ShEx & SHACL compared

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

Several common features...

	ShEx	SHACL
Employ the word "shape"	✓	✓
Validate RDF graphs	✓	✓
Node constraints	✓	✓
Constraints on incoming/outgoing arcs	✓	✓
Defining cardinalities on properties	✓	✓
RDF syntax	✓	✓
Extension mechanism	✓	✓

But several differences…

Underlying philosophy

Syntactic differences

Notion of a shape

Syntactic differences

Default cardinalities

Shapes and Classes

Recursion

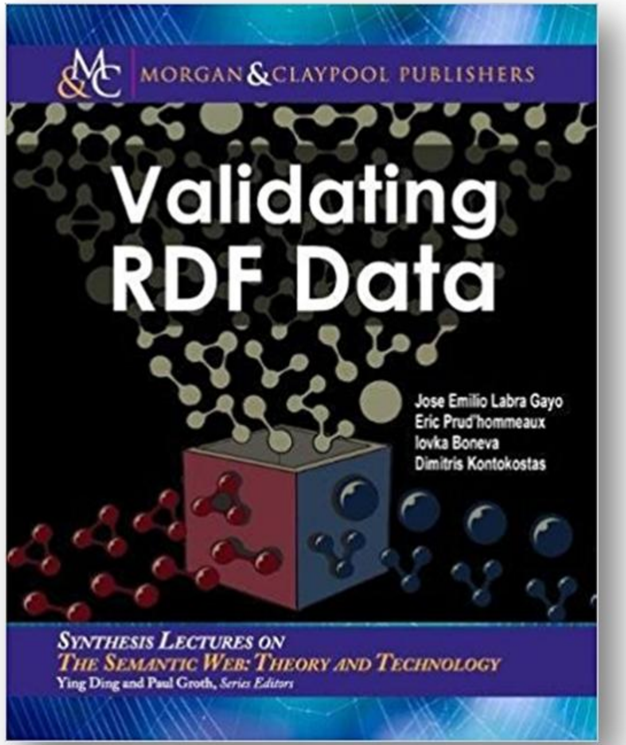
Repeated properties

Property pair constraints

Uniqueness

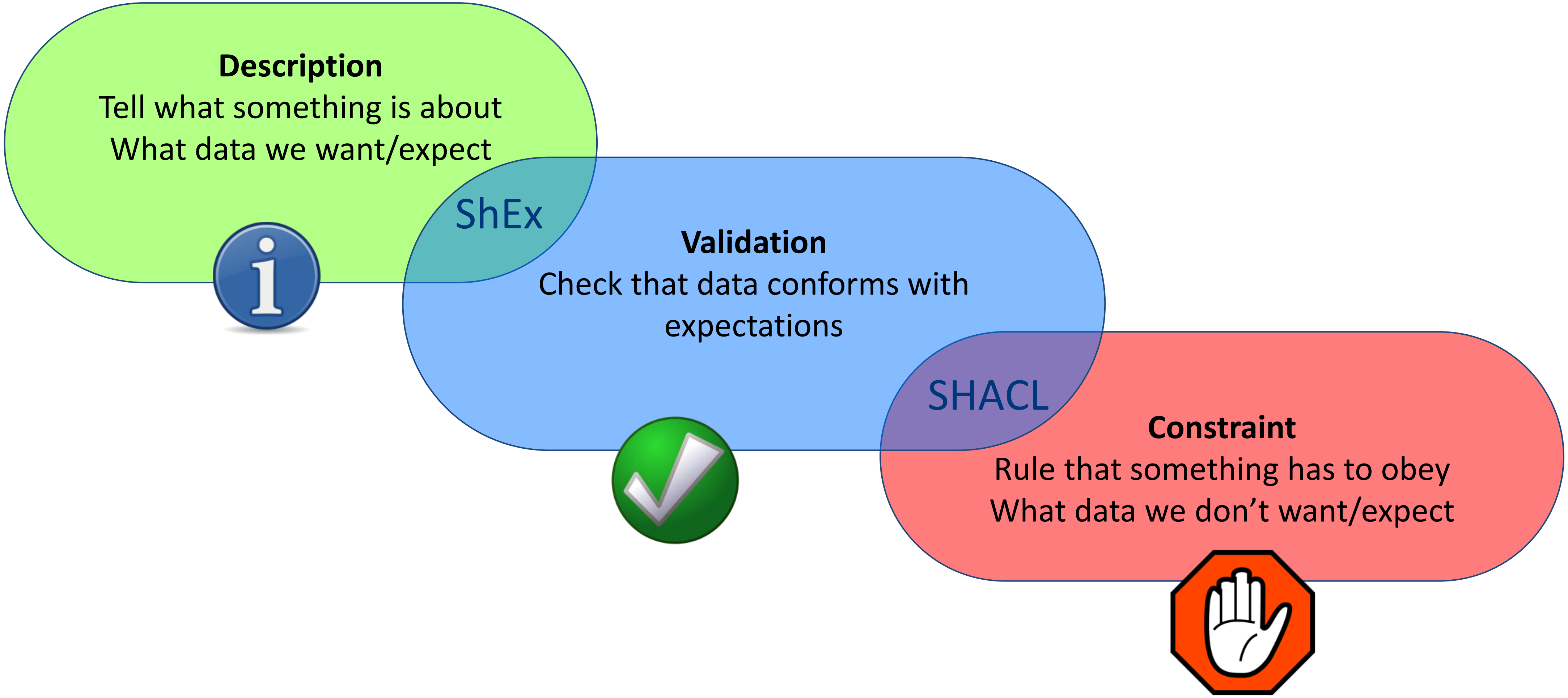
Extension mechanism

More info in Chapter 7 of:



Jose E. Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, Dimitris Kontokostas, *Validating RDF Data*, Synthesis Lectures on the Semantic Web, Vol. 7, No. 1, 1-328, DOI: 10.2200/S00786ED1V01Y201707WBE016, Morgan & Claypool (2018)Online version: http://book.validatingrdf.com/

Emphasis on
description - validation - constraints



Underlying philosophy

ShEx is more *schema* based

Shape ≈ grammar

More focus on validation results

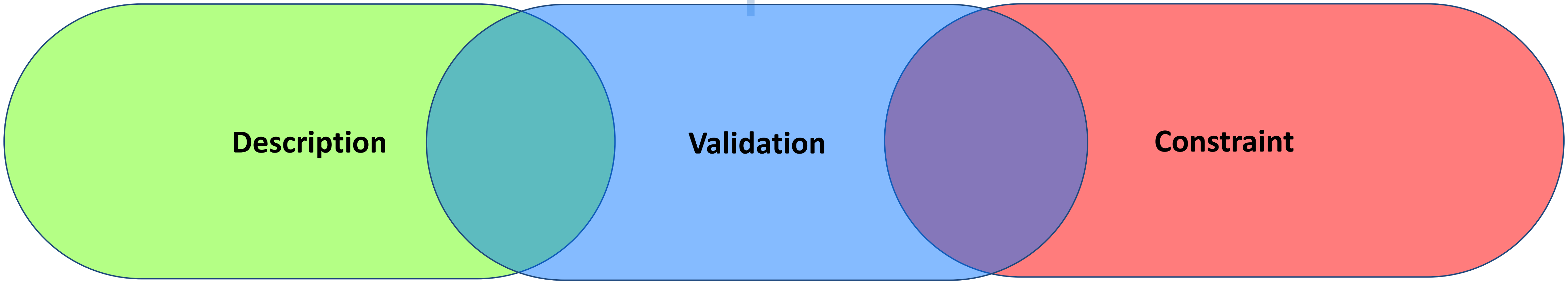
Result shape maps = Conforming and non-conforming nodes

SHACL is more constraint based

Shapes ≈ collections of constraints

More focus on validation errors

Validation report = set of violations



Design principles

ShEx = based on regular expressions

Cyclic data models = part of the language

SHACL = designed from Data Shapes WG

Cyclic data models = implementation dependent

Syntactic differences

ShEx design focused on human-readability

Followed 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

Compact syntax created after RDF syntax

Semantic specification

ShEx semantics: mathematical concepts

Well-founded semantics*

Support for recursión and negation

Inspired by type systems and RelaxNG

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

SHACL semantics = textual description + SPARQL

SHACL terms described in natural language

SPARQL fragments used as helpers

Recursion is implementation dependent*

Several proposals to add recursion:
*Semantics and Validation of Recursive SHACL
Julien Corman, Juan L. Reutter, Ognjen Savkovic, ISWC’18
And more recent ones based on ASP concepts

