Navigating and Updating Trees of Maps and Arrays

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper describes new features proposed for the 4.0 versions of XSLT, XPath, and XQuery, designed to make it easier and more efficient to query and update trees of maps and arrays, such as those typically derived by parsing JSON.

1. Terminology

Before we start, let's introduce some terminology.

- I'll use the term QT3 to refer to the family of specifications that includes XSLT 3.0, XPath 3.1, and XQuery 3.1 (all published in 2017); and I'll use QT4 to refer to the proposed 4.0 versions of these specifications which can be found at https://qt4cg.org/
- I'll use the term JTree to refer to an XDM data structure comprising a tree of maps and arrays; and I'll refer to the nodes in this tree as JNodes. This terminology isn't in the spec, but I find it convenient. The "J" stands for JSON, because these trees often originate from JSON data files, but it's important to remember that the XDM model for maps and arrays is much more liberal than JSON: for example, keys in a map can be any atomic type, not just strings.
- In XDM, the term "node" means a construct like an element, attribute, or text node in a tree that represents an XML document; and "tree" means a tree of such nodes. I won't be talking much about XML in this paper, so when I do, I will call these things "XML trees" and "XML nodes" to avoid any confusion.

2. What problem are we trying to solve?

One of the major new features in QT3 was support for maps and arrays as new data types. There were two main motivations for this. One was to extend the XDM data model with data structures that provided native representation of data found in JSON files: although we had no ambition to treat XML and JSON as equal citizens, we recognized that real-life applications had to be able to mix and match data from different sources. The second motivation was that users were

increasingly finding that representing everything (including transient working data) as XML could be cumbersome and inefficient.

Maps and arrays proved a very successful addition to the QT3 data model, but experience has shown that they are not yet as well supported by language features as XML node trees, and a significant part of the QT4 programme is aimed at filling the gaps.

At the same time, we're trying to come up with language features that exploit the intrinsic potential of maps and arrays to deliver performance benefits. In standards and specifications, the emphasis is on defining language features; performance objectives do not feature directly, but of course the design of language features is informed by the experience of implementors and users.

3. Previous work

Over the years I have written a number of applications that stretched the boundaries of what could be achieved with XSLT, and the challenges these posed are documented in a number of previous conference papers.

Back in 2007 [2] I experimented with using XSLT to write a query optimizer. I reckoned that implementing optimization rewrite rules is essentially a rule-based tree-to-tree transformation task, and in principle is therefore an ideal fit for XSLT. But my experiments suggested that doing tree rewrites in XSLT would be about 12 times slower than doing them in Java, which made the idea infeasible. But it set me thinking about why XSLT should be slower, and what could be done to fix it.

One potential way of avoiding the costs of transforming XML trees in such applications is to represent internal data using maps and arrays instead. So in 2016 [3] I explored the potential for using the QT3 facilities for transforming JSON. Although the XSLT 3.0 spec wasn't quite finished yet, development was on the home run and there wasn't really much scope for making further changes. That means the analysis in that paper of how to tackle JSON tree transformations still represents the state of the art; and it's clear that even simple transformations can be quite difficult. The conclusion of that paper was that there were significant limitations in the QT3 specifications for manipulating maps and arrays, and that indeed, the most practical way to implement several apparently-simple use cases was to convert the data to XML, transform it as XML, and then convert back to JSON.

At XML Prague in 2018 [4] I wrote about the problem of copying XML trees. In particular, I explored whether we could make tree copying faster by using a tree implementation that allowed sharing of subtrees. Node identities and parent pointers would be created on-the-fly during downwards navigation. The results were disappointing: while one particular operation (grafting a subtree) was dramatically faster, everything else (including all-important navigation operations)

slowed down a little, meaning there was no bottom-line benefit. But there's also an underlying problem: given a typical XSLT recursive-descent transformation, you can't tell when there's a particular subtree that isn't going to change, and that can therefore benefit from a fast copy operation. That's particularly true because even when you think a subtree is being copied unchanged, there's usually some hidden effect on the in-scope namespaces.

When I presented this paper, someone in the audience pointed out very politely that I had re-invented a technique well known in functional programming circles: zipper data structures, attributed to Gerard Huet [1].

At Markup UK in 2018 [5] I described the project that motivated some of this thinking: the task of writing an XSD schema validator using XSLT 3.0. This project (which was never finished to release quality) in fact made heavy use of maps and arrays as its internal data structures. (The use of maps here was nothing to do with JSON. It was primarily to enable a single pass over input XML trees to return multiple results: for example multiple ID values found in the tree, or multiple validation errors). The experience on this project led to the design of record types as found in the QT4 drafts, to enable stronger typing of maps used to hold heterogenous data; it also led to performance improvements in the way that maps are implemented in Saxon.

The following year, back in Prague [6], John Lumley and I described a project to implement an XSLT compiler in XSLT. The reason we used XSLT is that we needed a compiler that would run in the browser, and writing it in Javascript was too horrible to contemplate; in addition it seemed logical that since compilers are all about multiphase tree transformations, and that's what XSLT is supposed to be good at, it ought to be a natural fit. In any case, writing a compiler in its own language is always considered to be good for a software engineer's soul. But there were performance issues to tackle: issues already highlighted in the 2018 paper on efficient copy operations.

