated label must be violated.

$$
\begin{aligned}
&\mathit{ShapeVerdict} ::= \\
&\qquad \mathit{assert}\langle\!\langle \mathit{ShapeLabel} \rangle\!\rangle \mid \\
&\qquad \mathit{negate}\langle\!\langle \mathit{ShapeLabel} \rangle\!\rangle
\end{aligned}
$$

The notation `!T` corresponds to $\mathit{negate}(T)$.

4.10 *allowed*

Given a value constraint V, let $\mathit{allowed}(V)$ be the set of all allowed values defined by V.

$$
\begin{array}{|l}
\mathit{allowed} : \mathit{ValueConstr} \longrightarrow \mathbb{P}(\mathit{Lit} \cup \mathit{IRI}) \\
\hline
\forall\, \mathit{vs} : \mathbb{P}(\mathit{Lit} \cup \mathit{IRI}) \bullet \\
\qquad \mathit{allowed}(\mathit{valueSet}(\mathit{vs})) = \mathit{vs} \\
\\
\forall\, \mathit{dt} : \mathit{LiteralDatatype} \bullet \\
\qquad \mathit{allowed}(\mathit{datatype}(\mathit{dt}, \varnothing)) = \mathit{literalsOfDatatype}(\mathit{dt}) \\
\\
\forall\, \mathit{dt} : \mathit{LiteralDatatype};\, f : \mathit{XSFacet} \bullet \\
\qquad \mathit{allowed}(\mathit{datatype}(\mathit{dt}, \{f\})) = \mathit{literalsOfFacet}(\mathit{dt}, f) \\
\\
\forall\, k : \mathit{NodeKind} \bullet \\
\qquad \mathit{allowed}(\mathit{kind}(k)) = \mathit{termsOfKind}(k)
\end{array}
$$

4.10.1 DAG

A directed, acyclic graph is a directed graph in which no node is reachable from itself.

$$
\begin{array}{|l|}
\hline
\mathit{DAG}[X] \\
\mathit{DiGraph}[X] \\
\hline
\mathbf{let}\ g == \theta \mathit{DiGraph} \bullet \\
\qquad \forall\, T : \mathit{nodes} \bullet T \notin \mathit{reachable}(g, T) \\
\hline
\end{array}
$$

4.11 *ReplaceShape*

The semantics draft introduces the notation S_{ri} for a reduced schema where S is a schema, r is a rule-of-one node in a proof tree, and i corresponds to a premise of r. The reduced schema is constructed by replacing a shape with one in which the corresponding one-of component is eliminated. This replacement operation is described here. The full definition of S_{ri} is given below following the definition of proof trees.

Given a schema S, a shape label T defined in S, and a shape expression $Expr'$, the schema $replaceShape(S, T, Expr')$ is the schema S' that is the same as S except that $expr(T, S') = Expr'$.

$$
\begin{array}{|l}
\hline
\textit{ReplaceShape} \\
S, S' : \textit{SchemaWD} \\
T : \textit{ShapeLabel} \\
\textit{Expr}' : \textit{ShapeExpr} \\
l : \mathbb{N}_1 \\
d, d' : \textit{ShapeDefinition} \\
\textit{ecs} : \mathrm{seq}\ \textit{ExtensionCondition} \\
\hline
l \in \mathrm{dom}\ S \\
S(l) = (T, d, \textit{ecs}) \\
\forall\, o : \textit{OPTIONAL}[\textit{InclPropSet}]; \textit{Expr} : \textit{ShapeExpr} \mid \\
\qquad d = \textit{open}(o, \textit{Expr}) \bullet \\
\qquad\qquad d' = \textit{open}(o, \textit{Expr}') \\
\forall\, \textit{Expr} : \textit{ShapeExpr} \mid \\
\qquad d = \textit{close}(\textit{Expr}) \bullet \\
\qquad\qquad d' = \textit{close}(\textit{Expr}') \\
S' = S \oplus \{l \mapsto (T, d', \textit{ecs})\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{replaceShape} : \\
\qquad \textit{SchemaWF} \times \textit{ShapeLabel} \times \textit{ShapeExpr} \rightarrowtail\!\!\!\!\!+ \textit{SchemaWF} \\
\hline
\textit{replaceShape} = \{\, \textit{ReplaceShape} \bullet (S, T, \textit{Expr}') \mapsto S' \,\}
\end{array}
$$

4.12 *SchemaWD*

Given a well-formed schema S, it is said to be well-defined if for each negated label T in $negshapes(T)$, the dependency subgraph $dep\_subgraph(T, S)$ is a directed, acyclic graph.

$$
\begin{array}{l}
\textit{SchemaWD} == \\
\qquad \{\, S : \textit{SchemaWF} \mid \\
\qquad\qquad \forall\, T : \textit{negshapes}(S) \bullet \\
\qquad\qquad\qquad \textit{dep\_subgraph}(T, S) \in \textit{DAG}[\textit{ShapeLabel}] \,\}
\end{array}
$$

The semantics of shape expression schemas is sound only for well-defined schemas. Only well-defined schemas will be considered from this point forward.

5 Declarative semantics of shape expression schemas

Recall that negated triple and inverse triple shape expressions are represented by the corresponding non-negated expressions with cardinality $none$ = `[0;0]`.

5.1 *LabelledTriple*

A labelled triple is either an incoming or outgoing edge in an RDF graph.

$$
\begin{array}{l}
\mathit{LabelledTriple} ::= \\
\quad \mathit{out}\langle\!\langle \mathit{Triple} \rangle\!\rangle \mid \\
\quad \mathit{inc}\langle\!\langle \mathit{Triple} \rangle\!\rangle
\end{array}
$$

Sometimes labelled triples are referred to simply as triples.

5.2 *matches*

A labelled triple matches a directed triple constraint when they have the same direction and predicate.

$$
\begin{array}{|l}
\mathit{matches} : \mathit{LabelledTriple} \leftrightarrow \mathit{DirectedTripleConstraint} \\
\hline
\mathit{matches} = \mathit{matches\_out} \cup \mathit{matches\_inc}
\end{array}
$$

5.2.1 *matches_out*

matches_out matches outgoing triples to triple constraints.

$$
\begin{array}{l}
\mathit{matches\_out} == \\
\quad \{\, s, p, o : \mathit{TERM};\ C : \mathit{Constraint} \mid \\
\qquad (s, p, o) \in \mathit{Triple} \bullet \\
\qquad\quad \mathit{out}(s, p, o) \mapsto (\mathit{nop}(p), C) \,\}
\end{array}
$$

Note that this definition ignores any value constraints defined in C. The absence of restrictions imposed by value constraints makes matching weaker than it could be. This may be an error in the semantics draft.

The semantics drafts contains the following text.

> The following definition introduces the notion of satisfiability of a shape constraint by a set of triples. Such satisfiability is going to be used for checking that the neighbourhood of a node satisfies locally the constraints defined by a shape expression, without taking into account whether the shapes required by the triple constraints and inverse triple constraints are satisfied.

Read literally, only shape constraints should be ignored, so unless value constraints are handled elsewhere, the semantics draft has an error in the definition of *matches*.

5.2.2 *matches_inc*

matches_inc matches incoming triples to inverse triple constraints.

$$
\begin{aligned}
&\mathit{matches\_inc} == \\
&\quad \{\, s, p, o : \mathit{TERM};\ C : \mathit{ShapeConstr} \mid \\
&\qquad (s, p, o) \in \mathit{Triple} \bullet \\
&\qquad\quad \mathit{inc}(s, p, o) \mapsto (\mathit{inv}(p), C) \,\}
\end{aligned}
$$

5.3 *satifies*

A set of labelled triples *Neigh* is said to satisfy a shape expression *Expr* if the constraints, other than shape constraints, defined in *Expr* are satisfied.

Note that the definition of *matches* ignores both value and shape constraints.

$$
\mathit{satisfies} : \mathbb{F}\ \mathit{LabelledTriple} \leftrightarrow \mathit{ShapeExpr}
$$

This relation is defined recursively by inference rules for each type of shape expression.

$$
\begin{aligned}
&\mathit{satisfies} = \\
&\quad \mathit{rule\_empty} \cup \\
&\quad \mathit{rule\_triple\_constraint} \cup \\
&\quad \mathit{rule\_inverse\_triple\_constraint} \cup \\
&\quad \mathit{rule\_some\_of} \cup \\
&\quad \mathit{rule\_one\_of} \cup \\
&\quad \mathit{rule\_group} \cup \\
&\quad \mathit{rule\_repeat}
\end{aligned}
$$

5.3.1 *InfRule*

An inference rule defines a relation between a set of labelled triples and a shape expression. It is convenient to define a base schema for the inference rules.

