Extreme Markup Languages 2006®

Montréal, Québec
August 7-11, 2006

Higher-Order Functional Programming with XSLT 2.0 and FXSL

Dimitre Novatchev
Microsoft

Abstract

This article describes the added support in FXSL 2.0 for writing higher-order functions in XSLT 2.0 based on new fundamental XPath 2.0/XSLT 2.0 features such as the sequence datatype, strong typing and writing functions natively in XSLT. FXSL 2.0 makes nearly all standard XPath 2.0/XSLT 2.0 functions and operators higher-order by providing in the FXSL namespace the definition of their identically named higher-order wrappers and partial applications. The author argues that in effect, this makes XSLT 2.0 + FXSL a higher-order strongly-typed functional programming system.

This paper demonstrates how based on the even higher degree of abstraction and code reuse many challenging problems have now more compact or even one-line solutions.

A Conference of IDEAlliance
Extreme Markup 2006 Languages®
Montréal, August 7–11, 2006

Higher-Order Functional Programming with XSLT 2.0 and FXSL

Table of Contents

A Conference of IDEAlliance
Extreme Markup 2006 Languages®
Montréal, August 7–11, 2006

Higher-Order Functional Programming with XSLT 2.0 and FXSL

Dimitre Novatchev

§ 1 Introduction

XSLT has turned out to be very different from the typical programming languages in use today. One question that's being asked frequently is: *What kind of programming language is actually XSLT?* Until recently, the authoritative answer from some of the best specialists was that XSLT is a declarative (as opposed to imperative), but still not a FP [functional programming], language. Michael Kay notes in his article "What kind of language is XSLT" [Kay]:

> Although XSLT is based on functional programming ideas, it is not as yet a full functional programming language, as it lacks the ability to treat functions as a first-class data type.

My Extreme 2003 paper "*Functional Programming in XSLT using the FXSL Library*" [Nova5] described a way to implement higher-order functions in XSLT 1.0 and demonstrated the XSLT implementation of some of the most general FP design patterns.

This article goes further. It shows how the new features of XSLT 2.0 and XPath 2.0 make it even easier to write higher-order native `xsl:functions` that are compact and type-aware, and how otherwise complex processing tasks can now be implemented as one-liner XPath 2.0 expressions. What is more, even the standard XPath 2.0/XSLT 2.0 functions and operators [F & O], [XSLT2.0] have been turned into their higher-order-function equivalents.

1.1 Basic definitions

For readers not familiar with the basic functional programming concepts, some definitions and brief descriptions are provided below.

1.1.1 Imperative programming

The imperative style of programming describes a system as evolving from an initial state through a series of state changes to a set of desired final states. A program consists of commands that change the state of the system. For example,

```
y = y - 2
```

will bring the system into a new state, in which the variable `y` has a new value, which has been obtained by subtracting `2` from the value of `y` in the previous state of the system.

1.1.2 Declarative programming

The declarative programming style specifies relationships between different variables, *e.g.*, the equation

```
z = y - 2
```

declares `z` to have a value of two less than the value of `y`.

Variables, once declared, cannot change their value. Typically, there is no concept of *state, order of execution, memory, ..., etc.*

In XSLT [XSLT1.0], [XSLT2.0] the declarative approach is used, *e.g.*,

```
<xsl:variable name="z" select= "$y - 2" />
```

is the XSLT version of the mathematical equation above.

1.1.3 Functional programming

A function is a relationship between a set of *inputs* and an *output*. It can also be regarded as an operation, which when passed specific values as input produces a specific output.

A *functional program* is made up of a series of definitions of functions and other values [ThompSJ].

The functional programming style builds upon the declarative programming style by adding the ability to treat functions as first-class objects — that means, among other things, that functions can be passed as arguments to other functions. A function can also return another function as its result.

1.1.4 *Basic Haskell notation*

In the rest of this article, the following notations are borrowed from the programming language Haskell [ThompSJ], [JonesSP] and are used to show an abbreviated version of the XSLT code.

Function definition

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
```

is the definition of a function `f` having arguments `x`, `y`, and `z` and producing their sum. The first line in the definition is an optional declaration of the type of the function. It says that `f` is a function, which takes three arguments of type `Int` and produces a result of type `Int`. The type of `f` itself is:

```
Int -> Int -> Int -> Int
```

Function application

```
f x y z
```

is the application of the function `f` to the arguments `x`, `y`, and `z`

Lists

```
[a1, a2, a3]
```

is the list of elements a1, a2, a3.

```
[ ]
```

is the empty list.

```
xs
```

by convention should be a variable, which is a list of elements of type `x`

```
x:xs
```

is an operation which pre-pends an element `x` in front of a list `xs`

In a list `x:xs` the symbol `x` denotes the **head** and `xs` denotes the **tail** of the list. Operating on the head and tail is useful in defining list-processing functions recursively.

1.2 *HOF [Higher-Order Functions]*

A function is *higher order* if it takes a function as an argument or returns a function as a result, or both [ThompSJ].

Examples.

A classical example is the ***map*** function, which can be defined in Haskell in the following two ways:

```
map   f    xs     =  [ f  x  |   x  <-  xs ]                    (1)
```

or

```
  map   f   [ ]     =  [ ]
                                                                (2)
  map   f   (x:xs)   =   f  x   :   map   f    xs
```

In this definition `|` means "*such that*" and `<-` means "*belongs to*". `:` is the ***cons*** operator — this prepends an element at the start of a list.

The map function takes two arguments — another function ***f*** and a list ***xs***. The result is a list, every element of which is the result of applying ***f*** to the corresponding element of the list ***xs***.

If we define **f** as:

```
    f x = x + 5
```

and **xs** as

```
    [1, 2, 3]
```

Then the value of

```
    map   f    xs
```

is

```
       [6, 7, 8]
```

Another example – fold

```
fold f z0 [ ]  = z0
fold f z0 x:xs = fold  f  (f  z0  x)  xs

sum xs = fold (+) 0  xs

product xs = fold (*) 1 xs
```

> *The last two examples show how easy and convenient it is to produce new functions from a more general higher order function, simply by feeding it with different function-arguments.*

Whenever XSLT programmers try to do without higher order programming, then they repeatedly have to write similar recursive processing code (e.g. calculating product, maximum, minimum, etc.); this wastes programming effort each time an instance of such code is written, and means whenever a change is made, it has to be replicated in many locations — this is very error-prone and often becomes a maintenance nightmare.

§ 2 Higher Order Functions and XSLT

A simple analysis shows that there's nothing like higher-order functions in XSLT. In XSLT 1.0 even the notion of writing one own's function is not supported natively.

XSLT 2.0 provides the `xsl:function` instruction; now programmers can write their functions natively in XSLT. However, it is still not possible to pass an `xsl:function` as a parameter to another callable component and then invoke it dynamically.

Therefore, as a start, we have to define what is to be considered a higher order function in XSLT. Then, we will describe a mechanism that allows "handles" or "references" to such functions to be passed to other functions, or returned by functions. Finally, we show how to write an `xsl:function` so that it can be passed as a parameter in a most convenient and intuitive way.

2.1 Higher-Order Functions in XSLT 1.0

XSLT 1.0 templates come closest to the definition of functions. Templates accept parameters and produce what can be regarded as a single result. Not all types of templates, however, can have references that are intuitive and easily represented. An example is a named template without a "`match`" attribute. The following is illegal in XSLT:

```
<xsl:call-template name="$varName"/>    WRONG!
```

A template name should be a QName and so is the value of the "`name`" attribute of `xsl:call-template` [XSLT1.0]. A QName is static and must be completely specified — it cannot be dynamically produced by the contents of a variable. An XSLT processor produces a compile-time error.

2.2 Template References

Another way to instantiate a template dynamically has been known for quite some time [HainesC], but there was no evidence of using it in a systematic manner before 2002. Let's have a template, which matches only a single node belonging to a unique namespace. We'll call such nodes "*nodes having a unique type*" or "*template references*":

File: example1.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version = "1.0" >
 <xsl:template match=
     "*[namespace-uri()='f:8B9C63F4-F4AB5D11-994A0001-B4CD626F']">
```

```
    <xsl:param name = "pX" />
    <xsl:value-of select = "$pX + $pX" />
 </xsl:template>

 <xsl:template match=
    "*[namespace-uri()='f:AB02AC1C-1C65B3FF-77C5FFFE-4B329DA1']">
    <xsl:param name = "pX" />
    <xsl:value-of select = "5 * $pX" />
 </xsl:template>
</xsl:stylesheet>
```

We have defined two templates, each matching only a node of a unique type. The first template produces twice the value of its input parameter `pX`. The second template produces the value of its input parameter `pX` multiplied by 5.

