ShEx by example

RDF Validation tutorial

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

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

Harold Solbrig
Mayo Clinic, USA

Iovka Boneva
LINKS, INRIA & CNRS
University of Lille, France

ShEx

ShEx (Shape Expressions Language)

High level, concise Language for RDF validation & description

Official info: http://shex.io

Inspired by RelaxNG, Turtle

ShEx as a language

- Language based approach (domain specific language)
  - Specification repository: http://shexspec.github.io/
  - Abstract syntax & semantics http://shexspec.github.io/semantics/
  - Different serializations:
    - ShExC (Compact syntax): https://www.w3.org/2005/01/yacker/uploads/ShEx2/bnf
    - JSON http://shex.io/primer/ShExJ
    - RDF (in progress)

Short history of ShEx

2013 - RDF Validation Workshop

- Conclusions: "*SPARQL queries cannot easily be inspected and understood...to uncover the constraints that are to be respected*"
- Need of a higher level, concise language
- Agreement on the term "Shape"
- First proposal of Shape Expressions (ShEx) by Eric Prud'hommeaux

2014 - Data Shapes Working Group chartered

- Mutual influence between SHACL & ShEx

ShEx implementations

shex.js - Javascript

Source code: https://github.com/shexSpec/shex.js

Recent addition of a REST server

Installing the latest version locally

```
git clone git@github.com:shexSpec/shex.js.git
cd shex.js
npm install # wait 30s
cd rest
node server.js
```

shexcala - Scala (JVM)

Source code: https://github.com/labra/shExcala

shexpy - Python

Source code: https://github.com/hsolbrig/shexypy

Other prototypes: https://www.w3.org/2001/sw/wiki/ShEx

ShEx Online demos

- Fancy ShEx Demo https://www.w3.org/2013/ShEx/FancyShExDemo.html
  - Based on shex.js (Javascript)
  - Shows information about validation process
- RDFShape http://rdfshape.weso.es
  - Based on ShExcala
    - Developed using Play! framework and Jena
    - Can be used as a REST service and allows conversion between syntaxes
    - Recently added support for SHACL
- ShExValidata https://www.w3.org/2015/03/ShExValidata/
  - Based on an extension of shex.js
  - 3 deployments for different profiles HCLS, DCat, PHACTS

First example

User shapes must contain one property `schema:name` with a value of type `xsd:string`

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

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

Prefix declarations as in Turtle

Note: We will omit prefix declarations and use the aliases from:
http://prefix.cc

RDF Validation using ShEx

User shapes must contain one property schema:name with a value of type xsd:**string**

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

Schema

```
:alice schema:name "Alice" .                                    ☺

:bob   schema:name 234 .                                        ☹

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

:dave foaf:name "Dave" .                                        ☹

:emily schema:name "Emily" .                                    ☺
       schema:email <mailto:emily@example.org> .
```

Instance

A node fails if:

- there is a value of shema:name which is not xsd:**string**
- there are more than one value for schema:name
- there is no property schema:name

It doesn't fail if there are other properties apart of schema:name (Open Shape by default)

Try it (RDFShape): http://goo.gl/AuEldH Try it (ShExDemo): https://goo.gl/QCaQlu

ShExC - Compact syntax

BNF Grammar: https://www.w3.org/2005/01/yacker/uploads/ShEx2/bnf

Directly inspired by Turtle (reuses several definitions)

- Prefix declarations
- Comments starting by #
- a keyword for rdf:type
- Keywords aren't case sensitive (MinInclusive = MININCLUSIVE)

Shape Labels can be URIs or BlankNodes

ShEx-Json

Json serialization for Shape Expressions and validation results

See: http://shexspec.github.io/primer/ShExJ

```
<UserShape> {
  schema:name  xsd:string
}
```