InfRule

$$
\begin{aligned}
&\mathit{Neigh} : \mathbb{F}\ \mathit{LabelledTriple} \\
&\mathit{Expr} : \mathit{ShapeExpr}
\end{aligned}
$$

5.3.2 *rule_empty*

An empty set of triples satisfies the empty shape expression.

RuleEmpty

$$
\begin{aligned}
&\mathit{InfRule} \\
\hline
&\mathit{Expr} = \mathit{emptyshape} \\
&\mathit{Neigh} = \varnothing
\end{aligned}
$$

$$\begin{array}{l} rule\_empty : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\ \hline rule\_empty = \\ \quad \{\ RuleEmpty \bullet Neigh \mapsto Expr\ \} \end{array}$$

5.3.3 *rule_triple_constraint*

A set of triples satisfies a triple constraint shape expression when each triple matches the constraint and the total number of constraints is within the bounds of the cardinality.

$$\begin{array}{l} \textit{RuleTripleConstraint} \\ \hline InfRule \\ k : \mathbb{N} \\ p : IRI \\ C : Constraint \\ c : Cardinality \\ \hline Expr = triple((nop(p), C), c) \\ k = \#Neigh \\ k\ \underline{inBounds}\ c \\ \forall t : Neigh \bullet t\ \underline{matches}\ (nop(p), C) \end{array}$$

$$\begin{array}{l} rule\_triple\_constraint : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\ \hline rule\_triple\_constraint = \\ \quad \{\ RuleTripleConstraint \bullet Neigh \mapsto Expr\ \} \end{array}$$

5.3.4 *rule_inverse_triple_constraint*

A set of triples satisfies an inverse triple constraint shape expression when each triple matches the constraint and the total number of constraints is within the bounds of the cardinality.

$$\begin{array}{l} \textit{RuleInverseTripleConstraint} \\ \hline InfRule \\ k : \mathbb{N} \\ p : IRI \\ C : Constraint \\ c : Cardinality \\ \hline Expr = triple((inv(p), C), c) \\ k = \#Neigh \\ k\ \underline{inBounds}\ c \\ \forall t : Neigh \bullet t\ \underline{matches}\ (inv(p), C) \end{array}$$

$$
\begin{array}{|l}
rule\_inverse\_triple\_constraint : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\
\hline
rule\_triple\_constraint = \\
\quad \{\ RuleInverseTripleConstraint \bullet Neigh \mapsto Expr\ \}
\end{array}
$$

5.3.5 *rule_some_of*

A set of triples satisfies a some-of shape expression when the set of triples satisfies one of the component shape expressions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{RuleSomeOf} \\
\hline
InfRule \\
Exprs : \mathrm{seq}_1\ ShapeExpr \\
i : \mathbb{N} \\
\hline
Expr = someOf(Exprs) \\
i \in \mathrm{dom}\ Exprs \\
Neigh\ \underline{satisfies}\ Exprs(i) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
rule\_some\_of : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\
\hline
rule\_some\_of = \\
\quad \{\ RuleSomeOf \bullet Neigh \mapsto Expr\ \}
\end{array}
$$

5.3.6 *rule_one_of*

A set of triples satisfies a one-of shape expression when the set of triples satisfies one of the component shape expressions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{RuleOneOf} \\
\hline
InfRule \\
Exprs : \mathrm{seq}_1\ ShapeExpr \\
i : \mathbb{N} \\
\hline
Expr = oneOf(Exprs) \\
i \in \mathrm{dom}\ Exprs \\
Neigh\ \underline{satisfies}\ Exprs(i) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
rule\_one\_of : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\
\hline
rule\_one\_of = \\
\quad \{\ RuleOneOf \bullet Neigh \mapsto Expr\ \}
\end{array}
$$

The semantics draft contains the following text.

> Note that the conditions for some-of and one-of shapes are identical. The distinction between both will be made by taking into account also the non-local, shape constraints.

5.3.7 *rule_group*

A set of triples satisfies a group shape expression when the set of triples can be partitioned into a sequence of subsets whose length is the same as the sequence of component shape expressions, and each subset satisfies the corresponding component shape expression.

$$
\begin{array}{l}
\textbf{RuleGroup} \\
\hline
InfRule \\
Neighs : \mathrm{seq}_1(\mathbb{F}\ LabelledTriple) \\
Exprs : \mathrm{seq}_1\ ShapeExpr \\
\hline
Expr = group(Exprs) \\
Neighs\ \mathsf{partition}\ Neigh \\
\#Neighs = \#Exprs \\
\forall j : \mathrm{dom}\ Neighs \bullet \\
\quad Neighs(j)\ \underline{satisfies}\ Exprs(j) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
rule\_group : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\
\hline
rule\_group = \\
\quad \{\ RuleGroup \bullet Neigh \mapsto Expr\ \}
\end{array}
$$

5.3.8 *rule_repeat*

A set of triples satisfies a repetition shape expression when the set of triples can be partitioned into a sequence of subsets whose length is in the bounds of the cardinality, and each subset satisfies the component shape expression of the repetition shape expression.

$$
\begin{array}{l}
\textbf{RuleRepeat} \\
\hline
InfRule \\
Expr1 : ShapeExpr \\
Neighs : \mathrm{seq}_1(\mathbb{F}\ LabelledTriple) \\
k : \mathbb{N} \\
c : Cardinality \\
\hline
Expr = repetition(Expr1, c) \\
k = \#Neighs \\
k\ \underline{inBounds}\ c \\
Neighs\ \mathsf{partition}\ Neigh \\
\forall j : \mathrm{dom}\ Neighs \bullet \\
\quad Neighs(j)\ \underline{satisfies}\ Expr1 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
rule\_repeat : \mathbb{F}\ LabelledTriple \leftrightarrow ShapeExpr \\
\hline
rule\_repeat = \\
\quad \{\ RuleRepeat \bullet Neigh \mapsto Expr\ \}
\end{array}
$$

5.4 Proof Trees

The preceding definition of *satisfies* is based on the existence of certain characteristics of the set of triples. For example, a set of triples satisfies one of a sequence of shape expressions when it satisfies exactly one of the them, but the *satisfies* relation forgets the actual shape expression that the set of triples satisfies. We can remember this type of information in a proof tree.

5.4.1 *RuleTree*

A rule tree is a tree of inference rules and optional child rule trees. Child rule trees occur in cases where the inference rule depends on other inference rules.

$$
\begin{array}{l}
RuleTree ::= \\
\quad ruleEmpty\langle\langle RuleEmpty\rangle\rangle\ | \\
\quad ruleTripleConstraint\langle\langle RuleTripleConstraint\rangle\rangle\ | \\
\quad ruleInverseTripleConstraint\langle\langle RuleInverseTripleConstraint\rangle\rangle\ | \\
\quad ruleSomeOf\langle\langle RuleSomeOf \times RuleTree\rangle\rangle\ | \\
\quad ruleOneOf\langle\langle RuleOneOf \times RuleTree\rangle\rangle\ | \\
\quad ruleGroup\langle\langle RuleGroup \times \mathrm{seq}_1\ RuleTree\rangle\rangle\ | \\
\quad ruleRepeat\langle\langle RuleRepeat \times \mathrm{seq}_1\ RuleTree\rangle\rangle
\end{array}
$$

5.4.2 *baseRule*

Each node in a rule tree contains an inference rule and, therefore, a base inference rule.

$$
\begin{array}{|l}
\mathit{baseRule} : \mathit{RuleTree} \longrightarrow \mathit{InfRule} \\
\hline
\forall \mathit{RuleEmpty} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleEmpty}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleEmpty}(\mathit{rule})) = \mathit{base} \\
\forall \mathit{RuleTripleConstraint} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleTripleConstraint}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleTripleConstraint}(\mathit{rule})) = \mathit{base} \\
\forall \mathit{RuleInverseTripleConstraint} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleInverseTripleConstraint}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleInverseTripleConstraint}(\mathit{rule})) = \mathit{base} \\
\forall \mathit{RuleSomeOf}; \mathit{tree} : \mathit{RuleTree} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleSomeOf}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleSomeOf}(\mathit{rule}, \mathit{tree})) = \mathit{base} \\
\forall \mathit{RuleOneOf}; \mathit{tree} : \mathit{RuleTree} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleOneOf}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleOneOf}(\mathit{rule}, \mathit{tree})) = \mathit{base} \\
\forall \mathit{RuleGroup}; \mathit{trees} : \mathrm{seq}_1\ \mathit{RuleTree} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleGroup}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleGroup}(\mathit{rule}, \mathit{trees})) = \mathit{base} \\
\forall \mathit{RuleRepeat}; \mathit{trees} : \mathrm{seq}_1\ \mathit{RuleTree} \bullet \\
\quad \textbf{let } \mathit{rule} == \theta \mathit{RuleRepeat}; \\
\quad\quad \mathit{base} == \theta \mathit{InfRule} \bullet \\
\quad\quad\quad \mathit{baseRule}(\mathit{ruleRepeat}(\mathit{rule}, \mathit{trees})) = \mathit{base}
\end{array}
$$