The paper contains a very detailed account of the techiques we used to get the performance up to a level where it was only three times slower than the existing Java compiler. Two of the significant factors were the need to represent complex values (such as data types) as strings so that they could be held in XML attributes, and the cost of copying XML trees, which turned out to be significantly caused by the complexity of getting the namespaces right. Unlike some of the previous projects described, this one did in fact complete, and resulted in the XSLT compiler that we use today in the SaxonJS project: though we did end up writing some critical parts (notably the XPath parser) in Javascript.

Most recently, my paper at Balisage in 2022 [8] covered similar ground to this paper, but the thinking has evolved since then so I feel that an update is overdue.

A theme that runs through a number of these papers is the dilemma of parent pointers. Should trees be implemented with parent pointers or not? Closely associated with this is the question of whether nodes should have a persistent, perma-

nent identity. As far as I'm aware, all tree models for XML have node identity and parent pointers, and all tree models for JSON don't. There's nothing intrinsic to XML or JSON that accounts for this difference, but it has become part of the culture. Without parent pointers and node identity, it's easy to share subtrees, which means that updates can be very efficient (you only need to copy the parts of the tree that actually change). With parent pointers, you end up copying the whole tree every time you want to make a small change. But without the ability to reference data higher up in the tree, some queries (especially on recursive data structures) become much harder to express. Perhaps the reason XML trees always have parent pointers is that XML is primarily designed for text processing, and textual data structures are intrinsically recursive.

It's worth pointing out that despite the limitations identified in this paper, maps and arrays in their QT3 form can be extremely useful. They play a key role, for example, in the Java-to-C# transpiler described in [7], where they are used in a way that doesn't encounter any of these problems.

4. The QT4 project

The project to create the 4.0 specifications was announced at XML Prague in 2020 [https://archive.xmlprague.cz/2020/files/xmlprague-2020-proceedings.pdf]; it took a while to build momentum but it is now in full swing, with well-attended Zoom meetings taking place weekly. To date 640 issues have been tabled, and several hundred of these have resulted in new features in the specifications. Thousands of test cases have been written, and both Saxon and BaseX have announced experimental implementations.

The project operates under the auspices of a W3C Community Group. As such, it receives some lightweight support and endorsement from the W3C organisation, but is largely free to do what it chooses. There is no requirement, for example, for specifications to go through milestones such as proposed and candidate recommendations, or for transitions in status to be approved by a formal vote among the wider W3C membership.

There is a great deal of new functionality in the QT4 specifications, much of it associated with support for maps and arrays, but in this paper I am going to concentrate on three aspects: recursive query, point update, and rule-based transformation.

The adjective deep here means that we are looking at maps and arrays not just as individual data structures, but at their use in combination to represent more complex data sets, typified by the representation of a complete JSON document. I refer to these as JTrees.

5. Overview

In the following sections I'm going to present three areas where the QT4 specifications offer new language features: deep query, point update, and rule-based transformation.

What I desrcribe in the paper isn't necessarily identical with the current snapshot of the published specifications. In some cases there are proposals to change the current drafts that the community group is still working on. In some cases there is a broad consensus on the way forward, in other cases proposals have been put forward but not yet discussed, in other areas there have been discussions but alternatives are still on the table. So what I present here is a mix of what the current drafts actually say, and what I hope they will say in due course. It's very much a personal perspective. And with the best will in the world, the final specifications will have moved on from the current proposals. The process of agreeing language specifications may be slow, but it is fairly good and distinguishing what works and what doesn't, and at finding incremental improvements where they are needed.

6. Recursive Query

Let's look first at the question of recursive query.

The QT3 specifications offer the lookup operator `?` as a rough equivalent of the path operator `/` used for navigation in XML trees. It can be used both for selection by key within a map, and for selection by subscript within an array. Here is a simple example. Consider the example document used in the JSONPath specification, representing four books and a red bicycle:

```
{
  "store": {
    "book": [
      {
        "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      {
        "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      {
        "category": "fiction",
        "author": "Herman Melville",
```

```
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      {
        "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 399
    }
  }
}
```

It is possible to find the average price of books by Tolkien with the query:

```
$data ?store ?book ?[ ?author = "J. R. R. Tolkien" ] ?price => avg()
```

Most of this is QT3 syntax, but there's one new QT4 construct here, namely the array filter expression `ARRAY?[PREDICATE]`. The takes an array, and filters it to retain the array members that match the predicate, in exactly the same way that XPath filter expressions have always been used to filter sequences.

To achieve the same effect in QT3, it is necessary to convert the array to a sequence by writing `book ? * [ ?author = "J. R. R. Tolkien" ]`. This works fine in this case where all the members of the array are single items (in this case, single maps), but is not useful in the general case where array members are arbitrary sequences: in such cases the QT3 solution is the higher-order `array:filter` function.

The limitations of the lookup operator start to become apparent when the data becomes more complex.