≡

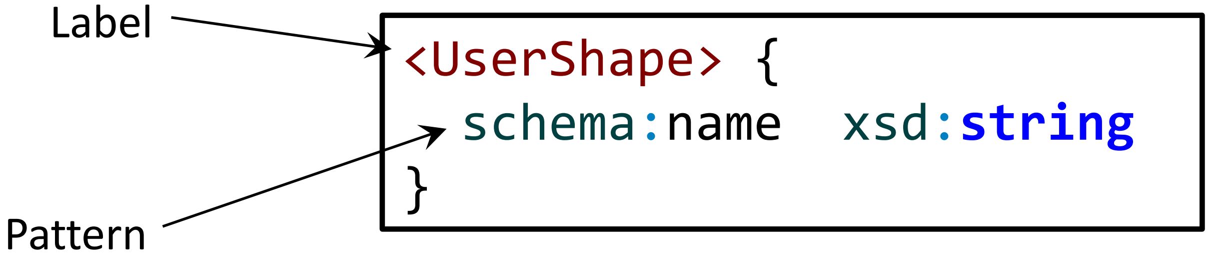
```
{
"type": "Schema",
 "shapes": {
  "User": {
  "type": "Shape",
   "expression" : {
   "type": "TripleConstraint",
   "predicate": "http://schema.org/name",
     "valueExpr": {
     "type": "ValueClass",
     "datatype": "http://www.w3.org/2001/XMLSchema#string"
}}}}}
```

Some definitions

Schema = set of Shape Expressions

Shape Expression = labeled pattern

```
<label> {
  ...pattern...
}
```



Label

```
<UserShape> {
  schema:name  xsd:string
}
```

Pattern

Focus Node and Neighborhood

Focus Node = node that is being validated

Neighborhood of a node = set of incoming/outgoing triples

```
:alice      schema:name      "Alice";
            schema:follows   :bob;
            schema:worksFor  :OurCompany .

:bob        foaf:name        "Robert" ;
            schema:worksFor  :OurCompany .

:carol      schema:name      "Carol" ;
            schema:follows   :alice .

:dave       schema:name       "Dave" .

:OurCompany schema:founder :dave ;
            schema:employee :alice, :bob .
```

```
Neighbourhood of :alice = {
  (:alice,       schema:name,      "Alice")
  (:alice,       schema:follows,   :bob),
  (:alice,       schema:worksFor,  :OurCompany),
  (:carol,       schema:follows,   :alice),
  (:OurCompany,  schema:employee,  :alice)
}
```

Validation process and node selection

Given a node and a shape, check that the neighborhood of the node matches the shape expression

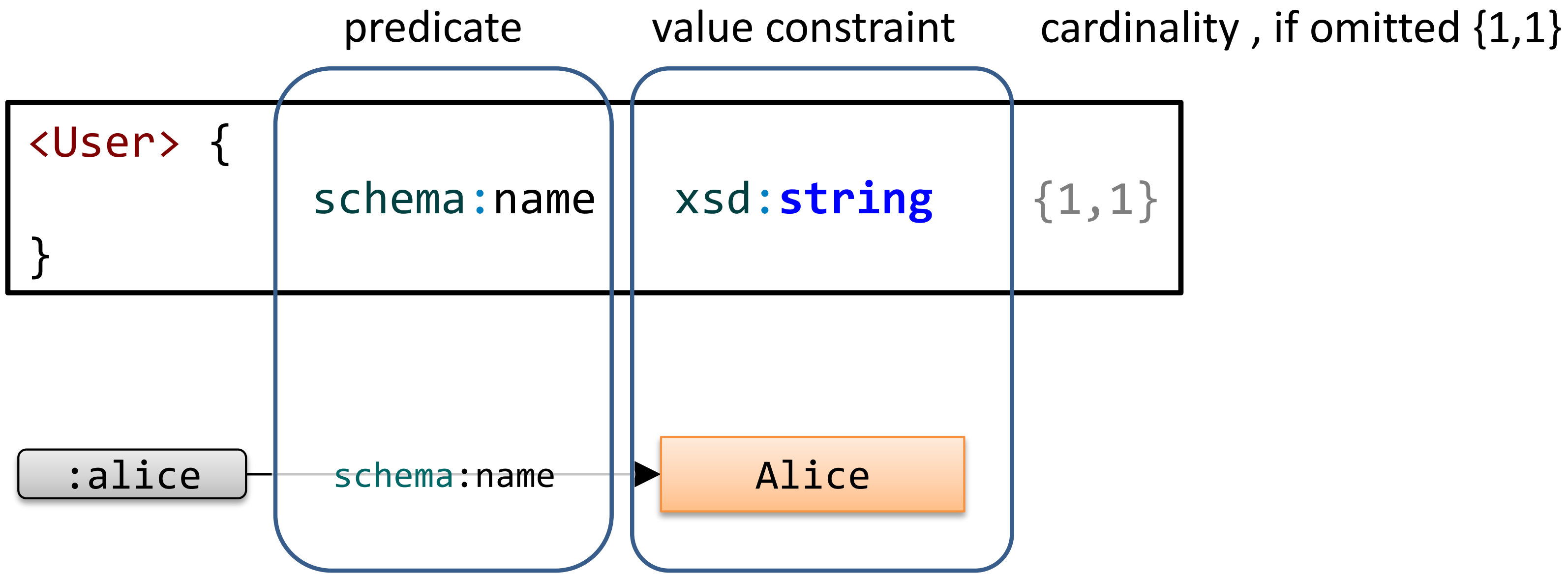
Which node and shape are selected?

