Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

A ShExML Perspective on Mapping Challenges

Already Solved Ones, Language Modifications and Future Required Actions

Herminio García González - garciaherminio@uniovi.es - @herminio_gg

Introduction

What are the mapping challenges?

- Declarative mapping rules
  - Single representation for heterogeneous data sources
    - Modifiability, shareability, efficiency
- However, some mappings are still not reachable
  - Defined and agreed by the community
  - Need to be solved

Mapping challenges

Summary

- 9 mapping challenges
  - Inputs defined by the community to test different scenarios
- ShExML mapping language
  - 5 challenges completely solved
  - 2 partially covered
  - 2 unaddressed

Brief ShExML Introduction

ShExML language overview

A flash introduction

ShExML language overview

A flash introduction

Prefixes 

```
PREFIX : <http://example.com/>
```

ShExML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

```
PREFIX : <http://example.com/>
SOURCE films_xml_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.xml>
SOURCE films_json_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json>
```

ShExML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

Iterators + Fields (extract content from input files using queries)

```
PREFIX : <http://example.com/>
SOURCE films_xml_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.xml>
SOURCE films_json_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json>
ITERATOR film_xml <xpath: //film> {
    FIELD id <@id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <directors/director>
}
ITERATOR film_json <jsonpath: $.films[*]> {
    FIELD id <id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <director>
}
```

ShExML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

Iterators + Fields (extract content from input files using queries)

Expressions to be applied to iterators or to actual values (result will be, accordingly, an iterator or a value)

```
PREFIX : <http://example.com/>
SOURCE films_xml_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.xml>
SOURCE films_json_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json>
ITERATOR film_xml <xpath: //film> {
    FIELD id <@id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <directors/director>
}
ITERATOR film_json <jsonpath: $.films[*]> {
    FIELD id <id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <director>
}
EXPRESSION films <films_xml_file.film_xml UNION films_json_file.film_json>
```

ShExML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

Iterators + Fields (extract content from input files using queries)

Expressions to be applied to iterators or to actual values (result will be, accordingly, an iterator or a value)

Shapes (give form to the output graph)

```
PREFIX : <http://example.com/>
SOURCE films_xml_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.xml>
SOURCE films_json_file <https://rawgit.com/herminiogg/ShExML/master/src/test/resources/films.json>
ITERATOR film_xml <xpath: //film> {
    FIELD id <@id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <directors/director>
}
ITERATOR film_json <jsonpath: $.films[*]> {
    FIELD id <id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <director>
}
EXPRESSION films <films_xml_file.film_xml UNION films_json_file.film_json>

:Films :[films.id] {
    :name [films.name] ;
    :year [films.year] ;
    :country [films.country] ;
    :director [films.directors] ;
}
```

ShExML iteration model

How it works?

- Iterator (tells the engine to traverse all results)
  - Fields (actual value/s to be outputted)
  - Nested iterators (to traverse down in the hierarchy)
- films_xml.actors.name
  - //film[i]/cast/actor[j]/name
- No need to join

```
ITERATOR film_xml <xpath: //film> {
    FIELD id <@id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <crew/directors/director>
    FIELD screenwritters <crew//screenwritter>
    FIELD music <crew/music>
    FIELD photography <crew/photography>
    ITERATOR actors <cast/actor> {
        FIELD name <name>
        FIELD role <role>
        FIELD film <../../@id>
    }
    ITERATOR actresses <cast/actress> {
        FIELD name <name>
        FIELD role <role>
        FIELD film <../../@id>
    }
}
```

Addressed Challenges

Datatype map

The problem

- Generate datatype from input data
- Dynamic vs static
- Different possible format inputs
  - http://www.w3.org/2001/XMLSchema#integer
  - xsd:integer
  - integer
  - int


```
{
  "persons": [
    {
      "firstname": "John",
      "lastname": "Doe",
      "lang": "en",
      "num": 3,
      "dt": "http://www.w3.org/2001/XMLSchema#integer"
    },
    {
      "firstname": "Jane",
      "lastname": "Smith",
      "lang": "fr",
      "num": "3.14",
      "dt": "http://www.w3.org/2001/XMLSchema#decimal"
    }
  ]
}
```

