Extreme Markup Languages 2003

Montréal, Québec
August 4-8, 2003

Functional programming in XSLT using the FXSL library

Dimitre Novatchev
IAEA

Abstract

Described is the implementation in XSLT of some major functional programming design patterns:

- Higher-order functions (HOF)
- Recursive iteration
- Primitive recursion (folding) over lists and trees
- Mapping of lists
- Functional composition
- Partial application and currying
- Dynamic creation of a new function

The author argues that using a functional programming library that supports these design patterns in XSLT makes programming easier and more effective by increasing the level of abstraction and code reuse.

Functional programming in XSLT using the FXSL library

Table of Contents

Functional programming in XSLT using the FXSL library

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.

It is possible to implement higher-order functions in XSLT, and the purpose of this article is to describe this and to demonstrate the implementation of some of the most general FP design patterns in XSLT. First let's start with a few definitions.

1.1 *Basic definitions*

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], 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 an empty list.

```
xs
```

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

```
x:xs
```

is an operation which prepends an element `x` in front of a list `xs`.

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

1. 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]
   ```

2. Another example is ***fold***

```
fold f z0 [ ]  = z0
fold f zo x:xs = fold  (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.

On the other side, without the facility of higher-order functions, XSLT programmers have to write almost the same recursive processing code repeatedly (*e.g.*, calculating product, maximum, minimum, etc.) over and over again. Not only does this cause less than optimum productivity (*e.g.*, number of lines of code per hour) but the probability of making mistakes is about the same at the time of every rewrite.

§ 2 Higher-order functions and XSLT

A simple analysis shows that there's nothing like higher-order functions in XSLT. Even the notion of writing one own's function is not supported natively. Therefore, as a start, we have to define what we shall understand to be a 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.

2.1 *Functions in XSLT*

XSLT 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 named templates. 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.

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. 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*":

Figure 1: File: example1.xsl

```
<xsl:stylesheet version = "1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<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 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; and then, as a result, produces the sum of the outputs of these instantiated templates.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
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 called the template named "**mySum**", passing to it two *references to templates* that accept a **pX** parameter and produce something out of it. The "**mySum**" 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 *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 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.

XSLT 1.0 does not provide for an efficient list implementation. In XSLT 2.0 [XSLT 2 WD], it would be possible to represent lists as sequences and to implement adding a new element at the front of the list as an O(1) operation, sharing the common tail of lists.

§ 3 FP design patterns

Based on the XSLT implementation of HOF, described above, we can implement now some of the most general FP design patterns [RalfL].

3.1 *Recursion patterns*

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

3.1.1 Iteration

Very frequently we need to perform an operation N times, or perform an operation, take the result, and perform the same operation on it, take the result, and perform the same operation on it, …, and so on N times.

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 implementation:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:vendor="urn:schemas-microsoft-com:xslt"
exclude-result-prefixes="xsl vendor">

  <!-- this implements functional power composition,
       that is f(f(...f(x)))))...)
```

```
       f composed with itself n - 1 times -->

  <xsl:template name="iter">
    <xsl:param name="pTimes" select="0"/>
    <xsl:param name="pFun" select="/.."/>
    <xsl:param name="pX" />

    <xsl:choose>
      <xsl:when test="$pTimes = 0" >
        <xsl:copy-of select="$pX"/>
      </xsl:when>
      <xsl:when test="$pTimes = 1">
        <xsl:apply-templates select="$pFun">
          <xsl:with-param name="arg1" select="$pX"/>
          </xsl:apply-templates>
      </xsl:when>
      <xsl:when test="$pTimes > 1">
        <xsl:variable name="vHalfTimes"
                      select="floor($pTimes div 2)"/>
        <xsl:variable name="vHalfIters">
          <xsl:call-template name="iter">
            <xsl:with-param name="pTimes" select="$vHalfTimes"/>
            <xsl:with-param name="pFun" select="$pFun"/>
            <xsl:with-param name="pX" select="$pX"/>
          </xsl:call-template>
        </xsl:variable>

          <xsl:call-template name="iter">
            <xsl:with-param name="pTimes"
                            select="$pTimes - $vHalfTimes"/>
            <xsl:with-param name="pFun" select="$pFun"/>
            <xsl:with-param name="pX"
                            select="vendor:node-set($vHalfIters)"/>
          </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:message>[iter]Error: the $pTimes argument must be
                     a positive integer or 0.
        </xsl:message>
      </xsl:otherwise>
    </xsl:choose>

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

Let's use `iter` to solve the following problem: "I want to create `N` 'input' elements, where `N` is a parameter".

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:addNode="f:addNode" exclude-result-prefixes="addNode">

<xsl:import href="iter.xsl"/>
<xsl:output omit-xml-declaration="yes" indent="yes"/>

<xsl:param name="N" select="10"/>

  <xsl:template match="/">
    <xsl:call-template name="iter">
      <xsl:with-param name="pFun" select="document('')/*/addNode:*[1]"/>
      <xsl:with-param name="pTimes" select="$N"/>
    </xsl:call-template>
  </xsl:template>

  <addNode:addNode/>
  <xsl:template match="addNode:*">
    <xsl:param name="arg1"/>
    <xsl:copy-of select="$arg1"/> <!-- The result so far -->
    <input type="text"/>
  </xsl:template>
</xsl:stylesheet>
```

3.1.2 *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 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 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 implementation of **foldl** and some of its useful applications:

Figure 2: foldl.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:vendor="http://icl.com/saxon">

    <xsl:template name="foldl">
      <xsl:param name="pFunc" select="/.."/>
      <xsl:param name="pA0"/>
      <xsl:param name="pList" select="/.."/>

      <xsl:choose>
         <xsl:when test="not($pList)">
            <xsl:copy-of select="$pA0"/>
         </xsl:when>
         <xsl:otherwise>