5.4.3 *baseNeigh*

Each node in a rule tree has a base set of labelled triples.

$$
\begin{array}{|l}
\mathit{baseNeigh} : \mathit{RuleTree} \longrightarrow \mathbb{F}\ \mathit{LabelledTriple} \\
\hline
\forall \mathit{tree} : \mathit{RuleTree} \bullet \\
\quad \mathit{baseNeigh}(\mathit{tree}) = (\mathit{baseRule}(\mathit{tree})).\mathit{Neigh}
\end{array}
$$

5.4.4 *baseExpr*

Each node in a rule tree has a base shape expression.

$$
\begin{array}{|l}
baseExpr : RuleTree \longrightarrow ShapeExpr \\
\hline
\forall\, tree : RuleTree \bullet \\
\qquad baseExpr(tree) = (baseRule(tree)).Expr
\end{array}
$$

5.4.5 *ProofTree*

A proof tree is a rule tree in which the child trees prove subgoals of their parent nodes.

$$
\begin{array}{|l}
ProofTree : \mathbb{P}\, RuleTree
\end{array}
$$

The definition of proof tree is recursive so it is given by a set of constraints, one for each type of node.

Any rule tree whose root node contains an empty shape expression is a proof tree since it has no subgoals.

$$\mathrm{ran}\, ruleEmpty \subset ProofTree$$

Any rule tree whose root node node contains a triple constraint shape expression is a proof tree since it has no subgoals.

$$\mathrm{ran}\, ruleTripleConstraint \subset ProofTree$$

Any rule tree whose root node node contains an inverse triple constraint shape expression is a proof tree since it has no subgoals.

$$\mathrm{ran}\, ruleInverseTripleConstraint \subset ProofTree$$

A rule tree whose root node contains a some-of shape expression is a proof tree if and only if its child rule tree correspond to the distinguished shape expression at index i and it is a proof tree.

$$
\begin{array}{l}
\forall\, RuleSomeOf; tree : RuleTree \bullet \\
\qquad ruleSomeOf(\theta RuleSomeOf, tree) \in ProofTree \Leftrightarrow \\
\qquad\qquad baseNeigh(tree) = Neigh \land \\
\qquad\qquad baseExpr(tree) = Exprs(i) \land \\
\qquad\qquad tree \in ProofTree
\end{array}
$$

A rule tree whose root node contains a one-of shape expression is a proof tree if and only if its child rule tree correspond to the distinguished shape expression at index i and it is a proof tree.

$$
\begin{array}{l}
\forall\, RuleOneOf; tree : RuleTree \bullet \\
\qquad ruleOneOf(\theta RuleOneOf, tree) \in ProofTree \Leftrightarrow \\
\qquad\qquad baseNeigh(tree) = Neigh \land \\
\qquad\qquad baseExpr(tree) = Exprs(i) \land \\
\qquad\qquad tree \in ProofTree
\end{array}
$$

A rule tree whose root node contains a group shape expression is a proof tree if and only if its sequence of child rule trees correspond to its sequence of component neighbourhood and shape expressions and each child rule tree is a proof tree.

$$
\begin{array}{l}
\forall \mathit{RuleGroup}; \mathit{trees} : \mathrm{seq}_1 \mathit{RuleTree} \bullet \\
\quad \mathit{ruleGroup}(\theta \mathit{RuleGroup}, \mathit{trees}) \in \mathit{ProofTree} \Leftrightarrow \\
\quad\quad \# \mathit{Exprs} = \# \mathit{trees} \land \\
\quad\quad (\forall i : \mathrm{dom}\, \mathit{trees} \bullet \\
\quad\quad\quad \mathit{baseNeigh}(\mathit{trees}(i)) = \mathit{Neighs}(i) \land \\
\quad\quad\quad \mathit{baseExpr}(\mathit{trees}(i)) = \mathit{Exprs}(i) \land \\
\quad\quad\quad \mathit{trees}(i) \in \mathit{ProofTree})
\end{array}
$$

A rule tree whose root node contains a repetition shape expression is a proof tree if and only if its sequence of child rule trees correspond to its sequence of component neighbourhoods and each child rule tree is a proof tree.

$$
\begin{array}{l}
\forall \mathit{RuleRepeat}; \mathit{trees} : \mathrm{seq}_1 \mathit{RuleTree} \bullet \\
\quad \mathit{ruleRepeat}(\theta \mathit{RuleRepeat}, \mathit{trees}) \in \mathit{ProofTree} \Leftrightarrow \\
\quad\quad \# \mathit{Neighs} = \# \mathit{trees} \land \\
\quad\quad (\forall i : \mathrm{dom}\, \mathit{trees} \bullet \\
\quad\quad\quad \mathit{baseNeigh}(\mathit{trees}(i)) = \mathit{Neighs}(i) \land \\
\quad\quad\quad \mathit{baseExpr}(\mathit{trees}(i)) = \mathit{Expr}1 \land \\
\quad\quad\quad \mathit{trees}(i) \in \mathit{ProofTree})
\end{array}
$$

We have the following relation between proof trees and the *satisfies* relation.

$$
\begin{array}{l}
\vdash \mathit{satisfies} = \\
\quad \{ \mathit{tree} : \mathit{ProofTree} \bullet \mathit{baseNeigh}(\mathit{tree}) \mapsto \mathit{baseExpr}(\mathit{tree}) \}
\end{array}
$$

5.5 Reduced Schema for rule-one-of

As mentioned above, inference rules and proof trees treat rule-one-of exactly the same as rule-some-of. The difference between these rules appears when considering valid typings, which are described in detail later.

Let t be a valid typing of graph G under schema S. Let n be a node in G and let T be a shape label in $t(n)$. Let $\mathit{Expr} = \mathit{expr}(T, S)$ be the shape expression for T. Let *tree* be a proof tree that the neighbourhood of n satisfies Expr. Let r be a node of the proof tree that contains an application of rule-one-of and let i be the index of the component expression used in the application of the rule. The intention of the one-of shape expression is that the triples match exactly one of the component expressions. Therefore, if the matched shape expression is removed from the one-of expression then there must not be any valid typings of G under the reduced schema S_{ri}.

Note that a one-of shape expression may have one or more components. The number of components is denoted by k in the inference rule. However, if it contains exactly one component then there no further semantic conditions that must hold and there is no corresponding reduced schema. Therefore, the definition of the reduced schema only applies to the case where the number of components is greater than one, i.e. $k > 1$.

Rule trees are ordered trees. A child tree can be specified by giving its index among all the children. The maximum index of a child depends on the type of rule. For leaf trees, the maximum child index is 0.

$$maxChild : RuleTree \longrightarrow \mathbb{N}$$

$$\forall\, tree : \operatorname{ran} ruleEmpty \bullet maxChild(tree) = 0$$

$$\forall\, tree : \operatorname{ran} ruleTripleConstraint \bullet maxChild(tree) = 0$$

$$\forall\, tree : \operatorname{ran} ruleInverseTripleConstraint \bullet maxChild(tree) = 0$$

$$\forall\, tree : \operatorname{ran} ruleSomeOf \bullet maxChild(tree) = 1$$

$$\forall\, tree : \operatorname{ran} ruleOneOf \bullet maxChild(tree) = 1$$

$$\forall\, r : RuleGroup;\ trees : \operatorname{seq}_1 RuleTree \bullet \\ \quad maxChild(ruleGroup(r, trees)) = \#trees$$

$$\forall\, r : RuleRepeat;\ trees : \operatorname{seq}_1 RuleTree \bullet \\ \quad maxChild(ruleRepeat(r, trees)) = \#trees$$

Given a tree *tree* and a valid child index j, the child tree at the index is $childAt(tree, j)$.