- Several possibilities...
  - All nodes against all shapes
  - One node against one shape
  - One node against all shapes
  - All nodes against one shape
  - Explicit declarations: `sh:nodeShape` `sh:scopeNode` `sh:scopeClass`

Triple constraints

A basic expression consists of a Triple Constraint

Triple constraint ≈ predicate + value constraint + cardinality



predicate

value constraint

cardinality , if omitted {1,1}

```
<User> {
           schema:name    xsd:string    {1,1}
}
```

:alice

schema:name

Alice

Simple expressions and grouping

, or ; can be used to group components

```
:User {
 schema:name  xsd:string ;
 foaf:name    xsd:integer ;
 schema:email xsd:string ;
}
```

```
:alice schema:name  "Alice";
       foaf:age     10 ;
       schema:email "alice@example.org" .

:bob   schema:name  "Robert Smith" ;
       foaf:age     45 ;                          ☹
       schema:email <mailto:bob@example.org> .

:carol schema:name   "Carol" ;
       foaf:age      56, 66 ;                     ☹
       schema:email  "carol@example.org" .
```

Try it (RDFShape): http://goo.gl/GbhaJX

Try it (ShexDemo): https://goo.gl/APtLt8

Cardinalities

Inspired by regular expressions

- If omitted, {1,1} = default cardinality*

`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{m}`	m repetitions
`{m,n}`	Between m and n repetitions
`{m,}`	m or more repetitions

*__Note__: In SHACL, cardinality by default = (0,unbounded)

Example with cardinalities

```
:User {
 schema:name xsd:string ;
 schema:worksFor IRI ? ;
 schema:follows  IRI *
}

:Company {
 schema:founder  IRI ?;
 schema:employee IRI {1,100}
}
```

```
:alice      schema:name     "Alice";
            schema:follows  :bob;
            schema:worksFor :OurCompany .

:bob        foaf:name       "Robert" ;
            schema:worksFor :OurCompany .

:carol      schema:name     "Carol" ;
            schema:follows  :alice .

:dave       schema:name      "Dave" .

:OurCompany schema:founder :dave ;
            schema:employee :alice, :bob .
```

Try it (RDFShape): http://goo.gl/YlzLU8   Try it (ShExDemo): http://tinyurl.com/jbxen2u

Choices

The operator | represents alternatives (either one or the other)

```
:User {
  schema:name  xsd:string ;
| schema:givenName xsd:string + ;
  schema:lastName  xsd:string
}
```

```
:alice schema:name      "Alice Cooper" .

:bob   schema:givenName "Bob", "Robert" ;
       schema:lastName  "Smith" .

:carol schema:name      "Carol King" ;
       schema:givenName "Carol" ;
       schema:lastName  "King" .

:emily foaf:name        "Emily" .
```

Try it (RDFShape): http://goo.gl/R3pjA2 (ShExDemo): https://goo.gl/xLZRLf

Value constraints

Type	Example	Description
Anything	`.`	The object can be anything
Datatype	`xsd:string`	Matches a value of type `xsd:string`
Kind	`IRI BNode` `Literal NonLiteral`	The object must have that kind
Value set	`[:Male :Female ]`	The value must be an element of a that set
Reference	`@<UserShape>`	The object must have shape `<UserShape>`
Composed	`xsd:string OR IRI`	The Composition of value expressions using `OR AND NOT`
IRI Range	`foaf:~`	Starts with the IRI associated with `foaf`
Any except...	`- :Checked`	Any value except `:Checked`

No constraint

A dot (.) matches anything ⇒ no constraint on values

```
:User {
 schema:name . ;
 schema:affiliation . ;
 schema:email . ;
 schema:birthDate .
}
```

```
:alice
   schema:name        "Alice";
   schema:affiliation [ schema:name "OurCompany" ] ;
   schema:email       <mailto:alice@example.org> ;
   schema:birthDate   "2010-08-23"^^xsd:date .
```