            <xsl:variable name="vFunResult">
              <xsl:apply-templates select="$pFunc[1]">
                <xsl:with-param name="arg0"
                                select="$pFunc[position() > 1]"/>
                <xsl:with-param name="arg1" select="$pA0"/>
                <xsl:with-param name="arg2" select="$pList[1]"/>
              </xsl:apply-templates>
            </xsl:variable>
```

```
            <xsl:call-template name="foldl">
            <xsl:with-param name="pFunc" select="$pFunc"/>
            <xsl:with-param name="pList"
                            select="$pList[position() > 1]"/>
            <xsl:with-param name="pA0"
                            select="vendor:node-set($vFunResult)/node()"/>

            </xsl:call-template>
          </xsl:otherwise>
        </xsl:choose>

    </xsl:template>

</xsl:stylesheet>
```

Figure 3: sum.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:sum-fold-func="sum-fold-func"
exclude-result-prefixes="xsl sum-fold-func">

   <xsl:import href="foldl.xsl"/>

    <xsl:template name="sum">
      <xsl:param name="pList" select="/.."/>

      <xsl:variable name="sum-fold-func:vFoldFun"
                    select="document('')/*/sum-fold-func:*[1]"/>

      <xsl:call-template name="foldl">
        <xsl:with-param name="pFunc" select="$sum-fold-func:vFoldFun"/>
        <xsl:with-param name="pList" select="$pList"/>
        <xsl:with-param name="pA0" select="0"/>
      </xsl:call-template>
    </xsl:template>

    <sum-fold-func:sum-fold-func/>
    <xsl:template name="add" match="sum-fold-func:*">
         <xsl:param name="arg1" select="0"/>
         <xsl:param name="arg2" select="0"/>

         <xsl:value-of select="$arg1 + $arg2"/>
    </xsl:template>
</xsl:stylesheet>
```

Figure 4: product.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:prod-fold-func="prod-fold-func"
exclude-result-prefixes="xsl prod-fold-func">

   <xsl:import href="foldl.xsl"/>

    <xsl:template name="product">
      <xsl:param name="pList" select="/.."/>

      <xsl:variable name="prod-fold-func:vFoldFun"
                    select="document('')/*/prod-fold-func:*[1]"/>

      <xsl:call-template name="foldl">
        <xsl:with-param name="pFunc" select="$prod-fold-func:vFoldFun"/>
        <xsl:with-param name="pList" select="$pList"/>
        <xsl:with-param name="pA0" select="1"/>
      </xsl:call-template>
    </xsl:template>

    <prod-fold-func:prod-fold-func/>
    <xsl:template name="multiply" match="prod-fold-func:*">
         <xsl:param name="arg1" select="0"/>
         <xsl:param name="arg2" select="0"/>

         <xsl:value-of select="$arg1 * $arg2"/>
```

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

As we can see, all functions using `foldl` are very simple because they do not implement any recursive processing. The recursion is implemented in `foldl` once and forever. The use of functions like `foldl` and `foldr` demonstrates that a library of generic FP design patterns provides a powerful infrastructure for efficient XSLT programming through sharing and reuse.

3.2 *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 `xsl:for-each` instruction can be used instead of recursion to iterate over the list.

Figure 5: map.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template name="map">
      <xsl:param name="pFun" select="/.."/>
      <xsl:param name="pList1" select="/.."/>

      <xsl:for-each select="$pList1">
        <xsl:copy>
          <xsl:apply-templates select="$pFun">
            <xsl:with-param name="arg1" select="."/>
          </xsl:apply-templates>
        </xsl:copy>
      </xsl:for-each>
    </xsl:template>

</xsl:stylesheet>
```

Using the map function one can easily solve a variety of tasks such as: "produce a list of numbers, every one of which is the double of its corresponding number from the original list", or "produce a list containing the squares of the elements of another list", etc.

As an 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:

Figure 6: sales.xml

```
<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>
```

This is a list of sales, and each sale is a list of numbers — a price, a quantity, and zero or more Discounts. The amount paid for each sale is the product of price, quantity, and all the Discounts. We want to find the total amount paid for all sales.

Figure 7: sumProducts.xsl

```
<xsl:stylesheet version = "1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:vendor="http://icl.com/saxon"
xmlns:test-map-product="test-map-product"
exclude-result-prefixes = "xsl test-map-product">

  <xsl:import href = "sum.xsl" />
  <xsl:import href = "map.xsl" />
  <xsl:import href = "product.xsl" />

  <xsl:output method = "text" />

  <xsl:template match = "/" >
    <!-- Get: map product /sales/sale -->
    <xsl:variable name = "vSalesTotals" >
      <xsl:variable name = "vTestMap"
           select = "document('')/*/test-map-product:*[1]" />
      <xsl:call-template name = "map" >
        <xsl:with-param name = "pFun" select = "$vTestMap" />
        <xsl:with-param name = "pList1" select = "/sales/sale" />
      </xsl:call-template>
    </xsl:variable>
    <!-- Get sum map product /sales/sale -->
    <xsl:call-template name = "sum" >
      <xsl:with-param name = "pList"
                select = "vendor:node-set($vSalesTotals)/*" />
    </xsl:call-template>
  </xsl:template>

  <test-map-product:test-map-product/>
  <xsl:template name = "makeproduct"
                match = "test-map-product:*" >
    <xsl:param name = "arg1" />
    <xsl:call-template name = "product" >
      <xsl:with-param name = "pList" select = "$arg1/*" />
    </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

The result is: `7.56000000000000005`

3.3 *Functional composition, partial application(currying), and lambda expressions*

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.

3.3.1 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.

3.3.2 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` of one argument (`f x = g`), defined below:

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

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

- A uniform form of expressing function application over one or many arguments:

  ```
  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
```

3.3.3 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 dynamically new functions.

3.3.4 Lambda expressions

A lambda expression (also known as anonymous function) is a concept derived directly from Church's lambda calculus [ChurchA]. The following are examples of lambda expressions:

```
\x -> x + 1
\x y  ->  x * y
```

The first expression above defines a (anonymous) function, which increments its argument.

The second expression above defines a (anonymous) function, which calculates the product of its two arguments.

Any partial application can be expressed as a lambda expression. For example:

```
(f x ) y =  \y -> (f x) y
```

expresses the partial application of `f x`.

The reverse is not true — lambda expressions are more powerful and can define functions, which are not partial applications. For example:

```
\x -> f x y
```

is not a partial application, because it is a function of the first argument `x`, while the second argument (not the first!) is bound to the value `y`.

Lambda expressions are useful in all cases when it is not necessary to name a function (*e.g.*, when the function is to be used only locally, or is dynamically created) but just to apply it. An example is often the case with functions to be used with the `map` function:

```
map (\x -> x+3/3) xs
```

3.4 *Implementation of functional composition in XSLT*

Below is the XSLT implementation of functional composition. The named template compose has three parameters, two of which are the functions to be composed — `pFun1` and `pFun2` — and the third is the argument `pArg1`, on which `pFun2` will be applied.

Figure 8: compose.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon">

  <xsl:template name="compose">
    <xsl:param name="pFun1" select="/.."/>
    <xsl:param name="pFun2" select="/.."/>
    <xsl:param name="pArg1"/>

    <xsl:variable name="vrtfFun2">
      <xsl:apply-templates select="$pFun2">
        <xsl:with-param name="pArg1" select="$pArg1"/>
      </xsl:apply-templates>
    </xsl:variable>

    <xsl:apply-templates select="$pFun1">
      <xsl:with-param name="pArg1"
       select="saxon:node-set($vrtfFun2)/node()"/>
    </xsl:apply-templates>

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

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 implementation is once again very simple:

Figure 9: compose-flist.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon">

  <xsl:template name="compose-flist">
    <xsl:param name="pFunList" select="/.."/>
    <xsl:param name="pArg1"/>

    <xsl:choose>
      <xsl:when test="not($pFunList)">
        <xsl:copy-of select="$pArg1"/>
      </xsl:when>
```

```
      <xsl:otherwise>
      <xsl:variable name="vrtfFunRest">
        <xsl:call-template name="compose-flist">
          <xsl:with-param name="pFunList"
                 select="$pFunList[position() > 1]"/>
          <xsl:with-param name="pArg1" select="$pArg1"/>
        </xsl:call-template>
      </xsl:variable>

      <xsl:apply-templates select="$pFunList[1]">
        <xsl:with-param name="pArg1"
             select="saxon:node-set($vrtfFunRest)/node()"/>
      </xsl:apply-templates>
      </xsl:otherwise>
    </xsl:choose>

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

Here's a test of `compose` and `compose-flist`.

Figure 10: testCompose.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:myFun1="f:myFun1"
xmlns:myFun2="f:myFun2"
xmlns:saxon="http://icl.com/saxon"
exclude-result-prefixes="xsl saxon myFun1 myFun2">

  <xsl:import href="compose.xsl"/>
  <xsl:import href="compose-flist.xsl"/>

  <!-- to be applied on any xml source -->

  <xsl:output method="text"/>


  <xsl:template match="/">

    <xsl:variable name="vFun1" select="document('')/*/myFun1:*[1]"/>
    <xsl:variable name="vFun2" select="document('')/*/myFun2:*[1]"/>
    Compose:
    (*3).(*2) 3 =  <xsl:text/>
    <xsl:call-template name="compose">
      <xsl:with-param name="pFun1" select="$vFun1"/>
      <xsl:with-param name="pFun2" select="$vFun2"/>
      <xsl:with-param name="pArg1" select="3"/>
    </xsl:call-template>

    <xsl:variable name="vrtfParam">
      <xsl:copy-of select="$vFun1"/>
      <xsl:copy-of select="$vFun2"/>
      <xsl:copy-of select="$vFun1"/>
    </xsl:variable>

    Multi Compose:
    (*3).(*2).(*3) 2 =  <xsl:text/>
    <xsl:call-template name="compose-flist">
      <xsl:with-param name="pFunList"
       select="saxon:node-set($vrtfParam)/*"/>
      <xsl:with-param name="pArg1" select="2"/>
    </xsl:call-template>
  </xsl:template>

  <myFun1:myFun1/>
  <xsl:template match="myFun1:*">
    <xsl:param name="pArg1"/>

    <xsl:value-of select="3 * $pArg1"/>
  </xsl:template>

  <myFun2:myFun2/>
  <xsl:template match="myFun2:*">
    <xsl:param name="pArg1"/>

    <xsl:value-of select="2 * $pArg1"/>
  </xsl:template>
</xsl:stylesheet>
```

The result of applying this transformation to any xml source (ignored) is:

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

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

The next example is strongly based on requirements that originated from routine, practical XSLT programming.

We need to implement a `trim` function, which accepts a string and produces its content stripped off any leading or trailing white-space characters.

A definition of `trim` in Haskell might look like this:

```
trim      :: [Char] -> [Char]
trim      = applyTwice (reverse . trim1)
            where  trim1 = dropWhile (‘elem‘ delim)
                   delim    = [’ ’, ’\t’, ’\n’, ’\r’]
                   applyTwice f = f . f
```

What this says is that we are actually doing the following:

- `trim1` (trim the start of) the string.
- `reverse` the left-trimmed string.
- `trim1` the reversed string (this will trim the left end of the reversed string, which is actually the right end of the original string).
- `reverse` finally the string (it has already been trimmed from both ends).

Note that here we’re using both functional composition and partial application almost in all possible places in the above definition.