$$childAt : RuleTree \times \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\!+ \; RuleTree$$

$$\operatorname{dom} childAt = \\ \quad \{\, tree : RuleTree;\ ci : \mathbb{N}_1 \mid ci \leq maxChild(tree) \,\}$$

$$\forall\, r : RuleSomeOf;\ tree : RuleTree \bullet \\ \quad childAt(ruleSomeOf(r, tree), 1) = tree$$

$$\forall\, r : RuleOneOf;\ tree : RuleTree \bullet \\ \quad childAt(ruleOneOf(r, tree), 1) = tree$$

$$\forall\, r : RuleGroup;\ trees : \operatorname{seq}_1 RuleTree \bullet \\ \quad \mathbf{let}\ tree == ruleGroup(r, trees) \bullet \\ \qquad \forall\, ci : 1 \,..\, maxChild(tree) \bullet \\ \qquad\quad childAt(tree, ci) = trees(ci)$$

$$\forall\, r : RuleRepeat;\ trees : \operatorname{seq}_1 RuleTree \bullet \\ \quad \mathbf{let}\ tree == ruleRepeat(r, trees) \bullet \\ \qquad \forall\, ci : 1 \,..\, maxChild(tree) \bullet \\ \qquad\quad childAt(tree, ci) = trees(ci)$$

The location of a node within a rule tree can be specified by giving a sequence of positive integers that specify the index of each child tree. The root of the tree is specified by the empty sequence. Such a sequence of integers is referred to as a rule path. Given a rule tree *tree*, the set of all of its rule paths is $rulePaths(tree)$.

$$
\begin{array}{|l}
rulePaths : RuleTree \longrightarrow \mathbb{F}(\operatorname{seq} \mathbb{N}_1) \\
\hline
\forall\, tree : RuleTree \mid maxChild(tree) = 0 \bullet \\
\quad rulePaths(tree) = \{\langle\rangle\} \\
\forall\, tree : RuleTree \mid maxChild(tree) > 0 \bullet \\
\quad rulePaths(tree) = \\
\qquad \bigcup\{\, ci : 1 \mathinner{..} maxChild(tree) \bullet \\
\qquad\quad \{\, path : rulePaths(childAt(tree, ci)) \bullet \langle ci \rangle \frown path \,\}\,\}
\end{array}
$$

Given a rule tree *tree* and a rule path *path*, the tree node specified by the path is $treeAt(tree, path)$,

$$
\begin{array}{|l}
treeAt : RuleTree \times \operatorname{seq} \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\!\!\!\;\; + \; RuleTree \\
\hline
\operatorname{dom} treeAt = \\
\quad \{\, tree : RuleTree;\ path : \operatorname{seq} \mathbb{N}_1 \mid path \in rulePaths(tree) \,\} \\
\forall\, tree : RuleTree \bullet treeAt(tree, \langle\rangle) = tree \\
\forall\, tree : RuleTree;\ ci : \mathbb{N}_1;\ path : \operatorname{seq} \mathbb{N}_1 \mid \\
\quad \langle ci \rangle \frown path \in rulePaths(tree) \bullet \\
\qquad treeAt(tree, \langle ci \rangle \frown path) = treeAt(childAt(tree, ci), path)
\end{array}
$$

Given a one-of shape expression *Expr* that has more than one component, and an index i of one component, $elimExpr(Expr, i)$ is the reduced expression in which component i is eliminated.

$$
\begin{array}{|l}
\hline
ElimExpr \\
Expr, Expr' : ShapeExpr \\
Exprs, ExprsL, ExprsR : \operatorname{seq}_1 ShapeExpr \\
i : \mathbb{N} \\
\hline
Expr = oneOf(Exprs) \\
\#Exprs > 1 \\
i \in \operatorname{dom} Exprs \\
Exprs = ExprsL \frown \langle Exprs(i) \rangle \frown ExprsR \\
Expr' = oneOf(ExprsL \frown ExprsR) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
elimExpr : ShapeExpr \times \mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\!\;\; + \; ShapeExpr \\
\hline
elimExpr = \{\, ElimExpr \bullet (Expr, i) \mapsto Expr' \,\}
\end{array}
$$

Given a proof tree *tree* with the shape expression *Expr* as its base, and a path *path* to some application r of rule-one-of in *tree* in which the rule-of expression has more than one component,

$$
\begin{array}{|l}
\hline
\textit{RuleOneOfApplication} \\
tree : ProofTree \\
path : \mathrm{seq}\,\mathbb{N}_1 \\
r, rChild : ProofTree \\
R : RuleOneOf \\
\hline
path \in rulePaths(tree) \\
r = treeAt(tree, path) = ruleOneOf(R, rChild) \\
\#R.Exprs > 1 \\
\hline
\end{array}
$$

- The path is a valid rule path in the proof tree.
- The tree at the path is an application of rule-one-of.
- There are more than one components in the one-of shape expression.

$reduceExpr(tree, path)$ is the reduced base shape expression with the corresponding one-of expression in $Expr$ replaced by the reduced one-of expression.

$$
\begin{array}{|l}
reduceExpr : ProofTree \times \mathrm{seq}\,\mathbb{N}_1 \nrightarrow ShapeExpr \\
\hline
\mathrm{dom}\, reduceExpr = \{\, RuleOneOfApplication \bullet (tree, path) \,\} \\
\forall\, RuleOneOfApplication \mid \\
\quad path = \langle\rangle \land \\
\quad tree = r \bullet \\
\qquad reduceExpr(r, \langle\rangle) = elimExpr(R.Expr, R.i)
\end{array}
$$

- The domain of this function requires that the path be a valid rule path in the proof tree.
- In the case of an empty path, the tree must be a one-of tree and the branch taken is eliminated.
- When the path is not empty, this function is defined recursively by additional constraints which follow. There are four possible cases in which the proof tree has children. These cases correspond to applications of rule-some-of, rule-one-of, rule-group, and rule-repeat. Each case is defined by a schema below.

ReduceSomeOf

$$
\begin{array}{l}
RuleOneOfApplication \\
RuleSomeOf \\
child : ProofTree \\
tail : \mathrm{seq}\, \mathbb{N}_1 \\
ExprsL, ExprsR : \mathrm{seq}\, ShapeExpr \\
Expr' : ShapeExpr \\
\hline
tree = ruleSomeOf(\theta RuleSomeOf, child) \\
path = \langle 1 \rangle \frown tail \\
Exprs = ExprsL \frown \langle Exprs(i) \rangle \frown ExprsR \\
Expr' = someOf(ExprsL \frown \langle reduceExpr(child, tail) \rangle \frown ExprsL)
\end{array}
$$

$$
\begin{array}{l}
\forall\, ReduceSomeOf \bullet \\
\quad reduceExpr(tree, path) = Expr'
\end{array}
$$

ReduceOneOf

$$
\begin{array}{l}
RuleOneOfApplication \\
RuleOneOf \\
child : ProofTree \\
tail : \mathrm{seq}\, \mathbb{N}_1 \\
ExprsL, ExprsR : \mathrm{seq}\, ShapeExpr \\
Expr' : ShapeExpr \\
\hline
tree = ruleOneOf(\theta RuleOneOf, child) \\
path = \langle 1 \rangle \frown tail \\
Exprs = ExprsL \frown \langle Exprs(i) \rangle \frown ExprsR \\
Expr' = oneOf(ExprsL \frown \langle reduceExpr(child, tail) \rangle \frown ExprsL)
\end{array}
$$

$$
\begin{array}{l}
\forall\, ReduceOneOf \bullet \\
\quad reduceExpr(tree, path) = Expr'
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ReduceGroup} \\
\textit{RuleOneOfApplication} \\
\textit{RuleGroup} \\
\textit{children} : \mathrm{seq}_1\, \textit{ProofTree} \\
\textit{ci} : \mathbb{N}_1 \\
\textit{tail} : \mathrm{seq}\, \mathbb{N}_1 \\
\textit{ExprsL}, \textit{ExprsR} : \mathrm{seq}\, \textit{ShapeExpr} \\
\textit{Expr}' : \textit{ShapeExpr} \\
\hline
\textit{tree} = \textit{ruleGroup}(\theta \textit{RuleGroup}, \textit{children}) \\
\textit{path} = \langle \textit{ci} \rangle \frown \textit{tail} \\
\textit{Exprs} = \textit{ExprsL} \frown \langle \textit{Exprs}(\textit{ci}) \rangle \frown \textit{ExprsR} \\
\textit{Expr}' = \textit{group}(\textit{ExprsL} \frown \langle \textit{reduceExpr}(\textit{children}(\textit{ci}), \textit{tail}) \rangle \frown \textit{ExprsL}) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{ReduceGroup} \bullet \\
\quad \textit{reduceExpr}(\textit{tree}, \textit{path}) = \textit{Expr}'
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ReduceRepeat} \\
\textit{RuleOneOfApplication} \\
\textit{RuleRepeat} \\
\textit{children} : \mathrm{seq}_1\, \textit{ProofTree} \\
\textit{ci} : \mathbb{N}_1 \\
\textit{tail} : \mathrm{seq}\, \mathbb{N}_1 \\
\textit{Expr}' : \textit{ShapeExpr} \\
\hline
\textit{tree} = \textit{ruleRepeat}(\theta \textit{RuleRepeat}, \textit{children}) \\
\textit{path} = \langle \textit{ci} \rangle \frown \textit{tail} \\
\textit{Expr}' = \textit{repetition}(\textit{reduceExpr}(\textit{children}(\textit{ci}), \textit{tail}), c) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{ReduceRepeat} \bullet \\
\quad \textit{reduceExpr}(\textit{tree}, \textit{path}) = \textit{Expr}'
\end{array}
$$

- Something looks wrong here because if a repetition expression has a one-of expression as a child then there is no way to associate the reduced one-of expression with just the path taken in the proof tree since all the children of a repetition expression share the same shape expression. However, a rule-repeat node in the proof tree has many children and there is no requirement that all children would use the same branch of the one-of expression. To make progress, I'll assume that all children of the repeat will eliminate the same branch of the one-of. I will report this to the mailing list later, along with the observation that the reduction should only one done when a one-of expression has more than one component.

