Semantics and Validation of Shapes Schemas for RDF

Iovka Boneva[1(✉)], Jose E. Labra Gayo[2], and Eric G. Prud'hommeaux[3]

[1] Univ. Lille - CRIStAL, 59000 Lille, France
iovka.boneva@univ-lille.fr
[2] University of Oviedo, Oviedo, Spain
labra@uniovi.es
[3] W3C and Stata Center, MIT, Cambridge, USA
eric@w3.org

Abstract. We present a formal semantics and proof of soundness for shapes schemas, an expressive schema language for RDF graphs that is the foundation of Shape Expressions Language 2.0. It can be used to describe the vocabulary and the structure of an RDF graph, and to constrain the admissible properties and values for nodes in that graph. The language defines a typing mechanism called shapes against which nodes of the graph can be checked. It includes an algebraic grouping operator, a choice operator and cardinality constraints for the number of allowed occurrences of a property. Shapes can be combined using Boolean operators, and can use possibly recursive references to other shapes.

We describe the syntax of the language and define its semantics. The semantics is proven to be well-defined for schemas that satisfy a reasonable syntactic restriction, namely stratified use of negation and recursion. We present two algorithms for the validation of an RDF graph against a shapes schema. The first algorithm is a direct implementation of the semantics, whereas the second is a non-trivial improvement. We also briefly give implementation guidelines.

1 Introduction

RDF's distributed graph model encouraged adoption for publication and manipulation of e.g. social and biological data. Coding errors in data stores like DBpedia have largely been handled in a piecemeal fashion with no formal mechanism for detecting or describing schema violations. Extending uptake into environments like medicine, business and banking requires structural validation analogous to what is available in relational or XML schemas.

While OWL ontologies can be used for limited structural validation, they are generally used for formal models of reusable classes and predicates describing objects in some domain. Applications typically consume and produce graphs composed of precise compositions of such ontologies. A company's human resources records may leverage terms from FOAF and Dublin Core, but only certain terms, composed into specific structures, and subject to additional use-specific constraints. We would no more want to impose the constraints of a single

C. d'Amato et al. (Eds.): ISWC 2017, Part I, LNCS 10587, pp. 104–120, 2017.
DOI: 10.1007/978-3-319-68288-4_7

human resources application suite on FOAF and Dublin Core than we would want to assert that such applications need to consume all ontologically valid permutations of FOAF and Dublin Core entities. Further, open-world constraints on OWL ontologies make it impossible to use conventional OWL tools to e.g. detect missing properties. Shape expression schemas (ShEx 1.0) [6,8] were introduced as a high level language in which it is easy to mix terms from arbitrary ontologies. They provide a schema language in which one can define structural constraints (arc labels, cardinalities, datatypes, etc.) and since version 2.0 (ShEx 2.0)[1], mix them using Boolean connectives (disjunction, conjunction and negation).

A schema language for any data format has several uses: communicating to humans and machines the form of input/output data; enabling machine-verification of data for production, publication, or consumption; driving query and input interfaces; static analysis of queries. In this, ShEx provides a similar role as relational and XML schemas. A ShEx schema validates nodes in a graph against a schema construct called a *shape*. In XML, validating an element against an XML Schema[2] type or element or Relax NG[3] production recursively tests nested elements against constituent rules. In ShEx, validating a node in a graph against a shape recursively tests the nodes which are the object of triples constrained in that shape. An essential difference however is that unlike trees, graphs can have cycles and recursive definitions can yield infinite computation. Moreover, ShEx 2.0 includes a negation operator, and it is well known that mixing recursion with negation can lead to incoherent semantics.

Contributions. In this paper we present *shapes schemas*, a schema language that is the foundation of ShEx 2.0 (Sect. 2). The precise relationship between shapes schemas and ShEx 2.0 is given at the end of Sect. 2. We formally define the semantics of shapes schemas and show that it is sound for schemas that mix recursion and negation in a stratified manner (Sect. 3). We then propose two algorithms for validating an RDF graph node against a shapes schema. Both algorithms are shown to be correct w.r.t. the semantics (Sect. 4). We finally discuss future research directions and conclude (Sect. 5).

Related Work. In [8] we gave semantics for ShEx 1.0. The latter does not use Boolean operators end because of negation, the extension to ShEx 2.0 (and thus to shapes schemas) is non trivial.

Closest to shapes schemas is the SHACL[4] language both in terms of purpose and expressiveness. SHACLalso defines named constraints called shapes to be checked on RDF graph nodes. Unlike ShEx, SHACLis not completely independent from the RDF Schema vocabulary: rdfs:Classes play a particular role there as a shape can be required to hold for all the nodes that are instances of some rdfs:Class. Therefore validation in SHACLrequires partial RDF Schema entailment

[1] Shape Expressions Language 2.0. http://shex.io/shex-semantics/index.html.
[2] W3C XML Schema. http://www.w3.org/XML/Schema.
[3] RELAX NG home page. http://relaxng.org.
[4] Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/.

in order to discover all rdfs:Classes of a node. Regarding expressiveness, the main differences between SHACLand shapes schemas are that SHACLallows to define constraints based on property paths and for comparison of values; SHACLdoes not have the algebraic operators some-of and each-of and uses Boolean connectives for defining complex shapes; finally SHACLdoes not define the semantics of recursive shapes.

Ontology languages such as OWL, description logics or RDF Schema are not meant to define (complex) constraints on the data and we do not compare shapes schemas with them. Proposals were made for using OWL with a closed world assumption in order to express integrity constraints [5,9]. They associate alternative semantics with the existing OWL syntax and can be misleading for users.

Some approaches use SPARQL to express constraints on graphs (SPIN[5], RDFUnit [3]), or compile a domain specific language into SPARQL queries [2]. SPARQL allows to express complex constraints but does not support recursion. While SPARQL constraints can be validated by standard SPARQL engines, they are harder to write and maintain compared to high-level schemas like ShEx and SHACL.

Description Set Profiles[6] is a constraint language that uses an RDF vocabulary to define templates and constrain the value and cardinality of properties. It does not have any equivalent of the each-of algebraic operator, and was not designed to be human-readable.