Now we'll define in another stylesheet a template (let's think of it as the `apply-and-sum-two-functions` template) that accepts as parameters references to two other templates (*template references*), instantiates the two templates that are referenced by the template-reference parameters, passing as parameter to each instantiation its `pX` parameter, then as result it produces the sum of the outputs of these instantiated templates.

```
<xsl:stylesheet  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:f1="f:8B9C63F4-F4AB5D11-994A0001-B4CD626F"
  xmlns:f2="f:AB02AC1C-1C65B3FF-77C5FFFE-4B329DA1" >

        <xsl:import href = "example1.xsl" />
        <xsl:output method = "text" />

        <f1:f1/>
        <f2:f2/>

        <xsl:template match = "/" >
            <xsl:variable name = "vFun1" select = "document('')/*/f1:*[1]" />
            <xsl:variable name = "vFun2" select = "document('')/*/f2:*[1]" />

            <xsl:call-template name = "sum2Functions" >
                <xsl:with-param name = "pX" select = "3" />
                <xsl:with-param name = "pFun1" select = "$vFun1" />
                <xsl:with-param name = "pFun2" select = "$vFun2" />
            </xsl:call-template>
        </xsl:template>

        <xsl:template name = "sum2Functions" >
            <xsl:param name = "pX" />
            <xsl:param name = "pFun1" select = "/.." />
            <xsl:param name = "pFun2" select = "/.." />

            <xsl:variable name = "vFx_1" >
                <xsl:apply-templates select = "$pFun1" >
                    <xsl:with-param name = "pX" select = "$pX" />
                </xsl:apply-templates>
            </xsl:variable>

            <xsl:variable name = "vFx_2" >
                <xsl:apply-templates select = "$pFun2" >
                    <xsl:with-param name = "pX" select = "$pX" />
                </xsl:apply-templates>
            </xsl:variable>

            <xsl:value-of select = "$vFx_1 + $vFx_2" />
        </xsl:template>
</xsl:stylesheet>
```

The result produced when this last stylesheet is applied on any (ignored) xml source document, is: `21`

What we have effectively done is we called the template named "**sum2Functions**", passing to it two *references to templates*, that accept a **pX** parameter and produce something out of it. The "**sum2Functions**" template successfully instantiates (applies) the templates that are uniquely identified by the template reference parameters, then finally produces the sum of their results. What guarantees that the XSLT processor will select exactly the necessary templates is the unique namespace-uri of the nodes they are matching. The most important property of a template reference is that it guarantees the unique matching of the template that it is referencing.

2.3 Higher-Order Functions in XSLT 2.0

In XSLT 2.0 the programmer can define functions written natively in XSLT using the new `xsl:function` instruction. Functions defined in this way can declare their return type and the types of

the parameters they accept. It is legal to have different overloads (`xsl:function`s defined with the same name) as long as they have different number of parameters.

However, there's no obvious way to pass an `xsl:function` as a parameter. The following XPath 2.0 expression is illegal:

```
$varFunc($params)    (: WRONG! :)
```

According to the latest XSLT 2.0 Candidate Recommendation [XSLT2.0] the value of the name attribute of `xsl:function` must be a QName – static and completely specified at compile time – it cannot be dynamically produced by the contents of a variable. An XSLT 2.0 processor produces a static (compile-time) error.

```
someFunction(funcName(), otherParams)    (: Not what we wish! :)
```

In the call of `someFunction()` above the first parameter is not the function `funcName()` itself, but the value of a function `funcName()` that is defined to have zero arguments.

Once again template references come to help.

We can write an XSLT 2.0 `xsl:function` that accepts as a parameter a template reference. The most important such function is `f:apply()`. It invokes its first argument via templates application, passing all its remaining arguments as parameters:

```
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:f="http://fxsl.sf.net/"
 exclude-result-prefixes="f">

  <xsl:function name="f:apply">
   <xsl:param name="pFunc" as="element()"/>
   <xsl:param name="arg1"/>

   <xsl:apply-templates select="$pFunc" mode="f:FXSL">
      <xsl:with-param name="arg1" select="$arg1"/>
   </xsl:apply-templates>
  </xsl:function>

  <xsl:function name="f:apply">
   <xsl:param name="pFunc" as="element()"/>
   <xsl:param name="arg1"/>
   <xsl:param name="arg2"/>

   <xsl:apply-templates select="$pFunc" mode="f:FXSL">
      <xsl:with-param name="arg1" select="$arg1"/>
      <xsl:with-param name="arg2" select="$arg2"/>
   </xsl:apply-templates>
</xsl:function>

<!-- More overloads for fns with 3,4,5,6,7,8,9,10 etc. parameters  -->
</xsl:stylesheet>
```

Now, it is easy to write an `xsl:function` producing the same result as the `apply-and-sum-two-functions` template:

```
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:f="http://fxsl.sf.net/"
 exclude-result-prefixes="f">
  <xsl:import href="../f/func-apply.xsl"/>
  <xsl:output omit-xml-declaration="yes" indent="yes"/>

  <xsl:template match="/">
    <xsl:sequence select=
     "f:applyAndSum($vfunTwice, $vfunFiveTimes, 3)"/>
  </xsl:template>

  <xsl:function name="f:applyAndSum">
    <xsl:param name="arg1" as="element()"/>
    <xsl:param name="arg2" as="element()"/>
    <xsl:param name="arg3"/>

    <xsl:sequence select=
      "f:apply($arg1,$arg3) + f:apply($arg2,$arg3)"/>
  </xsl:function>

  <xsl:variable name="vfunTwice" as="element()">
    <f:funTwice/>
  </xsl:variable>
  <xsl:variable name="vfunFiveTimes" as="element()">
```

```
    <f:funFiveTimes/>
  </xsl:variable>

  <xsl:template match="f:funTwice" mode="f:FXSL">
    <xsl:param name="arg1"/>

    <xsl:sequence select ="2*$arg1"/>
  </xsl:template>

  <xsl:template match="f:funFiveTimes" mode="f:FXSL">
    <xsl:param name="arg1"/>

    <xsl:sequence select ="5*$arg1"/>
  </xsl:template>
</xsl:stylesheet>
```

What have we achieved so far? We managed to replace a clumsy – looking 5-line `xsl:call-template` used in the XSLT 1.0 version with a simple one-line XPath expression:

```
f:applyAndSum($vfunTwice, $vfunFiveTimes, 3)
```

However, we are still passing explicit template references as parameters – could we do better? Could we write `xsl:function`s (not templates!) that do the Twice and FiveTimes processing and then pass "something similar to" these `xsl:function`s' names as parameters to the called function?

Ideally, we wish to be able to invoke `f:applyAndSum()` in the following way:

```
f:applyAndSum(f:funTwice(), f:funFiveTimes(), 3)
```

where `funTwice()` and `funFiveTimes()` are `xsl:function`s (not templates) of one argument.

It turns out that this can be done. We write `funTwice()` and `funFiveTimes()` in the usual way:

```
<xsl:function name="f:funTwice">
 <xsl:param name="arg1"/>

 <xsl:sequence select="2*$arg1"/>
</xsl:function>

<xsl:function name="f:funFiveTimes()">
 <xsl:param name="arg1"/>

 <xsl:sequence select="5*$arg1"/>
</xsl:function>
```

But now the question is how to be able to dynamically refer to these functions? And still use their names as the reference?

The solution is to write for each of these two functions an additional overload taking zero arguments and producing a template reference to a template that invokes the function:

```
<xsl:function name="f:funTwice" as="element()">
   <f:funTwice/>
</xsl:function>

<xsl:function name="f:funFiveTimes" as="element()">
   <f:funFiveTimes/>
</xsl:function>
```

The templates that are invoked on the template references returned by `f:funTwice()` and `f:funFiveTimes()` are now really simple – they simply pass-through to the corresponding `xsl:function`:

```
<xsl:template match="f:funTwice" mode="f:FXSL">
  <xsl:param name="arg1"/>

  <xsl:sequence select ="f:funTwice($arg1)"/>
</xsl:template>

<xsl:template match="f:funFiveTimes" mode="f:FXSL">
  <xsl:param name="arg1"/>

  <xsl:sequence select ="f:funFiveTimes($arg1)"/>
```

```
</xsl:template>
```

Now, in the calling code we invoke `f:applyAndSum()` exactly in the way we wanted:

```
f:applyAndSum(f:funTwice(), f:funFiveTimes(), 3)
```

Here's the complete code for our XSLT 2.0 implementation of these higher-order `xsl:function`s:

```
<xsl:import href="../f/func-apply.xsl"/>

<xsl:template match="/">

  <xsl:sequence select=
    "f:applyAndSum(f:funTwice(), f:funFiveTimes(), 3)"/>
</xsl:template>

<xsl:function name="f:applyAndSum">
  <xsl:param name="arg1" as="element()"/>
  <xsl:param name="arg2" as="element()"/>
  <xsl:param name="arg3"/>

  <xsl:sequence select=
    "f:apply($arg1,$arg3) + f:apply($arg2,$arg3)"/>
</xsl:function>

<xsl:function name="f:funTwice" as="element()">
   <f:funTwice/>
</xsl:function>

<xsl:function name="f:funFiveTimes" as="element()">
   <f:funFiveTimes/>
</xsl:function>

<xsl:template match="f:funTwice" mode="f:FXSL">
   <xsl:param name="arg1"/>

   <xsl:sequence select ="f:funTwice($arg1)"/>
</xsl:template>

<xsl:template match="f:funFiveTimes" mode="f:FXSL">
   <xsl:param name="arg1"/>

   <xsl:sequence select ="f:funFiveTimes($arg1)"/>
</xsl:template>

<xsl:function name="f:funTwice">
   <xsl:param name="arg1"/>

   <xsl:sequence select="2*$arg1"/>
</xsl:function>

<xsl:function name="f:funFiveTimes">
   <xsl:param name="arg1"/>

   <xsl:sequence select="5*$arg1"/>
</xsl:function>
```

Note the use of the `mode` attribute: `mode="f:FXSL"` In case our template references were not in a specific mode (either no mode was specified or `mode="#default"` was used, and the `<xsl:apply-templates>` instruction to select them also didn't specify a particular mode, then it would be very easy for the template-reference-nodes to be processed by completely unexpected templates selected for processing simply because of their generality and higher import precedence, such as `<xsl:template match="node()">`.