Try it: http://goo.gl/xBndCT

Datatypes

Datatypes are directly declared by their URIs

- Predefined datatypes from XML Schema:
  - `xsd:string xsd:integer xsd:date …`

```
:User {
 schema:name      xsd:string ;
 schema:birthDate xsd:date
}
```

```
:alice schema:name      "Alice";
       schema:birthDate "2010-08-23"^^xsd:date .

:bob   schema:name      "Robert" ;
       schema:birthDate "2012-10-23" .

:carol schema:name      _:unknown ;
       schema:birthDate 2012 .
```

Try it: http://goo.gl/hqYDqu

Facets on Datatypes

It is possible to qualify the datatype with XML Schema facets

See: http://www.w3.org/TR/xmlschema-2/#rf-facets

Facet	Description
MinInclusive, MaxInclusive MinExclusive, MaxExclusive	Constraints on numeric values which declare the min/max value allowed (either included or excluded)
TotalDigits, FractionDigits	Constraints on numeric values which declare the total digits and fraction digits allowed
Length, MinLength, MaxLength	Constraints on string values which declare the length allowed, or the min/max length allowed
Pattern	Regular expression pattern

Facets on Datatypes

```
:User {
 schema:name  xsd:string  MaxLength 10 ;
 foaf:age     xsd:integer MinInclusive 1 MaxInclusive 99 ;
 schema:phone xsd:string  Pattern "\\d{3}-\\d{3}-\\d{3}"
}
```

```
:alice schema:name  "Alice";
       foaf:age     10 ;
       schema:phone "123-456-555" .

:bob   schema:name  "Robert Smith" ;
       foaf:age     45 ;
       schema:phone "333-444-555" .

:carol schema:name  "Carol" ;
       foaf:age     23 ;
       schema:phone "+34-123-456-555" .
```

Try it: http://goo.gl/LMwXRi

Node Kinds

Define the kind of RDF nodes: Literal, IRI, BNode, ...

Value	Description	Examples

`Literal`	Literal values	`"Alice"`
		`"Spain"@en`
		`23`
		`true`

`IRI`	IRIs	`<http://example.org/alice>` `ex:alice`
`BNode`	Blank nodes	`_:1`

`NonLiteral`	Blank nodes or IRIs	`_:1`
		`<http://example.org/alice>`
		`ex:alice`

Example with node kinds

```
:User {
 schema:name    Literal ;
 schema:follows IRI
}
```

```
:alice a :User;
       schema:name    "Alice" ;
       schema:follows :bob .

:bob   a :User;
       schema:name  :Robert ;
       schema:follows :carol .

:carol a :User;
       schema:name    "Carol" ;
       schema:follows "Dave"  .
```

Try it: http://goo.gl/vouWCU

Value sets

- The value must be one of the values of a given set
  - Denoted by [and]

```
:Product {
 schema:color [ "Red" "Green" "Blue" ] ;
 schema:manufacturer [ :OurCompany :AnotherCompany ]
}
```

```
:x1 schema:color "Red";
    schema:manufacturer :OurCompany .

:x2 schema:color "Cyan" ;
    schema:manufacturer :OurCompany .

:x3 schema:color "Green" ;
    schema:manufacturer :Unknown .
```

Try it: http://goo.gl/TgaoJ0

Single value sets

- Value sets with a single element
  - A very common pattern

```
<SpanishProduct> {
 schema:country [ :Spain ]
}

<FrenchProduct> {
 schema:country [ :France ]
}

<VideoGame> {
 a [ :VideoGame ]
}
```

```
:product1 schema:country :Spain .

:product2 schema:country :France .

:product3 a :VideoGame ;
          schema:country :Spain .
```

Note: ShEx doesn't interact with inference
It just checks if there is an `rdf:type` arc
- Inference can be done before/after validating
- ShEx can even be used to test inference systems