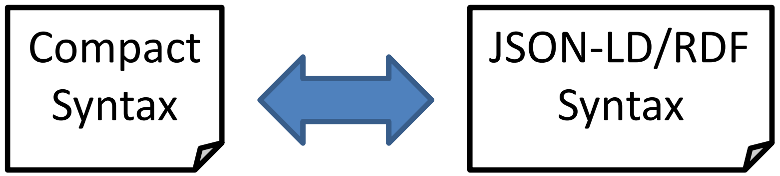
Most were proposed after the spec

Compact Syntax

ShEx compact syntax designed along the language

Test-suite with long list of tests

Round-trip with JSON-LD and RDF syntax



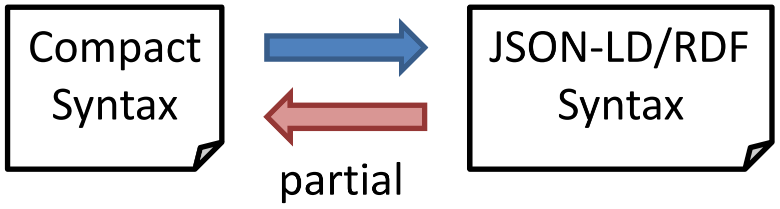
SHACL compact syntax is lossy

A WG Note proposed a compact syntax

It covers a subset of SHACL core

SHACL-C is complemented with several production rules

Some files can be parsed by the grammar but should be rejected by the rules



Compact syntax

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

:UserShape CLOSED EXTRA rdf:type {
    schema:name      xsd:string ?                   ;
    schema:gender    [ schema:Male schema:Female ] ;
    schema:birthDate xsd:date    ?                  ;
    schema:knows     @:User      *
}
```

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

shape :UserShape -> :User {
    closed=true ignoredProperties=[rdf:type]               .
    schema:name      xsd:string [0..1]                     .
    schema:gender    in = [ schema:Male schema:Female ]    .
    schema:birthDate xsd:date [0..1]                       .
    schema:knows     :User [0..*]                          .
}
```

RDF vocabulary

ShEx vocabulary ≈ abstract syntax

- ShEx RDF vocabulary obtained from abstract syntax
  - ShEx RDF serializations typically more verbose
  - They can be round-tripped to Compact syntax

```
:User a sx:Shape;
  sx:expression [ a sx:EachOf ;
    sx:expressions (
     [ a sx:TripleConstraint ;
       sx:predicate schema:name ;
       sx:valueExpr [ a sx:NodeConstraint ;
        sx:datatype xsd:string ]
     ]
     [ a sx:TripleConstraint ;
       sx:predicate schema:birthDate ;
       sx:valueExpr [ a sx:NodeConstraint ;
         sx:datatype xsd:date ] ;
         sx:min 0
     ] )
  ].
```

SHACL designed as an RDF vocabulary

- Some rdf:type declarations can be omitted
  - SHACL RDF serialization typically more readable

```
:User a sh:NodeShape ;
  sh:property [ sh:path schema:name ;
   sh:minCount 1; sh:maxCount 1;
   sh:datatype xsd:string
  ];
  sh:property [ sh:path schema:birthDate ;
   sh:maxCount 1;
   sh:datatype xsd:date
  ] .
```

Notion of Shape

In ShEx, shapes only define structure of nodes

Shape maps select which nodes are validated with which shapes

Goal: separation of concerns

Shape

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

Shape map

```
: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

Although target declarations can be written in a separate graph (recommended)

Shape

```
:User a sh:NodeShape, rdfs:Class ;
  sh:targetClass :Person ;
  sh:targetNode  :alice ;
  sh:nodeKind    sh:IRI ;
  sh:property [
   sh:path     schema:name ;
   sh:datatype xsd:string
  ] .
```

target declarations

structure

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 ;
 ] .
```

ShEx	Data	SHACL
☺	`: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 .`	☹

☺ = conforms to Shape

☹ = doesn't conform

Property paths

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

- Recursion paths can be emulated by nested shapes
  - Sometimes requiring auxiliary recursive shapes

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

SHACL shapes can also describe whole property paths following SPARQL paths

```
:GrandSon a sh:NodeShape ;
  sh:property [
   sh:path (schema:parent schema:parent) ;
   sh:minCount 1
  ];
  sh:property [
   sh:path [
    sh:alternativePath (:father :mother) ]
  ];
  sh:minCount 1
 ] ;
 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

```
:GrandParent {
  schema:knows @:PersonKnown*;
}
:PersonKnown @:Person {
  schema:knows @:PersonKnown*
}

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

SHACL shapes can use property paths

```
: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
 }
}
```

```
:i1 :payment  [ :amount  3.0 ] .

:i2 :payment  [ :amount  3.0 ] ;
    :payment  [ :amount  2.0 ] .

:i3 :payment  [ :amount  2.0 ] ;
    :payment  [ :amount  2.0 ] .
```

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

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/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, ...

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

Some implicit RDFS inference but not all

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

```
:Teacher rdfs:subClassOf :User .
:teaches rdfs:domain :Teacher .

:alice a :Teacher       ;
       schema:name 23 .

:bob   :teaches :Algebra ;
       schema:name 34     .

:carol :teaches     :Logic;
       schema:name "King" .
```

Node	ShEx: With or without RDFS inference	SHACL: No RDFS inference	SHACL: RDFS inference
:alice	☹	☹	☹
:bob	☹	Ignored	☹
:carol	☺	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

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

Direct approximation (wrong):

```
: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 handles repeated properties with qualifiedValueShapes

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

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

Solution with qualifiedValueShapes:

```
: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 implementation dependent in SHACL*

```
: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 with stratified negation

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

Recursive shapes are undefined in SHACL

Implementation dependent

Can be simulated with target declarations

- Example with target declatations
- It needs discriminating arcs

```
: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

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

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, 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 shapes and paths

Closed in ShEx acts on all properties

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

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 ] ;
.
```

☺

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

☹

☺ = conforms to Shape

☹ = doesn't conform

Property pair constraints

This feature was posponed

Proposal in github issue

Not supported yet

```
: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>
}
```

SHACL supports equals, disjoint, lessThan, ...

