Validating RDF data
ShEx & SHACL compared

Jose Emilio Labra Gayo
WESO Research group
University of Oviedo, Spain

Eric Prud'hommeaux
World Wide Web Consortium
MIT, Cambridge, MA, USA

Iovka Boneva
LINKS, INRIA & CNRS
University of Lille, France
About me…

WESO (Web Semantics Oviedo) research group
  Practical applications of semantic technologies
  Several domains: e-Government, e-Health

Some books:
  "Validating RDF data"
  "Semantic web" (in Spanish)

...and software:
  SHacLEX (Scala library, implements ShEx & SHACL)
  RDFShape (RDF playground)

HTML version: http://book.validatingrdf.com
Examples: https://github.com/labra/validatingRDFBookExamples
Contents

Short overview of RDF data model
Understanding the RDF validation problem
Short intro to ShEx
Short intro to SHACL
ShEx and SHACL compared
Conclusions & future work

For longer presentations about ShEx/SHACL, see the ISWC'18 tutorial slides:
RDF Data Model

RDF Graph = set of triples

Triple = (subject, predicate, object)

Example:

\[
\]
RDF Graph

Basic statement = a simple triple
RDF Graph

...we can add more statements

Circled nodes are IRIs
Yellow boxes are literals
RDF Graph

...more statements can be added...
RDF Graph

...and more...

Forming a graph

...which can contain cycles
RDF Graph

Graphs can be created independently
...and automatically merged
RDF helps information integration
RDF syntaxes

Several syntaxes:
N-triples, Turtle, ...

```
<http://example.org/alice> <http://example.org/name> "Alice" .
<http://example.org/bob> <http://example.org/name> "Robert" .
<http://example.org/bob> <http://example.org/age> "None" .
<http://example.org/carol> <http://example.org/name> "Carol" .
```
Turtle Syntax

prefix : <http://example.org/>

:alice :name "Alice" ;
:age 23 ;
:enrolledIn :cs101 ;
:knows :carol ;
:birthPlace :Oviedo .
:carol :name "Carol" ;
:enrolledIn :cs101 .
:cs101 :students :alice , :carol ;
:subject "Programming" .
:bob :name "Robert" ;
:age "None" ;
:enrolledIn :cs102 ;
:birthPlace :Oviedo .
:cs102 :students :bob ;
:subject "Algebra" .

Some simplifications

prefix declarations
;
when triples share the subject
,
when triples share subject and object

Try it: https://goo.gl/pK3Csh
Objects can also be literals

- Literals contain a lexical form and a datatype
- Common datatypes: XML Schema primitive datatypes
- If not specified, a literal has type xsd:string

Examples:

```xml
:bob :name "Robert" ;
:age 18 ;
:birthDate "2010-04-12"^^xsd:date .
```

```xml
:bob :name "Robert"^^xsd:string ;
:age "18"^^xsd:integer ;
:birthDate "2010-04-12"^^xsd:date .
```
Blank nodes

Subjects and objects can also be Blank nodes

Blank nodes can have local identifiers

\[
\exists x (\text{\text{:bob :knows} } x \land x : \text{age 23})
\]

or

\[
\text{:bob :knows} \ _:1 \ . \\
_:1 : \text{age 23} .
\]
Language tagged strings

String literals can be qualified by a language tag.
They have datatype `rdfs:langString`.

```
:spain rdfs:label "Spain"@en ; rdfs:label "España"@es .
```
...and that's all?

Yes, the RDF Data model is simple

Simple is better
RDF, the good parts...

RDF as an integration language
RDF as a *lingua franca* for semantic web and linked data
RDF data stores & SPARQL
RDF flexibility & integration
  Data can be adapted to multiple environments
  Open and reusable data by default
RDF for knowledge representation
RDF, the other parts

Consuming & producing RDF

- Multiple serializations: Turtle, RDF/XML, JSON-LD, ...
- Embedding RDF in HTML
- Describing and validating RDF content
Why describe & validate RDF?

For producers

- Developers can understand the contents they are going to produce
- Ensure they produce the expected structure
- Advertise and document the structure
- Generate interfaces

For consumers

- Understand the contents
- Verify the structure before processing it
- Query generation & optimization
## Similar technologies

<table>
<thead>
<tr>
<th>Technology</th>
<th>Schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational Databases</td>
<td>DDL</td>
</tr>
<tr>
<td>XML</td>
<td>DTD, XML Schema, RelaxNG, Schematron</td>
</tr>
<tr>
<td>Json</td>
<td>Json Schema</td>
</tr>
<tr>
<td>RDF</td>
<td>?</td>
</tr>
</tbody>
</table>

Fill that gap
Understanding the problem

RDF is composed by nodes and arcs between nodes
We can describe/check
  The form of the node itself (node constraint)
  The number of possible arcs incoming/outgoing from a node
  The possible values associated with those arcs

RDF Node:
:alice schema:name "Alice";
  schema:knows :bob .