2.4 *Standard Higher Order XPath 2.0 Functions*

The result so far is very useful – from now on we can write all our `xsl:function`s as higher-order functions in the way described above.

However, there are many functions we didn't write – the most useful ones being the standard XPath 2.0 Functions and Operators [F & O] and the standard XSLT 2.0 functions [XSLT2.0]. We could benefit greatly from being able to treat these functions as first class objects of the language in solving many real-world problems.

Fortunately, exactly the same approach can be used to create a higher-order function with the same name (but in another namespace) as any given function. The code below shows how this has been done with the `avg()` function.

```
<xsl:template match="f:avg" mode="f:FXSL">
  <xsl:param name="arg1" as="xdt:anyAtomicType*"/>

  <xsl:sequence select="f:avg($arg1)"/>
</xsl:template>

<xsl:function name="f:avg" as="element()">
  <f:avg/>
</xsl:function>

<xsl:function name="f:avg" as="xdt:anyAtomicType?">
  <xsl:param name="arg1" as="xdt:anyAtomicType*"/>

  <xsl:sequence select="avg($arg1)"/>
</xsl:function>
```

2.5 Standard Higher Order XPath 2.0 Operators

Using exactly the same approach we can make higher-order any standard XPath 2.0 Functions and Operators[F & O] operator. The code below shows how this has been done with the `op:numeric-multiply()` — the "*" operator.

```
<xsl:template match="f:multiply" mode="f:FXSL">
  <xsl:param name="arg1"/>
  <xsl:param name="arg2"/>

  <xsl:sequence select="f:mult($arg1, $arg2)"/>
</xsl:template>

<xsl:function name="f:mult" as="element()">
  <f:multiply/>
</xsl:function>

<xsl:function name="f:mult" as="item()">
  <xsl:param name="arg1" as="item()"/>
  <xsl:param name="arg2" as="item()"/>
  <xsl:sequence select="$arg1 * $arg2"/>
</xsl:function>
```

Now we can easily define as a one-line XPath expression many useful functions:

```
f:foldl(f:add(), 0, $vSeq)                  (: Sum :)

f:foldl(f:mult(), 1, $vSeq)                 (: Product :)

f:foldl(f:or(), false(),$vSeq)              (: Some true :)

f:foldl(f:and(), true(),$vSeq)              (: All true :)

f:someTrue(f:map(P(),$vSeq))                (: For some P is true :)

f:allTrue(f:map(P(),$vSeq))                 (: For all P is true :)

f:sum(f:zipWith(f:mult(), $vect1, $vect2))  (: Scalar product of
                                               two vectors :)
f:allTrue(f:zipWith(f:eq(),$vect1, $vect2)) (: Vec/tpl equality :)

f:zipWith(f:add(),$vect1, $vect2)           (: Sum of two vectors :)

f:zipWith(f:subtr(),$vect1, $vect2)         (: Diff. of 2 vectors :)
```

and this code is as compact as the equivalent Haskell code!

2.6 Higher Order XPath 2.0 Constructors

Using exactly the same approach we can make higher-order any XPath 2.0 constructor function. The code below shows how this has been done with the `xs:decimal()` constructor.

```
<xsl:template match="f:decimal" mode="f:FXSL">
   <xsl:param name="arg1" as="xdt:anyAtomicType?"/>
```

```
   <xsl:sequence select="f:decimal($arg1)"/>
</xsl:template>

<xsl:function name="f:decimal" as="element()">
   <f:decimal/>
</xsl:function>

<xsl:function name="f:decimal" as="xs:decimal?">
   <xsl:param name="arg1" as="xdt:anyAtomicType?"/>

   <xsl:sequence select="xs:decimal($arg1)"/>
</xsl:function>
```

Because of the strict type-checking in XPath 2.0 many situations arise, when in order to avoid a type error we must explicitly convert the arguments of a function to instances of the mandated type. In other cases, in order to preserve a desired degree of accuracy we may prefer to use a more strict argument type rather than the more general default type accepted by a function. For example, in many cases evaluating the following expression:

```
sum(/*/claim/claim_line/reimbursement_amount)
```

may produce more inaccurate results (because all items in the argument-sequence of `sum()` are cast to `xs:double`) than evaluating the corresponding expression where the elements to be summed are first converted to `xs:decimal`:

```
sum(f:map(f:decimal(),
          /*/claim/claim_line/reimbursement_amount)
    )
```

If the above two expressions were evaluated against the following xml document, with a given XSLT 2.0 processor such as Saxon 8.7.1, they would produce, correspondingly, `343.24999999999992` and `343.25`

```
<claim_file>
    <claim>
      <claim_line>
        <reimbursement_amount>45.00</reimbursement_amount>
      </claim_line>
      <claim_line>
        <reimbursement_amount>23.95</reimbursement_amount>
      </claim_line>
      <claim_line>
        <reimbursement_amount>56.36</reimbursement_amount>
      </claim_line>
    </claim>
    <claim>
      <claim_line>
        <reimbursement_amount>45.00</reimbursement_amount>
      </claim_line>
      <claim_line>
        <reimbursement_amount>23.95</reimbursement_amount>
      </claim_line>
      <claim_line>
        <reimbursement_amount>37.04</reimbursement_amount>
      </claim_line>
    </claim>
    <claim>
      <claim_line>
        <reimbursement_amount>45.00</reimbursement_amount>
      </claim_line>
      <claim_line>
        <reimbursement_amount>23.95</reimbursement_amount>
      </claim_line>
      <claim_line>
        <reimbursement_amount>43.00</reimbursement_amount>
      </claim_line>
    </claim>
</claim_file>
```

2.7 Higher Order Foreign Functions

The attentive reader may already have observed that the approach used to represent a non-higher order function by using an identically named higher-order wrapper can seamlessly be extended to functions defined and implemented outside the scope of XPath2.0/XSLT2.0, such as any extension function of interest.

§ 3 Representing lists in XSLT

Many functional programming design patterns involve operations on lists. Therefore, we have to choose a representation of a list in XSLT.

We represent in XSLT 1.0 a *list* of N elements `[x1, x2, …, xN]` as the following tree:

```
<list>
  <el>x1</el>
  <el>x2</el>
. . . . . .
  <el>xN</el>
</list>
```

where any names may be chosen for “list” and “el”.

As a special case, when x_i are lists themselves we get the following representation of a *list of lists*:

```
<list>
  <x1>
    <el>x11</el>
    <el>x12</el>
  . . . . . .
    <el>x1N1</el>
  </x1>
  <x2>
    <el>x21</el>
    <el>x22</el>
  . . . . . .
    <el>x2N2</el>
  </x2>
  . . . . . .
  <xM>
    <el>xM1</el>
    <el>xM2</el>
  . . . . . .
    <el>xMNM</el>
  </xM>
</list>
```

A *list of characters* is most naturally represented as a string. Therefore, for each design pattern, which operates on lists there will be two implementations – one for lists represented as node-sets and one for lists of characters represented as strings.

The XPath 2.0 Data Model [XPath2] used in XSLT 2.0 provides a new data type — the *sequence*. The sequence data type makes it possible to represent any type of list, not only list of nodes — it is the natural way of representing lists in XSLT 2.0. However, the sequence type is flat, meaning that an item of a sequence cannot be a sequence itself. Therefore, the way to represent a list of lists is still the one described for XSLT 1.0 above.

§ 4 FP Design Patterns

Based on the XSLT1.0/2.0 implementation of HOF, described above, we describe now the XSLT implementation of some of the most general FP design patterns [RalfL].

4.1 Recursion patterns

Recursion design patterns capture some of the most general and frequent uses of recursion.

4.1.1 Iteration

The `iter` function implements this design pattern. It is briefly defined in Haskell like this:

```
iter n f
     | n > 0     = f . iter (n-1) f
     | n == 0    = id
     | otherwise = error "[iter]: Negative argument!"
```

The `iter` function is defined using *guards* — boolean expressions used to express various cases in the definition of a function. Here the `"."` operator denotes *functional composition*:

```
(f.g) x = f(g x)
```

The `id` function is the identity — `id x = x`.

As defined, `iter` takes a non-negative integer `n` and a function `f` and returns another function, which is the composition of `f` with itself `n-1` times. Functional composition is another FP design pattern to be discussed later.