Datatype map

ShExML old syntax

- ShExML allowed to generate static datatypes
- It does not solve the previous case
- We should generalise the existing syntax for dynamic generation

```
ex:Person exPerson:[person.firstname] {
    ex:num [person.num] xsd:integer ;
}
```

```
exPerson:Jane  ex:num  "3.14"^^xsd:integer .

exPerson:John  ex:num  3 .
```

Datatype map

ShExML new syntax

```
ex:Person exPerson:[person.firstname] {
    ex:num [person.num] xsd:integer ;
}
```

```
ex:Person exPerson:[person.firstname] {
    ex:num [person.num] xsd:[person.dt] ;
}
```

```
exPerson:Jane  ex:num  3.14 .

exPerson:John  ex:num  3 .
```

- Expand the content generation expression but for datatypes
- It allows different types representations
- Prefixed or not
- Works like subject/object generation expressions but for datatypes

Language map

The problem

- Generate language tag from input data
- Dynamic vs static
- Different possible format inputs
  - en
  - English


```
{
  "persons": [
    {
      "firstname": "John",
      "lastname": "Doe",
      "lang": "en",
      "num": 3,
      "dt": "http://www.w3.org/2001/XMLSchema#integer"
    },
    {
      "firstname": "Jane",
      "lastname": "Smith",
      "lang": "fr",
      "num": "3.14",
      "dt": "http://www.w3.org/2001/XMLSchema#decimal"
    }
  ]
}
```

Language map

ShExML old syntax

- ShExML allowed to generate static language tags
- It does not solve the previous case
- We should generalise the existing syntax for dynamic generation

```
ex:Person exPerson:[person.firstname] {
    ex:lastName [person.lastname] @en ;
}
```

```
exPerson:Jane  ex:lastName  "Smith"@en .

exPerson:John  ex:lastName  "Doe"@en .
```

Language map

ShExML new syntax

```
ex:Person exPerson:[person.firstname] {
    ex:lastName [person.lastname] @en ;
}
```

- Expand the content generation expression but for datatypes
- It allows different types representations
  - We can use MATCHERS to generate BCP47 conformant tags
- Works like subject/object generation expressions but for datatypes

```
ex:Person exPerson:[person.firstname] {
    ex:lastName [person.lastname] @[person.lang] ;
}
```

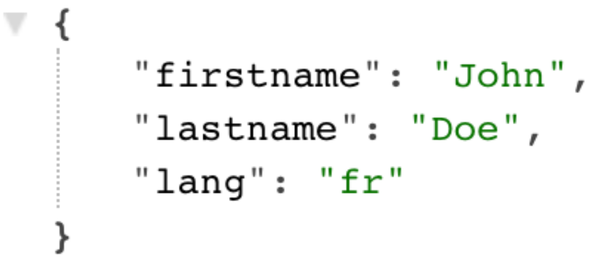
```
exPerson:Jane  ex:lastName  "Smith"@fr .

exPerson:John  ex:lastName  "Doe"@en .
```

Generate multiple values

The problem

- Multi-language or multi-datatype values for the same subject
- Additionally, default languages or datatypes

```
exPerson:John  ex:name  "John"@fr , "John"@en .
```



Generate multiple values

ShExML solution

- Multi-value generation bounds language and datatype generation to the current index
- Using old syntax we can generate additional default triples

```
ex:Person exPerson:[person.lastname] {
    ex:name [person.firstname.label] @[person.firstname.lang] ;
}
```

```
exPerson:John  ex:name  "John"@fr , "John"@en .
```

```
ex:Person exPerson:[person.firstname] {
    ex:name [person.firstname] @en ;
    ex:name [person.firstname] @[person.lang] ;
}
```

Join on literal

The problem

- Joins, by default, generate resources
- There is no way to output literals instead

```
experson:2  :affiliation  "Uni2" ;
            :lastName     "Dane" .

experson:1  :affiliation  "Uni1" ;
            :lastName     "Doe" .
```


```
{
  "author": [
    {
      "id": 1,
      "firstname": "John",
      "affiliation": "Uni1"
    },
    {
      "id": 2,
      "firstname": "Jane",
      "affiliation": "Uni2"
    }
  ],
  "people": [
    {
      "firstname": "John",
      "familyName": "Doe"
    },
    {
      "firstname": "Jane",
      "familyName": "Dane"
    }
  ]
}
```

Join on literal

ShExML solution

- ShExML allows (from its inception) to output resources and literals on joins
- Separation of concerns
  - How to extract and transform data
  - How to output data

