A Proof of the Turing-completeness of XSLT and XQuery

Stephan Kepser
SFB 441, University of Tübingen
kepser@sfs.uni-tuebingen.de

May 13, 2002

Abstract

The World Wide Web Consortium recommends both XSLT and XQuery as query languages for XML documents. XSLT, originally designed to transform XML into HTML, is nowadays a fully grown XML query language that is mostly suited for use by machines. XQuery on the other hand was particularly designed to be easily used by humans. Since both query languages receive a steady growth in user acceptance, it is important and natural to ask about their expressive power. We show here that both XSLT and XQuery are Turing-complete by reduction to μ-recursive functions.

Keywords: XML, XSLT, XQuery, Turing-completeness

1 Introduction

The World Wide Web Consortium (W3C) recommends both XSLT and XQuery as query languages for XML documents. XSLT (X Style sheet Language Transformations [6, 9]) is the recommendation of the W3C for an XML [8] style sheet language. The original primary role of XSLT was to allow users to write transformations of XML to HTML, thus describing the presentation of XML documents. Nowadays, many people use XSLT as their basic tool for XML to XML transformations, which renders XSLT into an XML query language. This naturally raises the question about the expressive power of XSLT as a query language. We show here that XSLT is Turing-complete by coding μ-recursive functions in XSLT.

We note that there are several claims in the XSLT programmers comunity that XSLT be Turing-complete. But none of these claims is supported by a proof. There is a web site (`http://www.unidex.com/turing/`) which provides a style sheet for a Universal Turing Machine, but it is that long there is hardly a chance to formally show that it does what it claims to do. We do not maintain the insight XSLT to be Turing-complete to be new. Rather we provide a relatively simple proof to firmly establish a result which may otherwise remain folklore.

Previous work on the expressive power of XSLT was mostly concerned with fragments of XSLT. Neven et al. [2, 12] investegated XSLT from the point of view of its intended model of structural recursion over trees.

There exist now several query languages for querying XML documents. The W3C recommends (as a working draft) XQuery [4], which is based on XPath [1, 7], a language originally designed to locate elements in an XML document. XQuery adds variables and recursion to this, but also features to produces structured output. Turing-completeness of XQuery can again be shown by coding μ-recursive functions. Indeed, the coding is very straight forward.

It is interesting to note that the W3C now recommends two XML query languages with the same expressive power, both based on XPath, but with quite some differences. XSLT's processing model is to transform a document by structural recursion starting with the root node and ending at the leaves. XQuery does not have a processing model. Instead it possesses an abstract semantics which is originally based on a query algebra. XQuery has a human-readable syntax while XSLT has an XML-based syntax, which is very verbose, and cumbersome to read for humans. XSLT is untyped while XQuery is strictly typed. So, apart from typing issues, XSLT is more machine oriented while XQuery is more suitable for human use.

2 μ-Recursive Functions

Amongst the many equivalent definition of Turing-complete computations there exists a proposal by Kleene [10] using so-called *μ-recursive functions* for computations on natural numbers. Our definition of μ-recursive functions follows the exposition by Lewis and Papadimitriou [11].

Definition 1 The *basic* functions are as follows:

1. For any $k \geq 0$, the k-ary zero function is defined as $\mathsf{zero}_k(n_1,\ldots,n_k) = 0$ for all $n_1,\ldots,n_k \in \mathbb{N}$.
2. For any $k \geq j > 0$ the j-th k-ary projection function is simply the function $\pi_{k,j}(n_1,\ldots,n_k) = n_j$ for all $n_1,\ldots,n_k \in \mathbb{N}$.
3. The successor function is defined as $\mathsf{succ}(n) = n+1$ for all $n \in \mathbb{N}$.

The *composition* is defined as follows: Let $k,l \geq 0$, let $g : \mathbb{N}^k \to \mathbb{N}$ be a k-ary function, and let $h_1,\ldots,h_k$ be l-ary functions. Then the composition of g with $h_1,\ldots,h_k$ is the l-ary function

$$f(n_1,\ldots,n_l) = g(h_1(n_1,\ldots,n_l),\ldots,h_k(n_1,\ldots,n_l)).$$

Primitive recursion: Let $k \geq 0$ and g be a k-ary function, and let h be a $k+2$-ary function. Then the function defined recursively by g and h is the $k+1$-ary function f with

$$\begin{aligned} f(n_1,\ldots,n_k,0) &= g(n_1,\ldots,n_k), \\ f(n_1,\ldots,n_k,m+1) &= h(n_1,\ldots,n_k,m,f(n_1,\ldots,n_k,m)) \end{aligned}$$

for all $n_1, \ldots, n_k, m \in \mathbb{N}$.