```
:UserShape a sh:NodeShape ;
 sh:property [
   sh:path schema:givenName ;
   sh:datatype xsd:string ;
   sh:disjoint schema:lastName
 ] ;
 sh:property [
   sh:path foaf:firstName ;
   sh:equals schema:givenName ;
 ] ;
 sh:property [
   sh:path schema:birthDate ;
   sh:datatype xsd:date ;
   sh:lessThan :loginDate
 ] .
```

Modularity

ShEx has EXTERNAL and import keywords

import imports shapes from URI

external declares that a shape definition can be retrieved elsewhere

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

http://example.org/UserShapes

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

:TeacherShape :UserShape AND {
 :teaches . ;
 :teacherCode external
}
```

SHACL supports owl:imports

SHACL processors follow `owl:imports`

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

http://example.org/UserShapes

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

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

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

Extending by composition

```
: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]" ]
  ) .
```

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

Extending 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 subclases

Reusability - Extending shapes (3)

ShEx 2.2 is planning to add extends

extends keyword added to the language

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

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

```
:phone     :code "P4356" .

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

SHACL doesn't have this feature

Basic cases can be emulated with AND

Exclusive-or and alternatives

ShEx operator | declares choices

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

```
: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" .
```

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

?

```
:PersonShape a sh:NodeShape;
 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

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

```
: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" .
```

SHACL solution with normalization...

```
:Person a sh:N
 sh:or (
  [ sh:propert
     sh:path f
     sh:minCou
     sh:maxCou
    ];
    sh:propert
     sh:path f
     sh:minCou
     sh:maxCou
    ];
   sh:property
    sh:path sc
    sh:maxCoun
   ];
   sh:property
    sh:path sc
    sh:maxCoun
   ];
 ]
```

```
[ sh:property [
    sh:path foaf:firstName;
    sh:maxCount 0
   ];
  sh:property [
    sh:path foaf:lastName;
    sh:maxCount 0
   ];
  sh:property [
    sh:path schema:givenName;
    sh:minCount 1;
    sh:maxCount 1
   ] ;
  sh:property [
    sh:path schema:familyName;
    sh:minCount 1;
    sh:maxCount 1
   ];
] )
.
```

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 ;
}
```

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 defines a result shape map

- It contains both positive and negative node/shape associations
- It doesn't specify the structure of errors

SHACL defines a validation report

- Describes only the structure of errors
- Some properties can be used to control which information is shown
  - sh:message
  - sh:severity

Extension mechanism

ShEx uses semantic actions

Semantic actions allow any future processor

They can be used also to transform RDF

```
:Event {
 schema:startDate xsd:date  %js:{ start = o %} ;
 schema:endDate   xsd:date  %js:{ end = o %}   ;
 %js: { start < end %}
}
```

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

Stems

ShEx can describe stems

Stems are built into the language

Example:

The value of :homePage starts by <http://company.com/>

```
:Employee {
 :homePage [ <http://company.com/> ~ ]
}
```

Stems are not built-in

But can be defined using SHACL-SPARQL