Extract/ transform

```
PREFIX : <http://example.com/>
PREFIX experson: <http://example.com/person/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>
SOURCE jsonfile <https://raw.githubusercontent.com/kg-construct/mapping-challenges/2aac9680cd731fd647abd33d44a7f400e4278cf3/challenges/join-on-literal/input-1/input.json>
ITERATOR author <jsonpath: $.author[*]> {
    FIELD id <id>
    FIELD firstname <firstname>
    FIELD affiliation <affiliation>
}
ITERATOR people <jsonpath: $.people[*]> {
    FIELD firstname <firstname>
    FIELD familyname <familyName>
}
EXPRESSION authors <jsonfile.author UNION jsonfile.people>
EXPRESSION familyName <jsonfile.people.familyname UNION jsonfile.author.firstname JOIN jsonfile.people.firstname>
```

Output

```
:Author experson:[authors.id] {
    :affiliation [authors.affiliation] ;
    :lastName [familyName] ;
}
```

Multi-value references

The problem

- How to deal with the expected output in a hierarchical file
- Cartesian product or respect the current relation
- Join condition poses problems in JSON files as it is not possible to go upwards


```
{
  "labName": "AmazingLab1",
  "articles": [
    {
      "title": "article1",
      "authors": [
        {
          "name": "Alice",
          "affiliation": [
            {
              "label": "Uni1"
            },
            {
              "label": "Company2"
            }
          ]
        },
        {
          "name": "Bob",
          "affiliation": [
            {
              "label": "Uni3"
            },
            {
              "label": "Company4"
            }
          ]
        }
      ]
    },
    { … } // 2 items
  ]
}
```

Multi-value references

ShExML solution

- ShExML allows to nest iterators
  - No need for join condition
  - More usable
- Thus, verbatim translation

```
ITERATOR lab <jsonpath: $> {
    FIELD labName <labName>
    ITERATOR articles <articles[*]> {
        FIELD title <title>
        ITERATOR authors <authors[*]> {
            FIELD name <name>
          ITERATOR affiliation <affiliation[*]> {
              FIELD label <label>
          }
        }
    }
}
```

Access fields outside iterators

The problem

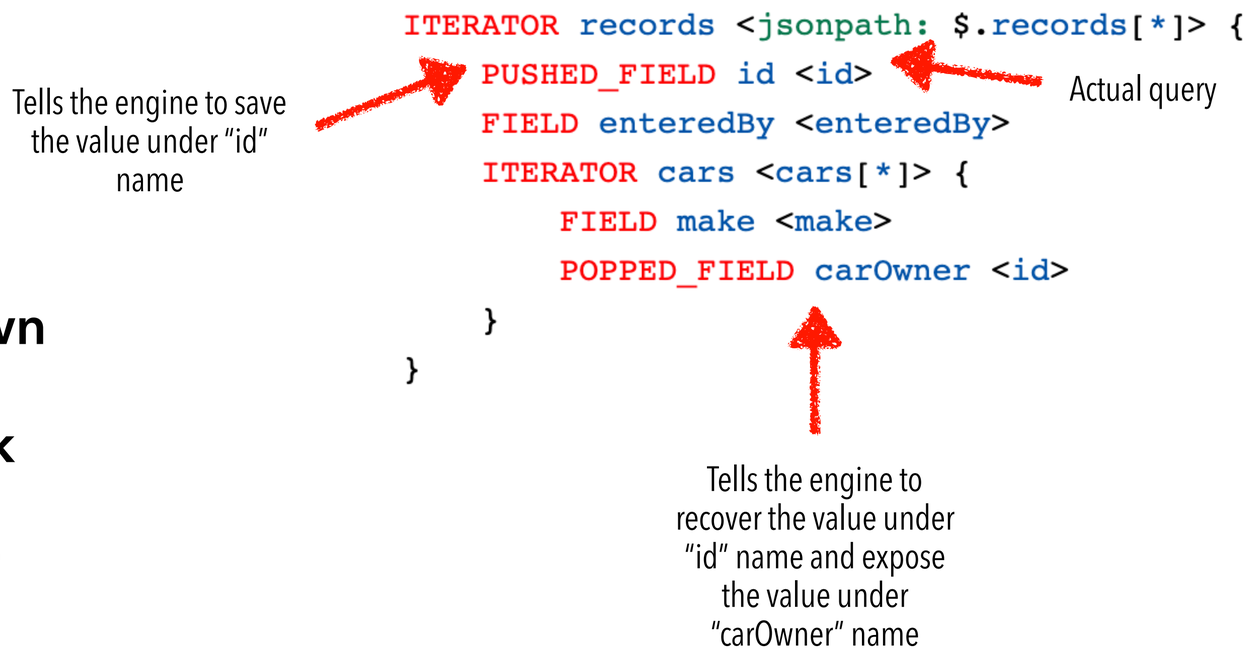
- How to access upper fields
- JSON path doesn’t allow going upwards
- From cars how to reach owners?


```
{
  "records": [
    {
      "id": "1",
      "enteredBy": "Alice",
      "cars": [
        {
          "make": "Mercedes"
        },
        {
          "make": "Honda"
        }
      ]
    },
    {
      "id": "2",
      "enteredBy": "Bob",
      "cars": [
        {
          "make": "Mercedes"
        },
        {
          "make": "Toyota"
        }
      ]
    }
  ]
}
```