μ-Recursion: Let $k \geq 0$ and g be a $k+1$-ary function. The minimisation of g is the k-ary function f defined as

$$f(n_1, \ldots, n_k) = \begin{cases} \text{the least } m \text{ such that } g(n_1, \ldots, n_k, m) = 0, \\ \quad \text{if such an } m \text{ exists;} \\ 0 \text{ otherwise} \end{cases}$$

for all $n_1, \ldots, n_k \in \mathbb{N}$.

We note in passing that primitive recursion is necessary even in the presence of μ-recursion. It is indeed the only way to define a function that truly depends on more than one argument.

3 XSLT

A full explanation of XSLT is of course far beyond the scope of this paper. The interested reader is referred to the official standard [6, 9], published by the World Wide Web Consortium, unfortunately not always easy to read. We will give here a short overview over the constructs we need for coding μ-recursive functions. In a nutshell, XPath provides the arithmetics, and XSLT recursion (with a little help from XPath). As we will see, we need only a very small subpart of XSLT, basically template calling, parameter passing, some basic arithmetics and a little bit of string handling. Arithmetics and string handling is defined in XPath [1, 7], the standard for expressions in XSLT.

Templates are XSLT's way of expressing procedures or functions.

```
<xsl:template name="f">
 ...
</xsl:template>
```

Templates may have a name. If a template has a name, it can be called by another template via this name:

```
<xsl:call-template name="f">
 ...
</xsl:call-template>
```

Instead of an identifier (*Qname* in XSLT terminology) like `f` there may be an expression that can be evaluated to an identifier, so that the template to be called may be determined at run-time. This is one of the features newly introduced in XSLT version 2.0 that we will make use of to simplify the exposition. `xsl:call-template` corresponds to jumping to a particular label in a program code. In particular, if you call a template and this called template completes its execution, program execution does *not* return to the calling template, it rather just stops there. If you want to do any further computations, there must be an explicit call to the next template.

Parameters can be used for passing information from one template to another. They have a name and a binding.

```
<xsl:param name="n"/>
```

to be placed at the beginning of a template states that this template can receive a parameter called `n`. When the template is called, i.e., inside a `<xsl:call-template>` ... `</xsl:call-template>` block, the parameter is transfered by

```
<xsl:with-param name="n" select="expr"/>
```

Variables are similar to parameters, but local to the template in which they occur. They are not used for passing information between templates.

```
<xsl:variable name="m" select="expr"/>
```

defines a local variable with name `m` and binds it to the value of the expression *expr*.

Conditionals XSLT provides constructs for conditional execution. We only need the simple form of

```
<xsl:if test='expr'>
 ...
</xsl:if>
```

If the expression *expr* evaluates to true, the block enclosed by `<xsl:if>` and `</xsl:if>` is executed. If it evaluates to false, the block is skipped. The expressions we will use in tests are very simple: We just test if the value of a parameter is equal to 0.

Arithmetics XPath provides natural numbers and addition and subtraction of them. That is all we need.

Strings XPath provides stings as data types and string functions which allow one to emulate stacks by strings. One needs a symbol separating objects on the stack. In our case it will be the slash (/). To push an element on the stack, we use the function *concat* to concatenate strings. It takes two or more arguments and returns their concatenation. To get the top element of the stack, we use the function *substring-before* which takes two strings as arguments and returns the substring of the first string before the first occurrence of the second string. When using the stack as the first and the separating symbol '/' as the second argument, the top of the stack is returned. To get the rest of the stack, we use *substing-after*.

We need stacks for two purposes: Firstly, when calling an XSLT-template for doing subcomputations we need to know the return point, the point at which computation shall resume after completing a subcomputation. Since subcomputations can call further subcomputations, we need a stack of return points and not just a single one. Secondly, we need a storage place for results of subcomputations. There is no way to determine beforehead how many results we will need to store, so we use a stack.

4 Coding μ-Recursive Functions in XSLT

While the coding of the basic functions is easy to grasp from the XSLT source code, the coding of complex functions is more demanding. We therefore explain the way these templates work each time after presenting the XSLT code for each of the complex functions.