```
: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:property [
  sh:path :homePage ;
  :stem <http://company.com/>
 ] .
```

End of first part

Introduction to ShEx/SHACL foundations

Warning: A more abstract perspective

Foundations of ShEx and SHACL

A short introduction to the theoretical foundations

The section is based on several papers (see references)

For ShEx/SHACL we present:

- Abstract syntax of a core ShEx/SHACL language
- Semantics
- Simple validation algorithm

RDF data model

Typical definition of an RDF graph

$$
\begin{array}{lcl}
\text{RDF Graph } \Sigma & = & \text{Set of triples } \langle s, p, o \rangle \\
 & \text{where} & s \in V_s,\ p \in V_p \text{ and } o \in V_o \\
 & \text{and} & V_s = \mathcal{I} \cup \mathcal{B} \text{ is the vocabulary of subjects} \\
 & \text{and} & V_p = \mathcal{I} \text{ is the vocabulary of predicates} \\
 & \text{and} & V_o = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L} \text{ is the vocabulary of objects}
\end{array}
$$

Example

$$
\begin{array}{lll}
\mathcal{G} & = \{ & \langle :alice, :name, \texttt{"alice"} \rangle \\
 & & \langle :alice, :knows, ::bob \rangle \\
 & & \langle :bob, :name, \texttt{"Robert"} \rangle \\
 & & \langle :bob, :knows, \_:1 \rangle \\
 & & \langle \_:1, :name, \texttt{"Unknown"} \rangle \\
 & \} & \\
 & \text{where} & \mathcal{I} = \{:alice, :bob\} \\
 & & \mathcal{B} = \{\_:1\} \\
 & & \mathcal{L} = \{\texttt{"alice"}, \texttt{"bob"} \}
\end{array}
$$

Operations on RDF graphs

$$
\begin{aligned}
\mathcal{G} = \{ & \langle :alice, :name, \texttt{"alice"} \rangle \\
& \langle :alice, :knows, ::bob \rangle \\
& \langle :bob, :name, \texttt{"Robert"} \rangle \\
& \langle :bob, :knows, \_:1 \rangle \\
& \langle \_:1, :name, \texttt{"Unknown"} \rangle \\
\} &
\end{aligned}
$$

$$neighs(n, \mathcal{G}) = \{\langle x, p, y \rangle \in \mathcal{G} | x = n\}$$

$$
\begin{aligned}
neighs(:alice, \mathcal{G}) &= \{ \langle :alice, :name, \texttt{"alice"} \rangle \\
& \quad\; \langle :alice, :knows, ::bob \rangle \} \\
neighs(:bob, \mathcal{G}) &= \{ \langle :bob, :name, \texttt{"Robert"} \rangle \\
& \quad\; \langle :bob, :knows, \_:1 \rangle \} \\
neighs(\_:1, \mathcal{G}) &= \{ \langle \_:1, :name, \texttt{"Unknown"} \rangle \}
\end{aligned}
$$

$$extract(\mathcal{G}) = (t, ts) \text{ such that } \{t\} \cup ts = \mathcal{G}$$

$$
\begin{aligned}
\text{extract}(\mathcal{G}) \quad & = (\text{t,ts}) \\
\text{where} \quad & \text{t} = \{ \langle :alice, :name, \texttt{"alice"} \rangle \} \\
\text{and} \quad & \text{ts} = \{ \langle :alice, :knows, ::bob \rangle \\
& \quad \langle :bob, :name, \texttt{"Robert"} \rangle \\
& \quad \langle :bob, :knows, \_:1 \rangle \\
& \quad \langle \_:1, :name, \texttt{"Unknown"} \rangle \\
& \}
\end{aligned}
$$

Operations on RDF graphs

Partition of a graph

$$partition(s) = \{(s_1, s_2) | s_1 \cup s_2 = s\}$$

$$
\begin{array}{lll}
\text{partition}(\{1,2,3\}) = & \{ & (\{\},\{1,2,3\}), \\
& & (\{1\},\{2,3\}), \\
& & (\{2\},\{1,3\}), \\
& & (\{3\},\{1,2\}), \\
& & (\{1,2\},\{3\}), \\
& & (\{1,3\},\{2), \\
& & (\{2,3\},\{1\}), \\
& & (\{1,2,3\},\{\}), \\
& \} &
\end{array}
$$

ShEx abstract syntax

$$
\begin{array}{lcll}
\mathcal{S} & ::= & l \mapsto se* & \\
se & ::= & \text{IRI} \mid \text{BNode} \mid \ldots & \text{Node constraints} \\
 & \mid & \text{cond} & \text{A boolean condition on nodes} \\
 & \mid & se_1 \ \texttt{AND} \ se_2 & \text{Conjunction} \\
 & \mid & se_1 \ \texttt{OR} \ se_2 & \text{Disjunction} \\
 & \mid & \texttt{NOT} \ se & \text{Negation} \\
 & \mid & @l & \text{Shape label reference for } l \in \Lambda \\