Try it: http://goo.gl/W464fn

Shape references

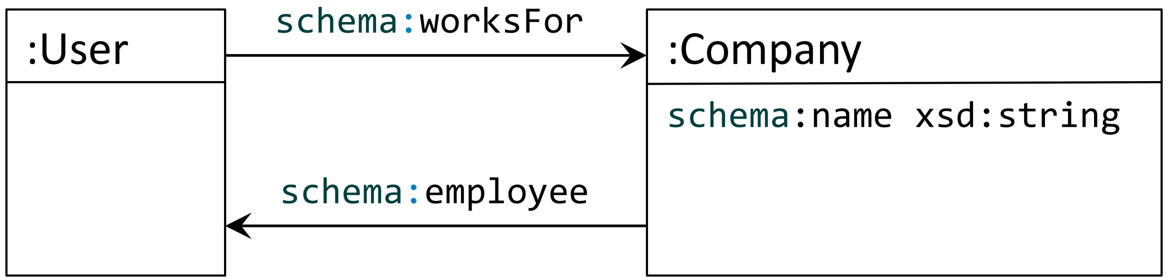
- Defines that the value must match another shape
  - References are marked as @

```
:User {
 schema:worksFor @:Company ;
}

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

```
:alice a :User;
       schema:worksFor :OurCompany .

:bob   a :User;
       schema:worksFor :Another .

:OurCompany
    schema:name "OurCompany" .

:Another
    schema:name 23 .
```

Try it: http://goo.gl/qPh7ry

Recursion and cyclic references



```
:User {
 schema:worksFor @:Company ;
}

:Company {
 schema:name      xsd:string ;
 schema:employee @:User
}
```

```
:alice a :User;
         schema:worksFor :OurCompany .

:bob   a :User;
         schema:worksFor :Another .

:OurCompany
     schema:name "OurCompany" ;
         schema:employee :alice .

:Another
     schema:name 23 .
```

Try it: http://goo.gl/507zss

Composed value constraints

It is possible to use AND and OR in value constraints

```
:User {
 schema:name     xsd:string ;
 schema:worksFor  IRI OR @:Company ?;
 schema:follows  IRI OR BNode *
}

:Company {
 schema:founder  IRI ?;
 schema:employee IRI {1,100}
}
```

```
:alice      schema:name     "Alice";
            schema:follows  :bob;
            schema:worksFor :OurCompany .
:bob        schema:name     "Robert" ;
            schema:worksFor [
             schema:Founder "Frank" ;
             schema:employee :carol ;
           ] .
:carol      schema:name     "Carol" ;
            schema:follows  [
              schema:name     "Emily" ;
           ] .
:OurCompany schema:founder :dave ;
            schema:employee :alice, :bob .
```

Try it:http://goo.gl/XXlKs4

IRI ranges

`uri:~` represents the set of all URIs that start with stem `uri`

```
prefix codes: <http://example.codes/>

:User {
  :status [ codes:~ ]
}
```

```
prefix codes: <http://example.codes/>
prefix other: <http://other.codes/>

:x1 :status codes:resolved .

:x2 :status other:done .

:x3 :status <http://example.codes/pending> .
```

☹

Try it: https://goo.gl/sNQi8n

Note: IRI ranges are not yet implemented in RDFShape

IRI Range exclusions

The operator - excludes IRIs or IRI ranges from an IRI range