ShEx:
<UserShape> IRI {
  schema:name xsd:string ;
  schema:knows IRI *
}

Shape:
IRI schema:name string 1
  schema:knows IRI 0, 1,...
RDF is a flexible language

Complex objects & literals can be mixed

Example:

Values of `schema:creator` can be:

- `string`
- `schema:Person`

```
:angie schema:creator "Keith Richards" ;
  schema:creator [  
    schema:firstName "Mick" ;  
    schema:lastName "Jagger"  
  ] .
```

Lots of examples at [http://schema.org](http://schema.org)
Understanding the problem

Repeated properties
The same property can be used for different purposes in the same data
Example: A product must have 2 codes with different structure

```
schema:productID "code456" .
```

A practical example from FHIR
See: http://hl7-fhir.github.io/observation-example-bloodpressure.ttl.html
Understanding the problem

Shapes ≠ types

Nodes in RDF graphs can have 0, 1 or many \texttt{rdf\:type} declarations

A type can be used in multiple contexts, e.g. \texttt{foaf\:Person}

Nodes are not necessarily annotated with discriminating types

Nodes with type \texttt{foaf\:Person} can represent friends, students, patients,...

Different meanings and different structure depending on context

Specific validation constraints for different contexts
Understanding the problem

RDF validation ≠ ontology definition ≠ instance data

Ontologies are usually focused on domain entities
RDF validation is focused on RDF graph features (lower level)

Different levels

Ontology

```
schema:knows a owl:ObjectProperty ;
rdfs:domain schema:Person ;
rdfs:range   schema:Person .
```

Constraints

RDF Validation

A user must have only two properties:
```
schema:name of value xsd:string
schema:knows with an IRI value
```

```
<User> IRI {
  schema:name   xsd:string ;
  schema:knows  IRI
}
```

Instance data

```
:alice schema:name "Alice";
  schema:knows :bob .
```
Why not using SPARQL to validate?

Pros:
- Expressive
- Ubiquitous

Cons
- Expressive
- Idiomatic
  - many ways to encode the same constraint

Example: Define SPARQL query to check:
There must be one `schema:name` which must be a `xsd:string`, and one `schema:gender` which must be `schema:Male` or `schema:Female`
Previous/other approaches

SPIN, by TopQuadrant [http://spinrdf.org/]
SPARQL templates, Influenced SHACL

Stardog ICV: [http://docs.stardog.com/icv/icv-specification.html]
OWL with UNA and CWA

OSLC resource shapes: [https://www.w3.org/Submission/shapes/]
Vocabulary for RDF validation

Dublin Core Application profiles (K. Coyle, T. Baker)
[http://dublincore.org/documents/dc-dsp/]

RDF Data Descriptions (Fischer et al)

RDFUnit (D. Kontokostas)
[http://aksw.org/Projects/RDFUnit.html]

...
ShEx and SHACL

2013 RDF Validation Workshop

Conclusions of the workshop:

*There is a need of a higher level, concise language for RDF Validation*

ShEx initially proposed (v 1.0)

2014 W3c Data Shapes WG chartered

2017 SHACL accepted as W3C recommendation

2017 ShEx 2.0 released as Community group draft

2018 ShEx 2.1 released
Short intro to ShEx

ShEx (Shape Expressions Language)
Concise and human-readable language for RDF validation & description
Syntax similar to SPARQL, Turtle
Semantics inspired by regular expressions & RelaxNG
2 syntaxes: Compact and RDF/JSON-LD
Official info: http://shex.io

ShEx implementations and playgrounds

<table>
<thead>
<tr>
<th>Libraries:</th>
<th>Online demos &amp; playgrounds</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Image" /></td>
<td><img src="image.png" alt="Image" /></td>
</tr>
<tr>
<td><strong>shex.js</strong>: Javascript</td>
<td><strong>ShEx-simple</strong></td>
</tr>
<tr>
<td><strong>SHaclEX</strong>: Scala (Jena/RDF4j)</td>
<td><strong>RDFShape</strong></td>
</tr>
<tr>
<td><strong>PyShEx</strong>: Python</td>
<td><strong>ShEx-Java</strong></td>
</tr>
<tr>
<td><strong>shex-java</strong>: Java</td>
<td><strong>ShExValidata</strong></td>
</tr>
<tr>
<td><strong>Ruby-ShEx</strong>: Ruby</td>
<td></td>
</tr>
</tbody>
</table>
Nodes conforming to `<User>` shape must:

- Be IRIs
- Have exactly one `schema:name` with a value of type `xsd:string`
- Have zero or more `schema:knows` whose values conform to `<User>`

```
prefix schema: <http://schema.org/>
prefix xsd:  <http://www.w3.org/2001/XMLSchema#>

<User> IRI {
    schema:name  xsd:string  ;
    schema:knows @<User>     *
}
```
RDF Validation using ShEx

Schema

```
<User> IRI {
    schema:name xsd:string ;
    schema:knows @<User> *
}
```

Shape map

```
:alice @<User> ✓
:bob @<User> ✓
:carol @<User> ✗
:dave @<User> ✗
:emily @<User> ✗
:frank @<User> ✗
:grace @<User> ✓
```

Try it (RDFShape): https://goo.gl/97bYdv
Try it (ShExDemo): https://goo.gl/Y8hBSw
Validation process

**Input:** RDF data, ShEx schema, Shape map

**Output:** Result shape map

**ShEx Schema**

```shex
:User {
  schema:name xsd:string ;
  schema:knows @:User *
}
```

**Shape map**

`:alice@:User, :bob@:User, :carol@:User`

**RDF data**

```turtle
:alice schema:name "Alice" ;
   schema:knows :alice .

:bob schema:knows :alice ;
   schema:name "Robert".

:carol schema:name "Carol", "Carole" .
```