 & \mid & \{ \text{ te } \} & \text{Triple expression } te \\
te & ::= & te_1 ; te_2 & \text{Each of } te_1 \text{ and } te_2 \\
 & \mid & te_1 \mid te_2 & \text{Some of } te_1 \text{ or } te_2 \\
 & \mid & \sqcup \xrightarrow{p} @l & \text{Triple with predicate } p \\
 & & & \text{that conforms to shape expression identified by } l \\
 & \mid & te* & \text{Zero or more } te
\end{array}
$$

Example of a shape expression

$\mathcal{S}$	$::=$	$l \mapsto se^*$	
se	$::=$	IRI $\mid$ BNode $\mid \ldots$	Node constraints
	$\mid$	cond	A boolean condition on nodes
	$\mid$	se_1 `AND` se_2	Conjunction
	$\mid$	se_1 `OR` se_2	Disjunction
	$\mid$	`NOT` se	Negation
	$\mid$	$@l$	Shape label reference for $l \in \Lambda$
	$\mid$	{ te }	Triple expression te
te	$::=$	$te_1 ; te_2$	Each of te_1 and te_2
	$\mid$	$te_1 \mid te_2$	Some of te_1 or te_2
	$\mid$	$\llcorner \xrightarrow{p} @l$	Triple with predicate p that conforms to shape expression identified by l
	$\mid$	$te*$	Zero or more te

$$\begin{array}{lcl}
:Issue & \mapsto & \{\llcorner \xrightarrow{:code} @:Positive; \llcorner \xrightarrow{:reportedBy} :User\} \\
:User & \mapsto & \text{IRI } \texttt{AND} \; \{ \llcorner \xrightarrow{:name} @:String; \llcorner \xrightarrow{:knows} @:User* \} \\
:Positive & \mapsto & \geq 0 \\
:String & \mapsto & \in xsd:string
\end{array}$$

Fixed Shape Maps

Denoted by τ

Also known as Shape assignments, Shape typing, ...

Pairs $node@label$ or $node@!label$

Example: $\{:alice@:User, :bob@!:User, :carol@:User\}$

Actions on shape map typings:

- Empty shape map is represented by: $[]$
- Adding a pair $n@l$ to a shape map τ is denoted as $n@l : \tau$
- Removing a pair $n@l$ from a shape map τ is denoted by $\tau \setminus n@l$
- Create a new shape map where n doesn't conform to l from an existing shape map τ can be done with $n@!l : (\tau \setminus n@l)$

Semantics: Shape Expressions (se)

$$IRI \frac{n \in \mathcal{I}}{\mathcal{G}, n, \tau \vDash \text{IRI}} \qquad BNode \frac{n \in \mathcal{B}}{\mathcal{G}, n, \tau \vDash \text{BNode}}$$

$$Cond \frac{cond(n) = true}{\mathcal{G}, n, \tau \vDash cond}$$

$$AND \frac{\mathcal{G}, n, \tau \vDash se_1 \qquad \mathcal{G}, n, \tau \vDash se_2}{\mathcal{G}, n, \tau \vDash se_1 \ \texttt{AND} \ se_2}$$

$$OR_1 \frac{\mathcal{G}, n, \tau \vDash se_1}{\mathcal{G}, n, \tau \vDash se_1 \ \texttt{OR} \ se_2} \qquad OR_2 \frac{\mathcal{G}, n, \tau \vDash se_2}{\mathcal{G}, n, \tau \vDash se_1 \ \texttt{OR} \ se_2}$$

$$Shape \frac{neighs(n, \mathcal{G}) = ts \qquad \mathcal{G}, ts, \tau \Vdash te}{\mathcal{G}, n, \tau \vDash \{te\}}$$

Semantics: Triple expressions (te)

$$EachOf\ \frac{partition(ts) = (ts_1, ts_2) \qquad \mathcal{G}, ts_1, \tau \Vdash te_1 \qquad \mathcal{G}, ts_2, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1 ; te_2}$$

$$OneOf_1\ \frac{\mathcal{G}, ts, \tau \Vdash te_1}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2} \qquad OneOf_2\ \frac{\mathcal{G}, ts, \tau \Vdash te_2}{\mathcal{G}, ts, \tau \Vdash te_1 \mid te_2}$$

$$TripleConstraint\ \frac{vs = \{t \in ts | pred(t) = p\} \qquad |vs| = 1 \qquad \langle x, p, y \rangle \in v_s \qquad \mathcal{G}, y, \tau \vDash se}{\mathcal{G}, ts, \tau \Vdash \textvisiblespace \xrightarrow{p} se}$$

$$Star_1\ \frac{}{\mathcal{G}, \{\}, \tau \Vdash te*} \qquad Star_2\ \frac{extract(ts) = (t, ts') \qquad \mathcal{G}, t, \tau \Vdash te \qquad \mathcal{G}, ts', \tau \Vdash te*}{\mathcal{G}, ts, \tau \Vdash te*}$$

Validation algorithm

Algorithm 1: ShEx validation - recursive algorithm with backtracking

Input: A shape map typing t_s, a graph $\mathcal{G}$ and a schema $\mathcal{S}$
