Version 2 Changes

1. Reflected the comments of Michael Kay: http://www.w3.org/Bugs/Public/show_bug.cgi?id=7350#c1

2. Updated the syntax for a literal function item so while it retains the ending brackets, there is no conflict with an invocation of a 0-argument function.

3. Removed the aliases as means to identify an overload, following Michael Kay's comment that they create more complexity than the one they are intended to solve.

4. Introduced the notion of *primary download* and *non-primary download.*

5. Specified how arity will be used with a more user-friendly syntax and only whenever a function has overloads.

6. Reflected a suggestion of Michael Kay, that the argument placeholder could be specified by the "?" character.

7. Noted the important fact that the function invocation pattern is applicable both to a literal function item and to an inline function item.

8. Added more examples.

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.

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 always specify function arity, even for functions with no 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.

As another example, let's express the partial application of a function with 5 arguments, with values supplied for its 2nd and 4th argument. Below are two different ways of doing this with the fn:partial-apply() function as defined in (P):

```
fn:partial-apply(fn:partial-apply(f:fun#5,2,7),3,4)

fn:partial-apply(fn:partial-apply(f:fun#5,4,4),2,7)
```

Not only are these two expressions completely unreadable, but it is extremely difficult to see that they are equivalent and mean the same thing (because in a functional language the result of evaluation does not depend on the order of evaluation).

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 (~) or with (~ arity)

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

```
LiteralFunctionItem ::= QName "(~)"                              (1)

                      | QName "(" "~" arity ")"                  (2)
```

The opening and closing brackets "()" are added in order to easily achieve better. This also disambiguates the function item from an element that may happen to have the same QName.

The "~" inside the brackets is used to distinguish a function item from an invocation of a 0-argument function. Thus,

```
current-time(~)
```

denotes the literal function item with name "`current-time`", while

```
  current-time()
```

denotes the invocation of the 0-argument function `current-time()`.

Rule (1) is used to specify the literal function item for any named function that has no overloads or for the *primary overload* (see the definition later in the document) of any function that has overloads.

Rule (2) **must** be used to specify the literal function item for any *non-primary overload* of a named function.

A *primary overload* is defined in the following way:

1. Any named function with no overload is its own *primary overload*.
2. Exactly one overload of a function with overloads is designated as the *primary overload.*
3. Any overload, which is not a primary overload according to 1. or 2. is a *non-primary overload*.

Whether a given named function is a primary overload is specified as part of the definition of this function. In XSLT the definition of an **xsl:function** will be specified as:

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

Because the default value for the "`primary-overload`" attribute is `true()`, it is not necessary to specify this attribute on any function definition for a function with no overloads or even for a function with overloads, if this function is the primary overload. This probably covers 90% or more of all function definitions.

The documentation describing any named function with overloads should specify which overload is primary. In case this is omitted, the first overload described is considered the primary one. All **F&O** functions follow this rule.

Examples:

```
  contains(~)
```

is the function item for the primary overload of the function **contains()**.

```
  contains(~3)
```

is the function item for the only non-primary overload of the same function.

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(?, 7, ?, 4, ?)
```

specifies the partial application of the 5-argument function f:fun(), in which the second argument is given value 7 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#5,2,7),3,4)
```

Further, we introduce the rule that if values for the first K arguments are supplied and the function doesn't have two overloads with more than K arguments, then the trailing underscores in the function invocation pattern may be omitted. This is always the case with a function that has no overloads. 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)

It is important to note that the function invocation pattern can be used with any function item – not only with a literal function item. Thus:

  $func(1, ?, 3, ?, 5)

denotes the partial application of the function item $func on its 1st, 3rd and 5th arguments.

Because the function invocation pattern can be used with both literal and inline function items, there remains no need for the `fn:partial-apply()` function and it may safely be discarded from the XPath 2.1 specification.

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)