Firstly, QT3 offers no equivalent to the `//` operator used for searching the descendant axis of XML trees. The `//` operator is useful for a number of reasons:

- It's a handy shorthand: it avoids having to spell out long and complex paths in detail.
- It's useful where the same structure can occur at different places in the tree, for example when searching the above structure for a price, which might be the price of a book or of a bicycle. (The sample data here is a joke, but unfortunately that's often also true of data found in the real world.)
- It's invaluable when handling recursive data such as an organisation chart, where the same structures occur at different levels.

Secondly, this path-based syntax can't be used for join queries. That's equally true of XML-based path expressions using `/`, and the solution is the same: FLWOR expressions. Let's suppose that we want to know (somewhat surreally) how many books there are in the store that cost less than a bicycle. We can write:

```
let $bicycle-cost := ?store ?bicycle ?price
return count( ?store ?book ?[ ?price lt $bicycle-cost ] )
```

This still leaves another usability problem: all selections have to be downwards. There's no equivalent to the sibling, parent, or ancestor axes used when navigating XML. Now, it's arguable that these axes are most useful when searching text, and are less needed for JSON because no sane person would use JSON for representing text. But even with structured data, they can be very handy, especially when the data is recursive. A classic query is to find everyone who earns more than their manager: `//person[salary > ../salary]`. Similarly, the sibling axes are useful when data is ordered.

The `map:find()` function was a late addition to the QT3 specifications which attempted to provide an equivalent to the `//` operator, but it has proved almost entirely useless. The reason is that it only returns the values, it tells you nothing about the context in which they were found. You can find all the values of `first-name` in your entire dataset, but what can you do with the knowledge that the names that appear are John, Jane, and Mary? Finding all the values of `employee` (which are likely to be maps) is a bit more promising, but without the ability to navigate up the tree to discover context (like the department or location of each employee) the data is still of very limited value.

So let's look at how lookup paths have been improved in the QT4 specifications. There are a number of changes, some quite minor, some more signficant. (In addition, I should emphasize, some of these are a solid part of the QT4 drafts, while others are proposals that are still being polished and refined.)

The most obvious change is that we've introduced the "deep lookup" variant, `A ?? B`. Like `//`, this recurses down through a JTree of maps and arrays to find its target. And like `//` (and like `map:find`), it returns a flattened sequence of items.

But there's a fundamental difference. When you return a flattened sequence of XML nodes, those nodes are located at a position within a structure, and you can get extra information about them: most commonly, you are selecting a sequence of elements, and from the elements you can get information about their attributes, their content, their siblings, their parents.

With `??`, all you get back are values. If you do `??name`, you get back a set of names. There's no context; no way of finding out any other information beyond the actual strings. If the values you get back are themselves maps, you can do a little bit better, you can drill down into the content of those maps. Remember, a map (unlike an XML element) doesn't even have a name. You might find it via a name (consider `store` and `bicycle` in the example above), but the name isn't part

of the map, it's not available as part of the information returned by the `??` operator.

The fact that maps and arrays (unlike XML elements) have no name, is rather fundamental. The expression `//event/*` selects elements that have names, and the names often serve to distinguish one event from another. The names are a property of the element. In JSON, names perform a subtly different role: they don't identify what kind of value you are dealing with, they identify its role in relation to a parent object, and an expression such as `??event?*` selects values without identifying what kind of object they are.

In JSON structures the best way to identify classes of value is often not by name, but by structure. If we want to process locations, we probably can't search for objects named `location`, we have to search for objects that have `longitude` and `latitude` properties. For that reason we've added type-based selectors to lookup expressions, so you can do `??type(record(longitude, latitude))` to find all the maps (objects) having `longitude` and `latitude` properties.

The fact that the lookup operators flatten their results brings additional problems. Suppose you have data like this:

```
{
  "readings": [
    { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}
    { "Week":2, "Mo": (4, 8), "Tu": (4, 5), "We": ... }
  ]
}
```

Now the result of `??Mo` is `(12, 16, 18, 4, 8)`, and the result of `??Tu` is `(4,5)` which is pretty meaningless - all the internal structure has been lost. We just have a set of numbers, and no idea where they came from.

So the next change we have made is to introduce modifiers, which enable you to return something other than the flattened result. For example, the result of `??entry::Mo` is a sequence of key-value pairs:

```
{"key": "Mo", "value": (12, 16, 18)},
{"key": "Mo", "value": (4, 8)}
```

This retains a lot more structure - it becomes possible to ask questions like *"On which Monday was the average reading at its highest?"*.

The syntax allows four modifiers: `entry::` returns the key-value pair as in the above example, `key::` returns the key (when selecting into an array, this is the numeric index), `value::` returns the value part, with each value wrapped as an array so that different values are kept separate, and `content::` returns the flattened sequence. For compatibility with QT3, `?x` is short for `?content::x`.

The biggest change, however, is that the results returned by the `?` and `??` operators are now labeled with their provenance. This means that you can now do queries like `??entry::*[?value?salary > ?parent()?salary]`.

How does that work?

The maps and arrays in a JTree don't have parent pointers, and we're not changing that. Parent pointers prevent a subtree being shared by two different trees, which is the major reason why copying subtrees in the XML model is so expensive. Instead, though, when we search for data by downwards navigation, we can remember how we reached the values that we found, and we can make this information available. This is essentially the idea behind zipper structures [1].