Here's the corresponding XSLT 2.0 implementation (note that it is based on a DVC (Divide and Conquer) approach in order to avoid stack overflow with some XSLT 2.0 processors that may not optimize tail-recursion well):

```
<!--
    Function: iter($pTimes, $pFun, $pX)
     Purpose: Iterate (compose a function with itself) N times
     Parameters:-
      $pTimes - number of times to iterate
      $pFun   - a template reference to the function that's to be
                iterated
      $pX     - an initial argument to the function
  ============================================================== -->

  <xsl:function name="f:iter">
    <xsl:param name="pTimes" as="xs:integer"/>
    <xsl:param name="pFun" as="element()"/>
    <xsl:param name="pX" />

    <xsl:sequence select=
     "if($pTimes = 0)
        then
          $pX
        else
          if($pTimes = 1)
            then
              f:apply($pFun, $pX)
            else
              if($pTimes > 1)
                then
                  for $vHalfTimes in $pTimes idiv 2 return
                    f:iter($pTimes - $vHalfTimes,
                           $pFun,
                            f:iter($vHalfTimes, $pFun, $pX)
                           )
                else
                  error((),'[iter]Error: the $pTimes argument must be
                      a positive integer or 0.')"/>
  </xsl:function>
```

We'll show `f:iter()` in an example later in this paper, when we discuss and demonstrate currying and partial application.

4.1.2 Recursion over a list (folding)

The function **sum** that computes the sum of the elements of a list can be defined as follows:

```
sum [] = 0
sum (n:ns) = n + sum ns
```

The function **product** that computes the product of the elements of a list can be defined as follows:

```
product [] = 1
product (n:ns) = n * product ns
```

There is something common and general in the above two function definitions — they define the same general operation over a list, but provide different arguments to this operation. The arguments to the general list operation are displayed in bold above.

They are a function **f** (+ and * in the described cases) that takes two arguments and an initial value (**0** and **1** in the described cases) to use as a second argument when applying this function to the first element of the list. Therefore, we can define this general operation on lists as a higher-order function:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

foldl processes a list from left to right. Its dual function, which processes a list from right to left is **foldr**:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

We can define many functions just by feeding `foldl` (or `foldr`) with appropriate functions and null elements:

```
sum = foldl add 0
product = foldl multiply 1
sometrue = foldl or false
alltrue = foldl and true
maximum = foldl1 max
minimum = foldl1 min
```

where `foldl1` is defined as `foldl`, which operates on non-empty lists, and `min(a1, a2)` is the lesser, while `max(a1, a2)` is the bigger of a pair of values.

```
append as bs = foldr (:) bs as
map f = foldr ((:).f ) [ ]
```

where **(:)** is the function, which adds an element to the front of a list.

Here's the corresponding XSLT 2.0 implementation of **foldl** and some of its useful applications:

```
<xsl:function name="f:foldl">
  <xsl:param name="pFunc" as="element()"/>
  <xsl:param name="pA0" as="item()*"/>
  <xsl:param name="pList" as="item()*"/>

  <xsl:sequence select=
         "if (empty($pList))
              then
                  $pA0
              else
                  f:foldl($pFunc,
                          f:apply($pFunc, $pA0, $pList[1]),
                          $pList[position() > 1]
                          )"/>
</xsl:function>
```

There are many useful functions that can be defined in just a one-line XPath expression using `f:foldl()`. Here are some examples:

```
f:foldl(f:add(), 0, $vSeq)                  (: Sum :)

f:foldl(f:mult(), 1, $vSeq)                 (: Product :)

f:foldl(f:or(), false(),$vSeq)              (: Some true :)

f:foldl(f:and(), true(),$vSeq)              (: All true :)
```

4.2 *Mapping of a list*

Another fundamental FP design pattern is the *mapping of a list*. The `map` function may be defined like this in Haskell:

```
map  f  xs   =  [ f x  |  x <- xs ]
```

In this definition, | means " "*such that*" and <- means "*belongs to*".

The `map` functions has two arguments — another function `f` and a list `xs`

The result of applying `map` on `f` and `xs` is a list `ys`, for which $\mathbf{y}_i = \mathbf{f}\,\mathbf{x}_i$.

The XSLT implementation is straightforward because the XPath 2.0 `for` expression can be used instead of recursion to iterate over the list.

```
<xsl:function name="f:map" as="item()*">
  <xsl:param name="pFun" as="element()"/>
  <xsl:param name="pList" as="item()*"/>

  <xsl:sequence select=
   "for $this in $pList
        return f:apply($pFun, $this)"/>
</xsl:function>
```

As an example, the following expression:

```
f:map(f:log2(), 1 to 10)
```

produces the list of 2-based logarithms of the list of numbers 1 to 10:

```
0
1
1.584962493378093
1.9999999939549516
2.321928090518739
2.58496249071745
2.807354909651913
2.9999999868509242
3.169924988315051
3.3219280785926497
```

As another example let's have a list of lists of numbers. We want to find the sum of the products of these lists.

The source xml document is:

```
<sales>
 <sale>
   <price>3.5</price>
   <quantity>2</quantity>
   <Discount>0.75</Discount>
   <Discount>0.80</Discount>
   <Discount>0.90</Discount>
 </sale>
 <sale>
   <price>3.5</price>
   <quantity>2</quantity>
   <Discount>0.75</Discount>
   <Discount>0.80</Discount>
   <Discount>0.90</Discount>
 </sale>
</sales>
```

On each `<sale>` element we need to perform two actions: get a list of all its children, get the product of these children. Finally, we must sum all results obtained this way on any <sale> element. We calculate the sum of products with the following XPath expression:

```
sum(f:map( f:compose(f:product(), f:childElements() ),
           /*/sale
         )
    )
```

In this case the result is: 7.56

The function `f:compose()` is the functional composition function – it is defined later in this article. It takes two functions as arguments and produces a new function, which is their functional composition.

We must note that XPath 2.0 provides a kind of mapping construct that can be applied on the last location-step of an XPath expression. The above expression could be re-written in a more compact form as:

```
sum(/*/*/f:product(*))
```

Unfortunately, XPath 2.0 does not provide a mapping construction for any other kind of a sequence, for example the following is illegal:

```
(1 to 10)/f:square()        (: Illegal! :)
```

In all such cases one has to use the `f:map()` function.

4.3 *Functional composition and currying (partial application)*

Some of the most important FP design patterns deal with producing new functions from existing ones.

Here we'll define functional composition and partial application.

4.3.1 Functional composition

The definition of functional composition is as follows:

Given two functions:

```
g :: a -> b   and   f :: b -> c
```

Their functional composition is defined as:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)

(f . g) x = f (g x)
```

This means that given two functions `f` and `g`, `(.)` glues them together by applying `f` on the result of applying `g`.

`(.)` is completely polymorphic, the only constraint on the types of `f` and `g` being that the type of the result of `g` must be the same as the type of the argument of `f`.

Functional composition is one of the most often used ways of producing a new function by applying two other functions in a successive pipe-like manner.

4.3.2 Curried functions

There are two ways of looking at a function that has more than one argument. For example, we could have:

```
f(x, y) = x * y      (1)
```

or

```
f x y   = x * y      (2)
```

The first expression above defines a function that takes a *pair* of arguments and produces their product.

In contrast, the second definition allows the function `f` to take its arguments one at a time. It is perfectly legal to have the expression:

```
f x
```

Not providing the second argument in this concrete case results in a new function `g` (`g = f x`) of one argument `y`, defined below:

```
g y  =  x * y
```

Defining a function as in `(2)` has the following advantages:

- A uniform style for expressing function application over one or many arguments - we write:

  ```
  f x,   f x y,   f x y z, ... etc.
  ```

- The ability to easily produce partial functions, resulting from applying a function only on the first several of its arguments. For example, we can also define the function `g` above as `(x *)` — a partial application of multiplication, in which the first argument has been bound to the value `x`.

A function as defined in `(2)` can be regarded as having just one argument `x`, and returning a function of `y`.

Functions defined as in `(2)` are called *curried* functions after Haskell Curry, who was one of the developers of the lambda calculus [ChurchA], and after whom the Haskell programming language was named.

Functions defined as in `(1)` are called uncurried. Uncurried functions can be turned into curried ones by the `curry` function, and curried functions can be turned into uncurried ones by the `uncurry` function. Here's how `curry` and `uncurry` are defined in the Haskell Prelude:

```
curry  f x y  =  f(x,y)

uncurry f (x, y) = f x y
```

4.3.3 Partial applications

A *partial application* is a function returned by any application of a curried function on the first several, but not all of its arguments. For example:

```
incr  = (1 +)

double = (2 *)
```

As defined above, `incr` is a function that adds one to its argument, and `double` is a function that multiplies its argument by `2`.

Partial application in the special case when the function is an operator is called an *operator section*. `(1 +)` and `(2 *)` above are examples of operator sections.

Using partial applications it is possible to simplify considerably many function definitions, which results in shorter, simpler, more understandable and maintainable code. For example, instead of defining `sum` and `product` as:

```
sum   xs   =  foldl (+) 0 xs

product xs =  foldl (*) 1 xs
```

it is possible to define them in an equivalent and much simpler way by omitting the last identically-named argument(s) from both sides of the definition:

```
sum      =  foldl (+) 0

product  =  foldl (*) 1
```

Producing partial applications is one of the most flexible and powerful ways of creating new functions dynamically.

4.4 *Implementation of functional composition in XSLT 2.0*

Below is the XSLT 2.0 implementation of functional composition.