**Result shape map**

`:alice@:User, :bob@:User, :carol@!:User`
Example with more ShEx features:

```
:AdultPerson EXTRA a {
  a [ schema:Person ] ;
  :name xsd:string ;
  :age MinInclusive 18 ;
  :gender [:Male :Female] OR xsd:string ;
  :address @:Address ? ;
  :worksFor @:Company +
}

:Address CLOSED {
  :addressLine xsd:string {1,3}
  :postalCode /[0-9]{5}/
  :state @:State
  :city xsd:string
}

:Company {
  :name xsd:string
  :state @:State
  :employee @:AdultPerson *
}

:State /[A-Z]{2}/

:alice a :Student, schema:Person ;
  :name "Alice" ;
  :age 20 ;
  :gender :Male ;
  :address [
    :addressLine "Bancroft Way" ;
    :city "Berkeley" ;
    :postalCode "55123" ;
    :state "CA" ] ;
  :worksFor [
    :name "Company" ;
    :state "CA" ;
    :employee alice ] .
```

Try it: https://tinyurl.com/yd5hp9z4
More info about ShEx

See:

ShEx by Example (slides):
https://figshare.com/articles/ShExByExample_pptx/6291464

ShEx chapter from Validating RDF data book:
Short intro to SHACL

W3C recommendation:
https://www.w3.org/TR/shacl/ (July 2017)

RDF vocabulary

2 parts: SHACL-Core, SHACL-SPARQL
<table>
<thead>
<tr>
<th>Name</th>
<th>Parts</th>
<th>Language - Library</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topbraid SHACL API</td>
<td>SHACL Core, SPARQL</td>
<td>Java (Jena)</td>
<td>Used by TopBraid composer</td>
</tr>
<tr>
<td>SHACL playground</td>
<td>SHACL Core</td>
<td>Javascript (rdflib.js)</td>
<td><a href="http://shacl.org/playground/">http://shacl.org/playground/</a></td>
</tr>
<tr>
<td>SHACLEX</td>
<td>SHACL Core</td>
<td>Scala (Jena, RDF4j)</td>
<td><a href="http://rdfshape.weso.es">http://rdfshape.weso.es</a></td>
</tr>
<tr>
<td>pySHACL</td>
<td>SHACL Core, SPARQL</td>
<td>Python (rdflib)</td>
<td><a href="https://github.com/RDFLib/pySHACL">https://github.com/RDFLib/pySHACL</a></td>
</tr>
<tr>
<td>Corese SHACL</td>
<td>SHACL Core, SPARQL</td>
<td>Java (STTL)</td>
<td><a href="http://wimmics.inria.fr/corese">http://wimmics.inria.fr/corese</a></td>
</tr>
<tr>
<td>RDFUnit</td>
<td>SHACL Core, SPARQL</td>
<td>Java (Jena)</td>
<td><a href="https://github.com/AKSW/RDFUnit">https://github.com/AKSW/RDFUnit</a></td>
</tr>
</tbody>
</table>
Basic example

Try it. RDFShape https://goo.gl/T1uuzy
Same example with blank nodes

prefix: <http://example.org/>
prefix sh: <http://www.w3.org/ns/shacl#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
prefix schema: <http://schema.org/>

:UserShape a sh:NodeShape ;
  sh:targetNode :alice, :bob, :carol ;
  sh:nodeKind sh:IRI ;
  sh:property [
    sh:path schema:name ;
    sh:minCount 1; sh:maxCount 1;
    sh:datatype xsd:string ;
  ] ;
  sh:property [
    sh:path schema:email ;
    sh:minCount 1; sh:maxCount 1;
    sh:nodeKind sh:IRI ;
  ].

:alice schema:name "Alice Cooper" ;
  schema:email <mailto:alice@mail.org> .

:bob schema:firstName "Bob" ;
  schema:email <mailto:bob@mail.org> . 😞

:carol schema:name "Carol" ;
  schema:email "carol@mail.org" . 😞

Try it. RDFShape https://goo.gl/ukY5vq
Some definitions about SHACL

Shape: collection of targets and constraints components

Targets: specify which nodes in the data graph must conform to a shape

Constraint components: Determine how to validate a node

```
:UserShape a sh:NodeShape ;
  sh:targetNode :alice, :bob, :carol ;
  sh:nodeKind sh:IRI ;
  sh:property :hasName,
    :hasEmail .
  :hasName sh:path schema:name ;
    sh:minCount 1;
    sh:maxCount 1;
    sh:datatype xsd:string .
  .
```
Validation Report

The output of the validation process is a list of violation errors

No errors $\Rightarrow$ RDF conforms to shapes graph

```
[ a sh:ValidationReport ;
  sh:conforms true ].
```

```
[ a sh:ValidationReport ;
  sh:conforms false ;
  sh:result [ a sh:ValidationResult ;
    sh:focusNode :bob ;
    sh:message
      "MinCount violation. Expected 1, obtained: 0" ;
    sh:resultPath schema:name ;
    sh:resultSeverity sh:Violation ;
    sh:sourceConstraintComponent sh:MinCountConstraintComponent ;
    sh:sourceShape :hasName
  ] ;
  ...
] .
```
SHACL processor

Shapes graph

```
:UserShape a sh:NodeShape ;
  sh:targetNode :alice, :bob, :carol ;
  sh:nodeKind sh:IRI ;
  sh:property :hasName,
    :hasEmail .
  :hasName sh:path schema:name ;
    sh:minCount 1;
    sh:maxCount 1;
    sh:datatype xsd:string .
...
```

