Function Composition and Recursion in XPath 3.0

Roger Costello
October 2012

XPath has become a much more functional language with the arrival of XPath 3.0. That is very exciting.

This paper shows how to implement, using XPath 3.0, two key functional capabilities:

1. function composition
2. recursion

These capabilities will be implemented using unnamed (anonymous) functions.

At the time of this writing the XPath specification is still a working draft. Nonetheless, SAXON has implemented all the capabilities described in this paper. So, you can experiment with these capabilities today.

Thanks to Dimitre Novatchev and Michael Kay for showing me how to implement these capabilities.

Function Composition

The ability to compose functions is arguably the most important feature of any functional programming language. So it is important to know how to compose functions in XPath 3.0.

The easiest way to explain function composition is with an example.

Example: Let's compose these two functions:

1. *increment*: this function adds 1 to its argument.
2. *double*: this function multiplies its argument by 2.

In the Haskell programming language the two functions are composed like so:

f = double . increment

The dot (.) is the composition operator. Function composition means the output of the *increment* function is to be used as the input to the *double* function.

f is a new function (it is the composition of *double* and *increment*). Function *f* can be applied to an argument:

f 2

The result is 6. That is the expected result:

- add 1 to the argument: (2 + 1)
- then multiply by 2: (2 + 1) * 2
- to produce the result: (2 + 1) * 2 = 6

XPath 3.0 Implementation

Function *increment* is implemented in XPath 3.0 like so:

```
$increment  :=  function($x as xs:integer) {$x + 1}
```

Read as: the variable *increment* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema, xs) integer *x*. The function returns the value of *x* plus 1.

Function *double* is implemented:

```
$double    :=  function($y as xs:integer) {$y * 2}
```

Read as: the variable *double* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema, xs) integer *y*. The function returns the value of *y* multiplied by 2.

Next, we create an (unnamed) function that composes two functions, *a* and *b*. It returns an (unnamed) function. The returned function takes one argument, *c*, and returns the result of applying *a* to *c* and then *b* to *a(c)*:

```
$compose   := function(
                          $a as function(item()*) as item()*,
                          $b as function(item()*) as item()*
                      )
                      as function(item()*) as item()*
                 {function($c as item()*) as item()* {$b($a($c))}}
```

Read as: the variable *compose* is assigned a value that is an (unnamed) function which takes two arguments, *a* and *b*. Both *a* and *b* are functions. The unnamed function returns an unnamed function. The returned function takes one argument, *c*. The implementation of the returned function is to apply *a* to *c* and then apply *b* to the result.

> This is a great example of a *higher-order function*. A higher-order function is a function that takes as its arguments function(s) and/or returns as its result a function. In this example, the function takes *two* functions as arguments *and* returns a function. Wow!

Next, we assign to variable *f* the result of composing function *increment* and *double*:

```
$f  :=  $compose($increment, $double)
```

Lastly, apply *f* to some value, say 2:

```
$f(2)
```

Here is the complete XPath 3.0 program:

```
let $increment  :=  function($x as xs:integer) {$x + 1},
    $double       :=  function($y as xs:integer) {$y * 2},
    $compose    :=  function(
                                        $a as function(item()*) as item()*,
                                        $b as function(item()*) as item()*
                                      )
                                     as function(item()*) as item()*
                               {function($c as item()*) as item()*  {$b($a($c))}},
     $f  :=  $compose($increment, $double)

return $f(2)
```

Observe these things:

- Each statement is separated from the next by a comma
- The first statement is initiated with the keyword, *let*
- The format is: *let statement, statement, …, statement return expression*

That XPath program can be used in an XSLT 3.0 program, an XQuery 3.0 program, and any language that supports XPath 3.0.

That is a beautiful program.

Recursion

XPath 3.0 does not support named functions. It only provides unnamed (anonymous) functions. So recursion is a bit tricky – how do you recurse in a function that has no name? This section shows how. The technique shown in this paper was discovered by Dimitre Novatchev.

The easiest way to explain the technique is with an example.

Example: Let's implement a recursive *until* function (loop until some condition is satisfied).

Function *until* takes three arguments:

1. *p* is a boolean function
2. *f* is a function on *x*

3. *x* is the value being processed

Here is an example using the function:

```
$until ($isNegative, $decrement, 3)
```

Read as: decrement 3 until it is negative. The result is (-1).

XPath 3.0 Implementation

Function *isNegative* is implemented in XPath 3.0 like so:

```
$isNegative  :=  function($x as xs:integer) {$x lt 0}
```

Read as: the variable *isNegative* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema, xs) integer *x*. The function returns True if *x* is less than 0 and False otherwise.

Function *decrement* is implemented:

```
$decrement  :=  function($x as xs:integer) {$x - 1}
```

Read as: the variable *decrement* is assigned a value that is an (unnamed) function which takes as its argument an (XML Schema, xs) integer *x*. The function returns the value of *x* minus 1.

Now it is time to implement function *until*. Here it is:

```
$until := function(
                        $p as function(item()*) as xs:boolean,
                        $f as function(item()*) as item()*,
                        $x as item()*
                    ) as item()*
        {$helper($p, $f, $x, $helper)}
```

Read as: the variable *until* is assigned a value that is an (unnamed) function which takes 3 arguments: a function, *p*, that takes an argument and returns a boolean value, a function, *f*, that takes an argument and returns a result, and a value, *x*. The function invokes another function that I called *helper*. Function *helper* is invoked with *p*, *f*, *x*, and itself.

This is key: to implement recursion with unnamed functions you must assign the unnamed function to a variable and invoke the function by passing it the variable. Wow!

Okay, let's look at the helper function:

```
$helper  := function(
                              $p as function(item()*) as xs:boolean,
                              $f as function(item()*) as item()*,
                              $x as item()*,
                              $helper as function(function(), function(), item()*, function()) as item()*

                         ) as item()*
                {if ($p($x)) then $x else $helper($p, $f, $f($x), $helper)}
```

Read as: the variable *helper* is assigned a value that is an (unnamed) function which takes 4 arguments: *p*, *f*, *x*, and *helper*. The function applies p to x and if the result is True then it returns x, otherwise it recurses, calling *helper* with *p*, *f*, *f(x)*, and *helper*.

Lastly, let's apply *until* to some value, say 3:

```
$until($isNegative, $decrement, 3)
```

The result is the value (-1).

Here is the complete XPath 3.0 program:

```
let $isNegative := function($x as xs:integer) {$x lt 0},
    $decrement  := function($x as xs:integer) {$x - 1},
    $helper  := function(
                              $p as function(item()*) as xs:boolean,
                              $f as function(item()*) as item()*,
                              $x as item()*,
                              $helper as function(function(), function(), item()*, function()) as item()*

                            ) as item()*
                {if ($p($x)) then $x else $helper($p, $f, $f($x), $helper)},
     $until := function(
                             $p as function(item()*) as xs:boolean,
                             $f as function(item()*) as item()*,
                             $x as item()*
                           ) as item()*
                {$helper($p, $f, $x, $helper)}
return $until($isNegative, $decrement, 3)
```

This is a fantastic technique.