Basic Functions

Let $k \geq 0$. We code $\mathsf{zero}_k(n_1, \ldots, n_k)$ as follows:

```
<xsl:template name="zero-k">
  <xsl:param name="n-1"/>
  ...
  <xsl:param name="n-k"/>
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:call-template name="substring-before($call-stack,'/')">
    <xsl:with-param name="call-stack"
                    select="substring-after($call-stack,'/')"/>
    <xsl:with-param name="value-stack" select="concat('0/',$value-stack)"/>
  </xsl:call-template>
</xsl:template>
```

Let $k \geq j > 0$. We code $\pi_{k,j}(n_1, \ldots, n_k)$ as follows:

```
<xsl:template name="pi-k-j">
  <xsl:param name="n-1"/>
  ...
  <xsl:param name="n-k"/>
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:call-template name="substring-before($call-stack,'/')">
    <xsl:with-param name="call-stack"
                    select="substring-after($call-stack,'/')"/>
    <xsl:with-param name="value-stack"
                    select="concat($n-j,'/',$value-stack)"/>
  </xsl:call-template>
</xsl:template>
```

We code $\mathsf{succ}(n)$ as follows:

```
<xsl:template name="succ">
  <xsl:param name="n"/>
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:call-template name="substring-before($call-stack,'/')">
    <xsl:with-param name="call-stack"
                    select="substring-after($call-stack,'/')"/>
    <xsl:with-param name="value-stack"
                    select="concat($n + 1,'/',$value-stack)"/>
  </xsl:call-template>
</xsl:template>
```

Composition

Let $k, l \geq 0$. We code the composition $f(n_1, \ldots, n_l) = g(h_1(n_1, \ldots, n_l), \ldots, h_k(n_1, \ldots, n_l))$ as follows:

```
<xsl:template name="f">
  <xsl:param name="n-1"/>
  ...
  <xsl:param name="n-l"/>
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:call-template name="h1">
    <xsl:with-param name="n-1"/ select="$n-1"/>
    ...
    <xsl:with-param name="n-l" select="$n-l"/>
    <xsl:with-param name="call-stack"
                    select="concat('f-s1/',$call-stack)"/>
    <xsl:with-param name="value-stack"
         select="concat($n-1,'/',$n-2,'/',...,'/',$n-l,'/',$value-stack)"/>
  </xsl:call-template>
</xsl:template>
```

For $0 < j < k-1$ a template to call h_{j+1}:

```
<xsl:template name="f-sj">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rv" select="substring-before($value-stack,'/')"/>
  <xsl:variable name="vs1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs1,'/')"/>
  <xsl:variable name="vs2" select="substring-after($vs1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs2,'/')"/>
  <xsl:variable name="vs3" select="substring-after($vs2,'/')"/>
  ...
  <xsl:variable name="n-l" select="substring-before($vsl,'/')"/>
  <xsl:variable name="vs(l+1)" select="substring-after($vsl,'/')"/>
  <xsl:variable name="vss1" select="concat($rv,'/',$vs(l+1))"/>
  <xsl:variable name="vss2"
       select="concat($n-1,'/',$n-2,'/',...,'/',$n-l,'/',$vss1)"/>
  <xsl:call-template name="h(j+1)">
    <xsl:with-param name="n-1"/ select="$n-1"/>
    ...
    <xsl:with-param name="n-l" select="$n-l"/>
    <xsl:with-param name="call-stack"
                          select="concat('f-s(j+1)/',$call-stack)"/>
    <xsl:with-param name="value-stack" select="$vss2"/>
  </xsl:call-template>
</xsl:template>
```

The template to call h_k. In opposite to the previous cases we do not need to store the parameters $n_1, \ldots, n_k$ on the value stack any more.

```
<xsl:template name="f-s(k-1)">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rv" select="substring-before($value-stack,'/')"/>
  <xsl:variable name="vs1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs1,'/')"/>
```

```
  <xsl:variable name="vs2" select="substring-after($vs1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs2,'/')"/>
  <xsl:variable name="vs3" select="substring-after($vs2,'/')"/>
  ...
  <xsl:variable name="n-l" select="substring-before($vsl,'/')"/>
  <xsl:variable name="vs(l+1)" select="substring-after($vsl,'/')"/>
  <xsl:variable name="vss" select="concat($rv,'/',$vs(l+1))"/>
  <xsl:call-template name="hk">
    <xsl:with-param name="n-1"/ select="$n-1"/>
    ...
    <xsl:with-param name="n-l" select="$n-l"/>
    <xsl:with-param name="call-stack"
                          select="concat('f-sk/',$call-stack)"/>
    <xsl:with-param name="value-stack" select="$vss"/>
  </xsl:call-template>
</xsl:template>
```