The idea of a zipper structure as that as you navigate into a structure, you keep a trail of where you have visited, so that you can back out. This effectively turns a one-way list into a two-way list, or a tree without parent pointers into one that allows upwards navigation; and it does so without making the structure mutable, or losing the properties that allow functional modifications (modifications that create a new list or tree without destroying the old one).

The solution to this in QT4 is taken straight from the zipper model. The result of a deep search (which can now be done using the deep lookup operator: `$data??person`) is a set of labeled values. The expression returns the required JNodes, but these carry an extra label identifying where they were found. The label is not part of the persistent data, it is additional information added by the search process, so the same value found by two different routes can carry different labels. We can immediately see how this allows common subtrees to be shared without the need for physical copying.

What information is available in the labels? We can describe this at two levels: an internal level that describes the data that is retained, and an external level that describes how this is exposed to user applications. At the internal level, there are two degrees of granularity associated with a downwards selection: selecting an entry with a given key (which in the case of an array means the array member at a given index), and selecting a specific item within the value part of the entry.

So, when selecting an entry (key-value pair) the information retained in the label consists of two things: the containing JNode (a map or array), and the key/index by which the entry was selected within that JNode. Of course the containing JNode will also have a label identifying its own provenance, so one can follow a chain of labels all the way to the root JNode of the query.

At the next level of granularity, when we select an individual item within a value, there is an additional piece of information: the integer position of the item within the value.

Looking back at the example where the query `??Mo` returned the sequence `(12, 16, 18, 4, 8)`, we can now see that these items carry hidden labels as follows:

- 12: `container: { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}, key: "Mo", position: 1`

- 16: `container: { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}, key: "Mo", position: 2`
- 18: `container: { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}, key: "Mo", position: 3`
- 4: `container: { "Week":2, "Mo": (4, 8), "Tu": (), "We": ...}, key: "Mo", position: 1`
- 8: `container: { "Week":2, "Mo": (4, 8), "Tu": (), "We": ...}, key: "Mo", position: 2`

There are two ways this data is currently exposed (this may change):

- Firstly, the function `selection-path` can be applied to any item. If the item has a label indicating how it was reached, the function returns a sequence of records, one for each step in the selection (in innermost-to-outermost order). The first record in the result, for the value 12 in the example above, is the record `{"key": "Mo", "position": 1, "container": { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}}` The second and subsequent records represent the selection path of the container, defined recursively. So if the `selection-path` function is called with the labeled value `12` as the argument, then (for example), `selection-path(12)?key` will return the sequence `("Mo", 1, "readings")`; reversing this sequence gives the sequence of keys needed to select the item from the root.
- Secondly, when the modifier `entry::` is used, the returned entries include functions that make the same information available, but in more digestible form. The proposed functions include `ancestors()`, which returns all the nested JNodes, innermost first, `parent()` which returns the same as `ancestors()[1]`, `root()` which returns `ancestors()[last()]`, and `ancestor-keys()` which returns the sequence of key values used to select the ancestors at each level.

7. Point Update

In my 2016 paper I presented two use cases for transformation of JSON trees, and I have continued to use these to test the adequacy of proposed new language features.

The first use case is what I call a "point update": *increase the price of all products tagged with the keyword "ice" by 10%*. I call it a "point update" because we identify the places in the tree structure than need to change, and then say how they should change. That requirement is very easily stated in English, and it's not very difficult to devise XQuery or XSLT syntax to express the requirement, for example in XQuery

```
update $root {
   replace ??product[?tag = "ice"]?price with  . * 1.1
}
```

or in XSLT

```
<xsl:update root="$root"
             replace="??product[?tag = 'ice']?price"
             with=". * 1.1"/>
```

In both cases the idea is that the instruction returns a new version of the tree rooted at `$root` (leaving the original intact) which differs from the original in that the selected prices have changed.

This is sometimes called *in-situ update*, but that's misleading, because existing data is not changed.

This is similar in concept to the kind of updating expressions available in the XQuery Update Facility (XQUF). However, it has far less complexity. XQUF essentially has two modes of operation:

- Pending Update Lists, where the updates are deferred until the query has finished execution, and the query itself has no opportunity to read the updated data;
- Copy-Modify operations, where the updates are performed on a copy of the data, leaving the original unchanged. But with XML (because of XML node identity and parent pointers) copying an XML tree typically takes time and memory proportional to the size of the tree.

The current proposed specification for this feature describes the semantics in terms of a recursive-descent rule based transformation, rather as if it were implemented in XSLT. The only complication is how to tell when you are processing a value selected by the expression `??product[?tag = 'ice']?price`. One way to do this would be to restrict the syntax of this expression to a pattern-like syntax so you can test each value in the tree when you get to it. But if you turn to the semantics of patterns in XSLT, the definition depends on being able to navigate upwards in the tree. Instead, the approach we have adopted is that you can use any selection expression you like, and the values you select are tagged with a temporary label, so that you can identify them during the recursive descent traversal, and process them accordingly.