Introductory Example. Let is: be a namespace prefix from some ontology, ex: be the prefix used in the example schema and instance, and foaf: and xsd: be the standard FOAF and XSD prefixes, respectively. The schema $\mathbf{S}_0$ is as follows

```
<UserShape>      → foaf:name @<StringValue> ; foaf:mbox @<IRIValue> [0;1]
<ProgShape>      → ex:expertise @<IRIValue> [0;*] ; ex:experience @<ExpValueSet>
<ClientShape>    → ex:clientNbr @<IntValue> | ex:clientAffil @<AnythgShape>
<IssueShape>     → is:reportedBy @<ClientAndUser> ;
                   is:reproducedBy @<ProgShape> [1;5] ;
                   is:relatedTo @<IssueShape> [0;*]
<AnythgShape>    → IRI @<AnythgShape> [0;*]
<ClientAndUser>  → Def_Client ; IRI − {ex:clientNbr, ex:clientAffil} @<AnythgShape> [0;*]
                   AND Def_User ; IRI − {foaf:name, foaf:mbox} @<AnythgShape> [0;*]
<StringValue>    → xsd:string
<IRIValue>       → IRI
<ExpValueSet>    → {ex:senior, ex:junior}
<IntValue>       → xsd:integer
```

where Def_{Client} is the definition of <ClientShape>, and similarly for Def_{User}, and $-$ in the definition of <ClientAndUser> is the set difference operator. The schema $\mathbf{S}_0$ defines four shapes intended to describe users, programmers, clients and issues, respectively. <UserShape> requires that a node has one foaf:name property with string value, and an optional foaf:mbox that is an IRI. The optional mailbox is specified by the cardinality constraint [0;1]. Other

[5] SPIN - Modeling Vocabulary. http://www.w3.org/Submission/spin-modeling/.

[6] Description Set Profiles: A constraint language for Dublin Core Application Profiles. http://dublincore.org/documents/dc-dsp/.

cardinality constraints used in $\mathbf{S}_0$ are [0;*] for zero or more, and [1;5] for one up to five. When no cardinality is given, the default is "exactly one". A <ProgShape> node has zero or more ex:expertise properties with values that are IRIs, and one ex:experience property whose value is one among ex:senior and ex:junior. A <ClientShape> has either a ex:clientNbr that is an integer, or a ex:clientAffil(iation) with unconstrained value (i.e. <AnythgShape>), but not both. Finally, an issue (<IssueShape>) is reported by somebody who is client and user, is reproduced by one to five programmers, and can be related to zero or more issues.

The shapes in $\mathbf{S}_0$ whose name contains Value specify the set of allowed values for a node. This can be the set of all values of some literal datatype (e.g. string, integer), the set of all nodes of some kind (e.g. IRI), or an explicitly given set (e.g. <ExpValueSet>). <AnythgShape> is satisfied by every node. It states that the node can have zero or more outgoing triples whose predicates can be any IRI, and whose objects match <AnythgShape>. Finally, <ClientAndUser> uses a conjunction to require that a node has both the client and the user properties. Its definition is a bit technical. The right hand side of the conjunction states that the node must have a foaf:name and an optional foaf:mbox (Def_{User}). Moreover (the ; operator), the node can have any number ([0;*]) of properties that can be any IRI except for foaf:name and foaf:mbox and whose value is unconstrained. The latter is necessary in order to allow the "client" properties required by the left hand side of the conjunction.

Graph $\mathbf{G}_0$ here after is described by schema $\mathbf{S}_0$. Nodes ex:issue1 and ex:issue2 have shape <IssueShape>; ex:fatima and ex:emin are <ClientAndUser>; ex:ren and ex:noa have shape <ProgShape>.

```
ex:issue1
   is:reportedBy ex:fatima ;
   is:reproducedBy ex:ren , ex:noa ;
   is:relatedTo ex:issue2 .
ex:issue2
   is:reportedBy ex:emin ;
   is:reproducedBy ex:ren ;
   is:relatedTo ex:issue1 .
ex:fatima ex:clientNbr 1 ;
   foaf:name "Fatima Smith".
ex:ren ex:expertise ex:semweb ;
   ex:experience ex:senior .
ex:noa ex:experience ex:junior .
ex:emin ex:clientAffil "ABC";
   foaf:name "Emin V. Petrov" ;
   foaf:mbox <mailto:evp@example.org> .
```

The RDF Graph Model. As usual, we assume three disjoint sets: IRI a set of IRIs, Lit a set of literals, and Blank a set of blank nodes. An *RDF graph* is a set of triples over $\mathsf{IRI} \cup \mathsf{Blank} \times \mathsf{IRI} \times \mathsf{IRI} \cup \mathsf{Lit} \cup \mathsf{Blank}$. For a triple (s, p, o) in some graph, s is called its subject, p is called its predicate, and o is called its object. We denote $\mathbf{N}\text{odes}(\mathbf{G})$ the set of *nodes* of the graph $\mathbf{G}$, that is, the elements that appear in a subject or object position in some triple of $\mathbf{G}$. The *neighbourhood* of node n in graph $\mathbf{G}$ is the set of triples in $\mathbf{G}$ that have n as subject, and is denoted $neigh_{\mathbf{G}}(n)$ or simply $neigh(n)$ when $\mathbf{G}$ is clear from the context. We use disjoint union on sets of triples, denoted $\uplus$: if N, N_1, N_2 are sets of triples, $N = N_1 \uplus N_2$ means that $N_1 \cup N_2 = N$ and $N_1 \cap N_2 = \emptyset$.

2 Shapes Schemas

A shapes schema $\mathbf{S}$ defines a set of named shapes. A shape is a description of the graph structure that can be visited starting from a particular node. It can talk about the value of the node itself and about its neighbourhood. Shapes can use (Boolean combinations of) other shapes and can be recursive.

Formally, a shapes schema $\mathbf{S}$ is a pair $(\mathbf{L}, \mathbf{def})$, where $\mathbf{L}$ is a set of *shape labels* used as names of shapes and $\mathbf{def}$ is a function that with every shape label associates a shape expression. In examples, we write $L \to S$ as short for $\mathbf{def}(L) = S$ (for a shape label L and a shape expression S).

Shape Expressions. The grammar for shape expressions is given on Fig. 1a. A *shape expression* (ShExpr) is a Boolean combination of two atomic components: value description and neighbourhood description. A neighbourhood description (NeigDescr) defines the expected neighbourhood of a node and is given by a triple expression (TExpr, see below). A value description (ValueDescr) is a set that declares the admissible values for a node. The set can contain IRIs, literals, and the special constant _b to indicate that the node can be a blank node. ShEx 2.0 proposes concrete syntax for different kinds of value description sets (literal datatypes, regex patterns to be matched by IRIs, intervals, etc.). Here we focus on defining the semantics so the concrete syntax for such sets is irrelevant. A ValueDescr can be an arbitrary set with the unique assumption that it has a finite representation for which membership can be effectively computed.