Output: A result shape map typing τ

def $check(t_s) = check(t_s, [])$

def $check(t_s, \tau) =$ **match** t_s
  case $[] \Rightarrow \tau$
  case $n@l : r_s \Rightarrow checkNodeLabel(n, l, \tau) \uplus check(r_s, \tau)$

def $checkNodeLabel(n, l, \tau) = checkNodeSe(n, l, @l, n@l : \tau)$

def $checkNodeSe(n, l, se, \tau) =$ **match** se
  case se_1 `AND` $se_2 \Rightarrow checkNodeSe(n, l, se_1) \uplus checkNodeSe(n, l, se_2)$
  case se_1 `OR` $se_2 \Rightarrow checkNodeSe(n, l, se_1) \parallel checkNodeSe(n, l, se_2)$
  case `NOT` $se \Rightarrow$ **if** $n@l \in checkNodeSe(n, l, se, \tau)$ **then** $n@!l : (\tau \setminus n@l)$
   else τ
  case $@l \Rightarrow checkNodeSe(n, l, \delta(l), \tau)$
  case $\{ te \} \Rightarrow checkTe(neighs(n), l, te, \tau)$
  case IRI $\Rightarrow$ **if** $n \in \mathcal{I}$ **then** τ **else** $n@!l : (\tau \setminus n@l)$
  case BNode $\Rightarrow$ **if** $n \in \mathcal{B}$ **then** τ **else** $n@!l : (\tau \setminus n@l)$
  case cond $\Rightarrow$ **if** $cond(n) = true$ **then** τ **else** $n@!l : (\tau \setminus n@l)$

def $checkTe(t_s, l, te, \tau) =$ **match** te
  case $te_1 ; te_2 \Rightarrow$ **let**
   $(ts_1, ts_2) = partition(t_s)$
   in $checkNodeTe(ts_1, l, te_1, \tau) \uplus checkNodeTe(ts_2, l, te_2, \tau)$
  case $te_1 \mid te_2 \Rightarrow checkNodeTe(ts, l, te_1, \tau) \parallel checkNodeTe(ts, l, te_1, \tau)$
  case $te* \Rightarrow$ **match** ts
   case $\{\} \Rightarrow \tau$
   case $t : ts \Rightarrow checkTe(\{t\}, l, te, \tau) \uplus checkTe(ts), l, te*, \tau)$
  case $\_ \xrightarrow{p} @l' \Rightarrow$ **if** $t_s = \{\langle n, p, n' \rangle\}$ **then**
   $checkNodeSe(n', l', @l', n'@l' : \tau)$
  else
   $n@!l : (\tau \setminus n@l)$

SHACL abstract syntax

$$
\begin{array}{rcll}
\mathcal{S} & ::= & l \mapsto (\phi, q)^* & \\
\phi & ::= & \text{IRI} \mid \text{BNode} \mid \ldots & \text{Node constraints} \\
 & \mid & \text{cond} & \text{A boolean condition on nodes} \\
 & \mid & \phi_1 \ \texttt{AND} \ \phi_2 & \text{Conjunction} \\
 & \mid & \phi_1 \ \texttt{OR} \ \phi_2 & \text{Disjunction} \\
 & \mid & \texttt{NOT} \ \phi & \text{Negation} \\
 & \mid & @l & \text{Shape label reference for } l \in \Lambda \\
 & \mid & \text{␣} \xrightarrow{p} @l\{\text{n},*\} & n \text{ or more arcs with predicate } p \\
 & & & \text{that conform to the shape identified by } l
\end{array}
$$

SHACL semantics

$$IRI \frac{n \in \mathcal{I}}{\mathcal{G}, n, \tau \vDash \text{IRI}} \qquad BNode \frac{n \in \mathcal{B}}{\mathcal{G}, n, \tau \vDash \text{BNode}}$$

$$Cond \frac{cond(n) = true}{\mathcal{G}, n, \tau \vDash cond}$$

$$AND \frac{\mathcal{G}, n, \tau \vDash \phi_1 \qquad \mathcal{G}, n, \tau \vDash \phi_2}{\mathcal{G}, n, \tau \vDash \phi_1 \texttt{ AND } \phi_2}$$

$$OR_1 \frac{\mathcal{G}, n, \tau \vDash \phi_1}{\mathcal{G}, n, \tau \vDash \phi_1 \texttt{ OR } \phi_2} \qquad OR_2 \frac{\mathcal{G}, n, \tau \vDash \phi_2}{\mathcal{G}, n, \tau \vDash \phi_1 \texttt{ OR } \phi_2}$$

$$ShapeRef \frac{\delta(l) = se \qquad \mathcal{G}, n, \tau \vDash se}{\mathcal{G}, n, \tau \vDash @l} \qquad Property \frac{|\langle n, p, n' \rangle \in \mathcal{G} \text{ such that } \mathcal{G}, n', \tau \Vdash @l| \geq n}{\mathcal{G}, n, \tau \vDash \llcorner \xrightarrow{p} @l\{n, *\}}$$

SHACL validation

Algorithm 1: SHACL validation - recursive algorithm with backtracking

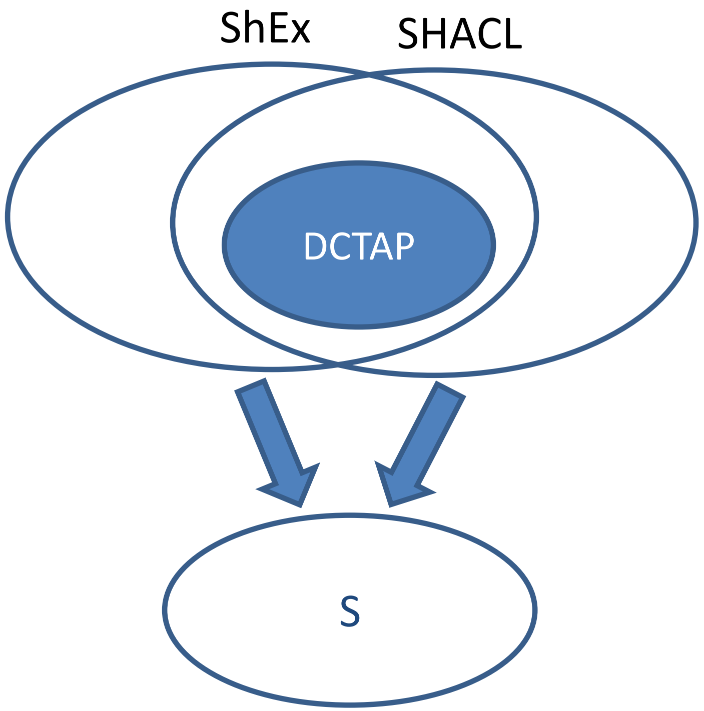
Input: A shape map typing t_s, a graph $\mathcal{G}$ and a schema $\mathcal{S}$
Output: A result shape map typing τ

def $check(t_s) = check(t_s, [])$