```
<xsl:function name="f:compose">
  <xsl:param name="pFun1" as="element()"/>
  <xsl:param name="pFun2" as="element()"/>
  <xsl:param name="arg1"/>

  <xsl:sequence select=
   "f:apply($pFun1, f:apply($pFun2, $arg1))"/>
</xsl:function>
```

The `xsl:function f:compose()` has three parameters, two of which are the functions to be composed — `pFun1` and `pFun2` — and the third is the argument `arg1`, on which `pFun2` will be applied.

In many cases, there are more than two functions that should be successively applied, each using as its argument the result returned from the previous function. In any such case, it would be inconvenient or even impossible (e.g., the number of functions to be composed is not known in advance) to compose the functions two at a time.

Therefore, we also provide the definition of a function, which takes two arguments: a list of functions and an argument on which the last function in the list is to be applied. This function produces the result of the functional composition of all functions in the list, using the provided initial argument:

```
multiCompose x [f]  =  f x

multiCompose x [f:fs] = f (multiCompose x fs)
```

Using the `foldr` function and the function application operator `$ (f $ x = f x)`, we can define the `multiCompose` function as follows:

```
multiCompose y fs = foldr ($) y fs
```

or even simpler:

```
multiCompose y = foldr ($) y
```

The XSLT 2.0 implementation is once again very simple:

```
<xsl:function name="f:compose-flist">
  <xsl:param name="arg1" as="element()*"/> <!-- fn list -->
  <xsl:param name="arg2"/>                 <!-- starting argument -->

  <xsl:sequence select=
   "if(empty($arg1))
     then $arg2
     else
       f:apply($arg1[1],
               f:compose-flist($arg1[position() > 1], $arg2))"/>
</xsl:function>
```

Here's a test of `f:compose()` and `f:compose-flist()`:

```
<xsl:template name="initial" match="/">

  Compose:
  (*3).(*2) 3 =
  <xsl:value-of select="f:compose(f:mult(3), f:mult(2), 3)"/>

  Multi Compose:
  (*3).(*2).(*3) 2 =
  <xsl:value-of select=
   "f:compose-flist((f:mult(3), f:mult(2), f:mult(3)), 2)"/>
</xsl:template>
```

And the result is:

```
  Compose:
  (*3).(*2) 3 = 18

  Multi Compose:
  (*3).(*2).(*3) 2 = 36
```

The expressions `f:mult(3)` and `f:mult(2)` are partial applications of the multiplication operator `f:mult()`. Essentially, they are functions of one argument, which return their argument multiplied, respectively, by 3 or by 2. We will describe the XSLT 2.0 implementation of currying and partial applications in the next section.

4.5 *Implementation of currying and partial application in XSLT 2.0*

The examples in the previous section demonstrated the importance of being able to use partial applications of functions. Below, we provide an XSLT implementation of currying and partial application. Let's recall the definition of `curry`:

```
curry  f x y  =  f(x,y)
```

This means that `curry f` returns a (curried) function of two arguments, and `curry f x` returns a function of one argument.

We have implemented a more generalised version of `curry`, which accepts a function of `N` (two or more) arguments and allows the first `k` (k < N) arguments to be specified, so that the result will be a partial application of the function passed as the first argument.

Our XSLT implementation of `curry` should, therefore, do the following:

1. Keep a reference to the function `f`. This will be needed if all arguments are passed and the value of `f` (as opposed to a partial application function) must be calculated.
2. Keep a record of the number of arguments of `f`. This is necessary, to know when all arguments for the function have been specified so that the function will be evaluated.
3. Provide a reference to an internal generic `f-curry` function that will record and accumulate the passed arguments and, in case all arguments have been provided, will apply (the function being curried) `f` on them and return the result. In case not all arguments' values have yet been provided, then `f-curry` will return a variant of itself, that knows about all the arguments' values accumulated so far.

The XSLT 2.0 implementation of `f:curry` and partial application is presented in the Appendix.

The `f:curry` function takes a function (a template reference) and a second argument, which is the number of the arguments of this function. It also takes up to ten optional arguments on which the provided function (the first argument to `f:curry`) is to be partially applied.

It then builds its internal `f-curry` function to produce the result as a partial application.

The internal `f-curry` function accepts up to ten arguments.

All specified arguments are appended to an internal structure, containing elements with names "`arg`" and values — the value specified for the "`arg`" argument. For example, if `f` has 6 arguments and only the first 3 of them were specified, the `f-curry`'s internal store could look like this:

```
<f-curry:f-curry xmlns:f-curry="http://fxsl.sf.net/curry">
     <fun><f:userFun xmlns:f="http://fxsl.sf.net/"/></fun>
     <cnArgs>6</cnArgs>
     <arg t="xs:integer">10</arg>
     <arg s="">
       <e t="xs:decimal">3.1415</e>
       <e t="xs:string">Hello World</e>
       <e t="xs:integer">15</e>
     </arg>
     <arg t="xs:double">2.718e0</arg>
</f-curry:f-curry>
```

Notice that not only the values of the specified arguments are recorded, but also their types have been determined and for a sequence-argument this is done on all items of the sequence. A new FXSL function: `f:type()` is used to determine the type of any argument or (in the case of a sequence-argument) the type of its items.

Below is the most essential code that produces the result of `f:curry()` when not all arguments have been specified:

```
<-- Some code ommitted  .  .  .  .  .  .  .  .  .  .  .  . -->
  <f-curry:f-curry>
    <fun><xsl:sequence select="$pFun"/></fun>
    <cnArgs><xsl:value-of select="$pNargs"/></cnArgs>
    <xsl:sequence select="int:makeArg($arg1)"/>
    <xsl:sequence select="int:makeArg($arg2)"/>
    <xsl:sequence select="int:makeArg($arg3)"/>
    <xsl:sequence select="int:makeArg($arg4)"/>
    <xsl:sequence select="int:makeArg($arg5)"/>
    <xsl:sequence select="int:makeArg($arg6)"/>
    <xsl:sequence select="int:makeArg($arg7)"/>
    <xsl:sequence select="int:makeArg($arg8)"/>
    <xsl:sequence select="int:makeArg($arg9)"/>
  </f-curry:f-curry>
</xsl:function>

<xsl:function name="int:makeArg" as="element()">
  <xsl:param name="arg1"/>
  <arg>
    <xsl:choose>
      <xsl:when test="exists($arg1[2])">
        <xsl:attribute name="s"/>

        <xsl:for-each select="$arg1">
          <e t="{f:type(.)}"><xsl:sequence select="."/></e>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
        <xsl:attribute name="t" select="f:type($arg1)"/>
        <xsl:sequence select="$arg1"/>
      </xsl:otherwise>
    </xsl:choose>
  </arg>
</xsl:function>
```

Whenever values for all arguments are provided, the implementation instantiates the template that implements the `fun` function (the function being curried) — the template reference to it is stored in the `<fun>` element.

Then the argument values are constructed in their proper type by using the stored raw values and the type information for every argument or item of an argument-sequence. A companion function of `f:type()` – `f:Constructor()` is used in this process.

The code of the internal function `int:getArg()` that reconstructs an argument is listed below:

```
<xsl:function name="int:getArg">
  <xsl:param name="pargNode" as="element()*"/>

  <xsl:sequence select=
   "if(not($pargNode/@s))
      then
         if(not($pargNode/@t) or $pargNode/@t = 'xml:node')
            then $pargNode/node()
            else
              f:apply(f:Constructor($pargNode/@t),$pargNode/node() )
      else
        for $varg in $pargNode/e/node()
          return
             if(not($varg/../@t) or $varg/../@t = 'xml:node')
```

```
                then $varg
                else
                   f:apply(f:Constructor($varg/../@t), $varg )"/>
 </xsl:function>
```

Let's see how `f:curry` can be used. In the next example, we curry the `f:mult()` multiplication operator (introduced earlier in this paper) specifying one argument's value – `3`. The result is a function of one argument that multiplies it by `3`.

So, this expression:

```
f:apply(f:curry(f:mult(),2,3), 5)
```

produces the wanted result: `15`.

As useful as this is, we'd like to hide the use of `f:curry()` and to achieve a more elegant and compact way of expressing partial applications.

This can be elegantly accomplished by defining a new overload of `f:mult()`, which takes one argument and returns the partial application, in the following way:

```
<xsl:function name="f:mult" as="element()">
  <xsl:param name="arg1" as="item()"/>

  <xsl:sequence select="f:curry(f:mult(), 2, $arg1)"/>
</xsl:function>
```

Now we can simply write:

```
f:apply(f:mult(3), 5)
```

and obtain the same correct result.

In FXSL 2.0 the definition of all possible partial applications has been specified for all standard XPath 2.0 functions and operators, with some very few exceptions. Listed below are some examples of the power of using higher-order functions and their partial applications:

```
f:add(1)                              (: Increment               :)

f:mult(2)                             (: Double                  :)

f:map(f:add($offset), $vSeq)          (: Translation of a vector :)

f:map(f:mult($scaleFactor), $vSeq)    (: Scaling a vector        :)

f:iter($N, f:mult($k), 1)             (: Exponentiation: $k ^ $N :)

f:iterUntil(f:le(100), f:mult(2), 1)  (: The first 2^N above 100 :)
```

We'll finish this section with examples of one-liner mathematical computations, something for which XSLT has never been considered strong:

Table 1: One-liner Math Calculations

Expression	Result
`f:pow(2,5)`	`32.000000498873237`
`f:flip(f:pow(), 2, 5)`	`25.000000210097641`

`f:map(f:flip(f:pow(), 2), 1 to 10)`	`1`
	`4.00000002487061`
	`8.999999997225032`
	`16.0000000654391`
	`25.000000210097641`
	`36.00000008032039`
	`49.00000001473974`
	`64.00000003117357`
	`81.00000012870524`
	`99.99999981533682`

`f:map(f:flip(f:pow(), 10), 1 to 10)`	`1`