ShExpr	::=	ValueDescr | NeigDescr | ShapeAnd | ShapeOr | ShapeNot
ValueDescr	::=	a subset of $\mathsf{IRI} \cup \mathsf{Lit} \cup \{\_\mathsf{b}\}$
NeigDescr	::=	TExpr
ShapeAnd	::=	ShExpr 'AND' ShExpr
ShapeOr	::=	ShExpr 'OR' ShExpr
ShapeNot	::=	'NOT' ShExpr

(a) Shape expressions.

TExpr	::=	TriplePattern | 'EMPTY' | SomeOfExpr | EachOfExpr | RepetExpr
TriplePattern	::=	PropSet '@'ShapeRef
SomeOfExpr	::=	TExpr '|' TExpr
EachOfExpr	::=	TExpr ';' TExpr
RepetExpr	::=	TExpr '[' min ';' max ']'
PropSet	::=	a subset of IRI
ShapeRef	::=	a shape label in $\mathbf{L}$

(b) Triple expressions.

Fig. 1. The grammar for shape expressions and triple expressions.

Triple expressions. Triple expressions describe the expected neighbourhood of a node. They are inspired by regular expressions likewise DTDs and XML Schema for XML. A triple expression will be matched by the neighbourhood of a node in a graph, similarly to type definitions in XML Schema that are matched by the children of some node. The main difference is that the neighbourhood of a node in an RDF graph is a (unordered) set, whereas the children of a node in an XML document form a sequence.

The grammar for triple expressions (TExpr) is given on Fig. 1b, in which *min* is a natural, and *max* is a natural or the special value $*$. The basic triple expression is a triple pattern and it constrains triples. A triple expression composed of each-of (separated by a ';'), some-of (separated by a '|') and repetition operators is satisfied if some distribution of the triples in the neighborhood of a node exactly satisfies the expression. Section 3.1 defines this and draws the analogy with regular expressions. In examples, we omit the braces for singleton PropSets, e.g. we write foaf:name @<StringValue> instead of {foaf:name} @<StringValue>.

Example 1 (Shape expressions, triple expressions). In schema $\mathbf{S}_0$ from the introductory example, the definitions of the five shapes with name ... Shape... are triple expressions and collectively make use of all the operators: each-of (;), some-of (|), repetition. All shapes with name ... Value... are defined by atomic ValueDescrs. The definition of <ClientAndUser> is a ShapeAnd expression. □

Relationship between Shapes Schemas and ShEx 2.0. Shapes schemas slightly generalizes ShEx 2.0 and thus allows for a more concise definition of syntax and semantics. For readers familiar with ShEx 2.0 we now explain how shapes schemas differ from ShEx 2.0. First, TriplePattern uses a set of properties, whereas the analogous triple constraint in ShEx 2.0 uses a single property. This slight generalization allows to encode the CLOSED and EXTRA constructs of ShEx 2.0. In shapes schemas, triple expressions are always closed (whereas in ShEx 2.0 they are non closed by default) but an expression E can be made non-closed by transforming it into $E; P$ @<AnythgShape>$[0; *]$, where P is the set of all IRIs not mentioned as properties in E, and <AnythgShape> is as defined in the introductory example. The EXTRA modifier is encoded in a similar way, using sets of properties in triple patterns and negation. Second, a ValueDescr is an arbitrary set of values that can be IRIs, literals or blank nodes, whereas the analogous node constraint in ShEx 2.0 defines a set of allowed values using a combination of elementary constraints such as XSD datatypes, facets, numerical intervals, node kinds. Using an arbitrary set of values allows to get rid of unnecessary (w.r.t. defining the semantics) details. Third, ShEx 2.0 allows to use shape labels in shape definitions; this is syntactic sugar and is equivalent to replacing the label by its definition. Finally, in shapes schemas we omit inverse properties which would make the proofs longer without representing any additional challenge w.r.t. the semantics.

3 Semantics of Shapes Schemas

A shape defines the structure of a graph when visited starting from a node that has that shape. In this section we give a precise meaning of the following statement

SHAPES_SEM: node n in graph $\mathbf{G}$ has shape (or type) L from schema $\mathbf{S}$[7]

[7] "type" is used as synonym of "shape", esp. in the notion of *typing* to be introduced shortly. The use of "type" must not be confused with rdf:type from RDF Schema. Shapes schemas are totally independent from the RDF Schema vocabulary.

To give a sound definition for SHAPES_SEM is not trivial because of the presence of recursion. It also requires to make a design choice that we explain now.

Example 2 (Simple recursive schema). Let schema $\mathbf{S}_1$ and graph $\mathbf{G}_1$ be:

```
<IssueSh> → is:reportedBy @<Str> ; is:relatedTo @<IssueSh> [0;*]
<Str>     → xsd:string

<i1> is:reportedBy "Ren" ; is:relatedTo <i2> .
<i2> is:reportedBy "Bob" ; is:relatedTo <i1> .
```

Example 2 captures the essence of recursion. If <i1> has shape <IssueSh> then <i2> also has shape <IssueSh>. If on the other hand <i1> does not have shape <IssueSh>, then neither does <i2>. This illustrates two important aspects of the semantics of shapes schemas. First, whether a node has some shape cannot be defined independently of the shapes of the other nodes in the graph. The consequence of this apparently simple fact is that we need a global statement about which nodes satisfy which shapes; we call this a typing. A typing must be correct, i.e. coherent with itself. Second, in the above example there is a (design) choice to make. Clearly, there are two acceptable alternatives: either (1) both <i1> and <i2> have shape <IssueSh>, or (2) none of them does. Such choice is well known for recursive languages: (1) corresponds to a maximal solution, and (2) to a minimal solution. Both choices can lead to sound semantics. In shapes schemas we choose the maximal solution. This is justified by applications: in the above example we *do want* to consider <i1> as a valid <IssueSh>. It would not be the case with semantics based on a minimal solution.

3.1 Typing and Correct Typing

The semantics is based on the notion of *typing*: this is a set of couples that associate a node of an RDF graph with a shape label (a type). In the sequel we consider a graph $\mathbf{G}$ and a schema $\mathbf{S} = (\mathbf{L}, \mathbf{def})$.