5.6 Witness Mappings

Given a set of labelled triples *Neigh*, a shape expression *Expr* and a proof tree *tree* that proves *Neigh* satisfies *Expr*, each labelled triple *triple* appears in a unique leaf node of the proof tree whose rule matches *triple* with a directed triple constraint *dtc*. This association of *triple* with *dtc* is called a witness mapping, $wm(triple) = dtc$.

5.7 *WitnessMapping*

$$WitnessMapping == LabelledTriple \pfun DirectedTripleConstraint$$

5.7.1 *witness*

$$
\begin{array}{l}
witness : ProofTree \longrightarrow WitnessMapping \\
\hline
\forall r : RuleEmpty \bullet \\
\quad \mathbf{let}\ tree == ruleEmpty(r) \bullet \\
\quad\quad witness(tree) = \emptyset \\[1ex]
\forall r : RuleTripleConstraint;\ dtc : DirectedTripleConstraint;\ c : Cardinality \mid \\
\quad r.Expr = triple(dtc, c) \bullet \\
\quad\quad \mathbf{let}\ tree == ruleTripleConstraint(r) \bullet \\
\quad\quad\quad witness(tree) = baseNeigh(tree) \times \{dtc\} \\[1ex]
\forall r : RuleInverseTripleConstraint;\ dtc : DirectedTripleConstraint;\ c : Cardinality \mid \\
\quad r.Expr = triple(dtc, c) \bullet \\
\quad\quad \mathbf{let}\ tree == ruleInverseTripleConstraint(r) \bullet \\
\quad\quad\quad witness(tree) = baseNeigh(tree) \times \{dtc\} \\[1ex]
\forall r : RuleSomeOf;\ subtree : ProofTree \bullet \\
\quad \mathbf{let}\ tree == ruleSomeOf(r, subtree) \bullet \\
\quad\quad tree \in ProofTree \Rightarrow \\
\quad\quad\quad witness(tree) = witness(subtree) \\[1ex]
\forall r : RuleOneOf;\ subtree : ProofTree \bullet \\
\quad \mathbf{let}\ tree == ruleOneOf(r, subtree) \bullet \\
\quad\quad tree \in ProofTree \Rightarrow \\
\quad\quad\quad witness(tree) = witness(subtree) \\[1ex]
\forall r : RuleGroup;\ subtrees : \mathrm{seq}_1\ ProofTree \bullet \\
\quad \mathbf{let}\ tree == ruleGroup(r, subtrees) \bullet \\
\quad\quad tree \in ProofTree \Rightarrow \\
\quad\quad\quad witness(tree) = \bigcup \{\ subtree : \mathrm{ran}\ subtrees \bullet witness(subtree)\ \} \\[1ex]
\forall r : RuleRepeat;\ subtrees : \mathrm{seq}_1\ ProofTree \bullet \\
\quad \mathbf{let}\ tree == ruleRepeat(r, subtrees) \bullet \\
\quad\quad tree \in ProofTree \Rightarrow \\
\quad\quad\quad witness(tree) = \bigcup \{\ subtree : \mathrm{ran}\ subtrees \bullet witness(subtree)\ \}
\end{array}
$$

5.8 *outNeigh*

The outgoing neighbourhood of a node n in an RDF graph G is the set of outgoing labelled triples that correspond to triples in G with subject n.

$$
\begin{array}{|l}
outNeigh : Graph \times TERM \longrightarrow \mathbb{F}\ LabelledTriple \\
\hline
\forall G : Graph;\ n : TERM \bullet \\
\quad outNeigh(G, n) = \{\, p, o : TERM \mid (n, p, o) \in G \bullet out(n, p, o) \,\}
\end{array}
$$

5.9 *incNeigh*

The ingoing neighbourhood of a node n in an RDF graph G is the set of ingoing labelled triples that correspond to triples in G with object n.

$$
\begin{array}{|l}
incNeigh : Graph \times TERM \longrightarrow \mathbb{F}\ LabelledTriple \\
\hline
\forall G : Graph;\ n : TERM \bullet \\
\quad incNeigh(G, n) = \{\, p, s : TERM \mid (s, p, n) \in G \bullet inc(n, p, s) \,\}
\end{array}
$$

5.10 *Typing*

Given a schema S and a graph G, a typing t is a map that associates to each node n of G a, possibly empty, set $t(n)$ of shape labels and negated shape labels such that if `T` is a negated shape label then either `T` or `!T` is in $t(n)$. Here I infer that `T` and `!T` are mutually exclusive.

A typing map associates a finite, possibly empty, set of shape verdicts to nodes.

$$Typing == TERM \mathrel{+\!\!\!\!\!\!\!\to\!\!\to} \mathbb{F}\ ShapeVerdict$$

$$
\begin{array}{|l}
\hline
\textit{TypingMap} \\
G : Graph \\
S : SchemaWD \\
t : Typing \\
\hline
\operatorname{dom} t = nodes(G) \\[4pt]
\forall n : nodes(G);\ T : ShapeLabel \mid assert(T) \in t(n) \bullet \\
\quad T \in shapes(S) \\[4pt]
\forall n : nodes(G);\ T : ShapeLabel \mid negate(T) \in t(n) \bullet \\
\quad T \in negshapes(S) \\[4pt]
\forall n : nodes(G);\ T : negshapes(S) \bullet \\
\quad assert(T) \in t(n) \lor negate(T) \in t(n) \\[4pt]
\forall n : nodes(G);\ T : shapes(S) \bullet \\
\quad assert(T) \notin t(n) \lor negate(T) \notin t(n) \\
\hline
\end{array}
$$

- The typing associates a set of shape verdicts to each node in the graph.
- If a node is required to satisfy T then T must be a shape label of the schema.
- If a node is required to violate T then T must be a negated shape label of the schema.
- If T is a negated shape label of the schema then each node must be required to either satisfy or violate it.
- No node must be required to both satisfy and violate the same shape.

$$
\begin{array}{|l}
\mathit{typings} : \mathit{Graph} \times \mathit{SchemaWD} \longrightarrow \mathbb{P}\ \mathit{Typing} \\
\hline
\forall\, G : \mathit{Graph};\ S : \mathit{SchemaWD} \bullet \\
\qquad \mathit{typings}(G, S) = \{\, m : \mathit{TypingMap} \mid m.G = G \land m.S = S \bullet m.t \,\}
\end{array}
$$

5.11 *TypingSatisfies*

Given a typing t, a node u, and a shape constraint C, the typing satisfies the constraint at the node if the boolean conditions implied by the shape constraint hold.

$$
\begin{array}{|l}
\hline
\textit{TypingSatisfies} \\
\mathit{TypingMap} \\
u : \mathit{TERM} \\
C : \mathit{ShapeConstr} \\
\mathit{Ts} : \mathrm{seq}_1\ \mathit{ShapeLabel} \\
\hline
u \in \mathit{nodes}(G) \\
C = \mathit{and}(\mathit{Ts}) \Rightarrow \\
\qquad (\forall\, T : \mathrm{ran}\ \mathit{Ts} \bullet \mathit{assert}(T) \in t(u)) \\
C = \mathit{or}(\mathit{Ts}) \Rightarrow \\
\qquad (\exists\, T : \mathrm{ran}\ \mathit{Ts} \bullet \mathit{assert}(T) \in t(u)) \\
C = \mathit{nand}(\mathit{Ts}) \Rightarrow \\
\qquad (\exists\, T : \mathrm{ran}\ \mathit{Ts} \bullet \mathit{negate}(T) \in t(u)) \\
C = \mathit{nor}(\mathit{Ts}) \Rightarrow \\
\qquad (\forall\, T : \mathrm{ran}\ \mathit{Ts} \bullet \mathit{negate}(T) \in t(u)) \\
\hline
\end{array}
$$

- The node is in the graph.
- The node is required to satisfy every shape in an and shape constraint.
- The node is required to satisfy some shape in an or shape constraint.
- The node is required to violate some shape in a nand shape constraint.