```
prefix codes: <http://example.codes/>
prefix other: <http://other.codes/>

:User {
  :status [
      codes:~ - codes:deleted
  ]
}
```

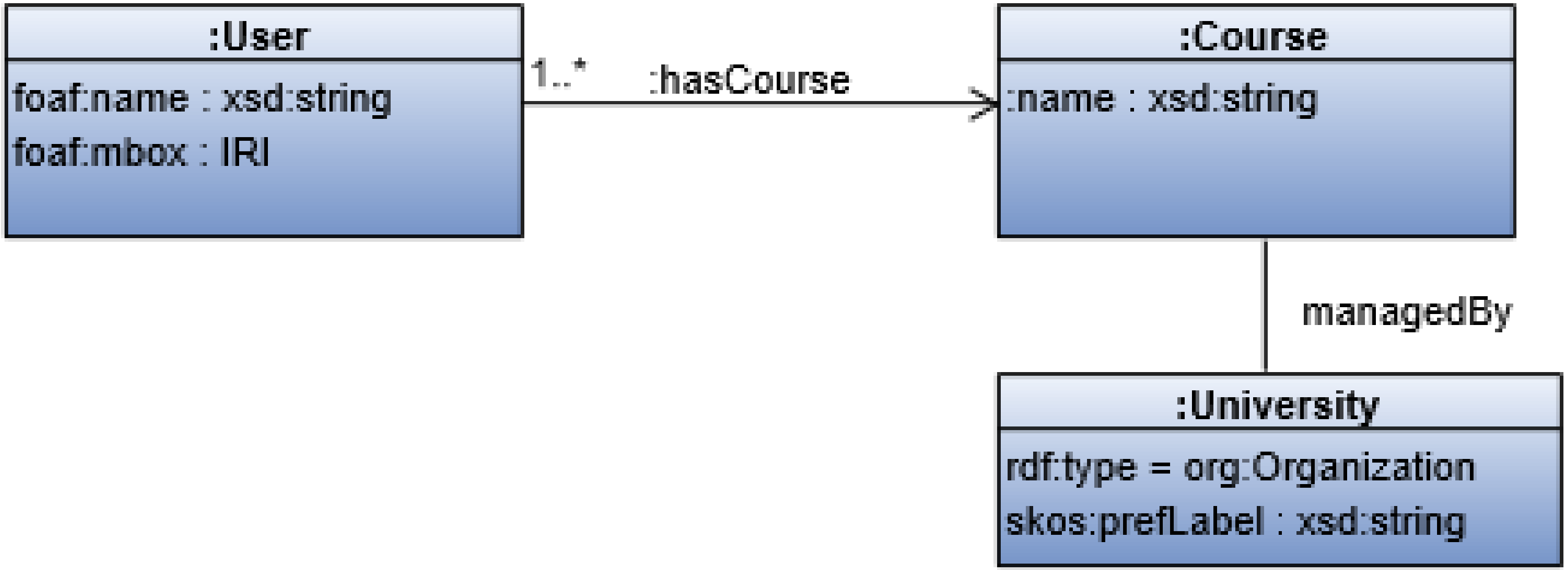
```
:x1 :status codes:resolved .

:x2 :status other:done.                                ☹

:x3 :status <http://example.codes/pending> .

:x4 :status codes:deleted .                            ☹
```

Try it: https://goo.gl/BvtTi2

Note: IRI range exclusions are not yet implemented in RDFShape

Exercise

Define a Schema for the following domain model


:User

foaf:name : xsd:string

foaf:mbox : IRI

1..*

:hasCourse

:Course

:name : xsd:string

managedBy

:University

rdf:type = org:Organization

skos:prefLabel : xsd:string

Nested shapes

Syntax simplification to avoid defining two shapes

- Internally, the inner shape is identified using a blank node

```
:User {
  schema:name xsd:string ;
  schema:worksFor {
    a [ schema:Company ]
  }
}
```

≡

```
User2 {
  schema:name xsd:string ;
  schema:worksFor _:1
}

_:1 a [ schema:Company ] .
```

```
:alice schema:name "Alice" ;
          schema:worksFor :OurCompany .

:OurCompany a schema:Company .
```

Try it (ShExDemo): https://goo.gl/z05kpL

Try it (RDFShape): http://goo.gl/aLt4rO

Combined value constraints

Value constraints can be combined (implicit AND)

```
:User {
  schema:name xsd:string ;
  schema:worksFor IRI @:Manager PATTERN "^http://hr.example/id#[0-9]+"
}

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

```
:alice schema:name "Alice";
       schema:worksFor <http://hr.example/id9> .

<http://hr.example/id> a :Manager .
```

Labeled constraints

- The syntax $label = <valueConstraint> allows to associate a value constraint to a label
  - It can later be used as $label

```
:CompanyConstraints =
  IRI @:CompanyShape PATTERN "^http://hr.example/id#[0-9]+"

<User> {
 schema:name        xsd:string;
 schema:worksFor    $:CompanyConstraints;
 schema:affiliation $:CompanyConstraints
}

<CompanyShape> {
  schema:founder xsd:string;
}
```

Inverse triple constraints

^ reverses the order of the triple constaint

```
:User {
 schema:name xsd:string ;
 schema:woksFor @:Company
}

:Company {
 a [schema:Company] ;
 ^schema:worksFor @:User+
}
```

```
:alice schema:name "Alice";
       schema:worksFor :OurCompany .

:bob schema:name "Bob" ;
     schema:worksFor :OurCompany .

:OurCompany a schema:Company .
```

Try it (ShEx demo): https://goo.gl/8omekl

Try it (RDFShape): http://goo.gl/CRj7J8

Negated property declarations

The ! operator negates a triple constraint

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

:Solitary {
 !schema:knows . ;
 !^schema:knows .
}
```

```
:alice schema:name "Alice" ;
       schema:knows :bob, :dave .

