XPath 2.1 HOFs Need Sugar

Recently there were reports about new, important developments in the evolution of XPath, XQuery and XSLT. What would truly be one of the most significant changes in XPath 2.1 (and its major hosting languages XSLT and XQuery), is the support of Higher Order Functions (HOF) in XQuery, "though … expect it to migrate to XSLT in due course. Unfortunately the XQuery 1.1 working draft that describes this feature isn't yet published outside W3C".

The next message in the same xsl-list thread provides even more details:

"Actually, although W3C hasn't published anything, John Snelson has published his proposal at

http://snelson.org.uk/~jpcs/higher-order-functions.html
which is essentially what Saxon 9.2 implements."

I have had some experience reading comments on XSLT/XPath-related W3C working drafts. Pretty often a comment is praised as correct but the decision is not to reflect it in the design as "sorry, it is too-late". This time the XSLT community has the unique chance to assess an original proposal even before the first working draft is published.

In the rest of this document I briefly provide my overall impression, then go on to point out the problems in the proposal, and finally I introduce solutions to each of these problems.

General impression

HOFs are a much needed addition to XPath and its hosting languages. The proposal by John Snelson (further referred to as ***the HOF P***, or just as ***the (P)***), if implemented, will provide this desirable feature. It defines and adds a "function item" to the XPath Data Model (XDM). Literal (statically defined) and inline (dynamically defined) functions are defined in a way that includes the function name, the variables in its scope, and its arity. A function item is referenced in an XPath expression using its name and arity (to distinguish between different overloads), or as a dynamically defined (inline) "function()".

Type checking for function items is briefly defined. Some modifications to XQuery that become necessary with the introduction of the "function item" type are defined.

Three additional functions: fn:function-name(), fn:function-arity() and fn:partial-apply() are defined.

The document ends by providing several examples of using the proposed feature.

As a whole I find that the document can be used as a base for a specification, though I would not recommend doing this without some necessary corrections.

Problems with the (P) proposal

Several problems, not critical but nevertheless significant, are immediately visible:

Problem 1: Literal function names must always have arity specified. The overload specification is unnatural.

Not all **F & O** functions have overloads and most user-written functions usually do not have overloads. It is very inconvenient to always have to specify the arity of a given literal function item, even though it is statically known that this function has no overloads. Not only it is rarely necessary to disambiguate an overload but specifying the function arity is only just one (and not the best) of many possible ways to disambiguate an overload.

It is very unnatural for a programmer to remember which overload has which arity. The programmer thinks in terms of what a function does, not how many arguments a function has.

Having to remember and specify function arity, even though very few functions have overloads, is plain inconvenient and annoying, and results in unnecessarily complex and difficult to understand and maintain code that is vulnerable to errors.

Problem 2: Operators cannot be used as functions.

The (P) doesn't provide any way to use operators as functions. In the examples it provides, a new function item, wrapping an operator is constructed, in order to use this operator as a HOF. This is very inconvenient and results in bloated and difficult to understand and maintain code that is vulnerable to errors. Here is **a particular example from (P)**:

```
let $sum :=
  fn:partial-apply(
    fn:partial-apply(local:foldl#3, function($a, $b) { $a + $b }),
    0)
```

Apart from other deficiencies of this code that are addressed later, see how an inline function item has to be created only because we cannot simply reference the "+" operator, which is already defined in **F & O** as **`op:numeric-add()`**

Having to wrap the XPath operators in inline or user-defined functions all over again and again is a bad "feature" that significantly decreases programmer's productivity and results in unnecessarily complicated and bloated code.

Problem 3: The partial-apply() function is "too-partial" and affects readability / understanding / maintainability.

The **fn:partial-apply()** is defined in (P) as:

```
fn:partial-apply($function as function(), $arg as item()*) as function()
```

It only allows argument values to be provided one at a time. In many cases a programmer needs to provide the value of more than one argument to produce a partial application function for a particular

purpose. In all such cases `fn:partial-apply()` has to be applied many times (as many as the number of arguments for which value is provided). This results in particularly bloated and unreadable code.

A good example of this problem is provided in (P) itself:

```
let $sum :=
  fn:partial-apply(
    fn:partial-apply(local:foldl#3, function($a, $b) { $a + $b }),
    0)
```

Here we need to construct a partial application of the `foldl()` function on the first (pass the "+" as the "acting" function) and the second (pass 0 as the initial accumulator value) of its three arguments.

Make a small experiment to get an idea how well people understand and value such code. Just show the above snippet to 100 average or even experienced programmers and ask them to spend five minutes and explain what this code does and whether they like it. Do not be surprised if those who understood the code can be counted with the fingers of one hand . . . Expect even less to like the code.

Solutions to problems 1 - 3.

Solution 1: A Literal function reference must end with (). #void "argument" for use when invoking 0-argument functions. Use function aliases or fixed namespaces for designating overloads, and only for a function with overloads.

The first step in solving Problem 1 is to change the syntax of a literal function reference.

```
LiteralFunctionItem ::= QName "()"
```

Here we've got rid of the "#" and the arity. The opening and closing brackets "()" are added in order to easily distinguish function references from element names and thus achieve better and more natural readability of the resulting XPath expressions. This also disambiguates the function item from an element that may happen to have the same QName.