- The node is required to violate every shape in a nor shape constraint.

$$
\begin{array}{|l}
\mathit{typingSatisfies} : \mathit{Typing} \times \mathit{TERM} \leftrightarrow \mathit{ShapeConstr} \\
\hline
\mathit{typingSatisfies} = \\
\qquad \{\, \mathit{TypingSatisfies} \bullet (t, u) \mapsto C \,\}
\end{array}
$$

5.12 *Matching*

Given a node n in graph G, a typing t, and a directed triple constraint X, let $\mathit{Matching}(G, n, t, X)$ be the set of triples in the graph with focus node n that match X under t.

$$
\begin{array}{|l}
\hline
\mathit{MatchingTriples} \\
\mathit{TypingMap} \\
n, p : \mathit{TERM} \\
X : \mathit{DirectedTripleConstraint} \\
C : \mathit{Constraint} \\
\mathit{triples} : \mathbb{F}\ \mathit{LabelledTriple} \\
\hline
C \in \mathit{ValueConstr} \land X = (\mathit{nop}(p), C) \Rightarrow \\
\qquad \mathit{triples} = \{\, u : \mathit{TERM} \mid (n, p, u) \in G \land \\
\qquad\qquad u \in \mathit{allowed}(C) \bullet \mathit{out}(n, p, u) \,\} \\
C \in \mathit{ShapeConstr} \land X = (\mathit{nop}(p), C) \Rightarrow \\
\qquad \mathit{triples} = \{\, u : \mathit{TERM} \mid (n, p, u) \in G \land \\
\qquad\qquad (t, u)\ \underline{\mathit{typingSatisfies}}\ C \bullet \mathit{out}(n, p, u) \,\} \\
C \in \mathit{ShapeConstr} \land X = (\mathit{inv}(p), C) \Rightarrow \\
\qquad \mathit{triples} = \{\, u : \mathit{TERM} \mid (u, p, n) \in G \land \\
\qquad\qquad (t, u)\ \underline{\mathit{typingSatisfies}}\ C \bullet \mathit{inc}(u, p, n) \,\} \\
\hline
\end{array}
$$

- An outgoing triple matches a value constraint if its object is an allowed value.
- An outgoing triple matches a shape constraint if the typing of its object satisfies the constraint.
- An incoming triple matches a shape constraint if the typing of its subject satisfies the constraint.

$$
\begin{array}{|l}
\mathit{Matching} : \mathit{Graph} \times \mathit{TERM} \times \mathit{Typing} \times \mathit{DirectedTripleConstraint} \longrightarrow \\
\qquad \mathbb{F}\ \mathit{LabelledTriple} \\
\hline
\mathit{Matching} = \{\, \mathit{MatchingTriples} \bullet (G, n, t, X) \mapsto \mathit{triples} \,\}
\end{array}
$$

5.13 *validTypings*

The definition of what it means for a graph to satisfy a shape schema is given in terms of the existence of a valid typing. Given a graph G and a schema S, a valid typing of G by S is a typing that satisfies certain additional conditions at each node n in G.

$$
\begin{array}{|l}
\mathit{validTypings} : \mathit{Graph} \times \mathit{SchemaWD} \longrightarrow \mathbb{P}\ \mathit{Typing} \\
\hline
\forall\ G : \mathit{Graph};\ S : \mathit{SchemaWD} \bullet \\
\qquad \mathit{validTypings}(G, S) \subseteq \mathit{typings}(G, S)
\end{array}
$$

5.13.1 *ValidTypingNodeLabel*

The definition of a valid typing is given in terms of a series of conditions that must hold at each node and for each shape verdict at that node. It is convenient to introduce the following base schema for conditions.

$$
\begin{array}{|l}
\hline
\mathit{ValidTypingNodeLabel} \\
\mathit{TypingMap} \\
n : \mathit{TERM} \\
T : \mathit{ShapeLabel} \\
\mathit{ruleT} : \mathit{Rule} \\
\mathit{defT} : \mathit{ShapeDefinition} \\
\mathit{Expr} : \mathit{ShapeExpr} \\
\mathit{Xs} : \mathbb{F}\ \mathit{DirectedTripleConstraint} \\
\hline
t \in \mathit{validTypings}(G, S) \\
\\
n \in \mathit{nodes}(G) \\
\\
\mathit{assert}(T) \in t(n) \lor \mathit{negate}(T) \in t(n) \\
\\
\mathit{ruleT} = \mathit{rule}(T, S) \\
\mathit{defT} = \mathit{shapeDef}(\mathit{ruleT}) \\
\\
\mathit{Expr} = \mathit{expr}(T, S) \\
\mathit{Xs} = \mathit{tripleConstraints}(\mathit{Expr}) \\
\hline
\end{array}
$$

5.13.2 *tripleConstraints*

Given a shape expression *Expr* let *tripleConstraints*(*Expr*) be the set of all triple or inverse triple constraints contained in it.

$$
\begin{array}{|l}
tripleConstraints : ShapeExpr \longrightarrow \mathbb{F}\ DirectedTripleConstraint \\
\hline
tripleConstraints(emptyshape) = \varnothing \\[1ex]
\forall\ dtc : DirectedTripleConstraint;\ c : Cardinality \bullet \\
\quad tripleConstraints(triple(dtc, c)) = \{dtc\} \\[1ex]
\forall\ Exprs : \mathrm{seq}_1\ ShapeExpr \bullet \\
\quad tripleConstraints(someOf(Exprs)) = \\
\quad tripleConstraints(oneOf(Exprs)) = \\
\quad tripleConstraints(group(Exprs)) = \\
\qquad \bigcup\{\ Expr : \mathrm{ran}\ Exprs \bullet tripleConstraints(Expr)\ \} \\[1ex]
\forall\ Expr : ShapeExpr;\ c : Cardinality \bullet \\
\quad tripleConstraints(repetition(Expr, c)) = tripleConstraints(Expr)
\end{array}
$$

5.13.3 *NegatedShapeLabel*

The semantics draft states:

> for all negated shape label !T, if !T ∈ t(n), then t1 is not a valid typing, where t1 is the typing that agrees with t everywhere, except for T ∈ t1(n)

$$
\begin{array}{|l}
\hline
\textit{NegatedShapeLabel} \\
ValidTypingNodeLabel \\
\hline
negate(T) \in t(n) \\
\hline
\end{array}
$$

- The shape T is negated at node n.

5.13.4 *AssertShape*

$$
\begin{array}{|l}
\hline
\textit{AssertShape} \\
NegatedShapeLabel \\
t1 : Typing \\
\hline
t1 = t \oplus \{n \mapsto (t(n) \setminus \{negate(T)\} \cup \{assert(T)\})\} \\
\hline
\end{array}
$$

- The typing $t1$ is the same as t except that at node n the shape label T is asserted instead of negated.

In a valid typing if any node has a negated shape, then the related typing with this shape asserted is invalid.

$$
\begin{array}{l}
\forall\ AssertShape \bullet \\
\quad t1 \notin validTypings(G, S)
\end{array}
$$

Although this condition on $t(n)$ is recursive in terms of the definition of *validTypings*, it is well-founded since $t1(n)$ has one fewer negated shapes than $t(n)$. Therefore it remains to define the meaning of *validTypings* for typings that contain no negated shapes.

5.13.5 *assertShape*

Given a typing t, node n, and shape label T such that $negate(T) \in t(n)$, define $assertShape(t, n, T)$ to be the typing $t1$ that is the same as t except that $assert(T) \in t1(n)$.

$$
\begin{array}{|l}
assertShape : Typing \times TERM \times ShapeLabel \rightarrowtail Typing \\
\hline
assertShape = \\
\quad \{ AssertShape \bullet \\
\quad\quad (t, n, T) \mapsto t1 \}
\end{array}
$$

5.13.6 *AssertedShapeLabel*

The semantics draft defines the meaning of valid typings t by imposing several conditions that must hold for all nodes n and all asserted shape labels $assert(T) \in t(n)$.

$$
\begin{array}{|l|}
\hline
AssertedShapeLabel \\
ValidTypingNodeLabel \\
\hline
assert(T) \in t(n) \\
\hline
\end{array}
$$

- The shape label T is asserted at node n.