:bob   schema:name "Bob" ;
       schema:knows :dave .

:carol schema:name "Carol" .

:dave  schema:name "Dave" ;
       schema:knows :alice .
```

Try it (ShExDemo): https://goo.gl/7gEb5g

Try it (RDFShape): http://goo.gl/yUcrmD

Repeated properties

```
<User> {
 schema:name   xsd:string;
 schema:parent @<Male>;
 schema:parent @<Female>
}

<Male> {
 schema:gender [schema:Male ]
}

<Female> {
 schema:gender [schema:Female]
}
```

```
:alice schema:name   "Alice" ;
       schema:parent :bob, :carol .

:bob   schema:name   "Bob" ;
       schema:gender schema:Male .

:carol schema:name   "Carol" ;
       schema:gender schema:Female .
```

Try it (RDFShape): http://goo.gl/ik4IZc

Permitting other triples

Triple constraints limit all triples with a given predicate to match one of the constraints

This is called *closing a property*

Example:

```
<Company> {
 a [ schema:Organization ] ;
 a [ org:Organization ]
}
```

```
:OurCompany a org:Organization,
  schema:Organization .

:OurUniversity a org:Organization,
  schema:Organization,
  schema:CollegeOrUniversity .
```

☹

Sometimes we would like to permit other triples (open the property)

Try it (RDFShape): http://goo.gl/m4fPzY

Permitting other triples

`EXTRA <listOfProperties>` declares that a list of properties can contain extra values

Example:

```
<Company> EXTRA a {
 a [ schema:Organization ] ;
 a [ org:Organization ]
}
```

```
:OurCompany a org:Organization,
  schema:Organization .

:OurUniversity a org:Organization,
  schema:Organization,
  schema:CollegeOrUniversity .
```

Try it (RDFShape): http://goo.gl/m4fPzY

Closed Shapes

- CLOSED can be used to limit the appearance of any predicate not mentioned in the shape expression

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

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

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

:dave schema:name "Dave" ;
 schema:knows :emily ;
 :link2virus <virus> .

:emily schema:name "Emily" ;
 schema:knows :dave .
```

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

Without closed, all match <User>

Try without closed: http://goo.gl/vJEG5G

With closed, only `:alice` and `:bob` match <User>

Try with closed: http://goo.gl/KWDEEs

Node constraints

Constraints on the focus node

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

```
:alice schema:name "Alice";
  :worksFor :OurCompany .

_:1 schema:name "Unknown";
  :worksFor :OurCompany .
```

☹

Conjunction of Shape Expressions

AND can be used to define conjunction on Shape Expressions

- Other top-level logical operators are expected to be added: NOT, OR

```
<User> { schema:name xsd:string ; schema:worksFor IRI }
   AND { schema:worksFor @<Company> }
```

*Conjunctions are employed in SHACL

Semantic Actions

- Arbitrary code attached to shapes
  - Can be used to perform operations with side effects
  - Independent of any language/technology
    - Several extension languages: GenX, GenJ (http://shex.io/extensions/)

```
<Person> {
 schema:name xsd:string,
 schema:birthDate xsd:dateTime
 %js:{ report = _.o; return true; %},
 schema:deathDate xsd:dateTime
 %js:{ return _[1].triple.o.lex > report.lex; %}
 %sparql:{
  ?s schema:birthDate ?bd . FILTER (?o > ?bd) %}
}
```

```
:alice schema:name "Alice" ;
 schema:birthDate "1980-01-23"^^xsd:date ;
 schema:deathDate "2013-01-23"^^xsd:date .

:bob schema:name "Robert" ;
 schema:birthDate "2013-08-12"^^xsd:date ;
 schema:deathDate "1990-01-23"^^xsd:date .
```

Other features

Current ShEx version: 2.0

Several features have been postponed for next version

- UNIQUE
- Inheritance (a Shape that extends another Shape)
- External logical operators: NOT, OR
- Language tag and datatype inspection

Future work & contributions

- Complete test-suite
  - See: https://github.com/shexSpec/shexTest (≈600 tests)
- More info http://shex.io
  - ShEx currently under active development
  - Curent work
    - Improve error messages
    - Language expressivity (combination of different operators)
- If you are interested, you can help
  - List of issues: https://github.com/shexSpec/shex/issues