And finally the template to call g:

```
<xsl:template name="f-sk">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rvk" select="substring-before($value-stack,'/')"/>
  <xsl:variable name="vs1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="rv(k-1)" select="substring-before($vs1,'/')"/>
  <xsl:variable name="vs2" select="substring-after($vs1,'/')"/>
  ...
  <xsl:variable name="rv1" select="substring-before($vs(k-1),'/')"/>
  <xsl:variable name="vsk" select="substring-after($vs(k-1),'/')"/>
  <xsl:call-template name="g">
    <xsl:with-param name="n-1"/ select="$rv1"/>
    ...
    <xsl:with-param name="n-k" select="$rvk"/>
    <xsl:with-param name="call-stack" select="$call-stack"/>
    <xsl:with-param name="value-stack" select="$vsk"/>
  </xsl:call-template>
</xsl:template>
```

Basically, we compute the values of $h_1(n_1,\ldots,n_l),\ldots,h_k(n_1,\ldots,n_l)$ first and use them then as input to the template for g. We use the value stack to store the parameters $n_1,\ldots n_k$ because we need them for each call to an h_j. We use the value stack also for accumulating the resulting values of the computations of $h_1,\ldots,h_k$. So, the template `f` pushes the parameters $n_1,\ldots n_k$ onto the value stack and '`f-s1`' onto the call stack. This is the name of the template to be called at the end of the computation of h_1. Template `f` calls the template for h_1 with parameters $n_1,\ldots,n_k$ and the call stack and value stack.

For $0 < j < k-1$ the template `f-s`j is called at the end of the computation for h_j. The top of the value stack now consists of
$h_j(n_1,\ldots,n_k),n_1,\ldots,n_k,h_{j-1}(n_1,\ldots,n_k),\ldots,h_1(n_1,\ldots,n_k)$.
Template `f-s`j pops the elements $h_j(n_1,\ldots,n_k),n_1,\ldots,n_k$ from the stack storing them in local variables. It pushes $h_j(n_1,\ldots,n_k)$ back onto the value stack and thereafter

also pushes $n_1,\ldots,n_k$ back onto that stack. Thus the order of return value from h_j and the parameters is now reversed on the stack. Template `f-s`j finishes by calling the template for h_{j+1} with parameters $n_1,\ldots,n_k$, the call stack with the next template '`f-s`$j+1$' pushed on top of it, and the value stack.

The template `f-s`$(k-1)$ is very similar to the above ones. It is called at the end of the computation for h_{k-1}. The top of the value stack now consists of
$h_{k-1}(n_1,\ldots,n_k),n_1,\ldots,n_k,h_{k-2}(n_1,\ldots,n_k),\ldots,h_1(n_1,\ldots,n_k)$.
Template `f-s`$(k-1)$ thus pops the elements $h_{k-1}(n_1,\ldots,n_k),n_1,\ldots,n_k$ from the stack to store them in local variables. It pushes $h_j(n_1,\ldots,n_k)$ back onto the value stack. The parameters $n_1,\ldots,n_k$ need not be pushed back onto the value stack, because we need them for the last time here for the call of h_k. Template `f-s`$(k-1)$ finishes by calling the template for h_k with parameters $n_1,\ldots,n_k$, the call stack with the template '`f-s`k' pushed on top of it, and the value stack.

Finally, the template `f-s`k is called at the end of the computation of h_k. The top of the value stack now consists of $h_k(n_1,\ldots,n_k),h_{k-1}(n_1,\ldots,n_k),\ldots,h_1(n_1,\ldots,n_k)$. These are popped from the stack and stored in local variables to be used as parameters in the call to `g` accompanied by the call stack and the value stack. Because the computation of g is the last step in the composition, we do not need to push a new continuation point onto the call stack. Rather the template `g` finishes by calling the continuation point left on the call stack by the function that called f.