A simpler solution is to use just `multiCompose`:

```
trim  =  multiCompose [reverse, trim1, reverse, trim1]
```

Here’s the XSLT implementation of the above definition of `trim`:

Figure 11: trim.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon"
xmlns:myTrimDropController="f:myTrimDropController"
xmlns:myTrim1="f:myTrim1"
xmlns:myReverse="f:myReverse"
exclude-result-prefixes="xsl saxon
myTrimDropController myTrim1 myReverse">

  <xsl:import href="str-dropWhile.xsl"/>
  <xsl:import href="compose-flist.xsl"/>
  <xsl:import href="reverse.xsl"/>

  <xsl:template name="trim">
    <xsl:param name="pStr"/>

    <xsl:variable name="vrtfParam">
      <myReverse:myReverse/>
      <myTrim1:myTrim1/>
      <myReverse:myReverse/>
      <myTrim1:myTrim1/>
    </xsl:variable>

    <xsl:call-template name="compose-flist">
        <xsl:with-param name="pFunList"
             select="saxon:node-set($vrtfParam)/*"/>
        <xsl:with-param name="pArg1" select="$pStr"/>
    </xsl:call-template>
  </xsl:template>
```

```
  <xsl:template name="trim1" match="myTrim1:*">
    <xsl:param name="pArg1"/>

  <xsl:variable name="vTab" select="'  '"/>
  <xsl:variable name="vNL" select="'
'"/>
  <xsl:variable name="vCR" select="'
'"/>
  <xsl:variable name="vWhitespace"
                select="concat(' ', $vTab, $vNL, $vCR)"/>

    <xsl:variable name="vFunController"
                  select="document('')/*/myTrimDropController:*[1]"/>

    <xsl:call-template name="str-dropWhile">
      <xsl:with-param name="pStr" select="$pArg1"/>
      <xsl:with-param name="pController" select="$vFunController"/>
      <xsl:with-param name="pContollerParam" select="$vWhitespace"/>
    </xsl:call-template>
  </xsl:template>

  <myTrimDropController:myTrimDropController/>
  <xsl:template match="myTrimDropController:*">
    <xsl:param name="pChar"/>
    <xsl:param name="pParams"/>

    <xsl:if test="contains($pParams, $pChar)">1</xsl:if>
  </xsl:template>

  <xsl:template name="myReverse" match="myReverse:*">
    <xsl:param name="pArg1"/>

    <xsl:call-template name="strReverse">
      <xsl:with-param name="pStr" select="$pArg1"/>
    </xsl:call-template>
</xsl:template>
</xsl:stylesheet>
```

A number of other functions are used in the code: `str-dropWhile`, `strReverse`, and `str-foldl`. These are part of the FXSL library [FXSL] as currently available on SourceForge.Net. It is a general convention in FXSL that a function named `strSomeName` is an implementation of the function `someName` for strings. This is necessary in an XSLT implementation, because a string in Xpath/XSLT cannot be naturally represented as a list of characters.

Here's a simple test of the XSLT `trim` function:

Figure 12: testTrim.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="trim.xsl"/>

  <!-- to be applied on trim.xml -->

  <xsl:output method="text"/>
  <xsl:template match="/">
    '<xsl:call-template name="trim">
        <xsl:with-param name="pStr" select="string(/*)"/>
    </xsl:call-template>'
  </xsl:template>
</xsl:stylesheet>
```

When applied on the following source xml document:

Figure 13: trim.xml

```
<someText>

   This is    some text
```

```
</someText>
```

The result is:

```
'This is    some text'
```

3.5 *Implementation of currying and partial application in XSLT*

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.

An 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, as we are departing from the above definition (taken from Haskell's Prelude) and will try to define and implement in XSLT a more generic currying function that can process functions having up to ten arguments.
3. Provide a reference to an internal generic `partial-applicator` function that will record and accumulate the name-value pairs of the passed arguments and, in case all arguments have been provided, will apply `f` on them and return the result. In case not all arguments' values have yet been provided, the `partial-applicator` will return a variant of itself, that knows about all the arguments' values accumulated so far. Here's another difference between our definition and XSLT implementation of partial application and its Haskell counterpart: because XSLT template parameters are known by name, they can be passed in any order — this means that in XSLT we can implement partial application not only when the first argument(s) are provided, but also in any case in which any subset of arguments' values have been specified. For example, given the function

   ```
   f x y z
   ```

   in Haskell, one can have the following partial applications: `f x` and `f x y`.

   Our implementation will allow us to obtain the following partial applications:

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

But, in case we're accepting named arguments, how do we know the names of the arguments of an arbitrary function to be curried and partially applied? The answer is that while arguments have names, only the names of type "`arg`"<N> are allowed — that is, "`arg1`", "`arg2`", ..., "`arg10`". This is a reasonable limitation resulting in a much more powerful partial application through the combination of naming and implicit ordering, indicated by the numeric part of the names of the arguments.

The XSLT implementation of `curry` and partial application is presented in the Appendix. It is a simplified version with error-checking code not included.

The `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 (as the first argument) function is to be partially applied.

Initially, it produces a partial application of the function with no arguments' values specified.

Then, it checks if values of arguments were specified, and if yes, it applies its internal `partialApplicator` function to produce the result as a partial application.

The internal `partialApplicator` function accepts up to ten arguments, whose names follow the convention "`arg`"<N>. All arguments have the empty node-set as default value and any of them may be omitted on an actual instantiation of the function.

The code finds any specified argument using the simple rule, that if the N-th argument is not specified, then "`arg`"<N> should be false (either the empty node-set or the empty string). All specified arguments are appended to an internal structure, containing elements with names "`arg`"<N> and values — the value specified for the "`arg`"<N> argument. For example, if `f` has 6 arguments and 3 of them were specified, the `partialApplicator`'s internal store could look like this:

```
<curryPartialApplicator:curryPartialApplicator>
  <fun><someFunNode /></fun>
  <curryNumargs>6</curryNumargs>
  <curryArgs>
    <arg1>a</arg1>
    <arg3>b</arg3>
    <arg5>c</arg5>
  </curryArgs>
</curryPartialApplicator:curryPartialApplicator>
```

Notice that exactly the above structure is returned as the result of applying f with three arguments (`arg1 = "a", arg3 = "b", arg5 = "c"`) — in fact, this is a new template reference to the implementation of `partialApplicator`, which has recorded all the (partial) arguments' values provided so far.

Whenever values for all arguments will be provided, the code above instantiates the template that implements the `f` function — the template reference to it is stored in the `<fun>` element.