The semantics draft states that the following conditions must hold for all valid typings t and all nodes n such that T is asserted at n:

> for all shape label T, if $T \in t(n)$, then there exist three mutually disjoint sets *Matching*, *OpenProp*, *Rest* such that
>
> 1. $out(G, n) \cup inc(G, n) = Matching \cup OpenProp \cup Rest$, and
> 2. $Rest = Rest_{out} \cup Rest_{inc}$, where
>   - $Rest_{out} = \{(out, n, p, u) \in out(G, n) \mid p \notin properties(expr(T, S))\}$, and
>   - $Rest_{inc} = \{(inc, u, p, n) \in inc(G, n) \mid p \notin invproperties(expr(T, S))\}$, and
> 3. *Matching* is the union of the sets $Matching(n, t, X)$ for all triple constraint or inverse triple constraint X that appears in $expr(T, S)$, and
> 4. if T is a closed shape, then $Rest_{out} = \varnothing$ and $OpenProp = \varnothing$
> 5. if T is an open shape, then $OpenProp \subseteq \{(out, n, p, u) \in out(G, n) \mid p \in incl(T, S)\}$
> 6. there exists a proof tree with corresponding witness mapping *wm* for the fact that *Matching* satisfies $expr(T, S)$, and s.t.

- for all outgoing triple (out, n, p, u), it holds $(out, n, p, u) \in Matching(n, t, wm((out, n, p, u)))$, and moreover if $wm((out, n, p, u))$ is a shape triple constraint, then there is no value triple constraint p::C in $expr(T, S) s.t. (out, n, p, u) \in Matching(n, t, p :: C)$, and
- for all incoming triple $(inc, u, p, n) \in G$, it holds $(inc, u, p, n) \in Matching(n, t, wm((inc, u, p, n)))$, and
- for all node r that corresponds to an application of rule-one-of in the proof tree, there does not exist a valid typing $t1$ of G by S_{ri} s.t. $T \in t1(n)$, and

7. for all extension condition $(lang, cond)$, associated with the type T, $f_{lang}(G, n, cond)$ returns true or undefined.

5.13.7 *MatchingOpenRest*

for all shape label T, if $T \in t(n)$, then there exist three mutually disjoint sets *Matching*, *OpenProp*, *Rest*

MatchingOpenRest
AssertedShapeLabel
MatchingNeigh, *OpenProp*, *Rest* : $\mathbb{F}$ *LabelledTriple*

disjoint $\langle$*MatchingNeigh*, *OpenProp*, *Rest*$\rangle$

- There are three mutually disjoint sets of labelled triples.
- Note that the name *MatchingNeigh* is used to avoid conflict with the previously defined *Matching* function.

$\forall$ *AssertedShapeLabel* $\bullet$
$\exists$ *MatchingNeigh*, *OpenProp*, *Rest* : $\mathbb{F}$ *LabelledTriple* $\bullet$
MatchingOpenRest

5.13.8 *PartitionNeigh*

$out(G, n) \cup inc(G, n) = Matching \cup OpenProp \cup Rest$

PartitionNeigh
MatchingOpenRest

$\langle$*MatchingNeigh*, *OpenProp*, *Rest*$\rangle$ partition
$outNeigh(G, n) \cup incNeigh(G, n)$

$\forall$ *AssertedShapeLabel* $\bullet$
$\exists$ *MatchingNeigh*, *OpenProp*, *Rest* : $\mathbb{F}$ *LabelledTriple* $\bullet$
PartitionNeigh

5.13.9 *RestDef*

$Rest = Rest_{out} \cup Rest_{inc}$, where

- $Rest_{out} = \{(out, n, p, u) \in out(G, n) \mid p \notin properties(expr(T, S))\}$, and
- $Rest_{inc} = \{(inc, u, p, n) \in inc(G, n) \mid p \notin invproperties(expr(T, S))\}$, and

$$
\begin{array}{|l}
\textit{RestDef} \\
\hline
\textit{MatchingOpenRest} \\
\textit{Rest\_out}, \textit{Rest\_inc} : \mathbb{F}\ \textit{LabelledTriple} \\
\hline
\textit{Rest} = \textit{Rest\_out} \cup \textit{Rest\_inc} \\
\textit{Rest\_out} = \\
\quad \{\, p, u : \textit{TERM} \mid \\
\qquad out(n, p, u) \in outNeigh(G, n) \land \\
\qquad p \notin properties(expr(T, S)) \bullet \\
\qquad\quad out(n, p, u) \,\} \\
\textit{Rest\_inc} = \\
\quad \{\, p, u : \textit{TERM} \mid \\
\qquad inc(u, p, n) \in incNeigh(G, n) \land \\
\qquad p \notin invproperties(expr(T, S)) \bullet \\
\qquad\quad inc(u, p, n) \,\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{MatchingOpenRest} \bullet \\
\quad \exists_1\, \textit{Rest\_out}, \textit{Rest\_inc} : \mathbb{F}\ \textit{LabelledTriple} \bullet \\
\qquad \textit{RestDef}
\end{array}
$$

5.13.10 *MatchingDef*

Matching is the union of the sets $Matching(n, t, X)$ for all triple constraint or inverse triple constraint X that appears in $expr(T, S)$

$$
\begin{array}{|l}
\textit{MatchingDef} \\
\hline
\textit{MatchingOpenRest} \\
\hline
\textit{MatchingNeigh} = \\
\quad \bigcup\{\, X : Xs \bullet Matching(G, n, t, X) \,\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{MatchingOpenRest} \bullet \\
\quad \textit{MatchingDef}
\end{array}
$$

5.13.11 *ClosedShapes*

if T is a closed shape, then $Rest_{out} = \emptyset$ and $OpenProp = \emptyset$

$$
\begin{array}{|l}
\hline
\textit{ClosedShapes} \\
\textit{RestDef} \\
\hline
\textit{defT} \in \operatorname{ran} \textit{close} \Rightarrow \\
\quad \textit{Rest\_out} = \emptyset \land \\
\quad \textit{OpenProp} = \emptyset \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall \textit{RestDef} \bullet \\
\quad \textit{ClosedShapes}
\end{array}
$$

5.13.12 *OpenShapes*

if T is an open shape, then

$$OpenProp \subseteq \{(out, n, p, u) \in out(G, n) \mid p \in incl(T, S)\}$$

$$
\begin{array}{|l}
\hline
\textit{OpenShapes} \\
\textit{MatchingOpenRest} \\
\hline
\textit{defT} \in \operatorname{ran} \textit{open} \Rightarrow \\
\quad \textit{OpenProp} \subseteq \\
\quad\quad \{\, p, u : \textit{TERM} \mid \\
\quad\quad\quad \textit{out}(n, p, u) \in \textit{outNeigh}(G, n) \land \\
\quad\quad\quad p \in \textit{incl}(T, S) \bullet \\
\quad\quad\quad\quad \textit{out}(n, p, u) \,\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\forall \textit{MatchingOpenRest} \bullet \\
\quad \textit{OpenShapes}
\end{array}
$$

5.13.13 *ProofWitness*

there exists a proof tree with corresponding witness mapping wm for the fact that $Matching$ satisfies $expr(T, S)$, and s.t.

- for all outgoing triple (out, n, p, u), it holds $(out, n, p, u) \in Matching(n, t, wm((out, n, p, u)))$, and moreover if $wm((out, n, p, u))$ is a shape triple constraint, then there is no value triple constraint p::C in $expr(T, S) s.t.(out, n, p, u) \in Matching(n, t, p :: C)$, and
- for all incoming triple $(inc, u, p, n) \in G$, it holds $(inc, u, p, n) \in Matching(n, t, wm((inc, u, p, n)))$, and

- for all node r that corresponds to an application of rule-one-of in the proof tree, there does not exist a valid typing $t1$ of G by S_{ri} s.t. $T \in t1(n)$, and

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ProofWitness}} \\ \hline
\textit{MatchingDef} \\
\textit{tree} : \textit{ProofTree} \\
\textit{wm} : \textit{WitnessMapping} \\ \hline
\textit{baseNeigh}(\textit{tree}) = \textit{MatchingNeigh} \\
\textit{baseExpr}(\textit{tree}) = \textit{Expr} \\
\\
\textit{wm} = \textit{witness}(\textit{tree}) \\ \hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{MatchingDef} \bullet \\
\quad \exists\, \textit{tree} : \textit{ProofTree};\ \textit{wm} : \textit{WitnessMapping} \bullet \\
\quad\quad \textit{ProofWitness}
\end{array}
$$

5.13.14 *OutgoingTriples*

for all outgoing triple (out, n, p, u), it holds

$$(out, n, p, u) \in Matching(n, t, wm((out, n, p, u))),$$

and moreover if $wm((out, n, p, u))$ is a shape triple constraint, then there is no value triple constraint p::C in $expr(T, S)$ s.t.

