Draft OAGi Interoperable Mapping Specification

OAGi Members

2021/11/17

1 Introduction

This document describes a data mapping language designed to serve as an *interoperable exchange form* for expressing the intent of many mapping needs. As an interoperable exchange form, it is intended that the language can be translated (by humans and machine) into mapping specification in other languages. For example, it should be possible to translate statements in the exchange form into mapping specification of commercial mapping tools.

The document is a draft and as of this writing (2021/11/17) is likely to be updated often. OAGi members can find the most recent version of this document in the *OAGi Mapping Specification Working Group* Confluence pages (under "Mapping Doc").

The mapping language described borrows predominantly from the JSONata mapping language [1], but also includes provisions for mapping to/from forms other than JSON. These forms include tables (e.g. Excel) XML, and networks of data. To support networks of data, the mapping language borrows ideas from the Object Management Group's Queries, Views and Transformation relational (QVT-r) [2] mapping language.

2 Quick start: example mapping tasks

In order to give you a sense of things, the next subsections describe the basics and how some common mapping challenges are addressed by the language.

2.1 Simple operations

2.2 Vectors and Iteration

2.2.1 Mapping over a vector

In the language iteration is typically performed implicitly in the sense that when the argument to an operation is a vector, the operation is applied to each element of the collection. The result returned is a vector of the result of applying the operator to each element. Thus in Figure 1, `.zipcode` is applied to the vector `$ADDRS` returning a vector `["20898", "07010-35445", "10878"]`.

Figure 1: Map over a vector of entities, collecting ZIP code.

```
(
  $ADDRS :=
     [{"name"     : "Peter",
       "street"   : "123 Mockingbird Lane",
       "zipcode"  : "20898"},

       {"name"    : "Bill",
        "street"  : "23 Main Street",
        "zipcode" : "07010-3545"},

       {"name"    : "Lisa",
        "street"  : "903 Forest Road",
        "zipcode" : "10878"}];

   $ADDRS.zipcode
)
```

In the case that the operator returns null for elements of the vector, no value is collected. For example, if "Bill" in the above didn't have a zipcode, `$ADDRS.zipcode` would have returned `["20898", "10878"]`. **That's JSONata behavior. I don't particularly like it. It would be better IMO to return 'null' and then if you'd like, filter out the nulls. Needs thought.**

2.2.2 Filtering a vector

Suppose, for some odd reason, we didn't like 9-digit Zip codes (what the US Postal Service calls "Zip+4") and that we simply want to exclude them from the collection. There are a few simple ways to do this. One of those is to append a test expression to the end of the previous expression. This "filtering" test expression needs to be enclosed in square brackets to indicate that it is filtering. `$ADDRS.zipcode[$match(/^\d[5,5]$/)]`. Regular expression syntax is arcane but what is shown is typical of JavaScript, Java and many other regex libraries. The above matches on strings that are exactly 5 number characters, thus excluding the Zip+4 Zip Codes.

2.2.3 Index variables

Index variables are things such as `i` that you find code and pseudocode such as `for i in [1..9] {...`. A goal of interoperable exchange is allow the expression of individual mapping statements detached from a larger program context. You find such things in "mapping tables", which describe individual mapping statements in the cells of a spreadsheet. With that in mind, index variables are discouraged by the design of the language; they'd add meandering procedural expression where concise declarative is desired. This is why the functional programming features known as *map*, *filter*, and *reduce* are used. That said, what if you really needed to enumerate the things you were working with, for example, to put an index number in front of them? You can do this with *map*, which in the language is similar to how it is implemented in JavaScript. See Figure 2 below.

Figure 2: Using an index variable.

```
$map($ADDRS.zipcode, function($v, $i) {'zip ' & ($i+1) & ' ' & $v})
```
returns

```
[ 'zip 1 20898', 'zip 2 07010-3545', 'zip 3 10878'].
```

The above could use some explaining. First, the mapping language uses JavaScript syntax of map, filter, and reduce (**but with $ in front of them like JSONata? Keep?**). In the above `function` introduces a function, and the following code in brackets is the body of the function, a single expression returned. In map, filter, and reduce, the function is called once each in with each member of the first argument (`$ADDRS.zipcode` here) in sequence. Those values (`[20898, 07010-3545, 10878]` in the above), are bound to the variable `$v` and `$i` is automatically bound to integers in the sequence `[0..3]`.

2.3 Working with code lists

A mapping tasks commonly needed is translating a code list, or subsetting one to a set of code values that actually is used in your business processes. For example, there is a very large set of codes for transportation events in the code list [**don't recall the spec**]. Suppose you are receiving messages with lots of these values but your inventory tracking application only wants to know whether the shipment is likely to be late. There is no getting around the fact that you need to explicitly describe a functional relationship to do this. The language defines a switch statement to do this kind of thing. An example is shown in Figure 3.