So far this is a good result in boosting readability, though a small problem remains: how to distinguish a function item from the invocation of a zero-argument function? What is the meaning of the following expression?

```
my:fun(current-time(), 3)
```

Is `current-time()` here denoting a function item or is this the *`xs:time`* that must be returned by the function `current-time()` ?

In order to eliminate any such ambiguity, **the following syntax rule** will be changed in the XPath specification as specified below:

[42] Literal ::= NumericLiteral | StringLiteral | "#void"

and any invocation of a function with zero arguments will be specified as:

```
QName "(" "#void" ")"
```

Then, following this rule we know that in the expression:

```
my:fun(current-time(), 3)
```

the first argument is the function item `current-time()` , not the value `current-time(#void)`.

One last problem is how to specify a particular overload for a function with overloads. While one possible solution is specifying the arity, it is not the only possible one and it is certainly not the best possible solution.

A better solution to the function overloads disambiguation problem is to provide a way for specifying an alias (alternative QName) for any function overload. For example, the definition of an **xsl:function** could be specified as:

```
<!-- Category: declaration -->
<xsl:function
  name = qname
  alias? = qname
  as? = sequence-type
  override? = "yes" | "no">
  <!-- Content: (xsl:param*, sequence-constructor) -->
</xsl:function>
```

An alias does not have to look too different from the primary function name; it can easily have the same local-name as the primary name. Then a function with three overloads can be referenced as:

```
f:myFun()

f2:myFun()

f3:myFun()
```

where `f:myFun()` is the primary name and `f2:myFyn()` and `f3:myFun()` are the aliases for the second and third overload. The `f2` and `f3` prefixes here are associated with two different namespaces, which themselves are different from the namespace of the primary function name.

Going even further, we can completely eliminate the need to specify an `alias` attribute. We introduce the following rule:

No explicit alias for an overload of a function is needed if a *conventional alias* is used. A conventional alias has the following syntax:

```
ConventionalAlias   ::=   Prefix  ":"  LocalName
```

where `Prefix` is associated with the following namespace:

```
ConventionalOverloadNamespace ::=

 "http://www.w3.org/2005/xpath-functions/overload/" Arity "/" PrimaryFunctionNamespace
```

Thus, if for the primary function name `f:myFun()` above, the prefix `"f"` is associated with the namespace `"my:fun"` and the prefixes `"f2"` and `"f3"` are associated, respectively, with the namespaces:

```
"http://www.w3.org/2005/xpath-functions/overload/2/my:fun"
```

and

```
"http://www.w3.org/2005/xpath-functions/overload/3/my:fun"
```

then the function overloads can be defined without providing any specific alias at all.

Another important point is that we can use any namespace for the primary function name, so in the 90% of the cases where a function has no overloads, we are not concerned at all with specifying its arity or putting it in a special namespace.

Solution 2. Allow any standard XPath operator to be referenced as a regular function item using the naming style op:OperatorName. Where applicable, provide shorter aliases.

This solution is simple and obvious. Recognize the well-known fact that the XPath 2.0 operators are legal functions and use them as such. Instead of having to write:

```
function($a, $b) { $a + $b }
```

We simply write:

```
op:numeric-add()
```

What remains to be done here is to provide in **F & O** shorter names for the operators – we all know that addition is a numeric operation. Then the above will be even simpler. Instead of:

```
function($a, $b) { $a + $b }
```

We write

```
op:add()
```

Solution 3: A function application construct to specify a partial application simultaneously on one or more arguments.

Let's return to the indicative example from (P):

```
let $sum :=
  fn:partial-apply(
    fn:partial-apply(local:foldl#3, function($a, $b) { $a + $b }),
    0)
```

In this example we have a partial application on two of the three arguments of a function and the code is already extremely unreadable and difficult to understand. What if we had a five-argument function and we needed a partial application on its second, third, and fourth argument?

The solution is to replace `fn:partial-apply()` with a syntactic construct that would allow us to specify the values of all arguments at the same time:

```
let $sum := f:foldl(op:add(), 0, _ )
```

More generally,

```
f:fun(_, _, ... , _ )
```

is a *function invocation pattern*.

The underscores are *argument placeholders*. The expression, as written above is equivalent to:

```
f:fun()
```

The function invocation pattern is used to simultaneously construct a partial application on one or more arguments in the following way:

```
f:fun(_, 2, _, 4, _)
```

specifies the partial application of the 5-argument function f:fun(), in which the second argument is given value 2 and the fourth argument is given the value 4.

The same expression would have to be written in (P) as:

```
fn:partial-apply(fn:partial-apply(f:fun#5,2,2),3,4)
```

Further, we introduce the rule that if values for the first K arguments are supplied and the function doesn't have an overload with K arguments, then the trailing underscores in the function invocation pattern may be omitted. Thus, we will usually only write:

```
f:func(1,2,3)
```

instead of:

```
f:func(1,2,3, _, _, _)
```

Notice, that as an added benefit the function invocation pattern allows for full static type-checking of the provided values, something that is impossible by definition with `fn:partial-apply` in (P)

To summarize, with the combined solution of problems 1 to 3 the programmer simply writes:

```
let $sum := f:foldl(op:add(), 0)
```

instead of being forced with (P) to write:

```
let $sum :=
  fn:partial-apply(
    fn:partial-apply(local:foldl#3, function($a, $b) { $a + $b }),
    0)
```