Definition 1 (node-type association, typing). *A* node-type association *is a couple* (n, L) *in* $\mathrm{Nodes}(\mathbf{G}) \times \mathbf{L}$. *A* typing of $\mathbf{G}$ by $\mathbf{S}$ *is a set of node-type associations.*

Example 3. With $\mathbf{S}_1$ and $\mathbf{G}_1$ from Example 2, the following are typings

$$\mathit{typing}_1 = \{(<\mathsf{i1}>,\ <\mathsf{IssueSh}>), (<\mathsf{i2}>,\ <\mathsf{IssueSh}>), (\text{"Ren"},\ <\mathsf{Str}>), (\text{"Bob"},\ <\mathsf{Str}>)\}$$
$$\mathit{typing}_2 = \{(\text{"Ren"},\ <\mathsf{Str}>), (\text{"Bob"},\ <\mathsf{Str}>)\}$$
$$\mathit{typing}_3 = \{(<\mathsf{i1}>,\ <\mathsf{IssueSh}>), (<\mathsf{i2}>,\ <\mathsf{IssueSh}>)\}$$
$$\mathit{typing}_4 = \emptyset.$$

A typing is correct if, intuitively, it contains an evidence for every node-type association in it. In the above example $typing_1$ and $typing_2$ are correct, whereas $typing_3$ is not correct as it contains e.g. the association (<i1>, <IssueSh>) but does not contain the association ("Ren", <Str>) that is required for <i1> to have type <IssueSh>. The empty typing ($typing_4$) is always correct.

Definition 2 (correct typing). *Let* $typing \subseteq \mathbf{N}\mathrm{odes}(\mathbf{G}) \times \mathbf{S}$. *We say that* $typing$ is a correct typing *if for any* $(n, L) \in typing$, *it holds* $typing, n \vdash \mathbf{def}(L)$, *where* $\vdash$ *is the relation defined on Fig. 2a.*

$$\text{se-value-descr1}\ \frac{n \in V}{typing, n \vdash V}$$

$$\text{se-value-descr2}\ \frac{\_\mathsf{b} \in V \quad n \in \mathsf{Blank}}{typing, n \vdash V}$$

$$\text{se-neig-descr}\ \frac{typing, neigh(n) \vDash E}{typing, n \vdash E}$$

$$\text{se-shape-and}\ \frac{typing, n \vdash S_1 \quad typing, n \vdash S_2}{typing, n \vdash S_1 \ \mathsf{AND}\ S_2}$$

$$\text{se-shape-or1}\ \frac{typing, n \vdash S_1}{typing, n \vdash S_1 \ \mathsf{OR}\ S_2}$$

$$\text{se-shape-or2}\ \frac{typing, n \vdash S_2}{typing, n \vdash S_1 \ \mathsf{OR}\ S_2}$$

$$\text{se-shape-not}\ \frac{typing, n \nvdash S}{typing, n \vdash \mathsf{NOT}\ S}$$

(a) Node satisfies a shape expression.

$$\text{te-tpattern}\ \frac{N = \{(subj, pred, obj)\} \quad pred \in P \quad (obj, L) \in typing}{typing, N \vDash P\ @L}$$

$$\text{te-empty}\ \frac{N = \emptyset}{typing, N \vDash \mathsf{EMPTY}}$$

$$\text{te-some-of1}\ \frac{typing, N \vDash E_1}{typing, N \vDash E_1 | E_2}$$

$$\text{te-some-of2}\ \frac{typing, N \vDash E_2}{typing, N \vDash E_1 | E_2}$$

$$\text{te-each-of}\ \frac{N = N_1 \uplus N_2 \quad typing, N_1 \vDash E_1 \quad typing, N_2 \vDash E_2}{typing, N \vDash E_1 ; E_2}$$

$$\text{te-repet}\ \frac{N = N_1 \uplus \ldots \uplus N_k \quad min \le k \le max \quad typing, N_i \vDash E \text{ for all } 0 \le i \le k}{typing, N \vDash E[min; max]}$$

(b) Set of triples matches a triple expression.

Fig. 2. Definitions of the $\vdash$ and $\vDash$ relations.

Discussion on $\vdash$. For a shape expression S, the definition of $typing, n \vdash S$ on Fig. 2a is by recursion on the structure of S. In Rules se-value-descr, V is a subset of $\mathsf{IRI} \cup \mathsf{Lit} \cup \{\_\mathsf{b}\}$ defining a ValueDescr. A node n satisfies the value description V if n belongs to the set V, or if n is a blank node and _b is in V. The other base case is Rule se-neig-descr, in which E is a TExpr representing a neighbourhood description. A node n satisfies the NeigDescr E if the neighbourhood of n matches the triple expression E. The matching relation $\vDash$ is defined on Fig. 2b and discussed below. The remaining four rules are for the Boolean operators. The rules for AND and OR are as one would expect. Regarding negation, a node satisfies a ShapeNot expression if it does not satisfy its sub-expression, as stated by Rule se-shape-not. The premise of that rule is $typing, n \nvdash S$ and means that (using the inference rules on Fig. 2a) it is impossible to construct a proof for $typing, n \vdash S$.

Discussion on $\vDash$. For a set of triples N, *a typing typing and a* TExpr E, *we say that N matches E with typing*, and we write $typing, N \vDash E$, as defined recursively on the structure of E on Fig. 2b. Note that the $\vDash$ relation is defined for an arbitrary set of triples N. In practice, N will be (a subset of) the neighbourhood of some node. In the basic Rule te-tpattern, $P@L$ is a TriplePattern with P a set of IRIs and L a shape label. A singleton set of triples $\{(subj, pred, obj)\}$ matches the triple pattern if the predicate *pred* belongs to P and the object has type L in *typing*. The other basic rule is Rule te-empty: an empty set of triples satisfies the EMPTY triple expression.

The remaining rules are about the composed triple expressions. A set of triples matches a SomeOfExpr if it matches one of its sub-expressions (Rules te-some-of). The semantics of a EachOfExpr is a bit more complex. A set N matches an each-of triple expression $E_1; E_2$ if N is the disjoint union of two sets N_1 and N_2, and N_1 matches the sub-expressions E_1, and N_2 matches the sub-expression E_2. Let us make a parallel between regular expressions and triple expressions. The each-of operator is analogous to concatenation. Recall that a string w matches a regular expression $R_1 \cdot R_2$ (where $\cdot$ is concatenation) whenever w can be "split" into two strings w_1 and w_2 such that their concatenation gives w ($w = w_1 \cdot w_2$), and w_1 matches R_1, and w_2 matches R_2. In the case of triple expressions, the set of triples N is "split" into two disjoint sets N_1 and N_2: disjoint union on sets is analogous to concatenation on words. Following the same analogy, repetition in triple expressions corresponds to Kleene star (the star operator) in regular expressions, with the difference that it allows to express arbitrary intervals for the number of allowed repetitions, whereas Kleene star is always $[0, *]$. So, in Rule te-repet, a set of triples N matches a repetition triple expression $E[min; max]$ if N can be split as the disjoint union of k sets $N_1, \ldots, N_k$ such that k is within the interval bound $[min; max]$ and each of these sets matches the sub-expression E. Note that $k = 0$ is possible only when $N = \emptyset$.