Let's test currying and partial application of a curried function with a simple example. We'll curry the `multiply` function, defined as:

```
multiply (x,y) = x*y
```

and then obtain its partial application for `x=3` (the function `(3*)`). Finally, we'll apply the so obtained `(3*)` on the argument with a value `7` — that is, we'll calculate:

```
(3*) 7
```

and the result is the expected: `21`.

Figure 14: testCurry.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon"
xmlns:myMultiply="f:myMultiply"
exclude-result-prefixes="saxon myMultiply">

<xsl:import href="curry.xsl"/>

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

  <xsl:template match="/">
    <xsl:variable name="vMyMultiply"
                  select="document('')/*/myMultiply:*[1]"/>

    <!-- Get the curried fn here -->
    <!-- Get the partial application (*3) -->
    <xsl:variable name="vrtfCurriedMultBy3">
      <xsl:call-template name="curry">
        <xsl:with-param name="pFun" select="$vMyMultiply"/>
        <xsl:with-param name="pNargs" select="2"/>
        <xsl:with-param name="arg1" select="3"/>
      </xsl:call-template>
    </xsl:variable>

    <xsl:variable name="vMultBy3"
                  select="saxon:node-set($vrtfCurriedMultBy3)/*"/>
```

```
    <xsl:apply-templates select="$vMultBy3"> <!-- Apply (*3) on 7 -->
      <xsl:with-param name="arg2" select="7"/>
    </xsl:apply-templates> <!-- The result must be 21 -->

  </xsl:template>

  <myMultiply:myMultiply/>     <!-- My multiply x y function -->
  <xsl:template match="myMultiply:*">
    <xsl:param name="arg1"/>
    <xsl:param name="arg2"/>

    <xsl:value-of select="$arg1 * $arg2"/>
  </xsl:template>
</xsl:stylesheet>
```

As a more realistic case of using the `curry` function, we provide an example of implementation of a `power` function.

A function `pow`, which takes a non-negative integer argument `n` and a real argument `x`, produces `x` to the power of `n` (`x^n`):

```
power n x = iter n (multByX x) 1
            where
              multByX x y = (*x) y
```

Note the use of the partial application in `multByX`. There are cases when we might know in advance that we need a particular partial application (*e.g.*, `(+1)`, `(*3)`, etc.), and in any such case, we can hardwire this knowledge into the code of a specific function. However, `multByX` is defined completely dynamically, depending on the value of `x`. To implement `power` as defined above, we must create the `multByX` function dynamically at run time, whenever `power` is invoked.

Figure 15: pow.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon"
xmlns:myMultiply="f:myMultiply"
exclude-result-prefixes="xsl saxon myMultiply">

  <xsl:import href="curry.xsl"/>
  <xsl:import href="iter.xsl"/>

  <xsl:template name="pow">
    <xsl:param name="pTimes" select="0"/>
    <xsl:param name="pX"/>

    <xsl:variable name="vMultiply"
         select="document('')/*/myMultiply:*[1]"/>

    <xsl:variable name="vrtfCurriedMultByX">
    <xsl:call-template name="curry">
      <xsl:with-param name="pFun" select="$vMultiply"/>
      <xsl:with-param name="pNargs" select="2"/>
      <xsl:with-param name="arg2" select="$pX"/>
    </xsl:call-template>
    </xsl:variable>

    <xsl:variable name="vCurriedMultByX"
         select="saxon:node-set($vrtfCurriedMultByX)/node()"/>

    <xsl:call-template name="iter">
      <xsl:with-param name="pTimes" select="$pTimes"/>
      <xsl:with-param name="pFun" select="$vCurriedMultByX"/>
      <xsl:with-param name="pX" select="1"/>
    </xsl:call-template>
  </xsl:template>

  <myMultiply:myMultiply/>
  <xsl:template match="myMultiply:*">
    <xsl:param name="arg1"/>
    <xsl:param name="arg2"/>

    <xsl:value-of select="$arg1 * $arg2"/>
```

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

As can be seen, we curry the `myMultiply` function and at the same time dynamically create its partial application, binding the second argument to the value of `pX`. Then, we pass this dynamically created partial application as argument to `iter`.

And here's how `pow` works:

Figure 16: testPow.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="pow.xsl"/>

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:for-each select="(document('')//node())
                          [position() < 10]">
      <xsl:value-of select="concat(position(),
                                   '^', position(),' = '
                                   )"/>
      <xsl:call-template name="pow">
        <xsl:with-param name="pTimes" select="position()"/>
        <xsl:with-param name="pX" select="position()"/>
      </xsl:call-template>
      <xsl:text>
      </xsl:text>
    </xsl:for-each>

(1+1/5000) ^ 5000 = <xsl:text/>
   <xsl:call-template name="pow">
      <xsl:with-param name="pTimes" select="5000"/>
      <xsl:with-param name="pX" select="1.0002"/>
   </xsl:call-template>
  </xsl:template>
</xsl:stylesheet>
```

The result is:

```
1^1 = 1
2^2 = 4
3^3 = 27
4^4 = 256
5^5 = 3125
6^6 = 46656
7^7 = 823543
8^8 = 16777216
9^9 = 387420489

(1+1/5000) ^ 5000 = 2.7180100501015563
```

3.6 *Creating a new function dynamically*

Here's an example, taken from the book by Simon Thompson, *Haskell: The Craft of Functional Programming* [ThompSJ].

Suppose we are to build a calculator for numerical expressions. As part of our system we need to be able to model the current values of the user-defined variables, which we might call the store of the calculator.

To model the store, we define an ADT [abstract data type], which has the following signature:

```
initial :: Store

value   :: Store -> Var -> Int

update  :: Store -> Var -> Int -> Store
```

The function "`initial`" produces an initial (empty) store.

The function "`value`" retrieves from the store the value (Int) of a variable name.

The function "`update`" updates the store with a new (variable-name, value) association

This ADT can have different implementations, *e.g.*, by keeping an internal list of pairs (varName, value), or by implementing a function that given a variable name returns its value.

The latter implementation is defined in Haskell in the following way:

```
newtype Store = Sto (Var -> Int)

initial :: Store
initial =  Sto (\v -> 0)

value   :: Store -> Var -> Int
value (Sto sto) v = sto v

update  :: Store -> Var -> Int -> Store
update (Sto sto) v n
  = Sto (\w -> if v == w then n else sto w)
```

Under this implementation:

- The initial `store` maps every variable to 0.
- To look up a value of a variable `v`, the store function `sto` is applied to `v`.
- In the case of an update, the new `store` returned has a function, which is identical to `sto` of the updated object, except on the variable, whose value is changed.

Here's the corresponding XSLT implementation:

Figure 17: store.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:getInitialStore="f:getInitialStore"
xmlns:getValue="f:getValue0"
xmlns:upd-getValue="f:upd-getValue"
xmlns:updateStore="f:updateStore">

  <xsl:template name="getInitialStore" match="getInitialStore:*">
    <store>
      <getInitialStore:getInitialStore/>
      <getValue:getValue/>
      <updateStore:updateStore/>
    </store>
  </xsl:template>

  <xsl:template match="getValue:*">
     <xsl:value-of select="0"/>
  </xsl:template>

  <xsl:template match="updateStore:*">
    <xsl:param name="pName"/>
    <xsl:param name="pValue"/>

    <store>
      <getInitialStore:getInitialStore/>
      <upd-getValue:getValue>
        <store><xsl:copy-of select="../*"/></store>
        <name><xsl:value-of select="$pName"/></name>
        <value><xsl:value-of select="$pValue"/></value>
      </upd-getValue:getValue>
      <updateStore:updateStore/>
    </store>
  </xsl:template>

  <xsl:template match="upd-getValue:*">
    <xsl:param name="pName"/>

    <xsl:choose>
      <xsl:when test="$pName = name">
        <xsl:value-of select="value"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-templates
             select="store/*[local-name()='getValue']">
           <xsl:with-param name="pName" select="$pName"/>
```

```
        </xsl:apply-templates>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

What is important to note in this example is that on every call of the `updateStore` function, it creates a new store containing a template reference for a new `getValue` function, like this:

```
<store>
  <getInitialStore:getInitialStore/>
  <upd-getValue:getValue>
    <store>
      <xsl:copy-of select="../*"/>
    </store>
    <name><xsl:value-of select="$pName"/></name>
    <value><xsl:value-of select="$pValue"/></value>
  </upd-getValue:getValue>
  <updateStore:updateStore/>
</store>
```

This template reference (`upd-getValue:getValue`) always initiates the same template rule, **but for a different function**. As we see, a function is not identical to the template rule that implements it; it is a tree, in which useful data can be stored, that fully defines the function.

This function may call a chain of previously defined `getValue` functions for the previous stores, until the first occurrence of a variable name is found, or if not found, then finally the very initially defined `getValue` function is instantiated to return the value 0.

This example has big importance in that it shows how we can implement objects in XSLT. Under this interpretation `store` is an object with `getInitialStore` as constructor. It also has two methods — `getValue`, which performs a lookup for a variable's value given its name, and `updateStore`, which creates a new `store` object. We have also showed how to model inheritance and virtual functions. Indeed, every newly created `store` object can be regarded as derived from the base `store` object and with a new `getValue` virtual function. The `getValue` function has two different implementations — one for the `getValue` function of an initially constructed `store` object (this function returns 0 regardless of any given variable name), and another, which performs the lookup. As can be seen in the test of our `store` implementation below, we can invoke any of these functions using the same linguistic construct — which function will be invoked depends on the `store` object, against the context of which the call is performed. Different implementations of a virtual function all have template references with the same `local-name()`, but belonging to different namespaces.

And here's the test of our implementation of `store`.

Figure 18: testStore.xsl

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:saxon="http://icl.com/saxon">

  <xsl:import href="store.xsl"/>

  <!-- to be applied on any xml source -->

  <xsl:output method="text"/>

  <xsl:template match="/">
    <!-- Get Initial Store -->
    <xsl:variable name="vrtfStore0">
      <xsl:call-template name="getInitialStore"/>
    </xsl:variable>

    <xsl:variable name="vStore0"
         select="saxon:node-set($vrtfStore0)/*"/>

    <!-- Get A, B, C from the initial store: must be 0-s -->
      vStore0:
      A='<xsl:apply-templates
```

```
      select="$vStore0/*[local-name() = 'getValue']">
      <xsl:with-param name="pName" select="'A'"/>
     </xsl:apply-templates>'
  B='<xsl:apply-templates
      select="$vStore0/*[local-name() = 'getValue']">
      <xsl:with-param name="pName" select="'B'"/>
     </xsl:apply-templates>'
  C='<xsl:apply-templates
      select="$vStore0/*[local-name() = 'getValue']">
      <xsl:with-param name="pName" select="'C'"/>
     </xsl:apply-templates>'

<!-- Update store0 with 'A=1' -->
<xsl:variable name="vrtfStore1">
  <xsl:apply-templates
   select="$vStore0/*[local-name() = 'updateStore']">
    <xsl:with-param name="pName" select="'A'"/>
    <xsl:with-param name="pValue" select="1"/>
  </xsl:apply-templates>
</xsl:variable>

<xsl:variable name="vStore1"
     select="saxon:node-set($vrtfStore1)/*"/>
<!-- Get A, B, C from the store1: A is 1, the rest must be 0-s -->
  vStore1:
  A='<xsl:apply-templates
      select="$vStore1/*[local-name() = 'getValue']">
      <xsl:with-param name="pName" select="'A'"/>
     </xsl:apply-templates>'
  B='<xsl:apply-templates
      select="$vStore1/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'B'"/>
     </xsl:apply-templates>'
  C='<xsl:apply-templates
      select="$vStore1/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'C'"/>
     </xsl:apply-templates>'

<!-- Update store1 with 'B=2' -->
<xsl:variable name="vrtfStore2">
  <xsl:apply-templates
   select="$vStore1/*[local-name() = 'updateStore']">
    <xsl:with-param name="pName" select="'B'"/>
    <xsl:with-param name="pValue" select="2"/>
  </xsl:apply-templates>
</xsl:variable>

<xsl:variable name="vStore2"
     select="saxon:node-set($vrtfStore2)/*"/>
<!-- Get A, B, C from the store2: A is 1, B is 2,
     the rest must be 0-s -->
  vStore2:
  A='<xsl:apply-templates
      select="$vStore2/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'A'"/>
     </xsl:apply-templates>'
  B='<xsl:apply-templates
      select="$vStore2/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'B'"/>
     </xsl:apply-templates>'
  C='<xsl:apply-templates
      select="$vStore2/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'C'"/>
     </xsl:apply-templates>'

<!-- Update store2 with 'C=3' -->
<xsl:variable name="vrtfStore3">
  <xsl:apply-templates
   select="$vStore2/*[local-name() = 'updateStore']">
    <xsl:with-param name="pName" select="'C'"/>
    <xsl:with-param name="pValue" select="3"/>
  </xsl:apply-templates>
</xsl:variable>

<xsl:variable name="vStore3"
     select="saxon:node-set($vrtfStore3)/*"/>
<!-- Get A, B, C from the store3: A is 1, B is 2, C is 3,
     the rest must be 0-s -->
  vStore3:
  A='<xsl:apply-templates
      select="$vStore3/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'A'"/>
     </xsl:apply-templates>'
  B='<xsl:apply-templates
      select="$vStore3/*[local-name() = 'getValue']">
       <xsl:with-param name="pName" select="'B'"/>
     </xsl:apply-templates>'
  C='<xsl:apply-templates
```

```
          select="$vStore3/*[local-name() = 'getValue']">
           <xsl:with-param name="pName" select="'C'"/>
         </xsl:apply-templates>'
    </xsl:template>
</xsl:stylesheet>
```