The QT3 data model says:

> *This version of the XPath Data Model does not specify whether or not maps have any identity other than their values. No operations currently defined on maps depend on the notion of map identity. Other specifications, for example, the XQuery Update Facility, may make identity of maps explicit.*

That was a committee compromise. The WG knew that maintenance of identity is expensive in many ways: not only does create the need to make physical copies of

data that is otherwise unchanged during a transformation, it also complicates the semantics of the language because functions are no longer purely functional (if a function creates new node each time it is called, then optimizations such as loop lifting become invalid). But the WG at the time couldn't see how update operations could be defined without some notion of object identity.

The solution adopted in QT4 is to have a notion of identity, but one that is transient and exists only while an updating operation is in progress. Conceptually, the update starts by making a copy of the input in which all JNodes have identity; it then modifies this copy in-situ, and then strips off the identifiers.

The great thing is, however, that this complexity exists only in the formal semantics of the operation. As far as users are concerned, its fairly intuitive what's meant by

```
update $root {
     replace ??product[?tag = "ice"]?price with  . * 1.1
}
```

Performing this operation in QT3, whether in XQuery or XSLT, is surprisingly difficult, as I showed in my 2016 paper. I ended up converting the data to an XML tree, updating the XML tree, and then converting back to maps and arrays.

The implementation of this update expression in Saxon is very different from the formal semantics[1]. Rather than doing a top-down traversal of the whole tree structure (which would take time proportional to the size of the tree), Saxon dives straight in to the selected nodes and makes the changes locally. Internally, you have to find the affected nodes in the tree, make local copies of those nodes with modified properties, and then you have to work your way back up the tree making copies of the affected parent nodes so they point to the modified children rather than to the originals. This needs to take into account that when creating a modified parent node, you might need to incorporate multiple modified child nodes. Finally, when you've worked your way back up to creating a modified copy of the original root node, you can return that as the result of the expression.

So for this use case, we don't need to expose the zipper model to the user; but it needs to be there in the background; the route to each modified node in the tree needs to be retained so that the ancestor nodes can be reconstructed.

The second use case in my 2016 paper involved a hierachic inversion. Starting with a JSON dataset representing courses and the students attending each course, the requirement is to invert this to create a dataset organised first by student, listing the courses taken by each student. This is more of a tree transformation task than a point update task, and I tackle it in the next section.

[1]Saxon has had an implementation of something very similar for some years: see https://www.saxonica.com/documentation12/index.html#!extensions/instructions/deep-update.

8. Rule-based Transformation

For earlier thoughts on this subject, see the section "Template-based Transformation" in my Balisage 2022 paper: [8].

Transformation by applying template rules during the course of a recursive tree walk is the characteristic processing model of the XSLT language, and there is no intrinsic reason why it shouldn't work just as well with JTrees as with XML trees. The main challenge is that it's tricky to define match patterns when the things we are matching don't have distinguishing names.

There are three parts to the problem: how to break arrays and maps down into components that can conveniently be matched by template rules; how to define the corresponding match patterns; and how to construct new arrays and maps within the body of the corresponding template rules.

There are proposals for how to do this in the current XSLT 4.0 draft specification, and further ideas in my 2022 Balisage paper, but the more recent work on deep query suggests enhancements to these features that are not yet fully worked out. So what follows is my current proposal for a revision to the draft specification.

A good way to approach this is to start by thinking about what the standard default template rules should do. The desired effect is that we can define a mode of processing (say with `<xsl:mode on-no-match="traversal"/>`) which has the effect that if there are no user-written template rules in the mode, the effect is an identity transformation. But we want to design it so that it becomes easy to inject user-written template rules to customize the processing of particular constructs.

So let's start by defining a set of built-in rules for this new processing mode.

8.1. Built-In Rules

For arrays, we want the default rule to process all the members of the array. The question is, how should the array members be represented? Remember that an array member is in general a sequence rather than a single item. Although QT4 has generalised the concept of the context item so that it is now a context value (which can be any sequence), for the time being, XSLT templates are still applied to individual items, so we need to package up each array member as a single item.

In my Balisage 2022 paper I proposed representing an array member as a "parcel", which was essentially a sequence wrapped up as a single item. That raises the question, should parcels be a new kind of item in the data model, or should we reuse some existing kind of item (candidates being an array, a zero-arity function, or a single-entry map). The problem with introducing a new kind of item is that extensions to the type system are expensive and disruptive. The problem with reusing existing kinds of item is that it's harder to define match patterns that match them nicely.

My current proposal is to represent array members as records (that is maps) with (at least) the following fields:

- `array-member`: a boolean, always true. This field exists purely for convenience in writing match patterns: a template rule can use `match="record(array-member, *)"` with a high level of confidence that the rule won't accidentally match something else.
- `member`: the actual value of the array member, an arbitrary sequence.
- `index`: the 1-based index position of the array member within the array (just in case different processing is applied to different members based on their position).

The template rule for an array member is required to return a new array member; this must be represented by a record containing a `member` field, but the other fields are optional and ignored.