The laws of the Boole algebra can be used to put a shape expression in disjunctive normal form in which only atomic sub-expressions ValueDescr and NeigDescr are negated. From now on we consider only shape expressions in disjunctive normal form. Note also that the each-of and some-of operators are associative and commutative and we use them as operators of arbitrary arity, as e.g. in schema $\mathbf{S}_0$ from the introductory example.

3.2 Stratified Negation

Because of the presence of recursion and negation, the notion of correct typing is not sufficient for defining sound semantics of shapes schemas.

Example 4 (Negation and recursion). Let schema $\mathbf{S}_2$ and graph $\mathbf{G}_2$ below:

<L1> → NOT(ex:p <L2>)	<n1> ex:p <n2> .
<L2> → NOT(ex:p <L1>)	<n2> ex:p <n1> .

These two typings of $\mathbf{G}_2$ by $\mathbf{S}_2$ are both correct: $typing_5 = \{($<n1>, <L1>$)\}$ and $typing_6 = \{($<n2>, <L2>$)\}$. □

The two typings in Example 4 strongly contradict each other. In order to prove that node <n1> has shape <L1> (in $typing_5$), we need to prove that <n1> does **not** have shape <L2>. The latter however does hold in $typing_6$. Such strong contradictions are possible only in presence of negation. In comparison, in Example 2 we also have two contradicting typings, but none of them uses in its proof a negative statement that is positive in the other typing.

This problem is well known in logic programming e.g. in Datalog, see Chap. 15 in [1] for an overview. The literature considers several solutions for defining coherent semantics in this case, among which the most popular are negation-as-failure, stratified negation and well-founded semantics. For instance, well-founded semantics would answer *undefined* to the question "does n have shape L" whenever there exist two proofs that contradict each-other on that fact. We exclude this solution for two reasons: it is not helpful for users, and it might require to compute all possible typings which is costly. We opt for stratification semantics instead. It imposes a syntactic restriction on the use of recursion together with negation, so that schemas as the one on Example 4 are not allowed. This is a reasonable restriction because negation in ShEx is expected to be used mainly locally, e.g. to forbid some property in the neighbourhood of a node.

We now define of stratified negation. The *dependency graph* of $\mathbf{S}$ is a graph whose set of nodes is $\mathbf{L}$, and that has two kinds of edges labelled dep^- and dep^+ defined by (recall that shape expressions in disjunctive normal form):

- There is a *negative dependency* edge $dep^-(L_1, L_2)$ from L_1 to L_2 iff the shape label L_2 appears in $\mathbf{def}(L_1)$ under an occurrence of the NOT operator;
- There is a *positive dependency* edge $dep^+(L_1, L_2)$ from L_1 to L_2 iff the shape label L_2 appears in $\mathbf{def}(L_1)$ but never under an occurrence of NOT.

Definition 3 (schema with stratified negation). *A schema* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *is* with stratified negation *if there exists a natural number k and a mapping strat from $\mathbf{L}$ to the interval $[1; k]$ such that for all shape labels L_1, L_2:*

- *if $dep^-(L_1, L_2)$, then $strat(L_1) > strat(L_2)$;*
- *if $dep^+(L_1, L_2)$, then $strat(L_1) \geq strat(L_2)$.*

The mapping *strat* is called a *stratification* of $\mathbf{S}$. A well known property of stratified negation is that the dependency graph does not have a cycle that goes through a negative dependency edge. This intuitively means that if shape L_1 depends negatively on shape L_2, then L_2 does not (transitively) depend on L_1. Positive interdependence is allowed in an unrestricted manner, as in $\mathbf{S}_1$ from Example 2. $\mathbf{S}_2$ from Example 4 is not with stratified negation because $dep^-($<L1>, <L2>$)$ and $dep^-($<L2>, <L1>$)$.

Example 5 (Stratification). Let schema $\mathbf{S}_3$ below.

<L1> → NOT(ex:a @<L2> ; ex:b @<Str>)	<L3> → ex:c @<L2>
<L2> → ex:c @<L3>	<Str> → xsd:string

The dependency graph contains the edges $dep^-(\text{<L1>}, \text{<L2>})$, $dep^-(\text{<L1>}, \text{<Str>})$, $dep^+(\text{<L2>}, \text{<L3>})$, $dep^+(\text{<L3>}, \text{<L2>})$. The unique loop is around <L2> and <L3> and it goes through positive dependencies only, so the schema is stratified. A stratification should be such that <Str> and <L2> are on stratums strictly lower than <L1>, and <L2> and <L3> are on the same stratum. One possible stratification is <L1> on stratum 2 and the other three shape labels on stratum 1. Another one is <L2> and <L3> on stratum 1, <Str> on stratum 2, and <L1> on stratum 3. The latter is called a most refined stratification as none of the stratums can be split.

3.3 Maximal Correct Typing

Recall from Example 3 that both $typing_1$ and $typing_4$ are correct. Note that <i1> has shape <IssueSh> according to $typing_1$ but not according to $typing_4$. Then what is the correct answer of SHAPES_SEM for <i1> and <IssueSh>? Does <i1> have shape <IssueSh> at the end? This section provides an answer to that question. In one sentence: we trust $typing_1$ because it is greater; actually it is the greatest (maximal) typing. The comparison is based on set inclusion.

The following Lemma 1 establishes that a maximal typing always exists in absence of negation. The proof is based on Lemma 2 in [8] that can be easily extended for the richer schemas we have here.

Lemma 1. *Let* $\mathbf{S}$ *be a schema that does not use the negation operator* NOT*. Then for all graphs* $\mathbf{G}$*, there exists a correct typing* $typing_{\mathrm{g}}$ *of* $\mathbf{G}$ *by* $\mathbf{S}$ *such that for every* $typing'$*, if* $typing'$ *is a correct typing of* $\mathbf{G}$ *by* $\mathbf{S}$*, then* $typing' \subseteq typing_{\mathrm{g}}$*.*

The typing $typing_{\mathrm{g}}$ can be computed as the union of all correct typings $typing'$.