Data Graph

```
:alice schema:name "Alice Cooper" ;
  schema:email <mailto:alice@mail.org>.

:bob schema:name "Bob" ;
  schema:email <mailto:bob@mail.org> .

:carol schema:name "Carol" ;
  schema:email <mailto:carol@mail.org> .
```

Validation report

```
[ a sh:ValidationReport ;
  sh:conforms true ].
```
Target declarations

Targets specify nodes that must be validated against the shape

Several types

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>targetNode</td>
<td>Directly point to a node</td>
</tr>
<tr>
<td>targetClass</td>
<td>All nodes that have a given type</td>
</tr>
<tr>
<td>targetSubjectsOf</td>
<td>All nodes that are subjects of some predicate</td>
</tr>
<tr>
<td>targetObjectsOf</td>
<td>All nodes that are objects of some predicate</td>
</tr>
</tbody>
</table>
Target node

Directly declare which nodes must validate the against the shape

:UserShape a sh:NodeShape ;
  sh:targetNode :alice, :bob, :carol ;
  sh:property [ 
    sh:path schema:name ;
    sh:minCount 1;
    sh:maxCount 1;
    sh:datatype xsd:string ;
  ] ;
  sh:property [ 
    sh:path schema:email ;
    sh:minCount 1;
    sh:maxCount 1;
    sh:nodeKind sh:IRI ;
  ].

:alice schema:name "Alice Cooper" ;
  schema:email <mailto:alice@mail.org> .

:bob schema:givenName "Bob" ;
  schema:email <mailto:bob@mail.org> .

:carol schema:name "Carol" ;
  schema:email "carol@mail.org" .
**Target class**

Selects all nodes that have a given class

Looks for `rdfs:subClassOf`/*`/`rdf:type` declarations*

:UserShape a sh:NodeShape ;
  sh:targetClass :User ;
  sh:property [ 
    sh:path schema:name ;
    sh:minCount 1;
    sh:maxCount 1;
    sh:datatype xsd:string ;
  ];
  sh:property [ 
    sh:path schema:email ;
    sh:minCount 1;
    sh:maxCount 1;
    sh:nodeKind sh:IRI ;
  ] .

:alice a :User;
  schema:name "Alice Cooper" ;
  schema:email <mailto:alice@mail.org> .

:bob a :User;
  schema:givenName "Bob" ;
  schema:email <mailto:bob@mail.org> .

:carol a :User;
  schema:name "Carol" ;
  schema:email "carol@mail.org" .
Implicit class target

A shape with type sh:Shape and rdfs:Class is a target class of itself

The targetClass declaration is implicit

```sparql
:alice a :User;
   schema:name "Alice Cooper" ;
   schema:email <mailto:alice@mail.org> .

:bob a :User;
   schema:givenName "Bob" ;
   schema:email <mailto:bob@mail.org> .

:carol a :User;
   schema:name "Carol" ;
   schema:email "carol@mail.org" .
```
Node and property shapes

2 types of shapes:

NodeShape: constraints about shapes of nodes
PropertyShapes: constraints on property path values of a node
Paths in property shapes

Subset of SPARQL property paths using the following predicates:

inversePath
alternativePath
zeroOrMorePath
oneOrMorePath
zeroOrOnePath

:User a sh:NodeShape, rdfs:Class ;
    sh:property [ sh:path [sh:inversePath schema:follows ];
                sh:nodeKind sh:IRI ;
] .
SHACL Core built-in constraint components

<table>
<thead>
<tr>
<th>Type</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td>minCount, maxCount</td>
</tr>
<tr>
<td>Types of values</td>
<td>class, datatype, nodeKind</td>
</tr>
<tr>
<td>Values</td>
<td>node, in, hasValue, property</td>
</tr>
<tr>
<td>Range of values</td>
<td>minInclusive, maxInclusive</td>
</tr>
<tr>
<td></td>
<td>minExclusive, maxExclusive</td>
</tr>
<tr>
<td>String based</td>
<td>minLength, maxLength, pattern</td>
</tr>
<tr>
<td>Language based</td>
<td>languageIn, uniqueLang</td>
</tr>
<tr>
<td>Logical constraints</td>
<td>not, and, or, xone</td>
</tr>
<tr>
<td>Closed shapes</td>
<td>closed, ignoredProperties</td>
</tr>
<tr>
<td>Property pair constraints</td>
<td>equals, disjoint, lessThan, lessThanOrEquals</td>
</tr>
<tr>
<td>Non-validating constraints</td>
<td>name, description, order, group</td>
</tr>
<tr>
<td>Qualified shapes</td>
<td>qualifiedValueShape, qualifiedValueShapesDisjoint</td>
</tr>
<tr>
<td></td>
<td>qualifiedMinCount, qualifiedMaxCount</td>
</tr>
</tbody>
</table>
Example with more features

In ShEx

:AdultPerson EXTRA a {
  a [ schema:Person ] ;
  :name xsd:string ;
  :age MinInclusive 18 ;
  :gender [:Male :Female] OR xsd:string ;
  :address @:Address ? ;
  :worksFor @:Company + ;
}

:Address CLOSED {
  :addressLine xsd:string {1,3} ;
  :postalCode /[0-9]{5}/ ;
  :state @:State ;
  :city xsd:string
}