```
                                1024.000031977348
                                59048.99991721205
                                1.0485760215547919E6
                                9.765625412399698E6
                                6.046617668837886E7
                                2.8247524947463E8
                                1.0737418267311203E9
                                3.486784428898773E9
                                9.999999907930431E9


sum(f:map(f:flip(f:pow          1.4914341865157238E10
(),10), 1 to 10))

                                10.407835264401298
     f:pow(
       sum(f:map(f:flip(f:pow(),10), 1 to 10)),
        0.1
            )

f:log(2,4)                      0.5000000015112621

f:flip(f:log(), 2, 4)           1.9999999939549516

f:map(f:flip(f:log(),
2), 2 to 10)                            1
                                1.584962493378093
                                1.9999999939549516
                                2.321928090518739
                                2.58496249071745
                                2.807354909651913
                                2.9999999868509242
                                3.169924988315051
                                3.3219280785926497


     f:map(                             1
      f:round-half-to-even(f:sqrt(21.4
           )                    1.41
                                1.414
                                1.4142
                                1.41421
                                1.414214
                                1.4142136
                                1.41421356
                                1.414213562
                                1.4142135624
                                1.41421356237
                                1.414213562375
                                1.4142135623747
```

4.6 *Limitations of wrapping non-higher order functions*

There are just a very few limitations to the powerful approach of wrapping an existing function by an identically named higher-order wrapper:

4.6.1 *Functions with unknown number of arguments*

The standard XPath function `concat()` accepts any number of arguments. Passing a reference to this function is not useful for any practical purposes.

4.6.2 *Functions with zero number of arguments*

Functions with zero number of arguments cannot be curried. Also,

```
f:zeroArgsFun()
```

should not be confused with

```
zeroArgsFun()
```

While the first evaluates only to a reference to the function, the second evaluates to the result of the original non-higher order function.

This can actually be useful when we want to accomplish delayed (lazy) function evaluation, for example, to prepare a list of functions to be evaluated by another function.

4.6.3 *Functions having overloads*

When an `xsl:function` has overloads, a wrapper can be provided only for one of this non-higher order function. If two overloads are:

```
F(arg1, arg2, ..., argN)
```

and

```
F(arg1, arg2, ..., argK)
```

where `N > K`

and we have created a wrapper f:F() for the first overload (having N arguments), then the expression

```
f:F(arg1, arg2, ..., argK)
```

should not be confused with evaluating the second overload via the wrapper. In fact, this expression denotes the partial application of the first wrapped overload on its first K arguments.

It is still possible to provide a wrapper for the second (or any other existing) overload, but the name of every wrapper must be different (at least by namespace).

4.6.4 *Type detection for arguments of user-defined type*

The argument type-detection that `f:curry()` uses has been implemented only for the builtin XML Schema datatypes. It will not recognize the user-defined type of an argument, which can be created by an user of a Schema-Aware (SA) XSLT 2.0 processor.

§ 5 *FXSL 2.0*

This article is only a brief reflection of the work accomplished in the development of the FXSL 2.0 — a functional programming library for XSLT 2.0 [FXSL].

While it incorporates the existing FXSL 1.X library for functional programming with XSLT 1.0, FXSL 2.0 provides a new and strong support for such fundamental XPath 2.0/XSLT 2.0 features as the sequence datatype, strong typing, writing functions natively in XSLT. FXSL 2.0 makes nearly all standard XPath 2.0/XSLT 2.0 functions and operators higher-order by providing in the "`http://fxsl.sf.net/`" namespace the definition of their identically named higher-order wrappers and partial applications.

In effect, this makes XSLT 2.0 + FXSL a higher-order strongly-typed functional programming system.

Additionally, FXSL 1.X, which is incorporated in FXSL 2.0, provides an XSLT implementation of all the major FP design patterns described in this paper and, based on them, goes further to provide more specific functionality. The home site of FXSL contains materials, describing the implementation of:

- Fundamental functions on lists and trees as well as some numerical methods [Nova1].
- Functional composition, partial application and curryng, dynamic creation of new functions [Nova2].
- Generation of random numbers within a given range and with a specified distribution [Nova3].
- Trigonometric, hyperbolic-trigonometric, exponential and logarithmic functions, inverse trigonometric functions, finding the roots of continuous functions with one real variable [Nova4].

The next table summarizes the main functionality implemented in or with FXSL.

Table 2: *Main Functionality Implemented in or with FXSL*

• Higher-order functions • Functional composition • Partial application, currying • Dynamic creation of functions	• Generic iteration • Generic recursion over lists • Generic recursion over trees • Mapping, zipping, splitting, filtering of lists • Generic binary search in Ordered • Generic sort in Ordered	• Generic recursion over the characters of a string • Mapping, zipping, splitting, filtering of lists of characters (strings) • String tokenisation, trimming and reversal • Text justification

- Spelling checking and generation of correct close words
- Concordance of text corpora

- Numerical differentiation
- Numerical integration
- Limits of sequences
- Trigonometric functions, hyperbolic trigonometric functions
- Logarithmic and exponentiation functions
- Inverse trigonometric functions
- Roots of a continuous function with one real variable

- Random numbers
- Random numbers with specified distribution
- Randomisation of lists/ node-sets

- Implementation of lazy evaluation
- Implementation of a mechanism (closely matching the "Monad someType" class) for using reliably functions with side effects

§ 6 *Conclusion*

More than three years ago FXSL 1.0 challenged our understanding about XSLT programming by providing the definition and implementation of basic FP design patterns and making it much easier to solve a large class of problems that until then had been considered very difficult or not appropriate for XSLT.

The next generation of FXSL adds support for new fundamental XPath 2.0/XSLT 2.0 features such as the sequence datatype, strong typing and writing functions natively in XSLT. FXSL 2.0 makes nearly all standard XPath 2.0/XSLT 2.0 functions and operators higher-order by providing in the "`http://fxsl.sf.net/`" namespace the definition of their identically named higher-order wrappers and partial applications.

In effect, this makes XSLT 2.0 + FXSL a higher-order strongly-typed functional programming system.

Based on the even higher degree of abstraction and code reuse many challenging problems have now more compact or even one-line solutions.

§ 7 Appendix

This appendix contains the code for `f:curry()` and the auxiliary functions it uses: `f:type()` and `f:Constructor()`.

```
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:f="http://fxsl.sf.net/"
 xmlns:int="http://fxsl.sf.net/internal/curry"
 xmlns:f-curry="http://fxsl.sf.net/curry"
 exclude-result-prefixes="xs f f-curry int">
  <xsl:import href="../f/func-type.xsl"/>

  <xsl:function name="f:curry">
    <xsl:param name="pFun" as="node()"/>
    <xsl:param name="pNargs" as="xs:integer"/>
    <xsl:param name="arg1"/>

    <xsl:if test="$pNargs < 2 or $pNargs > 10">
      <xsl:message terminate="yes">
         [curry]Error: pNargs (number of arguments) of a fn to be
                       curried must be between 2 and 10 inclusive
      </xsl:message>
    </xsl:if>
    <!-- else                                    -->
    <f-curry:f-curry>
      <fun><xsl:sequence select="$pFun"/></fun>
        <cnArgs><xsl:value-of select="$pNargs"/></cnArgs>
        <xsl:sequence select="int:makeArg($arg1)"/>
    </f-curry:f-curry>
  </xsl:function>
```

```
 <xsl:function name="f:curry">
   <xsl:param name="pFun" as="node()"/>
   <xsl:param name="pNargs" as="xs:integer"/>
   <xsl:param name="arg1"/>
   <xsl:param name="arg2"/>

   <xsl:if test="$pNargs < 2 or $pNargs > 10">
     <xsl:message terminate="yes">
        [curry]Error: pNargs (number of arguments) of a fn to be
                      curried must be between 2 and 10 inclusive
     </xsl:message>
   </xsl:if>
   <xsl:if test="$pNargs < 3">
     <xsl:message terminate="yes">
        [curry]Error: pNargs (number of arguments) of a fn to be
                      curried must be greater than the number of
                      partial applications.
     </xsl:message>
   </xsl:if>
   <!-- else                                    -->
   <f-curry:f-curry>
     <fun><xsl:sequence select="$pFun"/></fun>
       <cnArgs><xsl:value-of select="$pNargs"/></cnArgs>
       <xsl:sequence select="int:makeArg($arg1)"/>
       <xsl:sequence select="int:makeArg($arg2)"/>
   </f-curry:f-curry>
 </xsl:function>

 <xsl:function name="f:curry">
   <xsl:param name="pFun" as="node()"/>
   <xsl:param name="pNargs" as="xs:integer"/>
   <xsl:param name="arg1"/>
   <xsl:param name="arg2"/>
   <xsl:param name="arg3"/>