Primitive Recursion

For primitive recursion, let $k \geq 0$ and let

$$\begin{aligned} f(n_1,\ldots,n_k,0) &= g(n_1,\ldots,n_k), \\ f(n_1,\ldots,n_k,m+1) &= h(n_1,\ldots,n_k,m,f(n_1,\ldots,n_k,m)) \end{aligned}$$

We code f as follows:

```
<xsl:template name="f">
  <xsl:param name="n-1"/>
  ...
  <xsl:param name="n-k"/>
  <xsl:param name="m"/>
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:if test='$m = 0'>
    <xsl:call-template name="g">
      <xsl:with-param name="n-1"/ select="$n-1"/>
      ...
      <xsl:with-param name="n-k" select="$n-k"/>
      <xsl:with-param name="call-stack" select="$call-stack"/>
      <xsl:with-param name="value-stack" select="$value-stack"/>
    </xsl:call-template>
  </xsl:if>
  <xsl:call-template name="f">
    <xsl:with-param name="n-1" select="$n-1"/>
```

```
    ...
    <xsl:with-param name="n-k" select="$n-k"/>
    <xsl:with-param name="m" select="$m - 1"/>
    <xsl:with-param name="call-stack" select="concat('f-c/',$call-stack)"/>
    <xsl:with-param name="value-stack"
                    select="concat($n-1,'/',...,'/',$n-k,'/',$m - 1,
                                   '/',$value-stack)"/>
  </xsl:call-template>
</xsl:template>
```

And the template to call *h*:

```
<xsl:template name="f-c">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rv" select="substring-before($value-stack,'/')"/>
  <xsl:variable name="vs1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs1,'/')"/>
  <xsl:variable name="vs2" select="substring-after($vs1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs2,'/')"/>
  <xsl:variable name="vs3" select="substring-after($vs2,'/')"/>
  ...
  <xsl:variable name="n-k" select="substring-before($vsk,'/')"/>
  <xsl:variable name="vs(k+1)" select="substring-after($vsk,'/')"/>
  <xsl:variable name="m" select="substring-before($vs(k+1),'/')"/>
  <xsl:variable name="vs" select="substring-after($vs(k+1),'/')"/>
  <xsl:call-template name="h">
    <xsl:with-param name="n-1" select="$n-1"/>
    ...
    <xsl:with-param name="n-k" select="$n-k"/>
    <xsl:with-param name="n-(k+1)" select="$m"/>
    <xsl:with-param name="n-(k+2)" select="$rv"/>
    <xsl:with-param name="call-stack" select="$call-stack"/>
    <xsl:with-param name="value-stack" select="$vs"/>
  </xsl:call-template>
</xsl:template>
```

In principle, template `f` provides a division by cases. If the last argument, *m*, is equal to 0, the template for *g* is called. If *m* is greater then 0, firstly template *f* is called recursively with *m* decreased by 1 and then the template for *h* is called to complete the computation. Thus template `f` tests if $m = 0$. If so, it calls the template for *g* with parameters $n_1, \ldots, n_k$, the call stack and the value stack. If $m \neq 0$, it pushes the parameters $n_1, \ldots, n_k, m-1$ onto the value stack for further use and calls itself recursively with parameters $n_1, \ldots, n_k, m-1$, the call stack with continuation point '`f-c`' pushed onto it, and the value stack.

Template `f-c` is called at the end of the recursive computation of $f(n_1, \ldots, n_k, m)$ and hence the value of $f(n_1, \ldots, n_k, m)$ lies on top of the value stack followed by the parameters $n_1, \ldots, n_k, m$. These elemets are taken from the stack and stored in local variables. Then the template for *h* is called with parameters $n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)$ and the call stack and value stack. Because the computation of *h* is the last step in the primitive

recursion, we do not need to push a new continuation point onto the call stack. Rather the template h finishes by calling the continuation point left on the call stack by the function that called f.