:Company {
  :name xsd:string ;
  :state @:State ;
  :employee @:AdultPerson * ;
}

:State /[A-Z]{2}/

Try it: https://tinyurl.com/ycl3mkzr

In SHACL

:AdultPerson a sh:NodeShape ;
  sh:property [
    sh:path rdf:type ;
    sh:qualifiedValueShape [
      sh:hasValue schema:Person
    ];
    sh:qualifiedMinCount 1 ;
  ];

:Address a sh:NodeShape ;
  sh:qualifiedMinCount 1 ;
  sh:qualifiedMinCount 1 ;

:Company a sh:NodeShape ;
  sh:property [ sh:path :name ;
    sh:property [ sh:path :state ;
      sh:node :State
    ];
  ];

:State a sh:NodeShape ;
  sh:pattern "[A-Z]{2}" .

Its recursive!!! (not well defined SHACL)
Implementation dependent feature
More info about SHACL

SHACL by example (slides):
https://figshare.com/articles/SHACL_by_example/6449645

SHACL chapter at Validating RDF data book
ShEx and SHACL compared
Several common features...

Similar goal: describe and validate RDF graphs
Both employ the word "shape"
Node constraints similar in both languages
Constraints on incoming/outgoing arcs
Both allow to define cardinalities
Both have RDF syntax
Both have an extension mechanism
But some differences...

Underlying philosophy
Syntactic differences
Notion of a shape
Syntactic differences
Default cardinalities
Shapes and Classes
Recursion
Repeated properties
Property pair constraints
Uniqueness
Extension mechanism
Underlying philosophy

**ShEx is more schema based**
Shapes schemas look like grammars
Focus on description & validation results
  - Result shape maps
  - Info about conforming and non-conforming nodes

**SHACL is more constraint based**
Shapes ≈ collections of constraints
Main focus: validation errors
No info about conforming nodes
  - How to difficult to distinguish between conforming nodes and nodes that have been ignored?
  - RDFShape offers info about conforming node also
Semantic specification

ShEx semantics: mathematical concepts

Well-founded semantics*
Recursion and negation using stratification
Inspired by type systems and RelaxNG

SHACL semantics = textual description + SPARQL

SHACL terms described in natural language
SPARQL fragments used as helpers
Recursion is implementation dependent
SHACL-SPARQL based on pre-binding

*Semantics and Validation of Shapes Schemas for RDF
Iovka Boneva  Jose Emilio Labra Gayo  Eric Prud'hommeaux
ISWC'17
Syntactic differences

ShEx design focused on human-readability
Programming language design methodology
  1. Abstract syntax
  2. Different concrete syntaxes
     Compact
     JSON-LD
     RDF
     ...

SHACL design focused on RDF vocabulary
Design centered on RDF terms
Lots of rules to define valid shapes graphs
https://w3c.github.io/data-shapes/shacl/#syntax-rules
No compact syntax
Compact Syntax

ShEx compact syntax designed along the language
Test-suite with long list of tests
Round-trippable with JSON-LD syntax

SHACL has no compact syntax
A WG Note proposed a compact syntax
It covered a subset of SHACL core
No longer supported and no implementations
RDF vocabulary

ShEx vocabulary ≈ abstract syntax

ShEx RDF vocabulary obtained from the abstract syntax

ShEx RDF serializations typically more verbose

Can be round-tripped to Compact syntax

SHACL is designed as an RDF vocabulary

Some rdf:type declarations can be omitted

SHACL RDF serialization typically more readable
Notion of Shape

In ShEx, shapes only define structure of nodes

Validation triggering by shape maps
Select nodes/shapes to validate
Goal: flexibility and reusability

Shape

```json
:User IRI {
   schema:name xsd:string
}
```

Shape map

```json
:alice@:User,
{FOCUS rdf:type :Person}@:User
```

In SHACL, shapes define structure and can have target declarations

Shapes can be associated with nodes or sets of nodes through target declarations

Shapes may be less reusable in other contexts

Shape

```json
:User a sh:NodeShape, rdfs:Class ;
   sh:targetClass :Person ;
   sh:targetNode :alice ;
   sh:nodeKind sh:IRI ;
   sh:property [ sh:path schema:name ;
```
Default cardinalities

**ShEx:** default = (1,1)

```
:User {
  schema:givenName xsd:string
  schema:lastName xsd:string
}
```

**SHACL:** default = (0, unbounded)

```
:User a sh:NodeShape ;
  sh:property [ sh:path schema:givenName ;
    sh:datatype xsd:string ;
  ];
  sh:property [ sh:path schema:lastName ;
    sh:datatype xsd:string ;
  ] .
```

**Examples:**

- Alice: schema:givenName "Alice";
  schema:lastName "Cooper".

- Bob: schema:givenName "Bob", "Robert";
  schema:lastName "Smith", "Dylan".

- Carol: schema:lastName "King".