Access fields outside iterators

ShExML solution

- Pushed fields allow to save value information for later use
- Popped fields allow to recover values from pushed fields
- Inspired in xR2RML's xrr:pushDown
- Pushed and popped like in a stack
  - But popped is not gone forever



Access fields outside iterators

Explicit vs implicit

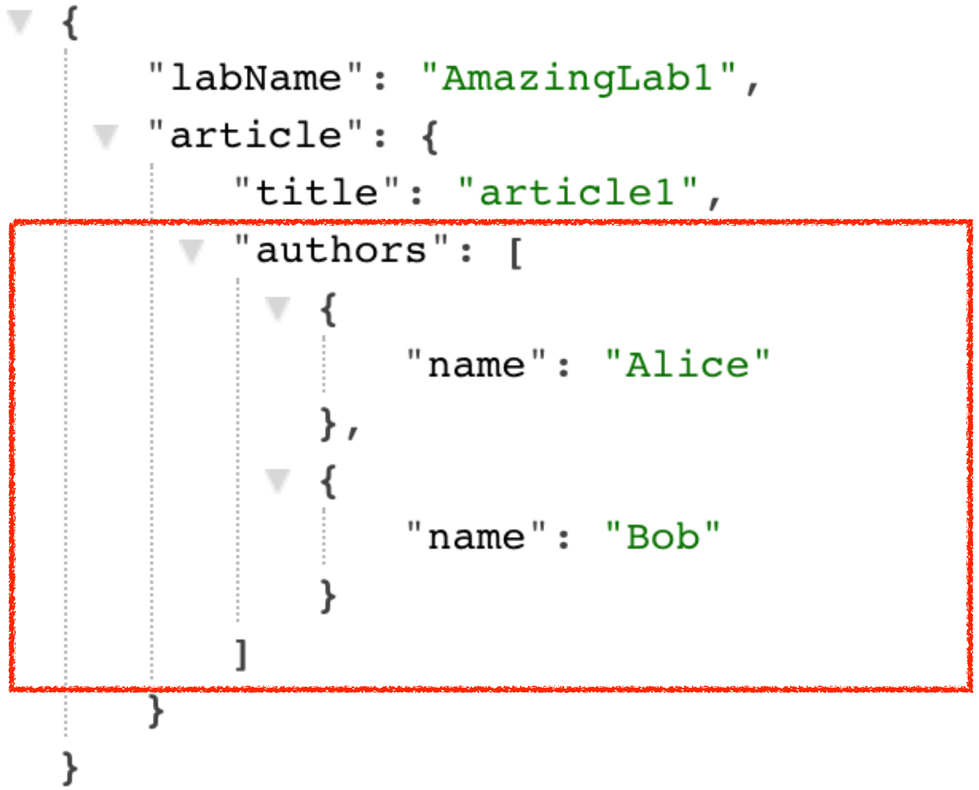
- Delva T. et al. [1] proposed an algorithm that saves iteration information
  - No need to push and pop fields, transparent to the user
  - But, it saves a lot of information -> Possible bottlenecks
- Further challenge, quantify the best option in terms of:
  - Usability
  - Performance

[1] Delva, T., Van Assche, D., Heyvaert, P., De Meester, B., & Dimou, A. (2021). Integrating nested data into knowledge graphs with RML fields.