$$(out, n, p, u) \in Matching(n, t, p :: C)$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OutgoingTriples}} \\ \hline
\textit{ProofWitness} \\ \hline
\forall\, \textit{triple} : \textit{outNeigh}(G, n);\ p, u : \textit{TERM} \mid \\
\quad \textit{triple} = \textit{out}(n, p, u) \bullet \\
\quad\quad \textbf{let } X == \textit{wm}(\textit{triple}) \bullet \\
\quad\quad\quad \textit{triple} \in \textit{Matching}(G, n, t, X) \wedge \\
\quad\quad\quad (\textit{constrDTC}(X) \in \textit{ShapeConstr} \Rightarrow \\
\quad\quad\quad\quad \neg\, (\exists\, C : \textit{ValueConstr} \mid (\textit{nop}(p), C) \in \textit{Xs} \bullet \\
\quad\quad\quad\quad\quad \textit{triple} \in \textit{Matching}(G, n, t, (\textit{nop}(p), C)))) \\ \hline
\end{array}
$$

$$
\begin{array}{l}
\forall\, \textit{ProofWitness} \bullet \\
\quad \textit{OutgoingTriples}
\end{array}
$$

5.13.15 *IncomingTriples*

for all incoming triple $(inc, u, p, n) \in G$, it holds

$$(inc, u, p, n) \in Matching(n, t, wm((inc, u, p, n)))$$

$$\begin{array}{|l}
\hline
\textit{IncomingTriples} \\
\textit{ProofWitness} \\
\hline
\forall\, triple : incNeigh(G, n) \bullet \\
\quad \mathbf{let}\ X == wm(triple) \bullet \\
\quad\quad triple \in Matching(G, n, t, X) \\
\hline
\end{array}$$

$\forall\, ProofWitness \bullet$
$\quad IncomingTriples$

5.13.16 *OneOfNodes*

for all node r that corresponds to an application of rule-one-of in the proof tree, there does not exist a valid typing $t1$ of G by S_{ri} s.t. $T \in t1(n)$

Let *OneOfNodes* describe the situation where we are given a graph G, a schema S, a typing t of G under S, a node n in G, a shape label T in $t(n)$, a proof tree *tree* for the triples *MatchNeigh* and the expression $Expr = expr(T, S)$ and an application of rule-one-of r in the proof tree.

$$\begin{array}{|l}
\hline
\textit{OneOfNodes} \\
\textit{ProofWitness} \\
\textit{RuleOneOfApplication} \\
Expr_ri : ShapeExpr \\
S_ri : SchemaWD \\
\hline
Expr_ri = reduceExpr(tree, path) \\
S_ri = replaceShape(S, T, Expr_ri) \\
\hline
\end{array}$$

Whenever rule-one-of is applied in the proof tree, there must not be any valid typings $t1$ for the reduced schema S_ri in which the selected component of the one-of shape expression is eliminated.

$\forall\, OneOfNodes \bullet$
$\quad \neg\, (\exists\, t1 : validTypings(G, S\_ri) \bullet$
$\quad\quad assert(T) \in t1(n))$

5.13.17 *ExtensionConditions*

for all extension condition $(lang, cond)$, associated with the type T, $f_{lang}(G, n, cond)$ returns true or undefined

The semantics of an extension condition is given by a language oracle function that evaluates the extension condition *cond* on a pointed graph (G, n) and returns a code indicating whether the pointed graph satisfies the extension condition, or if an error condition holds, or if the extension condition is undefined.

$$
\begin{array}{|l}
f : ExtLangName \times Graph \times TERM \times ExtDefinition \longrightarrow ReturnCode \\
\hline
\forall\, G : Graph;\ n : TERM \mid (G, n) \in PointedGraph \bullet \\
\quad \forall\, lang : ExtLangName;\ cond : ExtDefinition \bullet \\
\quad\quad \mathbf{let}\ returnCode == f(lang, G, n, cond) \bullet \\
\quad\quad\quad returnCode = trueRC \Rightarrow (G, n) \notin violatedBy(lang, cond) \land \\
\quad\quad\quad returnCode = falseRC \Rightarrow (G, n) \in violatedBy(lang, cond)
\end{array}
$$

- If the oracle returns true then the pointed graph satisfies the extension condition.
- If the oracle returns false then the pointed graph violates the extension condition.

Let the return codes for the language oracles be *ReturnCode*.

$$ReturnCode ::= trueRC \mid falseRC \mid errorRC \mid undefinedRC$$

- true means the extension condition is satisfied.
- false means the extension condition is violated.
- error means an error occurred.
- undefined means the extension condition is undefined.

$$
\begin{array}{|l}
\hline
ExtensionConditions \\
MatchingOpenRest \\
lang : ExtLangName \\
cond : ExtDefinition \\
\hline
\mathbf{let}\ ecs == extConds(ruleT) \bullet \\
\quad (lang, cond) \in \operatorname{ran} ecs \\
\hline
\end{array}
$$

- $(lang, cond)$ is an extension condition for T.

$$
\begin{array}{l}
\forall\, ExtensionConditions \bullet \\
\quad f(lang, G, n, cond) \in \{trueRC, undefinedRC\}
\end{array}
$$

6 Issues

Some areas of the semantics draft have multiple interpretations or appear to be wrong and therefore require further clarification. These areas are discussed below.

6.1 `dep-subgraph(T,S)`

In the definition of `dep-subgraph(T,S)`, is the shape `T` considered to be reachable from itself?

6.2 `negshapes(S)`

In the definition of `negshapes(S)`, the third bullet states:

```
there is a shape label T1 and a shape triple constraint p::C,
or an inverse shape triple constraints ^p::C in expr(T1, S),
and T appears in C.
```

This statement looks wrong because it omits mention of negation. If there is no negation involved, why would `T` be in `negshapes(S)`?

Does this definition only select directed triple constraints that have cardinality `[1,1]` because that is the default? If not then `negshapes(S)` is the set of all labels that are referenced in any shape definition ($\mathit{refs}(S)$), which seems wrong.

6.3 Triple matches constraint

The definition of matching `p:C` and `^p:C` omits consideration of `C`. The explanation is as follows.

```
The following definition introduces the notion of satisfiability
of a shape constraint by a set of triples. Such satisfiability
is going to be used for checking that the neighborhood of
a node satisfies locally the constraints defined by a shape
expression, without taking into account whether the shapes
required by the triple constraints and inverse triple constraints
are satisfied.
```

This statement implies that only shape constraints should be ignored here. However, the definition ignores the value set constraints too. This looks wrong.

6.4 `rule-triple-constraint`

Add the condition that all the outgoing triples must be distinct.

6.5 `rule-inverse-triple-constraint`

Add the condition that all the incoming triples must be distinct.

6.6 `rule-group`

Add the condition that `i` and `j` must be different.

6.7 `rule-repeat`

Add the condition that `i` and `j` must be different.

6.8 Reduced Schema for `rule-one-of`

This is an edge case. It only makes sense to reduce the schema if there are more than one components. Applying `rule-one-of` to a sequence of one shape is equivalent to requiring that shape. Add this condition to the definition.

6.9 Reduced Schema for `rule-one-of` under a repetition expression

Something looks wrong here because if a repetition expression has a one-of expression as a child then there is no way to associate the reduced one-of expression with just the path taken in the proof tree since all the children of a repetition expression share the same shape expression. However, a rule-repeat node in the proof tree has many children and there is no requirement that all children would use the same branch of the one-of expression.

7 Conclusion

The exercise of formalizing the semantics draft has resulted in a considerable expansion in the size of the document. The result has been the identification of a number of quality issues. This exercise has also established that the recursive definitions in the semantics draft are well-founded. However, it is not clear that these definitions produce results that agree with our intuition, or that they can be computed efficiently.

One possible way to further validate the semantics draft is to translate it into an executable formal specification system such as Coq [1] and test it on a set of examples, including both typical documents and corner cases.

References

[1] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development.* Springer-Verlag, 2004.

[2] Boneva, I., and Prud'hommeaux, E. Core SHACL Semantics. W3C Editor's Draft, World Wide Web Consortium, June 2015. http://w3c.github.io/data-shapes/semantics/.

[3] Le Hors, A. RDF Data Shapes Working Group. wiki page, World Wide Web Consortium, Sept. 2014. https://www.w3.org/2014/data-shapes/wiki/Main_Page.

[4] Ryman, A. agryman/z-core-shacl-semantics. source code repository, GitHub, 2015. https://github.com/agryman/z-core-shacl-semantics.

[5] Ryman, A. Recursion in RDF Data Shape Languages. Article, World Wide Web Consortium, May 2015. http://arxiv.org/abs/1505.04972.

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

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