So the default template rule for arrays breaks up the arrays into its members, applies templates to each of them individually, and then reassembles the result:

```
<xsl:template match="array(*)">
  <xsl:array use="?member">
     <xsl:apply-templates select="
             for member $m at $index in .
             return {'array-member':true(), 'member':$m, 'index':$index}"
         mode="#current"/>
  </xsl:array>
</xsl:template>
```

A few observations on this code:

- `match="array(*)"` is one of a new class of match patterns that match items by type. In XSLT 3.0 this was written, clumsily, as `match=".[. instance of array(*)]`.
- `xsl:array` is a new instruction to construct an array. The contained sequence constructor delivers one item for each member of the new array. If a `use` attribute is present, then it is evaluated once for each of these items, to convert the supplied item to the required array member. In this case we expect the applied template rule to return a value of type `record(member)`, and the `use` expression extracts the contents of the `member` field to form the array member.
- `for member` is a new variant of the `for` expression (or XQuery FLWOR expression) that binds the variable `$m` to each member of the array in turn, as well as binding `$index` to its 1-based index position.
- The expression `{'x':1, 'y':2}` is a map constructor, equivalent to the XPath 3.1 expression `map{'x':1, 'y':2}`. The `map` keyword is no longer needed. (It was required in XPath 3.1 because some members of the design team wanted

to reserve "bare brace" syntax with no leading keyword for a different purpose.)

The default processing rule for members of an array is to apply templates to each item in the value of the member individually: recall that in the general case, an array member includes zero or more items. The processing returns a new array member that replaces the original:

```
<xsl:template match="record(array-member as xs:boolean,
                            member as item()*,
                            index as xs:integer,
                            *)">
     <xsl:apply-templates select="
         for $item at $pos in ?member
         return {'array-member-item': true(),
                 'member': $m,
                 'index': $index,
                 'item': $item,
                 'position': $pos}"
         mode="#current"/>
</xsl:template>
```

Observations:

- It's probably unlikely that many users would want to override processing at this level; but it provides the option for completeness. A template rule at this level has access to the entire array member and its index position within the array, and also to an individual item within the array member and its position within the array member.
- A template rule at this level is expected to return one or more items, which will substitute for the original item in the new array member.

The default processing for individual items is then to apply templates to the item:

```
<xsl:template match="record(array-member-item as xs:boolean,
                            member as item()*,
                            index as xs:integer,
                            item as item(),
                            position as xs:integer,
                            *)">
     <xsl:apply-templates select="?item" mode="#current"/>
</xsl:template>
```

And the default processing for items is to move the item to the output unchanged:

```
<xsl:template match="item()">
     <xsl:sequence select="."/>
</xsl:template>
```

Note that this just creates a reference to the existing item, it doesn't require making a deep copy of the item. This makes a big difference if an array or map contains XML nodes. The transformed JTree will contain the original XML nodes, rather than copies of these nodes. If copying or transformation of the XML nodes is required, this can be achieved by overriding the template rule for the individual XML nodes.

Maps are handled in a similar way to arrays. The top-level default template rule for maps splits it into its constituent entries (key-value pairs), marking each one with a `map:entry` field for ease of matching:

```
<xsl:template match="map(*)">
  <xsl:map on-duplicates="op(',')">
     <xsl:apply-templates select="
             for entry {$key, $value} in .
             return {'map-entry':true(), 'key': $key, 'value':$m}"
         mode="#current"/>
  </xsl:array>
</xsl:template>
```

Observations:

- `match="map(*)"` is another new type-based match pattern, matching any instance of the specified type.
- The `xsl:map` instruction exists in XSLT 3.0, but the `on-duplicates` attribute is new: it controls what happens when the sequence constructor delivers more than one map entry with the same key. The value is a function that in general takes two values having the same key and returns a single value; the expression `op(',')` returns an arity-2 function equivalent to the dyadic , (comma) operator which forms the sequence-concatenation of two values.
- The `for entry {$key, $value}` construct is a proposed extension to the `for` or `FLWOR` expression syntax for iterating over the entries in a map; it has been mooted as proposed XPath 4.0 syntax, but is not yet in the draft specification.

The default template rule for processing map entries looks like this:

```
<xsl:template match="record(map-entry as xs:boolean,
                            key as xs:anyAtomicType,
                            value as item()*,
                            *)">
   <xsl:map-entry key="?key">
     <xsl:apply-templates select="
         for $item at $pos in ?value
         return {'map-entry-item': true(),
                 'key': ?key
                 'value': ?value,
                 'item': $item,
                 'position': $pos}"
```

```
          mode="#current"/>
    </xsl:map-entry>
</xsl:template>
```

And again, the default processing at this level is to apply templates to the individual items directly:

```
<xsl:template match="record(map-entry-item as xs:boolean,
                            key as xs:anyAtomicType,
                            value as item()*,
                            item as item(),
                            position as xs:integer,
                            *)">
    <xsl:apply-templates select="?item" mode="#current"/>
</xsl:template>
```

Once again, it is unlikely that user applications would want to override the processing at this level, but the option is there for completeness.

Not stated in the details above is that each of these built-in templates passes any template parameters through unchanged.