- Dave: schema:givenName 23;
  schema:lastName :Unknown.
Property paths

ShEx shapes describe neighborhood of focus nodes: direct/inverse properties

Examples with paths can be emulated using nested shapes

```json
:GrandSon {  
  :parent { :parent . + } + ;  
  (:father . | :mother .) + ;  
  ^:knows :Person  
}
```

SHACL shapes can also describe whole property paths following property paths

```json
:GrandSon a sh:NodeShape ;  
  sh:property [  
    sh:path (schema:parent schema:parent) ;  
    sh:minCount 1  
  ] ;  
  sh:property [  
    sh:path [  
      sh:alternativePath (:father :mother) ]  
  ] ;  
  sh:property [  
    sh:path [sh:inversePath :knows ] ]  
  sh:node :Person ;  
  sh:minCount 1  
} .
```
Property paths

ShEx shapes describe neighborhood of focus nodes: direct/inverse properties

Recursion paths can be emulated with auxiliary shapes

```shex
:GrandParent {
  schema:knows @:PersonKnown*;
}
:PersonKnown @:Person {
  schema:knows @:PersonKnown*
}
:Person {
  schema:name xsd:string
}
```

SHACL shapes can use property paths

```shex
:GrandParent a sh:NodeShape ;
sh:property [
  sh:path [ sh:zeroOrMorePath schema:knows] ;
  sh:node :Person ;
] .

:Person a sh:NodeShape ;
sh:property [
  sh:path schema:name ;
  sh:datatype xsd:string ;
  sh:minCount 1; sh:maxCount 1
] .
```
Property paths

ShEx shapes describe neighborhood of focus nodes: direct/inverse properties

Control about cardinalities

```
:Invoice {
  :payment {
    :amount xsd:decimal
  }
}
```

SHACL shapes can use property paths

Some pathological cases

```
:Invoice a sh:NodeShape ;
  sh:property [  
    sh:path (:payment :amount ) ;
    sh:datatype xsd:decimal ;
    sh:minCount 1; sh:maxCount 1
  ] .
```

Try it: https://tinyurl.com/y97npq5s

Try it: https://tinyurl.com/yabl4v95
**Inference**

**ShEx doesn't mess with inference**
Validation can be invoked before or after inference.

*rdf*:type is considered an arc as any other
No special meaning
The same for *rdfs*:Class, *rdfs*:subClassOf,
    *rdfs*:domain, *rdfs*:range, ...

**SHACL: Some constructs have special meaning**

The following constructs have special meaning in SHACL

- *rdf*:type
- *rdfs*:Class
- *rdfs*:subClassOf
- *owl*:imports

Other constructs like *rdfs*:domain,
    *rdfs*:range,... have no special meaning

*sh*:entailment can be used to indicate that some inference is required
Inference and triggering mechanism

**ShEx has no interaction with inference**

It can be used to validate a reasoner

```
:User {
    schema:name xsd:string
}
```

**In SHACL, RDF Schema inference can affect which nodes are validated**

Partial implicit RDFS inference but not all

```
:User a sh:NodeShape, rdfs:Class ;
    sh:property [ sh:path schema:name ;
        sh:datatype xsd:string; ].
```

With or without RDFS inference

No RDFS inference

RDFS inference

Ignored

Ignored

= conforms to Shape

= doesn't conform
Repeated properties

**ShEx (;) operator handles repeated properties**

**SHACL needs qualifiedValueShapes for repeated properties**

Example. A person must have 2 parents, one male and another female

```json
:Person {
  :parent { :gender [ :Male ] } ;
  :parent { :gender [ :Female ] }
}
```

Direct approximation (wrong):

```json
:Person a sh:NodeShape;
  sh:property [ sh:path :parent;
    sh:node [ sh:property [ sh:path :gender ;
      sh:hasValue :Male ; ] ] ] ;
  sh:property [ sh:path :parent;
    sh:node [ sh:property [ sh:path :gender ;
      sh:hasValue :Female ] ];
} .
```

This says that a person must have a parent which is at the same time male and female
Repeated properties

ShEx (;) operator handles repeated properties

SHACL needs qualifiedValueShapes for repeated properties

Example. A person must have 2 parents, one male and another female

Solution with qualifiedValueShapes:

```datavocabulary
:Person { 
  :parent { :gender [:Male ] } ;
  :parent { :gender [:Female ] } 
}
```

```datavocabulary
:Person a sh:NodeShape, rdfs:Class ;
  sh:property [ sh:path :parent ;
    sh:qualifiedValueShape [ sh:property [ sh:path :gender ;
      sh:hasValue :Male ] ] ;
    sh:qualifiedMinCount 1; sh:qualifiedMaxCount 1 ] ;
  sh:property [ sh:path :parent ;
    sh:qualifiedValueShape [ sh:property [ sh:path :gender ;
      sh:hasValue :Female ] ] ;
    sh:qualifiedMinCount 1; sh:qualifiedMaxCount 1 ] ;
  sh:property [ sh:path :parent ;
    sh:minCount 2; sh:maxCount 2 ] . 
```

It needs to count all possibilities
Recursion

ShEx handles recursion
Well founded semantics

```
:Person {
  schema:name xsd:string ;
  schema:knows @Person*
}
```

Recursive shapes are undefined in SHACL*
Implementation dependent
Direct translation generates recursive shapes

```
:Person a sh:NodeShape ;
sh:property [ sh:path schema:name ;
  sh:datatype xsd:string ];
sh:property [ sh:path schema:knows ;
  sh:node :Person ] .
```

Undefined because it is recursive

*Semantics and Validation of Recursive SHACL
Julien Corman, Juan L. Reutter and Ognjen Savkovic, ISWC’18
Recursion (with target declarations)