The result from the above test is:

```
vStore0:
A='0'
B='0'
C='0'

vStore1:
A='1'
B='0'
C='0'

vStore2:
A='1'
B='2'
C='0'

vStore3:
A='1'
B='2'
C='3'
```

§ 4 *The FXSL functional programming library*

This article is only a brief reflection of the work accomplished in the development of the FXSL — a functional programming library for XSLT [FXSL]. It 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, and 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 following table summarizes the main functionality implemented in or with FXSL.

Table 1: ***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 tokenization, trimming and reversal • Text justification • Spelling checking and generation of correct close words

- 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
- Randomization 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

§ 5 Conclusion

A new infrastructure for XSLT programming has been built and is provided by the FXSL library. It contains the FP design patterns described in this article and many more. The practice shows that, based on this infrastructure, it has become much easier to solve a large class of problems that until now has been considered very difficult or not appropriate for XSLT. With support for FP, XSLT programming is much easier and more effective due to the increased level of abstraction and code reuse.

§ 6 Appendix: the implementation of curry()

6.1 *currySimplified.xsl*

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:vendor="http://icl.com/saxon"
xmlns:curryPartialApplicator="f:curryPartialApplicator"
exclude-result-prefixes="vendor curryPartialApplicator">

  <xsl:template name="curry">
    <xsl:param name="pFun" select="/.."/>
    <xsl:param name="pNargs" select="2"/>
    <xsl:param name="pStripAuxNamespace"/>
    <xsl:param name="args" select="/.."/>
    <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:param name="arg10" select="/.."/>

    <!-- Build the Resulting fn with an empty Arguments store -->
    <xsl:variable name="vrtfCurriedNoArgs">
   <curryPartialApplicator:curryPartialApplicator>
     <fun><xsl:copy-of select="$pFun"/></fun>
     <curryNumargs><xsl:value-of select="$pNargs"/></curryNumargs>
     <curryStrip><xsl:value-of select="$pStripAuxNamespace"/></curryStrip>
   </curryPartialApplicator:curryPartialApplicator>
    </xsl:variable>

    <xsl:variable name="vCurriedNoArgs"
                  select="vendor:node-set($vrtfCurriedNoArgs)/*"/>

    <xsl:choose>
      <xsl:when test="not($args)
                      and
                        not($arg1 or $arg2 or $arg3 or $arg4 or $arg5
                         or $arg6 or $arg7 or $arg8 or $arg9 or $arg10
                            )">
        <xsl:copy-of select="$vCurriedNoArgs"/>
      </xsl:when>
```

```
      <xsl:otherwise>
        <xsl:apply-templates select="$vCurriedNoArgs">
          <xsl:with-param name="args" select="$args"/>
          <xsl:with-param name="arg1" select="$arg1"/>
          <xsl:with-param name="arg2" select="$arg2"/>
          <xsl:with-param name="arg3" select="$arg3"/>
          <xsl:with-param name="arg4" select="$arg4"/>
          <xsl:with-param name="arg5" select="$arg5"/>
          <xsl:with-param name="arg6" select="$arg6"/>
          <xsl:with-param name="arg7" select="$arg7"/>
          <xsl:with-param name="arg8" select="$arg8"/>
          <xsl:with-param name="arg9" select="$arg9"/>
          <xsl:with-param name="arg10" select="$arg10"/>

        </xsl:apply-templates>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="curryPartialApplicator:*">
    <xsl:param name="args" select="/.."/>
    <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:param name="arg10" select="/.."/>