In addition, the code for the built-in templates fails to show explicitly that each item passed to any of these template rules is labeled with provenance information indicating its position within the JTree being navigated. The existence of this label means that the function `selection-path` can be applied to the item to obtain information about the route used to select the item. The result of the function is a sequence of records, from innermost to outermost order, each containing some or all of the following fields:

- `container`: the map or array containing the value.
- `key`: the key (for an entry in a map) or index (for a member in an array) of the value within its container.
- `position`: the index position of an individual item within an array member or map value, when that value is a sequence of items.

Note that while built-in template rules maintain this information and pass it on to the template rules they call, user-written template rules may or may not do so, depending on how they are written. If they use the lookup operators `?` and `??` to make downwards selections, then the provenance of the values they select is maintained.

8.2. Use Cases

The most common use case is to override processing at the level of an individual entry in a map. Template rules at this level are expected to return zero or more map entries, in the format delivered by the `xsl:map-entry` instruction. Here are some examples of template rules that do this:

```
<xsl:template match="record(map-entry, *)[?key='note']"/>

 <xsl:template match="record(map-entry, *)[?key='mark']">
   <xsl:map:entry key="'mark'" select="upper-case(?value)"/>
 </xsl:template>

 <xsl:template match="record(map-entry, *)[?key='price']">
   <xsl:next-match/>
   <xsl:map:entry key="'currency'" select="'USD'"/>
 </xsl:template>
```

The first of these template rules (with `[?key='note']`) returns no output, so the matching map entries are deleted.

The second (with `[?key='mark']`) returns a map entry having the same key, but with the value converted to upper case.

The third (with `[?key='price']`) performs the built-in processing for map entries (by invoking `xsl:next-match`) and then adds a new map entry, setting `currency` to `"USD"`.

The transformation presented earlier (increase the price of all products tagged with the keyword "ice" by 10%) can be achieved in a number of ways. One way is to match selected product entries and then to modify the price directly using `map:put`:

```
<xsl:template match="record(map-entry, *)
          [?key='product']
          [?value?tag = 'ice']">
    <xsl:map-entry key="'product'">
      <xsl:sequence select="map:put(?value, 'price', ?value?price *
1.1)"/>
    </xsl:map-entry>
</xsl:template>
```

8.3. Grouping

While some transformations benefit from the rule-based recursive-descent paradigm, others involve wholesale reconstruction of a new document. Grouping and hierarchic inversion tasks often fall into this category.

Consider this use case from my 2016 paper, involving hierarchic inversion. The aim is to start with a JSON file containing a list of faculties and courses and the students enrolled on each course, and to invert this to produce a file containing a list of students, with the courses that each one is taking. Specifically, this is the input:

```
[{
    "faculty": "humanities",
    "courses": [
```

```
        {
            "course": "English",
            "students": [
                {
                    "first": "Mary",
                    "last": "Smith",
                    "email": "mary_smith@gmail.com"
                },
                {
                    "first": "Ann",
                    "last": "Jones",
                    "email": "ann_jones@gmail.com"
                }
            ]
        },
        {
            "course": "History",
            "students": [
                {
                    "first": "Ann",
                    "last": "Jones",
                    "email": "ann_jones@gmail.com"
                },
                {
                    "first": "John",
                    "last": "Taylor",
                    "email": "john_taylor@gmail.com"
                }
            ]
        }
    ]
},
{
    "faculty": "science",
    "courses": [
        {
            "course": "Physics",
            "students": [
                {
                    "first": "Anil",
                    "last": "Singh",
                    "email": "anil_singh@gmail.com"
                },
                {
                    "first": "Amisha",
                    "last": "Patel",
```

```
                    "email": "amisha_patel@gmail.com"
                }
            ]
        },
        {
            "course": "Chemistry",
            "students": [
                {
                    "first": "John",
                    "last": "Taylor",
                    "email": "john_taylor@gmail.com"
                },
                {
                    "first": "Anil",
                    "last": "Singh",
                    "email": "anil_singh@gmail.com"
                }
            ]
        }
    ]
}]
```

and this is the desired output:

```
[
    {
        "email": "ann_jones@gmail.com",
        "courses": [
            "English",
            "History"
        ]
    },
    {
        "email": "amisha_patel@gmail.com",
        "courses": ["Physics"]
    },
    {
        "email": "anil_singh@gmail.com",
        "courses": [
            "Physics",
            "Chemistry"
        ]
    },
    {
        "email": "mary_smith@gmail.com",
        "courses": ["English"]
    },
```

```
    {
        "email": "john_taylor@gmail.com",
        "courses": [
            "History",
            "Chemistry"
        ]
    }
]
```

In XSLT terms, this is clearly a grouping query. We want to select the students from the input file, group them by email address, and for each email address, output the courses attended by that student.