Let us now define a maximal typing in presence of negation. Let *strat* be a stratification of $\mathbf{S}$ that has k strata, with $k \geq 1$. For any $1 \leq i \leq k$, the schema $\mathbf{S}_i$ is the restriction of $\mathbf{S}$ that uses only the shape labels whose stratum is less than i. Formally, $\mathbf{S}_i = (\mathbf{L}_i, \mathbf{def}_i)$ with $\mathbf{L}_i = \{L \in \mathbf{L} \mid strat(L) \leq i\}$, and their respective definitions $\mathbf{def}_i(L) = \mathbf{def}(L)$. Remark that if $\mathbf{S}$ is stratified, then $\mathbf{S}_1$ is negation-free.

For a set of labels $\mathbf{L}_i \subseteq \mathbf{L}$, $typing_{|\mathbf{L}_i} = \{(n, L) \in typing \mid L \in \mathbf{L}_i\}$ is the restriction of *typing* on the labels from $\mathbf{L}_i$.

Definition 4 (stratification-maximal correct typing). *Let* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *be a schema,* $\mathbf{G}$ *be a graph, and strat be a stratification of* $\mathbf{S}$ *with* k *stratums (for* $k \geq 1$*). For any* $1 \leq i \leq k$*, let* $typing_i$ *be the typing of* $\mathbf{G}$ *by* $\mathbf{S}_i$*, defined by:*

- *$typing_1$ is the maximal correct typing of $\mathbf{G}$ by $\mathbf{S}_1$, as defined in Lemma 1;*
- *for any $1 \leq i < k$, $typing_{i+1}$ is the union of all correct typings $typing'$ of $\mathbf{G}$ by $\mathbf{S}_{i+1}$ s.t. $typing'_{|\mathbf{L}_i} = typing_i$.*

The stratification-maximal correct typing of $\mathbf{G}$ by $\mathbf{S}$ with stratification *strat* is $Typing(\mathbf{G}, \mathbf{S}, strat) = typing_k$.

$\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat})$ from the above definition is indeed a correct typing for $\mathbf{G}$ by $\mathbf{S}$, as shown in the following proposition that is the core of the proof of soundness for the semantics of shapes schemas.

Proposition 1. *For any schema* $\mathbf{S}$*, any stratification* strat *of* $\mathbf{S}$ *and any graph* $\mathbf{G}$*,* $\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat})$ *is a correct typing of* $\mathbf{G}$ *by* $\mathbf{S}$*.*

Proof. Goes by induction on the number of stratums. The base case (1 stratum) is Lemma 1. For the induction case and stratum $i+1$, by induction hypothesis typing_i is correct for $\mathbf{G}$ and $\mathbf{S}_i$. It is enough to show that if typing' and typing'' are two correct typings for $\mathbf{G}$ by $\mathbf{S}_{i+1}$ and $\mathit{typing}'_{|\mathbf{L}i} = \mathit{typing}''_{|\mathbf{L}i} = \mathit{typing}_i$, then their union $\mathit{typing} = \mathit{typing}' \cup \mathit{typing}''$ is correct for $\mathbf{G}$ by $\mathbf{S}_{i+1}$. Let $(n, L) \in \mathit{typing}$ and suppose that $(n, L) \in \mathit{typing}'$. Because typing' is correct, we have $\mathit{typing}', n \vdash \mathbf{def}(L)$. We will show that (*) the proof for $\mathit{typing}', n \vdash \mathbf{def}(L)$ can be used as a proof for $\mathit{typing}, n \vdash \mathbf{def}(L)$. If $\mathbf{def}(L)$ does not contain a negation of a triple expression, then (*) easily follows from the definition of $\vdash$.

So suppose $\mathbf{def}(L)$ contains a negation operator on top of the triple expressions $E_1, \ldots, E_l$. That is, (recall that shape expressions are in disjunctive normal form), $\mathsf{NOT}\, E_j$ is a sub-expression of $\mathbf{def}(L)$ for every $1 \leq j \leq l$. Then the proof for $\mathit{typing}', n \vdash \mathbf{def}(L)$ contains applications of Rule se-shape-not for $\mathsf{NOT}\, E_j$ that witness that there does not exist a proof for $\mathit{typing}', n \vdash E_j$, for every $1 \leq j \leq l$. We need to show that a proof $\mathit{typing}, n \vdash E_j$ cannot exist. Suppose by contradiction that P is a proof for $\mathit{typing}, n \vdash E_j$, for some $1 \leq j \leq l$. Let $\mathbf{L}'$ be the set of all shape labels that appear in E_j, then P uses only node-type associations with labels from $\mathbf{L}'$. That is, $\mathit{typing}_{|\mathbf{L}'}, n \vdash E_j$ holds. As E_j is negated in $\mathbf{def}(L)$, we have $\mathbf{L}' \subseteq \mathbf{L}_i$, so $\mathit{typing}_{|\mathbf{L}_i}, n \vdash E_j$ also holds. But $\mathit{typing}_{|\mathbf{L}_i} = \mathit{typing}_i \subseteq \mathit{typing}'$. Contradiction. □

Lemma 2 below establishes that $\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat})$ does not depend on the stratification being chosen. This allows to define the maximal correct typing (Definition 5) and to give a precise meaning of SHAPES_SEM (Definition 6) which was the objective of this section.

Lemma 2. *Let* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *be a schema and* $\mathbf{G}$ *be a graph. Let* strat_1 *and* strat_2 *be two stratifications of* $\mathbf{S}$*. Then* $\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat}_1) = \mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat}_2)$*.*

Proof. (Idea) The proof uses a classical technique as e.g. for stratified Datalog. There exists a unique (up to permutation on the numbering of stratums) most refined stratification $\mathit{strat}_{\mathrm{ref}}$ such that for any other stratification strat', each stratum of strat' can be obtained as a union of stratums of $\mathit{strat}_{\mathrm{ref}}$. Then we show that for any stratification strat', $\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat}') = \mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat}_{\mathrm{ref}})$.

Definition 5 (maximal correct typing). *Let* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *be a schema and* $\mathbf{G}$ *be a graph. The* maximal correct typing of $\mathbf{G}$ by $\mathbf{S}$ *is denoted* $\mathit{Typing}(\mathbf{G}, \mathbf{S})$ *and is defined as* $\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat})$ *for some stratification* strat *of* $\mathbf{S}$*.*

Input: **G**: a graph, **S** = (**L**, **def**): a schema, *strat* a stratification for **S** with k strata
Output: *Typing*(**G**, **S**)

```
1  typing ← ∅;
2  for i from 1 to k do
       // Add all node-type associations for stratum i
3      foreach n in Nodes(G) do
4          foreach L in L_i do
5              add (n, L) to typing;
       // Refine w.r.t the types on stratum i
6      changing ← true;
7      while changing do
8          changing ← false;
9          foreach (n, L) in typing s.t. L ∈ L_i do
10             if not typing, n ⊢ def(L) then
11                 remove (n, L) from typing;
12                 changing ← true;
13 return typing
```

Algorithm 1. The algorithm *refine*(**G**, **S**, *strat*).