def $check(t_s, \tau) =$ **match** t_s
  case $[] \Rightarrow \tau$
  case $n@l : r_s \Rightarrow checkNodeLabel(n, l, \tau) \uplus check(r_s, \tau)$
def $checkNodeLabel(n, l, \tau) = checkNodeSe(n, l, @l, n@l : \tau)$
def $checkNode(n, l, \phi, \tau) =$ **match** ϕ
  case ϕ_1 `AND` $\phi_2 \Rightarrow checkNode(n, l, \phi_1) \uplus checkNode(n, l, \phi_2)$
  case ϕ_1 `OR` $\phi_2 \Rightarrow checkNode(n, l, \phi_1) \parallel checkNode(n, l, \phi_2)$
  case `NOT` $\phi \Rightarrow$ **if** $n@l \in checkNode(n, l, \phi, \tau)$ **then** $remove(n, l, \tau)$ **else** τ
  case $@l \Rightarrow$ **if** $n@l \in \tau$ **then** τ **else** **if** $n@!l \in \tau$ **then** $\mathbb{E}$ **else** $checkNode(n, l, \delta(l), \tau)$

  case $\_ \xrightarrow{p} @l'\{m, *\} \Rightarrow$
    $count = 0$
    foreach n' *such that* $\langle n, p, n' \rangle \in \mathcal{G}$
      if $n'@l'$ *OR* $checkNode(n', l, @l, \tau) \neq \mathbb{E}$ **then** $count{+}{+}$
    if $count \geq m$ **then** τ **else** $remove(n, l, \tau)$
  case IRI $\Rightarrow$ **if** $n \in \mathcal{I}$ **then** τ **else** $remove(n, l, \tau)$
  case BNode $\Rightarrow$ **if** $n \in \mathcal{B}$ **then** τ **else** $remove(n, l, \tau)$
  case cond $\Rightarrow$ **if** $cond(n) = true$ **then** τ **else** $remove(n, l, \tau)$
def $remove(n, l, \tau) = n@!l : (\tau \setminus n@l)$

Foundations ShEx/SHACL

ShEx language [Prud'hommeaux, 14]
ShEx Complexity [Staworko, 15]
Negation but no recursion [SHACL Rec.]
Stratification (ShEx) [Boneva 17]
Recursion + negation for SHACL [Corman 18]
Converting between ShEx and SHACL [SHaclEx]
Common language S [Labra 19]
DCTAP [https://dcmi.github.io/dctap/]
Stable model semantics SHACL [Andresel 20]



References

[Prud'hommeaux 14] Eric Prud'hommeaux, José Emilio Labra Gayo, Harold R. Solbrig: Shape expressions: an RDF validation and transformation language. SEMANTICS 2014: 32-40

[Staworko 15] Slawek Staworko, Iovka Boneva, José Emilio Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, Harold R. Solbrig: Complexity and Expressiveness of ShEx for RDF. ICDT 2015: 195-211

[Boneva 17] Iovka Boneva, José Emilio Labra Gayo, Eric G. Prud'hommeaux: Semantics and Validation of Shapes Schemas for RDF. International Semantic Web Conference (1) 2017: 104-120

[Corman 18] Julien Corman, Juan L. Reutter, Ognjen Savkovic: Semantics and Validation of Recursive SHACL. International Semantic Web Conference (1) 2018: 318-336

[Labra 19] Labra G. J.E., García-González H., Fernández-Alvarez D., Prud'hommeaux E. (2019) Challenges in RDF Validation. Current Trends in Semantic Web Technologies: Theory and Practice. Studies in Computational Intelligence, vol 815. Springer, Cham. http://doi-org-443.webvpn.fjmu.edu.cn/10.1007/978-3-030-06149-4_6

[Andresel 20] Medina Andresel, Julien Corman, Magdalena Ortiz, Juan L. Reutter, Ognjen Savkovic, Mantas Simkus: Stable Model Semantics for Recursive SHACL. WWW 2020: 1570-1580

Further info

Further reading:

- Validating RDF data, chapter 7. http://book.validatingrdf.com/bookHtml013.html

Other resources:

- SHACL WG wiki: https://www.w3.org/2014/data-shapes/wiki/SHACL-ShEx-Comparison
- Phd Thesis: Thomas Hartmann, Validation framework of RDF-based constraint languages. 2016, https://publikationen.bibliothek.kit.edu/1000056458

Acknowledgments

Irene Polikoff provided feedback on a previous version of these slides

Vladimit Alexiev helped with this list of ShEX/SHACL implementations:

https://github.com/validatingrdf/validatingrdf.github.io/wiki/Updated-list-of-implementations

End

This presentation was part:

http://www.validatingrdf.com/tutorial/iswc2024/