RDF Collections

The problem

- Generate collections from multi-value references
- Different RDF Collections and Containers
  - List
  - Bag, Seq, Alt



RDF Collections

ShExML solution

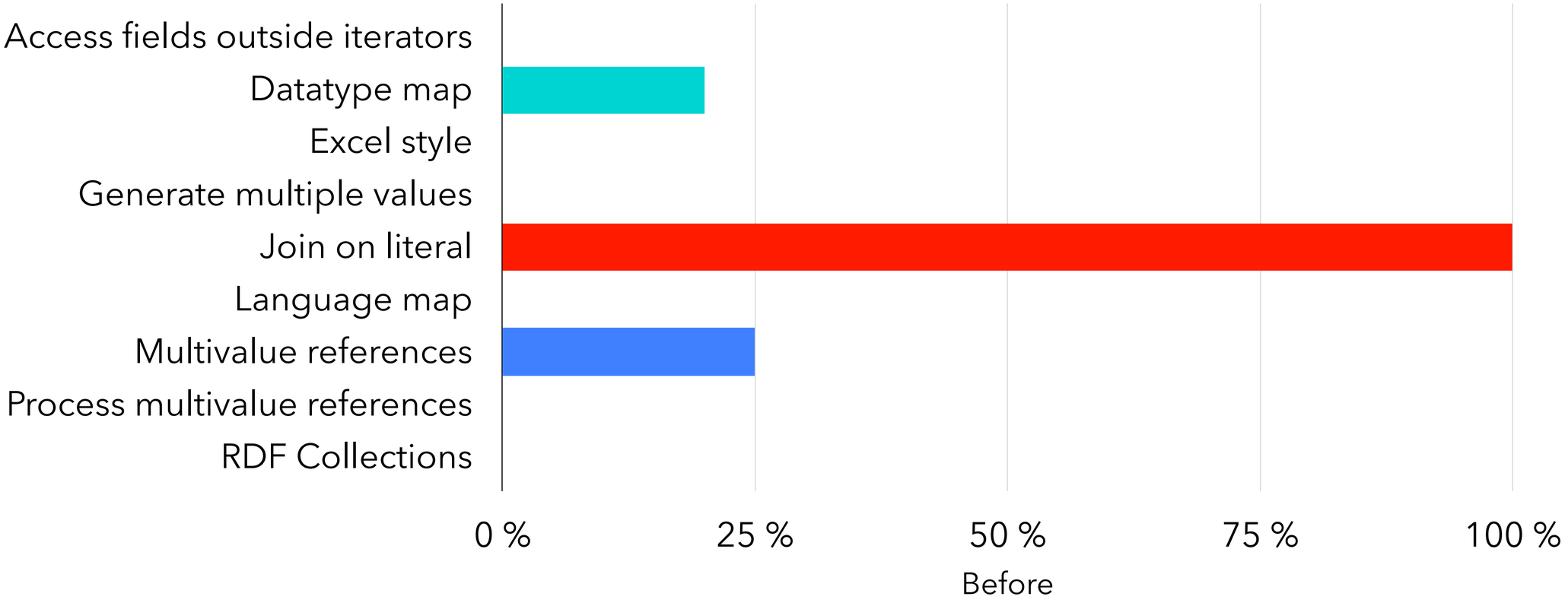
- Indicate the collection in the generation expression
- AS + (RDFList, RDFBag, RDFSeq, RDFAlt)

```
exArticle:article1  a  ex:Article ;
        ex:hasAuthors  ( exAuthor:Alice exAuthor:Bob ) .
```

```
ex:Article exArticle:[labValues.articles.title] {
    a ex:Article ;
    ex:hasAuthors exAuthor:[labValues.articles.authors.name AS RDFList] ;
}
```