   <xsl:if test="$pNargs < 2 or $pNargs > 10">
     <xsl:message terminate="yes">
        [curry]Error: pNargs (number of arguments) of a fn to be
                      curried must be between 2 and 10 inclusive
     </xsl:message>
   </xsl:if>
   <xsl:if test="$pNargs < 4">
     <xsl:message terminate="yes">
        [curry]Error: pNargs (number of arguments) of a fn to be
                      curried must be greater than the number of
                      partial applications.
     </xsl:message>
   </xsl:if>
   <!-- else                                    -->
   <f-curry:f-curry>
     <fun><xsl:sequence select="$pFun"/></fun>
       <cnArgs><xsl:value-of select="$pNargs"/></cnArgs>
       <xsl:sequence select="int:makeArg($arg1)"/>
       <xsl:sequence select="int:makeArg($arg2)"/>
       <xsl:sequence select="int:makeArg($arg3)"/>
   </f-curry:f-curry>
 </xsl:function>

<!—Equivalent code for overloads of f:curry with up to 9 args
   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
 -->


 <xsl:function name="int:makeArg" as="element()">
   <xsl:param name="arg1"/>
   <arg>
     <xsl:choose>
       <xsl:when test="exists($arg1[2])">
         <xsl:attribute name="s"/>

         <xsl:for-each select="$arg1">
           <e t="{f:type(.)}"><xsl:sequence select="."/></e>
         </xsl:for-each>
       </xsl:when>
       <xsl:otherwise>
         <xsl:attribute name="t" select="f:type($arg1)"/>
         <xsl:sequence select="$arg1"/>
       </xsl:otherwise>
     </xsl:choose>
   </arg>
 </xsl:function>

 <xsl:function name="int:getArg">
   <xsl:param name="pargNode" as="element()*"/>

   <xsl:sequence select=
    "if(not($pargNode/@s))
       then
          if(not($pargNode/@t) or $pargNode/@t = 'xml:node')
```

```
            then $pargNode/node()
            else
              f:apply(f:Constructor($pargNode/@t),
                      $pargNode/node() )
      else
        for $varg in $pargNode/e/node()
          return
            if(not($varg/../@t) or $varg/../@t = 'xml:node')
               then $varg
               else
                  f:apply(f:Constructor($varg/../@t), $varg )"/>
</xsl:function>

<xsl:template match="f-curry:*" mode="f:FXSL">
  <xsl:param name="arg1" select="()"/>
  <xsl:param name="arg2" select="()"/>
  <xsl:param name="arg3" select="()"/>
  <xsl:param name="arg4" select="()"/>
  <xsl:param name="arg5" select="()"/>
  <xsl:param name="arg6" select="()"/>
  <xsl:param name="arg7" select="()"/>
  <xsl:param name="arg8" select="()"/>
  <xsl:param name="arg9" select="()"/>

  <xsl:variable name="vLastArg" as="xs:integer"
   select="if(exists($arg9)) then 9
            else if(exists($arg8)) then 8
            else if(exists($arg7)) then 7
            else if(exists($arg6)) then 6
            else if(exists($arg5)) then 5
            else if(exists($arg4)) then 4
            else if(exists($arg3)) then 3
            else if(exists($arg2)) then 2
            else if(exists($arg1)) then 1
            else
             error((),
              '[Error]Currying: At least one non-empty argument must be provided to a curried
              function.')"/>

   <xsl:variable name="vTotalArgs" as="xs:integer"
    select="$vLastArg + count(arg)"/>

    <xsl:choose>
      <xsl:when test="$vTotalArgs > 10">
        <xsl:sequence select=
        "error((), '[Error]Currying: More than 10 arguments provided')"/>
      </xsl:when>

      <xsl:otherwise>
        <xsl:variable name="vCurried" as="element()">
          <f-curry:f-curry>
            <xsl:copy-of select="*"/>
            <xsl:for-each select="1 to $vLastArg">
              <xsl:sequence select=
              "int:makeArg(if(. eq 1)then $arg1
                         else if(. eq 2) then $arg2
                         else if(. eq 3) then $arg3
                         else if(. eq 4) then $arg4
                         else if(. eq 5) then $arg5
                         else if(. eq 6) then $arg6
                         else if(. eq 7) then $arg7
                         else if(. eq 8) then $arg8
                         else if(. eq 9) then $arg9
                         else(error((),'[Error]Currying: Must not happen 1')))"/>
            </xsl:for-each>
          </f-curry:f-curry>
        </xsl:variable>

        <xsl:choose>
        <xsl:when test="$vTotalArgs eq xs:integer(cnArgs)">
          <xsl:apply-templates select="fun/*[1]" mode="f:FXSL">
            <xsl:with-param name="arg1"
                 select="int:getArg($vCurried/arg[1])"/>
            <xsl:with-param name="arg2"
                 select="int:getArg($vCurried/arg[2])"/>
            <xsl:with-param name="arg3"
                 select="int:getArg($vCurried/arg[3])"/>
            <xsl:with-param name="arg4"
                 select="int:getArg($vCurried/arg[4])"/>
            <xsl:with-param name="arg5"
                 select="int:getArg($vCurried/arg[5])"/>
            <xsl:with-param name="arg6"
                 select="int:getArg($vCurried/arg[6])"/>
            <xsl:with-param name="arg7"
                 select="int:getArg($vCurried/arg[7])"/>
            <xsl:with-param name="arg8"
                 select="int:getArg($vCurried/arg[8])"/>
            <xsl:with-param name="arg9"
                 select="int:getArg($vCurried/arg[9])"/>
            <xsl:with-param name="arg10"
```