    <xsl:variable name="vrtvArgs">
    <xsl:copy-of select="$args"/>
    <xsl:if test="$arg1">
      <arg1>
        <xsl:copy-of select="$arg1"/>
      </arg1>
    </xsl:if>
    <xsl:if test="$arg2">
      <arg2>
        <xsl:copy-of select="$arg2"/>
      </arg2>
    </xsl:if>
    <xsl:if test="$arg3">
      <arg3>
        <xsl:copy-of select="$arg3"/>
      </arg3>
    </xsl:if>
    <xsl:if test="$arg4">
      <arg4>
        <xsl:copy-of select="$arg4"/>
      </arg4>
    </xsl:if>
    <xsl:if test="$arg5">
      <arg5>
        <xsl:copy-of select="$arg5"/>
      </arg5>
    </xsl:if>
    <xsl:if test="$arg6">
      <arg6>
        <xsl:copy-of select="$arg6"/>
      </arg6>
    </xsl:if>
    <xsl:if test="$arg7">
      <arg7>
        <xsl:copy-of select="$arg7"/>
      </arg7>
    </xsl:if>
    <xsl:if test="$arg8">
      <arg8>
        <xsl:copy-of select="$arg8"/>
      </arg8>
    </xsl:if>
    <xsl:if test="$arg9">
      <arg9>
        <xsl:copy-of select="$arg9"/>
      </arg9>
    </xsl:if>
    <xsl:if test="$arg10">
      <arg10>
```

```
      <xsl:copy-of select="$arg10"/>
    </arg10>
  </xsl:if>
  </xsl:variable>

  <xsl:variable name="vArgs"
       select="vendor:node-set($vrtvArgs)/*"/>

  <!-- Normal  Processing -->
  <xsl:variable name="vrtfNewFun">
  <curryPartialApplicator:curryPartialApplicator>
    <xsl:copy-of select="fun"/>
    <xsl:copy-of select="curryStrip"/>
    <xsl:copy-of select="curryNumargs"/>
    <curryArgs>
      <xsl:copy-of select="curryArgs/*"/>
      <xsl:copy-of select="$vArgs"/>
    </curryArgs>
  </curryPartialApplicator:curryPartialApplicator>
  </xsl:variable>

  <xsl:variable name="vNewFun"
       select="vendor:node-set($vrtfNewFun)/*"/>

  <xsl:choose>
    <xsl:when test="curryNumargs > count($vNewFun/curryArgs/*)">
      <xsl:copy-of select="$vNewFun"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:choose>
        <xsl:when test="not(string(curryStrip))">
      <xsl:apply-templates select="fun/*[1]">
        <xsl:with-param name="arg1"
                  select="$vNewFun/curryArgs/arg1/node()"/>
        <xsl:with-param name="arg2"
                  select="$vNewFun/curryArgs/arg2/node()"/>
        <xsl:with-param name="arg3"
                  select="$vNewFun/curryArgs/arg3/node()"/>
        <xsl:with-param name="arg4"
                  select="$vNewFun/curryArgs/arg4/node()"/>
        <xsl:with-param name="arg5"
                  select="$vNewFun/curryArgs/arg5/node()"/>
        <xsl:with-param name="arg6"
                  select="$vNewFun/curryArgs/arg6/node()"/>
        <xsl:with-param name="arg7"
                  select="$vNewFun/curryArgs/arg7/node()"/>
        <xsl:with-param name="arg8"
                  select="$vNewFun/curryArgs/arg8/node()"/>
        <xsl:with-param name="arg9"
                  select="$vNewFun/curryArgs/arg9/node()"/>
        <xsl:with-param name="arg10"
                  select="$vNewFun/curryArgs/arg10/node()"/>
      </xsl:apply-templates>
        </xsl:when>
        <xsl:otherwise>
          <xsl:variable name="vrtf-strippedArgs">
            <xsl:apply-templates select="$vNewFun/curryArgs/*"
                                 mode="stripNamespace"/>
          </xsl:variable>

          <xsl:variable name="vstrippedArgs"
               select="vendor:node-set($vrtf-strippedArgs)"/>

          <xsl:apply-templates select="fun/*[1]">
            <xsl:with-param name="arg1"
                      select="$vstrippedArgs/arg1/node()"/>
            <xsl:with-param name="arg2"
                      select="$vstrippedArgs/arg2/node()"/>
            <xsl:with-param name="arg3"
                      select="$vstrippedArgs/arg3/node()"/>
            <xsl:with-param name="arg4"
                      select="$vstrippedArgs/arg4/node()"/>
            <xsl:with-param name="arg5"
                      select="$vstrippedArgs/arg5/node()"/>
            <xsl:with-param name="arg6"
                      select="$vstrippedArgs/arg6/node()"/>
            <xsl:with-param name="arg7"
                      select="$vstrippedArgs/arg7/node()"/>
            <xsl:with-param name="arg8"
                      select="$vstrippedArgs/arg8/node()"/>
            <xsl:with-param name="arg9"
```

```
                        select="$vstrippedArgs/arg9/node()"/>
              <xsl:with-param name="arg10"
                        select="$vstrippedArgs/arg10/node()"/>
            </xsl:apply-templates>

          </xsl:otherwise>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template match="node()" mode="stripNamespace">
    <xsl:choose>
      <xsl:when test="self::*">
    <xsl:element name="{name()}">
      <xsl:copy-of
            select="namespace::*
                    [name() != 'curryPartialApplicator']"/>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select="node()"
                           mode="stripNamespace"/>
    </xsl:element>
   </xsl:when>
   <xsl:otherwise>
     <xsl:copy-of select="."/>
   </xsl:otherwise>
    </xsl:choose>

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

Bibliography

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

[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**.

[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.

[XSLT 2 WD] W3C (Draft), *XSL Transformations (XSLT) Version 2.0*, W3C Working Draft, 15 November 2002, At **http://www.w3.org/TR/xslt20/**.

[XSLT1.0] World Wide Web Consortium, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 16 November 1999, At **http://www.w3.org/TR/xslt**.

The Author

Dimitre Novatchev
IAEA
Vienna
Austria
dnovatchev@yahoo.com
http://fxsl.sf.net

Dimitre Novatchev is the author of the FXSL functional programming library for XSLT and of the popular tool for learning XPath — the XPath Visualizer

Extreme Markup Languages 2003
Montréal, Québec, August 4-8, 2003
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.