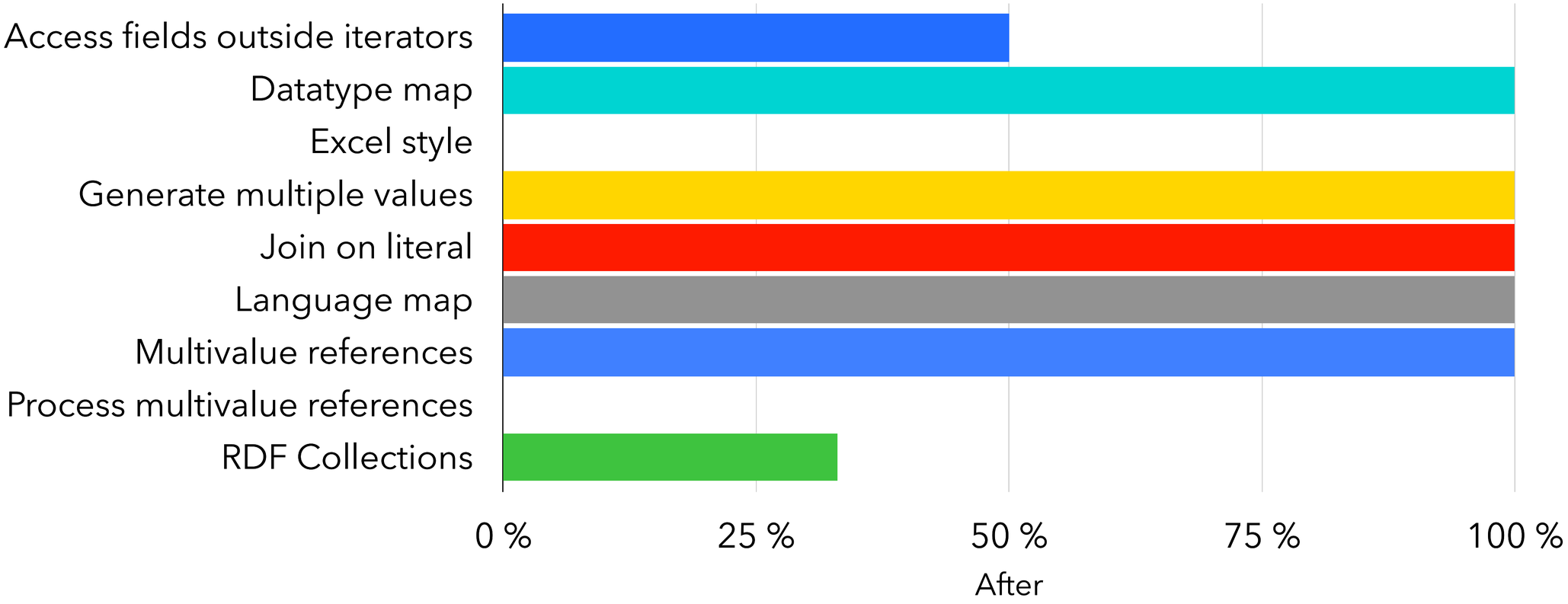
Addressed challenges

Coverage summary



Addressed challenges

Coverage summary



Unaddressed Challenges

And How They Could Be Addressed

Excel style

Possible solution A

- How to process Excel sheets values and styles?
  - Preprocess and add more columns with style
  - Input as CSV

No need to change the language and the engine

Need to preprocess (not ideal for users)

Excel style

Possible solution B

- How to process Excel sheets values and styles?
  - Support Excel sheets directly
  - How to access styles?
    - Need for a specific query language

```
ITERATOR excel_iterator <perRow> {
    FIELD value <A1>
    FIELD style <A1:TextColor>
}
```

Easy and straightforward for users

Too much design and implementation for a working solution

Process multivalue references

Possible solution A

- How to add data transformation functions in ShExML?
  - Add support for FnO functions library
  - No need to develop a dedicated function infrastructure

All function infrastructure outside ShExML language and engine

More dependencies to users which need to learn a third-party environment
We lose control of this part

Process multivalue references

Possible solution B

- How to add data transformation functions in ShExML?
  - Allow to define inline functions
    - Extension mechanism as semantic actions in ShEx
  - All in the same environment

```
FUNCTION splitFunction <n => n.split(',')>

ex:Tag ex:[lab.articles.tag WITH splitFunction] {
  ex:label [lab.articles.tag WITH splitFunction] ;
}
```

No third-party dependencies
Higher flexibility
No need to learn another tool

Higher complexity due to the necessity to know about functional programming

Discussion and Conclusions

Discussion & Conclusions

- Added solutions try to maintain ShExML usability
  - Using similar syntax constructions
- Some challenges were already solved by ShExML
  - Separation of concerns can help to solve some of them
- Further challenges
  - Need to be carefully designed and included in the language
- First step on how the challenges can be solved
  - Solutions from other languages and joint discussion -> Unified solutions

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

A ShExML Perspective on Mapping Challenges

Already Solved Ones, Language Modifications and Future Required Actions

Herminio García González - garciaherminio@uniovi.es - @herminio_gg