ShEx handles recursion
Well founded semantics

Recursive shapes are undefined in SHACL
Implementation dependent
Can be simulated with target declarations
  Example with target declarations
  It needs discriminating arcs

```rml
:Person a sh:NodeShape, rdfs:Class ;
  sh:property [ sh:path schema:name ;
    sh:datatype xsd:string ];
  sh:property [ sh:path schema:knows ;
    sh:class :Person ].
```
It requires all nodes to have `rdf:type Person`
Recursion (with property paths)

ShEx handles recursion
Well founded semantics

Recursive shapes are undefined in SHACL
Implementation dependent
Can be simulated property paths

:Person a sh:NodeShape ;
  sh:property [ 
    sh:path schema:name ; sh:datatype xsd:string ];
  sh:property [ 
    sh:path [sh:zeroOrMorePath schema:knows];
    sh:node :PersonAux ].

:PersonAux a sh:NodeShape ;
  sh:property [ 
    sh:path schema:name ; sh:datatype xsd:string ].
Closed shapes

In ShEx, **CLOSED** affects all properties

```
:Person CLOSED {
  schema:name xsd:string 
  | foaf:name xsd:string
}
```

In SHACL, **sh:closed** only affects properties declared at top-level

Properties declared inside other shapes are ignored

```
:Person a sh:NodeShape ;
  sh:targetNode :alice ;
  sh:closed true ;
  sh:or ( 
    [ sh:path schema:name ; sh:datatype xsd:string ]
    [ sh:path foaf:name ; sh:datatype xsd:string ]
  ) .
```

**:alice schema:name "Alice" .**

= conforms to Shape

= doesn't conform
Closed in ShEx acts on all properties

In SHACL, closed ignores properties mentioned inside paths

Person a sh:NodeShape;
sh:closed true;
sh:property [
  sh:path [
    sh:alternativePath
      ( schema:name foaf:name )
  ];
  sh:minCount 1; sh:maxCount 1;
  sh:datatype xsd:string ];
.

:Person CLOSED {
  schema:name xsd:string |
  foaf:name xsd:string
}

:alice schema:name "Alice".

= conforms to Shape
= doesn't conform
Property pair constraints

This feature was postponed in ShEx 2.1

SHACL supports equals, disjoint, lessThan, ...

```javascript
:UserShape {
  <$givenName> schema:givenName xsd:string ;
  <$firstName> schema:firstName xsd:string ;
  <$birthDate> schema:birthDate xsd:date ;
  <$loginDate> :loginDate xsd:date ;
  <$givenName> = <$firstName> ;
  <$givenName> != <$lastName> ;
  <$birthDate> < <$loginDate>
}
```
This feature was postponed in ShEx 2.1

No support for generic unique keys

sh:uniqueLang offers partial support for a very common use case

Uniqueness can be done with SHACL-SPARQL

```
:UserShape {
  schema:givenName xsd:string ;
  schema:lastName xsd:string ;
  UNIQUE(schema:givenName, schema:lastName)
}
```

```
:Country a sh:NodeShape ;
  sh:property [ 
    sh:path schema:name ;
    sh:uniqueLang true
  ] .
```
Modularity

ShEx has EXTERNAL and import keywords

- **import** imports shapes from URI
- **external** declares that a shape definition can be retrieved elsewhere

---

SHACL supports **owl:imports**

SHACL processors follow **owl:imports**

```shex
import <http://example.org/UserShapes>

:UserShape { 
schema:name xsd:string
} http://example.org/UserShapes

:TeacherShape :UserShape AND 
  :teaches . ;
  :teacherCode external

<owl:imports <http://example.org/UserShapes>>

:TeacherShape a sh:NodeShape;
  sh:node :UserShape ;
  sh:property [ sh:path :teaches ;
    sh:minCount 1; ] ;

http://example.org/UserShapes
```

```owl
<UserShape a sh:NodeShape ;
  sh:property [ sh:path schema:name ;
    sh:datatype xsd:string
  ] .

http://example.org/UserShapes>
```
Reusability - Extending shapes (1)

ShEx shapes can be extended by conjunction

```
:Product { 
  schema:productId xsd:string 
  schema:price xsd:decimal 
}

:SoldProduct @:Product AND { 
  schema:purchaseDate xsd:date ; 
  schema:productId /^[A-Z]/
}
```

SHACL shapes can also be extended by conjunction

```
:Product a sh:NodeShape, rdfs:Class ;
  sh:property [ sh:path schema:productId ;
    sh:datatype xsd:string ];
  sh:property [ sh:path schema:price ;
    sh:datatype xsd:decimal ].

:SoldProduct a sh:NodeShape, rdfs:Class ;
  sh:and ( 
    :Product 
    [ sh:path schema:purchaseDate ;
      sh:datatype xsd:date]
    [ sh:path schema:productId ;
      sh:pattern "^[A-Z]" ]
  ).
```

Extending by composition
Reusability - Extending shapes (2)

ShEx: no special treatment for `rdfs:Class`, `rdfs:subClassOf`, ...