Here's the solution. The critical dependency is the call on `ancestors` which enables us to trace back to the JNodes visited during the deep lookup operation `json-doc('courses.json')??email`.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="4.0">

   <xsl:template name="xsl:initial-template">
      <xsl:array>
         <xsl:for-each-group select="json-doc('courses.json')??
entry::email" group-by="?value">
            <xsl:sort select="current-grouping-key()"/>
            <xsl:map>
               <xsl:map-entry key="'email'"
                              select="current-grouping-key()"/>
               <xsl:map-entry key="'courses'">
                  <xsl:array select="current-group() ? ancestors() ?
course"/>
               </xsl:map-entry>
            </xsl:map>
         </xsl:for-each-group>
      </xsl:array>
   </xsl:template>

</xsl:transform>
```

Notes on this solution:

- `entry::email` selects a sequence of map entries or key-value pairs in the form `record(key, value)`; the record also includes an `ancestors` field which is a function providing access to the containing maps and arrays.

  The grouping key `?value` is the actual email address.
- The expression `current-group() ? ancestors() ? course` doesn't currently work. The LHS of a dynamic function call must be a single function, but `current-group() ? ancestors` is a sequence of functions. I've raised an issue

to fix that. To make it work without this fix, it needs to be written as `(current-group() ! ?ancestors()) ? course`

We can also solve this use case using XQuery, taking advantage of the `group by` clause in a `FLWOR` expression. The following should work:

```
array {
  for $email in json-doc('courses.json'))??entry::email
  order by $email?value
  group by $email?value
  return {
     'email': $email?value,
     'courses': array { for $e in $email
                        return $e?ancestors()?course }
  }
}
```

In the interests of full disclosure, I am using a development version of Saxon that implements a slightly different version of the syntax from the snapshot presented in this paper (we are dealing here with rapidly evolving specifications and proposals, and the implementation is often either ahead of the spec or a little behind it). With this version of Saxon, the XSLT code that actually works is:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="4.0">

   <xsl:template name="xsl:initial-template">
      <xsl:array>
         <xsl:for-each-group select="pin(json-doc('courses.json'))??
email" group-by=".">
            <xsl:sort select="current-grouping-key()"/>
            <xsl:map>
               <xsl:map-entry key="'email'"
                              select="current-grouping-key()"/>
               <xsl:map-entry key="'courses'">
                  <xsl:variable name="labels" select="current-group() =!
> label()"/>
                  <xsl:variable name="ancestors" select="($labels ! ?
ancestors())[. instance of record(course, students)]"/>
                  <xsl:array select="$ancestors?course"/>
               </xsl:map-entry>
            </xsl:map>
         </xsl:for-each-group>
      </xsl:array>
   </xsl:template>

</xsl:transform>
```

And the working XQuery code is:

```
array {
  for $email in pin(json-doc('courses.json'))??email
  order by $email
  group by $email
  return {
     'email': $email,
     'courses': array { for $e in $email
                        return label($e)?ancestors()[. instance of
map(*)]?course }
  }
}
```

9. Conclusions

A major theme running through the proposals for XSLT 4.0 and XQuery 4.0 is improved support for manipulation of maps and arrays, both to make the languages more suitable for processing JSON, and also to improve the usability and performance of maps and arrays when used for internal working data within an XML transformation.

My 2016 paper at XML Prague demonstrated the shortcomings of the 3.0/3.1 specifications for achieving some simple use cases in this regard; with that version of the language, the best solution for JSON processing has often been to convert the JSON to XML, transform the XML, and then convert back to JSON.

This paper describes how these challenges have been addressed in the proposals for XSLT 4.0 and XQuery 4.0. It describes new language features in three main areas:

- Recursive Query
- Point Update
- Rule-based Transformation

and revisits the 2016 use cases to show how the new features work together to solve the problem.

References

[1] Gerard Huet. *The Zipper*. Journal of Functional Programming. 7 (5): 549–554 doi:10.1017/s0956796897002864. S2CID 31179878.

[2] Michael Kay. *Writing an XSLT Optimizer in XSLT*. Extreme Markup Languages, Montreal, 2007. Available at `http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html`.

[3] Michael Kay. *Transforming JSON using XSLT 3.0*. XML Prague 2016. Available at https://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf and at https://www.saxonica.com/papers/xmlprague-2016mhk.pdf.

[4] Michael Kay. *XML Tree Models for Efficient Copy Operations*. XML Prague 2018. Available at https://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf and at https://www.saxonica.com/papers/xmlprague-2018mhk.pdf.

[5] Michael Kay. *An XSD 1.1 Schema Validator Written in XSLT 3.0*. Markup UK 2018. Available at https://markupuk.org/2018/Markup-UK-2018-proceedings.pdf and at https://www.saxonica.com/papers/markupuk-2018mhk.pdf.

[6] Michael Kay and John Lumley. *An XSLT compiler written in XSLT: can it perform?*. XML Prague 2019. Available at https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf and at https://www.saxonica.com/papers/xmlprague-2019mhk.pdf.

[7] Michael Kay. *<transpile from="Java" to="C#" via="XML" with="XSLT"/>*. Markup UK 2021. Available at https://markupuk.org/2018/Markup-UK-2021-proceedings.pdf and at https://www.saxonica.com/papers/markupuk-2021mhk.pdf.

[8] Michael Kay. *XSLT Extensions for JSON Processing*. Balisage 2022. Available at https://www.balisage.net/Proceedings/vol27/html/Kay01/BalisageVol27-Kay01.html .