μ-Recursion

For μ-recursion, let $k \geq 0$ and f defined as

$$f(n_1,\ldots,n_k) = \begin{cases} \text{the least } m \text{ such that } g(n_1,\ldots,n_k,m) = 0, \\ \quad \text{if such an } m \text{ exists;} \\ 0 \text{ otherwise} \end{cases}$$

We code f as follows:

```
<xsl:template name="f">
  <xsl:param name="n-1"/>
  ...
  <xsl:param name="n-k"/>
  <xsl:param name="m" select="0"/>
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:call-template name="g">
    <xsl:with-param name="n-1" select="$n-1"/>
    ...
    <xsl:with-param name="n-k" select="$n-k"/>
    <xsl:with-param name="n-(k+1)" select="$m"/>
    <xsl:with-param name="call-stack" select="concat('mu-f/',$call-stack)"/>
    <xsl:with-param name="value-stack"
         select="concat($n-1,'/',...,'/',$n-k,'/',$m,'/',$value-stack)"/>
  </xsl:call-template>
</xsl:template>
```

And the template to process the result of the call to g:

```
<xsl:template name="mu-f">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rv" select="substring-before($value-stack,'/')"/>
  <xsl:variable name="vs1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs1,'/')"/>
  <xsl:variable name="vs2" select="substring-after($vs1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs2,'/')"/>
  <xsl:variable name="vs3" select="substring-after($vs2,'/')"/>
  ...
  <xsl:variable name="n-k" select="substring-before($vsk,'/')"/>
  <xsl:variable name="vs(k+1)" select="substring-after($vsk,'/')"/>
  <xsl:variable name="m" select="substring-before($vs(k+1),'/')"/>
  <xsl:variable name="vs" select="substring-after($vs(k+1),'/')"/>
  <xsl:if test='$rv != 0'>
    <xsl:call-template name="f">
      <xsl:with-param name="n-1" select="$n-1"/>
```

```
      ...
      <xsl:with-param name="n-k" select="$n-k"/>
      <xsl:with-param name="m" select="$m + 1"/>
      <xsl:with-param name="call-stack" select="$call-stack"/>
      <xsl:with-param name="value-stack" select="$vs"/>
    </xsl:call-template>
  </xsl:if>
  <xsl:call-template name="substring-before($call-stack, '/')">
    <xsl:with-param name="call-stack"
                    select="substring-after($call-stack, '/')"/>
    <xsl:with-param name="value-stack" select="concat($m,'/',$vs)"/>
  </xsl:call-template>
</xsl:template>
```

Function f is coded by a loop on parameter m starting with 0. The core of the loop consists of a call to g with the current value of m. If we found a null for g, we are finished and return m. If not, we increment m by 1 and loop on. Thus the template for f pushes the parameters $n_1,\ldots,n_k,m$ onto the value stack for later use by `mu-f`, pushes the continuation point '`mu-f`' onto the call stack and calls the template for g with parameters $n_1,\ldots,n_k,m$, the call stack and value stack. Note the line

```
<xsl:param name="m" select="0"/>
```

in the parameter block of template `f`. Here, we use the fact that a template may be called with some parameters left uninstantiated by the caller. The first call to `f` will have m uninstantiated, because m is the loop variable. The `select`-part provides a default value of 0. In later calls to `f` by template `mu-f` the variable m will be instantiated.

Template `mu-f` is called at the end of the computation of g. The top of the value stack consists of the elements $g(n_1,\ldots,n_k,m),n_1,\ldots,n_k,m$. These are popped from the stack and stored in local variables. If $g(n_1,\ldots,n_k,m)\neq 0$, we call `f` recursively with parameters $n_1,\ldots,n_k,m+1$, the call stack and the value stack to loop on. If $g(n_1,\ldots,n_k,m)=0$ we found the null we are looking for, push m as return value on the value stack and finish by calling the next continuation point from the call stack. Note that if $g(n_1,\ldots,n_k,m)$ has no null, we loop forever.

A Complete Style Sheet

The above section showed the translation of recursive functions into XSLT. There are still two minor items missing to complete the translation. First, we have to provide some framework information to get a well-defined style sheet. And second, we want to output the result of the computation. We therefore introduce additional XSLT-code at the beginning and end of the translation. Assuming that we want to calculate $f(m_1,m_2,\ldots,m_k)$ we introduce before the translation

```
<?xml version="1.0"?>

<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
               version="2.0">

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

```
<xsl:template match="/">
  <xsl:call-template name="f">
    <xsl:with-param name="n-1" select="m1"/>
    <xsl:with-param name="n-2" select="m2"/>
    ...
    <xsl:with-param name="n-3" select="mk"/>
    <xsl:with-param name="call-stack select="out/"/>
    <xsl:with-param name="value-stack" select="/">
  </xsl:call-template>
</xsl:template>