Definition 6 (shapes_sem). *Let* $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ *be a schema and* **G** *be a graph. We say that node* n *(of* **G***) has shape* L *(from* **S***) if* $(n, L) \in \mathit{Typing}(\mathbf{G}, \mathbf{S})$.

4 Validation

In Sect. 3 we have given a declarative semantics of the shapes language. We now consider the related computational problem. We are again interested by the SHAPES_SEM statement (as defined in Sect. 3), i.e. checking whether a given node has a given shape.

4.1 Refinement Algorithm

Algorithm 1 computes $\mathit{Typing}(\mathbf{G}, \mathbf{S}, \mathit{strat})$. The i-th iteration of the loop on line 2 computes typing_i from Definition 4. The algorithm is correct thanks to Lemma 2 from [8] applied to every stratum i. According to that lemma, the maximal typing defined as the union of all correct typings (i.e. typing_i) can be computed by iteratively removing unsatisfied node-type associations (done on line 11) until a fixed point is reached (detected when *changing* remains *false*). The advantage of the *refine* algorithm is that once $\mathit{Typing}(\mathbf{G}, \mathbf{S})$ is computed, testing whether node n has shape L is done with no additional cost by testing whether (n, L) belongs to $\mathit{Typing}(\mathbf{G}, \mathbf{S})$. The drawback is that it considers all node-type associations which is not always necessary, as shown here after.

Input: n: node in $\mathbf{G}$, L: label in $\mathbf{L}$, Hyp: a stack over $\mathbf{N}\text{odes}(G) \times \mathbf{L}$
Output: *true* if n has label L, *false* otherwise

1 $Hyp = Hyp.\,push((n, L))$;
2 $Dep = \emptyset$;
3 **foreach** (n', L') *in* $dep(n, L) \setminus Hyp$ **do**
4   **if** $prove(n', L', Hyp)$ **then**
5         $Dep = Dep \cup \{(n', L')\}$;
6 result $= Dep \cup Hyp, n \vdash \mathbf{def}(L)$;
7 $Hyp = Hyp.\,pop()$;
8 **return** result ;

Algorithm 2. $prove(n, L, Hyp)$. Graph $\mathbf{G}$ and schema $\mathbf{S}$ are global variables.

4.2 Recursive Algorithm

Algorithm 2 allows to check whether node n has shape L without constructing $Typing(\mathbf{G}, \mathbf{S})$. The idea is to visit only a sufficiently large portion of $Typing(\mathbf{G}, \mathbf{S})$.

Example 6 (Motivation of the prove algorithm). Considering schema $\mathbf{S}_3$ from Example 5 and graph $\mathbf{G}_3$ below:

```
ex:n1 ex:a ex:n2        ex:n2 ex:c ex:n3 .
ex:n1 ex:b 4 .          ex:n3 ex:c ex:n2 .
```

We want to check whether ex:n1 has shape <L1>. Remark that the neighbor nodes of ex:n1 are ex:n2 and 4, whereas the shape labels on which the definition of <L1> depends are <L2> and <Str>. Any correct proof for $typing, \mathsf{ex{:}n1} \vdash <\mathsf{L1}>$ (or for $typing, \mathsf{ex{:}n1} \nvdash <\mathsf{L1}>$) would have as leaves either applications of Rule se-value-descr that do not depend on *typing*, or applications of Rule te-tpattern that uses node-type associations (n, L) where n is a neighbor of ex:n1 and L' is a label such that $dep^+(<\mathsf{L1}>, L')$ or $dep^-(<\mathsf{L1}>, L')$.

Assume schema $\mathbf{S} = (\mathbf{L}, \mathbf{def})$ and graph $\mathbf{G}$. For a shape label L in $\mathbf{S}$ and a node n in $\mathbf{G}$, we denote $dep(n, L)$ the set of node-type associations (n', L') s.t. n' is a neighbor of n (that is, $(n, p, n') \in neigh(n)$ for some IRI p) and L' appears as a shape reference in $\mathbf{def}(L)$. Algorithm 2 uses this easy to show property: $typing, n \models \mathbf{def}(L)$ iff $typing \cap dep(n, L), n \models \mathbf{def}(L)$. In order to check whether n has shape L, Algorithm 2 will (recursively) check whether n' has shape L' for all (n', L') in $dep(n, L)$. The parameter Hyp is a stack of node-type associations that is also seen (on line 3) as the set of node-type associations it contains. Dep is a set of node-type associations.

Example 7 (Execution trace of the prove algorithm). Here is the tree of recursive calls generated during the evaluation of $prove(\mathsf{ex{:}n1}, <\mathsf{L1}>, [])$ for graph $\mathbf{G}_3$ and schema $\mathbf{S}_3$, where [] is the empty stack. The returned value is given on the right. $prove(\mathsf{ex{:}n1}, <\mathsf{L1}>, [])$ generates four recursive calls that correspond to

$dep(\mathsf{ex{:}n1}, <\mathsf{L1}>)$. The call for $\mathsf{ex{:}n3}$ and $<\mathsf{L3}>$ does not generate any recursive call: $dep(\mathsf{ex{:}n3}, <\mathsf{L3}>)$ contains only $(\mathsf{ex{:}n2}, <\mathsf{L2}>)$ which is on the stack.

$prove(\mathsf{ex{:}n1}, <\mathsf{L1}>, [])$	*true*
$\vert{-}prove(\mathsf{ex{:}n2}, <\mathsf{L2}>, [(\mathsf{ex{:}n1}, <\mathsf{L1}>)])$	*true*
$\vert\ \vert{-}prove(\mathsf{ex{:}n3}, <\mathsf{L3}>, [(\mathsf{ex{:}n1}, <\mathsf{L1}>), (\mathsf{ex{:}n2}, <\mathsf{L2}>)])$	*true*
$\vert{-}prove(\mathsf{ex{:}n2}, <\mathsf{Str}>, [(\mathsf{ex{:}n1}, <\mathsf{L1}>)])$	*false*
$\vert{-}prove(4, <\mathsf{L2}>, [(\mathsf{ex{:}n1}, <\mathsf{L1}>)])$	*false*
$\vert{-}prove(4, <\mathsf{Str}>, [(\mathsf{ex{:}n1}, <\mathsf{L1}>)])$	*false*

The correctness of the *prove* algorithm is stated by the following:

Proposition 2 (Correctness of the *prove* algorithm). *For any node n and any shape label L, the evaluation of $prove(n, L, [])$ terminates and returns true if $(n, L) \in Typing(\mathbf{G}, \mathbf{S})$ and false otherwise.*

Proof (Sketch). For termination: the recursion cannot be infinite-breadth as *prove* generates a finite number of recursive calls on line 4. Infinite-depth recursion is also impossible because *Hyp* is a call stack and the condition on line 3 prevents from (recursively) calling *prove* with the same node and label.