Figure 3: Mapping code values.

```
$Codes2OurCodes := function($theirs)
 {switch $theirs {
     #{2, 5, 52, 66} : 'late';
     #{3, 6, 77, 85} : 'on time';
     * : 'unknown'}
 };
```

The code in Figure 3 also shows how names are associated with functions. Functions in the language do not have names. If you need to use a function in multiple places, assign it to a variable as shown in the figure (`$Codes2OurCodes`).

2.4 Working with tabular data

Being able to read information from a spreadsheet is a very handy capability in mapping. Of course, Excel-like spreadsheets can have multiple sheets and the content can be non-uniform, including merged cells and formula. The language does not provide means to deal with these complexities. However simple tables with no surprises and a header row naming columns can easily be viewed as a vector of map structures. For example, Table 1 can be viewed as the structure shown in Figure 4.

Table 1: A simple table oriented such that columns name properties.

Ship Date	**Item**	**Qty**	**Unit Price**
6/15/21	Widget 123	1	$10.50
6/15/21	Gadget 234	2	$12.80
6/15/21	Foobar 344	1	$100.00

Figure 4: Table 1 viewed as a vector of map structures.

```
[{"Shipment_Date":  "2021-06-15", "Item":  "Widget 123", "Qty": 1.0, "Unit Price": 10.5},
 {:Shipment_Date":  "2021-06-15", "Item":  "Gadget 234", "Qty": 2.0, "Unit Price": 12.8},
 {:Shipment_Date":  "2021-06-15", "Item":  "Foobar 344", "Qty": 1.0, "Unit Price": 100.0}]
```

Though you might not need to know such things if your target usage is mapping tables, making reference to such a table could be achieve by a statement such as:

```
$readSpreadsheet("data/spreadsheets/ExampleInvoiceInfo.xlsx" "Sales Info")
```

where the first argument names an Excel file and the second names a sheet in that spreadsheet.

If the table is transposed (so that all the properties are in its first column), a third argument with value `true` can be specified.

2.5 Working with relational data

[This section will introduce provisions for expressing maps to/from relational DBs, where schema are assumed.]

Everything to this point is similar to what can be done with JSONata; working with relational data, and exchange forms like EDI requires more functionality. The language borrows from QVT-r to achieve these requirements.

3 Specification

[This section will include a feature-by-feature description of the language similar to a language reference manual. Right now I've just listed the functions we think are needed.]

3.1 Table of elementary language features

Task	**Example**
Access (or navigate)	Address.PhoneNumber
Select from array (simple)	[2]
Select from array (from end)	[-2]
Select all from array	(done implicitly)
Select a subset of array	[[0..2]]
Entire input document	$
Simple query	Phone[type='mobile']
Select values of all fields	Address.*
Select value from any child	*.Postcode
Select arbitrarily deep	**.Postcode
Concat	'a' & 'b'
Usual numeric operators	+,-,*,/
Array constructors	
Object Constructors	Phone{type: number}
Object Constructors (force array)	Phone{type: number[]}
Code Block	(exp1; exp2; exp3)
Value as key in result	Account.Order.Product{'Product Name': Price}
Map	seq.exp
Filter	seq[exp]
Reduce (group and aggregate)	seq{exp:exp, exp:expâĂę}
Sort	seq ˆ(exp)
Index	seq # $var
Join	seq @ $var
Functions (in-line)	function($x) : {(âĂę)};
Naming functions	$myfunc = function$(x)âĂę
Context variable	$
Root context variable	
Conditional expression	pred ? exp : exp
Variable binding	$my_var := "value"
Function chaining	value ∼> $f2-$ >f3
Regular expressions (like JS)	/ab123/
Time	*now*(),millis()
Transform	head ∼>— location — update [,delete] —
Example variable binding	[needs investigation]
Type coercion	asType(obj, typespec)
Type checking	isTypeOf(obj, typespec)
Coercion to integer	toInteger(x)
Coercion to real	toReal(x)
Returning a substring	substring()

3.2 Table of language features for use with collections

Task	**Example**
Check for inclusion	$includes(col, x)
Check for exclusion	$excludes(col, x)
Size of a collection	$size(x)
Count elements	$count(col, obj)
sum of numbers	$sum(x)
Cartesian product	$product(x, y)
union of collections	$union(x, y)
symmetric difference of sets	$symmetricDifference(x, y)
coerce to ordered set	$asOrderedSet(col)
first element of ordered collection	$first(col)
last element of ordered collection	$last(col)
intersection of collections	$intersection(x,y)
find one	$any(col, predicate)
check for presence (return T/F)	$one(col, predicate)
filter collection	$filter(col, predicate)
sort collection	$sortedBy(predicate)
check every (return T/F)	$forAll(predicate)
add value to collection	$including(col, val)
flatten collection of collections	$flatten(x)

References

[1] Jsonata.org. JSONata: query and transformation language, 2021.

[2] Object Management Group. MOF Query/View/Transformation. Technical report, Object Management Group, Needham, MA, 2016.