```
                    select="int:getArg($vCurried/arg[10])"/>
            </xsl:apply-templates>
          </xsl:when>
          <xsl:otherwise>
            <xsl:sequence select="$vCurried"/>
          </xsl:otherwise>
          </xsl:choose>
        </xsl:otherwise>
      </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

```
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:xdt="http://www.w3.org/2005/04/xpath-datatypes"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:f="http://fxsl.sf.net/"
 exclude-result-prefixes="f xs xdt">
  <xsl:import href="../f/func-XpathConstructors.xsl"/>

  <xsl:output omit-xml-declaration="yes"/>

  <xsl:key name="kConstructor" match="*" use="@t"/>

  <xsl:variable name="f:vTypeConstructors">
    <f:unsignedByte t="xs:unsignedByte"/>
    <f:unsignedShort t="xs:unsignedShort"/>
    <f:unsignedInt t="xs:unsignedInt"/>
    <f:unsignedLong t="xs:unsignedLong"/>
    <f:positiveInteger t="xs:positiveInteger"/>
    <f:nonNegativeInteger t="xs:nonNegativeInteger"/>
    <f:nonPositiveInteger t="xs:nonPositiveInteger"/>
    <f:byte t="xs:byte"/>
    <f:short t="xs:short"/>
    <f:int t="xs:int"/>
    <f:long t="xs:long"/>
    <f:integer t="xs:integer"/>
    <f:decimal t="xs:decimal"/>
    <f:double t="xs:double"/>
    <f:float t="xs:float"/>
    <f:NMTOKEN t="xs:NMTOKEN"/>
    <f:NMTOKENS t="xs:NMTOKENS"/>
    <f:ENTITIES t="xs:ENTITIES"/>
    <f:ENTITY t="xs:ENTITY"/>
    <f:IDREF t="xs:IDREF"/>
    <f:IDREFS t="xs:IDREFS"/>
    <f:ID t="xs:ID"/>
    <f:NCName t="xs:NCName"/>
    <f:Name t="xs:Name"/>
    <f:language t="xs:language"/>
    <f:token t="xs:token"/>
    <f:normalizedString t="xs:normalizedString"/>
    <f:boolean t="xs:boolean"/>
    <f:duration t="xs:duration"/>
    <f:dateTime t="xs:dateTime"/>
    <f:time t="xs:time"/>
    <f:date t="xs:date"/>
    <f:gYearMonth t="xs:gYearMonth"/>
    <f:gYear t="xs:gYear"/>
    <f:gMonthDay t="xs:gMonthDay"/>
    <f:gDay t="xs:gDay"/>
    <f:gMonth t="xs:gMonth"/>
    <f:base64Binary t="xs:base64Binary"/>
    <f:hexBinary t="xs:hexBinary"/>
    <f:anyURI t="xs:anyURI"/>
    <f:QName t="xs:QName"/>
    <f:NOTATION t="xs:NOTATION"/>
    <f:string t="xs:string"/>
    <f:yearMonthDuration t="xdt:yearMonthDuration"/>
    <f:dayTimeDuration t="xdt:dayTimeDuration"/>
  </xsl:variable>

  <xsl:function name="f:Constructor" as="element()">
    <xsl:param name="pTypename" as="xs:string"/>

    <xsl:sequence select="key('kConstructor', $pTypename,$f:vTypeConstructors)"/>
  </xsl:function>

  <xsl:function name="f:typeConstructor" as="element()">
    <xsl:param name="pThis"/>

    <xsl:sequence select="key('kConstructor', f:type($pThis),$f:vTypeConstructors)"/>
  </xsl:function>

  <xsl:function name="f:type" as="xs:string">
    <xsl:param name="pThis"/>
```

```
    <xsl:choose>
      <xsl:when test="$pThis instance of xs:decimal">
        <xsl:choose>
          <xsl:when use-when="system-property('xsl:is-schema-aware')='yes'"  test="true()">
            <xsl:choose>
<!--         Not supported by a Basic XSLT Processor -->
               <xsl:when test="$pThis instance of xs:unsignedByte">xs:unsignedByte</xsl:when>
               <xsl:when test="$pThis instance of xs:unsignedShort">xs:unsignedShort</xsl:when>
               <xsl:when test="$pThis instance of xs:unsignedInt">xs:unsignedInt</xsl:when>
               <xsl:when test="$pThis instance of xs:unsignedLong">xs:unsignedLong</xsl:when>

               <xsl:when test="$pThis instance of xs:positiveInteger">xs:positiveInteger</xsl:when>
               <xsl:when test="$pThis instance of xs:nonNegativeInteger">xs:nonNegativeInteger
               </xsl:when>

               <xsl:when test="$pThis instance of xs:negativeInteger">xs:negativeInteger</xsl:when>
               <xsl:when test="$pThis instance of xs:nonPositiveInteger">xs:nonPositiveInteger
               </xsl:when>

               <xsl:when test="$pThis instance of xs:byte">xs:byte</xsl:when>
               <xsl:when test="$pThis instance of xs:short">xs:short</xsl:when>
               <xsl:when test="$pThis instance of xs:int">xs:int</xsl:when>
               <xsl:when test="$pThis instance of xs:long">xs:long</xsl:when>
            </xsl:choose>
<!--       End of SA only types  -->
          </xsl:when>
          <xsl:when test="$pThis instance of xs:integer">xs:integer</xsl:when>

          <xsl:otherwise>xs:decimal</xsl:otherwise>

        </xsl:choose>
      </xsl:when>

      <xsl:when test="$pThis instance of xs:double">xs:double</xsl:when>
      <xsl:when test="$pThis instance of xs:float">xs:float</xsl:when>

      <xsl:when test="$pThis instance of xs:string">
        <xsl:choose>
          <xsl:when use-when="system-property('xsl:is-schema-aware')='yes'"  test="true()">
<!--         Not supported by a Basic XSLT Processor -->
            <xsl:choose>
              <xsl:when test="$pThis instance of xs:NMTOKEN">xs:NMTOKEN</xsl:when>
              <xsl:when test="$pThis instance of xs:ENTITY">xs:ENTITY</xsl:when>
              <xsl:when test="$pThis instance of xs:IDREF">xs:IDREF</xsl:when>

              <!-- TODO: What to do with list simple types?
              <xsl:when test="$pThis instance of xs:NMTOKEN+">xs:NMTOKENS</xsl:when>
              <xsl:when test="$pThis instance of xs:ENTITY+">xs:ENTITIES</xsl:when>
              <xsl:when test="$pThis instance of xs:IDREF+">xs:IDREFS</xsl:when>
              -->

              <xsl:when test="$pThis instance of xs:ID">xs:ID</xsl:when>
              <xsl:when test="$pThis instance of xs:NCName">xs:NCName</xsl:when>
              <xsl:when test="$pThis instance of xs:Name">xs:Name</xsl:when>

              <xsl:when test="$pThis instance of xs:language">xs:language</xsl:when>
              <xsl:when test="$pThis instance of xs:token">xs:token</xsl:when>
              <xsl:when test="$pThis instance of xs:normalizedString">xs:normalizedString
              </xsl:when>
              <xsl:otherwise>xs:string</xsl:otherwise>
            </xsl:choose>
          </xsl:when>
          <xsl:when test="true()">xs:string</xsl:when>
        </xsl:choose>
      </xsl:when>

      <xsl:when test="$pThis instance of xs:boolean">xs:boolean</xsl:when>

      <xsl:when test="$pThis instance of xs:duration">xs:duration</xsl:when>

      <xsl:when test="$pThis instance of xs:dateTime">xs:dateTime</xsl:when>

      <xsl:when test="$pThis instance of xs:time">xs:time</xsl:when>

      <xsl:when test="$pThis instance of xs:date">xs:date</xsl:when>

      <xsl:when test="$pThis instance of xs:gYearMonth">xs:gYearMonth</xsl:when>

      <xsl:when test="$pThis instance of xs:gYear">xs:gYear</xsl:when>

      <xsl:when test="$pThis instance of xs:gMonthDay">xs:gMonthDay</xsl:when>

      <xsl:when test="$pThis instance of xs:gDay">xs:gDay</xsl:when>

      <xsl:when test="$pThis instance of xs:gMonth">xs:gMonth</xsl:when>

      <xsl:when test="$pThis instance of xdt:yearMonthDuration">xdt:yearMonthDuration</xsl:when>

      <xsl:when test="$pThis instance of xdt:dayTimeDuration">xdt:dayTimeDuration</xsl:when>
```

```
      <xsl:when test="$pThis instance of xs:base64Binary">xs:base64Binary</xsl:when>

      <xsl:when test="$pThis instance of xs:hexBinary">xs:hexBinary</xsl:when>

      <xsl:when test="$pThis instance of xs:anyURI">xs:anyURI</xsl:when>

      <xsl:when test="$pThis instance of xs:QName">xs:QName</xsl:when>

      <xsl:when use-when="system-property('xsl:is-schema-aware')='yes'"
           test="$pThis instance of xs:NOTATION">xs:NOTATION</xsl:when>

<!--
      <xsl:when test="$pThis instance of xdt:untypedAtomic">xdt:untypedAtomic</xsl:when>
      <xsl:otherwise>Unknown xdt:untypedAtomic</xsl:otherwise>
-->
      <xsl:when test="$pThis[1] instance of node()">xml:node</xsl:when>
      <xsl:otherwise>xs:string</xsl:otherwise>
    </xsl:choose>
  </xsl:function>
</xsl:stylesheet>
```

Bibliography

[ChurchA] Church, A., *The Calculi of Lambda Conversion* , Princeton, NJ: Princeton University Press, 1941

[F & O] *XQuery 1.0 and XPath 2.0 Functions and Operators*, Ashok Malhotra, Jim Melton, and Norman Walsh, Editors. World Wide Web Consortium, 3 Nov 2005, At **http://www.w3.org/TR/xpath-functions/**.

[FXSL] *The FXSL Functional Programming Library for XSLT* , At **http://fxsl.sourceforge.net/**

[HainesC] Haines, Correy, *Personal Communication on Template Referencing* , 2000

[JonesSP] Jones, Simon P., *Haskell 98 Language and Libraries – The Revised Report* , Cambridge University Press 2001

[Kay] Kay, Michael H., *What Kind of Language is XSLT*, At **http://www-106.ibm.com/developerworks/xml/library/x-xslt**.

[Nova1] Novatchev, Dimitre, *The Functional Programming Language XSLT - A proof through examples* , At **http://fxsl.sourceforge.net/articles/FuncProg/Functional%20Programming.html**

[Nova2] Novatchev, Dimitre, *Dynamic Functions using FXSL: Composition, Partial Applications and Lambda Expressions* , At **http://fxsl.sourceforge.net/articles/PartialApps/Partial%20Applications.html**

[Nova3] Novatchev, Dimitre, *Casting the Dice with FXSL: Random Number Generation Functions in XSLT* , At **http://fxsl.sourceforge.net/articles/Random/Casting%20the%20Dice%20with%20FXSL-htm.htm**

[Nova4] Novatchev, Dimitre, *An XSL Calculator: The Math Modules of FXSL* , At **http://fxsl.sourceforge.net/articles/xslCalculator/The%20FXSL%20Calculator.html**

[Nova5] Novatchev, Dimitre, *Functional programming in XSLT using the FXSL library*, In *Proc. Of the Extreme Markup Languages Conference 2003*, At **http://www.mulberrytech.com/Extreme/Proceedings/html/2003/Novatchev01/EML2003Novatchev01.html**.

[RalfL] Ralf Lämmel and Joost Visser, *Design Patterns for Functional Strategic Programming* , In Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02, **http://www.cwi.nl/~ralf/dp-sf.pdf**

[ThompSJ] Thompson, Simon J., *Haskell, The Craft of Functional Programming, Second Edition* , Addison-Wesley, 1999

[XPath2] *XML Path Language (XPath) 2.0*, Don Chamberlin , Anders Berglund, Scott Boag, et. al., Editors. World Wide Web Consortium, 3 Nov 2005. At **http://www.w3.org/TR/xpath20/**.

[XSLT1.0] *XSL Transformations (XSLT) Version 1.0*, James Clark, Editor. World Wide Web Consortium, 16 Nov 1999. At **http://www.w3.org/TR/xslt**.

[XSLT2.0] W3C (Draft), *XSL Transformations (XSLT) Version 2.0*, Michael Kay, Editor, W3C Candidate Recommendation, 3 November 2005, At **http://www.w3.org/TR/xpath-functions/**.

The Author

Dimitre Novatchev
Microsoft

Dimitre Novatchev is known as the developer of the FXSL functional programming library for XSLT, the popular XPath Visualizer tool and EXSLT for MSXML4. He works in the Data Programmability team at Microsoft.

Extreme Markup Languages 2006®
Montréal, Québec, August 7-11, 2006
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.