The proof of correctness goes by induction on the stratum of L using the most refined stratification *strat*. For every stratum i we show that whenever *Hyp* contains only node-type associations (n', L') with $strat(L') > i$, and for any L s.t. $strat(L) = i$, $Typing, n \vdash \mathbf{def}(L)$ iff $prove(n, L, Hyp)$ returns true. For the $\Rightarrow$ direction, the main argument is that if $Typing, n \vdash \mathbf{def}(L)$ then also $Typing \cup Hyp, n \vdash \mathbf{def}(L)$. This is not true in general because of negation, but is true if *Hyp* is on stratum $\geq strat(L)$ as in this case no type in *Hyp* is negated in $\mathbf{def}(L)$. For the $\Leftarrow$ direction, we need to show that if $prove(n, L, Hyp)$ returns *true* then $(n, L) \in Typing(\mathbf{G}, \mathbf{S})$. The problematic case is when $prove(n, L, Hyp)$ returns *true* whereas n does not have label L. Such error necessarily comes from the fact that on line 6 the algorithm used some $(n', L') \in Hyp \setminus Typing(\mathbf{G}, \mathbf{S})$ in the proof for $Dep \cup Hyp, n \vdash \mathbf{def}(L)$. Consequently, $strat(L) = strat(L')$, and because we consider the most refined stratification, it follows that L and L' mutually depend on each other in the dependency graph of $\mathbf{G}$. Then we need to distinguish two cases. Either all shape labels on stratum i only depend on each others, as for instance $<\mathsf{L2}>$ and $<\mathsf{L3}>$ from Example 5. In that case $prove(n, L, Hyp)$ returns *true* based only on hypotheses in *Hyp*, which is correct w.r.t. the semantics based on maximal solution: if nothing outside stratum i allows to disprove that n has label L, then it is indeed the case. The other possibility is that a shape label L' on stratum i depends also on shapes from lower stratums, as $<\mathsf{IssueSh}>$ from Example 2 that depends on $<\mathsf{Str}>$. Then the test on line 6 of the call of *prove* with L' will take this dependency into account and return *true* only if all conditions, including those that depend on the lower stratums, are satisfied. $\square$

4.3 On Implementation of the Validation Algorithms

Both algorithms use a test for $typing, n \vdash \mathbf{def}(L)$, which non trivial part is the test of the $\models$ relation required in Rule se-neig-descr. The latter is equivalent to checking whether a word (a string) matches a regular expression disregarding the ordering of the letters of the word. Here the word is over the alphabet of triple patterns that occur in the triple expression. In [4] we presented an algorithm for this problem based on regular expression derivatives. In [8] we gave another algorithm for so called deterministic single-occurrence triple expressions. That algorithm can be extended to general expressions, and was used in several of the implementations of ShEx available as open source[8].

The *prove* algorithm was presented in a form that is easier to understand but not optimized. An implementation could reduce considerably the search space of the algorithm by exploring only relevant node-shape associations from $dep(n, L)$ For instance, in Example 7 checking 4 against L2 is useless first because 4 is accessible from ex:n1 by ex:b whereas <L2> in the schema is accessible from <L1> by ex:a.

A more involved version of the *prove* algorithm could memorize portion of $Typing(\mathbf{G}, \mathbf{S})$ to be reused. This however should be done carefully: one should not memorize all node-shape associations (n, L) for which the algorithm returned *true*, as some of these can be false positives as discussed in the proof of Proposition 2.

5 Conclusion

In this paper we introduced shapes schemas that formalize the semantics of ShEx 2.0 and we showed that the semantics of ShEx 2.0 is sound. We also presented two algorithms for validating an RDF graph against a shapes schema.

ShEx and the underlying formalism presented here are still evolving, and there are several promising directions some of which are already being explored: introduce operators for value comparison, use property paths in triple patterns, define an RDF transformation language based on ShEx. We also plan to consider several heuristics and optimizations as the ones discussed in Sect. 4.3 in order to accelerate the validation of shapes schemas. These will be validated on examples. Another open problem is error reporting in ShEx: how to give useful feedback for correcting validation errors. We also plan to explore the exact relationship between shapes schemas and SHACLand establish whether shapes schemas can be encoded in SPARQL extended with recursion as the one defined in [7].

Acknowledgments. This work was partially supported by CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015–2020, ANR project DataCert ANR-15-CE39-0009.

[8] A list of the available ShEx implementations can be found on http://shex.io/.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Fischer, P.M., Lausen, G., Schätzle, A., Schmidt, M.: RDF constraint checking. In: Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference. CEUR-WS.org (2015)
3. Kontokostas, D., Westphal, P., Auer, S., Hellmann, S., Lehmann, J., Cornelissen, R., Zaveri, A.: Test-driven evaluation of linked data quality. In: Proceedings of the 23rd International Conference on World Wide Web (WWW 2014) (2014)
4. Labra Gayo, J.E., Prud'hommeaux, E., Boneva, I., Staworko, S., Solbrig, H.R., Hym, S.: Towards an RDF validation language based on regular expression derivatives. In: Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference. CEUR-WS.org (2015)
5. Motik, B., Horrocks, I., Sattler, U.: Adding integrity constraints to OWL. In: OWL: Experiences and Directions 2007 (OWLED 2007) (2007)
6. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.R.: Shape expressions: an RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems, SEMANTICS 2014. ACM (2014)
7. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 19–35. Springer, Cham (2015). doi:10.1007/978-3-319-25007-6_2
8. Staworko, S., Boneva, I., Labra Gayo, J.E., Hym, S., Prud'hommeaux, E.G., Solbrig, H.R.: Complexity and expressiveness of ShEx for RDF. In: 18th International Conference on Database Theory (ICDT). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
9. Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity constraints in OWL. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence. AAAI (2010)