By design, ShEx has no concept of Class
Not possible to extend by declaring subClass relationships
No interaction with inference engines

SHACL shapes can also be extended by leveraging subclasses

```
:Product a sh:NodeShape, rdfs:Class ;
   ...as before...

:SoldProduct a sh:NodeShape, rdfs:Class ;
  rdfs:subClassOf :Product ;
  sh:property [ sh:path schema:productId ;
                sh:pattern "^[A-Z]"
    ] ;
  sh:property [ sh:path schema:purchaseDate ;
                sh:datatype xsd:date
    ] .
```

SHACL subclasses may differ from RDFS/OWL subclasses
Reusability - Extending shapes (3)

ShEx 2.2 Extension by aggregation

`extends` keyword added to the language

```json
:Product {
    :code /P[0-1]{4}/ ;
}

:Book extends :Product {
    :code /^isbn:[0-1]{10}/
}

:phone :code "P4356" .

:mobyDick :code "P4789",
    "isbn:9876543210"
```

SHACL: no extensión by aggregation
Exclusive-or and alternatives

ShEx operator | declares choices

```
:Person {
  foaf:firstName . ; foaf:lastName .
  | schema:givenName . ; schema:familyName .
}
```

SHACL provides `sh:xone` for exactly one, but...

```
:PersonShape a sh:NodeShape;
sh:targetClass :Person ;
sh:xone (  
  [ sh:property [  
    sh:path foaf:firstName;  
    sh:minCount 1; sh:maxCount 1  
  ]];  
  sh:property [  
    sh:path foaf:lastName;  
    sh:minCount 1; sh:maxCount 1  
  ]])  
[ sh:property [  
    sh:path schema:givenName;  
    sh:minCount 1; sh:maxCount 1  
  ]];  
  sh:property [  
    sh:path schema:familyName;  
    sh:minCount 1; sh:maxCount 1  
  ])
).
```

It doesn't check partial matches
Exclusive-or and alternatives

ShEx operator | declares choices

```shex
:Person {
  foaf:firstName . ; foaf:lastName .
  | schema:givenName . ; schema:familyName .
}
```

SHACL solution with normalization...

```shex
:Person a sh:NodeShape;
  sh:or (  
    [ sh:property [  
      sh:path foaf:firstName ;  
      sh:minCount 1 ;  
      sh:maxCount 1  
    ]  
    , sh:property [  
      sh:path foaf:lastName ;  
      sh:minCount 1 ;  
      sh:maxCount 1  
    ]  
    , sh:property [  
      sh:path schema:givenName ;  
      sh:minCount 1 ;  
      sh:maxCount 1  
    ]  
    , sh:property [  
      sh:path schema:familyName ;  
      sh:minCount 1 ;  
      sh:maxCount 1  
    ]  
  ]  
).
```

 alice foaf:firstName "Alice" ;
   foaf:lastName "Cooper" .

 bob schema:givenName "Robert" ;
   schema:familyName "Smith" .

 carol foaf:firstName "Carol" ;
   foaf:lastName "King" ;
   schema:givenName "Carol" ;
   schema:familyName "King" .

 dave foaf:firstName "Dave" ;
   foaf:lastName "Clark" ;
   schema:givenName "Dave" .
Annotations

ShEx allows annotations but doesn't have predefined annotations yet

Annotations can be declared by //

```
:Person {
  // rdfs:label "Name"
  // rdfs:comment "Name of person"
  schema:name xsd:string ;
}
```

Any kind of annotations can be defined, some proposals:
Form generation for UI
Severities: MUST, SHOULD, etc.

SHACL allows any kind of annotations and has some non-validating built-in annotations

Built-in properties: sh:name, sh:description, sh:defaultValue, sh:order, sh:group

```
:Person a sh:NodeShape ;
sh:property [
  sh:path schema:name ;
  sh:datatype xsd:string ;
  sh:name "Name" ;
  sh:description "Name of person"
  rdfs:label "Name";
].
```

Apart of the built-in annotations, SHACL can also use any other annotation
Validation report

**ShEx 2.0 defines a result shape map**
It contains both positive and negative node/shape associations

**SHACL defines a validation report**
Describes only the structure of errors

In practice, difficult to distinguish between:
- nodes that are valid
- nodes that are ignored (not validated)

Some properties can be used to control which information is shown
- `sh:message`
- `sh:severity`
ShEx uses semantic actions
Semantic actions allow any future processor
They can be used also to transform RDF

SHACL has SHACL-SPARQL
SHACL-SPARQL allows new constraint components defined in SPARQL
[See example in next slide]
It is possible to define constraint components in other languages, e.g.
Javascript

```javascript
:Event {
  schema:startDate xsd:date %js:{  start = o %} ;
  schema:endDate xsd:date %js:{  end = o %} ;
  %js: { start < end %}
}
```
**ShEx can describe stems**

Stems are built into the language

**Example:**
The value of :homepage starts by <http://company.com/>

```plaintext
```

**Stems are not built-in**

Can be defined using SHACL-SPARQL

```plaintext
:StemConstraintComponent
   a sh:ConstraintComponent ;
   sh:parameter [ sh:path :stem ] ;
   sh:validator [ a sh:SPARQLAskValidator ;
                  sh:message "Value does not have stem {$stem}";
                  sh:ask """
                  ASK {
                     FILTER (!isBlank($value) &&
                              strstarts(str($value),str($stem)))
                  }"""
   ] .

:Employee a sh:NodeShape ;
   sh:targetClass :Employee ;
   sh:property [ sh:path :homepage ;
```
Some future work ideas

ShEx ⇔ SHACL

Recursion & negation

Inference of shapes from data

Visualization & authoring tools

Validation usability (better error information/visualization)

RDF Stream validation
End of presentation