<xsl:template name="out">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack">
Result: <xsl:value-of select="substring-before($value-stack,'/')"/>
<xsl:text>&#10;</xsl:text>
</xsl:template>
```

The `<xsl:output .../>` is just there to produce a nicer output. The first template is there to start the computation. It matches with the super-root of the input document, the only node that must be present at every input document, and the first node to be processed. It calls the template for f, the function we want to compute, pasing the arguments to f. And it places the call to the output template on the call-stack to ensure that the output routine will be called at the end of the computation. The second template is the output template. It pops the result of the computation from the value-stack and prints it out.

After the translation we just add

```
</xsl:transform>
```

to complete the style sheet.

5 Correctness

Proposition 2 *Let f be a k-ary μ-recursive function and $n_1, \ldots, n_k \in \mathbb{N}$. After calling the XSLT-coding of f, the top of the value stack is the value of $f(n_1, \ldots, n_k)$.*

Proof. If f is a basic function, then $f(n_1, \ldots, n_k)$ is directly pushed onto the value stack:
If f is zero_k, then the line
`<xsl:with-param name="value-stack" select="concat('0/',$value-stack)"/>`
sets the returned value stack to be the old one with a 0 pushed onto it.
If f is $\pi_{k,j}$, then the line

```
    <xsl:with-param name="value-stack"
                    select="concat($n-j,'/',$value-stack)"/>
```

sets the returned value stack to be the old one with the value of the parameter n_j pushed onto it.
If f is succ, then the line

```
<xsl:with-param name="value-stack"
                select="concat($n + 1,'/',$value-stack)"/>
```

sets the returned value stack to be the old one with the value of the parameter $n+1$ pushed onto it.

If f is defined by primitive recursion, we distinguish two cases. If $n_k = 0$ then the template for f calls the template for g handing over all needed parameters. By hypothesis we can assume that after calling the coding of g there is $g(n_1,\ldots,n_{k-1})$ on top of the value stack. Since $f(n_1,\ldots,n_{k-1},0) = g(n_1,\ldots,n_{k-1})$ the value of $f(n_1,\ldots,n_k)$ forms the top the value stack.
If $n_k > 0$ then the template for f calls itself recursively with parameters $n_1,\ldots,n_{k-1},n_k-1$ pushing `f-c` on the call stack and the parameters $n_1,\ldots,n_{k-1},n_k-1$ on the value stack. By hypothesis we can assume that at the end of the recursive call to f the value of $f(n_1,\ldots,n_{k-1},n_k-1)$ will be on top of the value stack, and the next elements are the parameters $n_1,\ldots,n_{k-1},n_k-1$. Since `f-c` is on top of the call stack, computation continues with the template `f-c`. This template call h with parameters $n_1,\ldots,n_{k-1},n_k-1,f(n_1,\ldots,n_{k-1},n_k-1)$. By hypothesis we can assume that at the end of the call to h there will be $h(n_1,\ldots,n_{k-1},n_k-1,f(n_1,\ldots,n_{k-1},n_k-1))$, on the value stack, which is by definition equal to $f(n_1,\ldots,n_k)$.

If f is defined by μ-recursion then the template of f calls the template of g with parameters $n_1,\ldots,n_k,m$ where $m=0$ initially, pushing `mu-f` on the call stack and the parameters $n_1,\ldots,n_k,m$ on the value stack. By hypothesis we can assume that at the end of the call to g there will be $g(n_1,\ldots,n_k,m)$ on top of the value stack followed by the parameters $n_1,\ldots,n_k,m$. Computation continues with the call to the template `mu-f`. This template introduces a case distinction depending on the top of the value stack. If $g(n_1,\ldots,n_k,m)=0$ then m is pushed on the value stack, and computation is complete. This m is by definition the value of $f(n_1,\ldots,n_k)$. If $g(n_1,\ldots,n_k,m)\neq 0$ then then the template for f is called recursively with parameters $n_1,\ldots,n_k,m+1$. By hypothesis we can assume that at the end of the recursive call to f there will be $f(n_1,\ldots,n_k)$ on top of the value stack.

If f is defined by composition as $f(n_1,\ldots,n_k) = g(h_1(n_1,\ldots,n_k),\ldots,h_l(n_1,\ldots,n_k))$, then the template for f calls the template for h_1 with parameters $n_1,\ldots,n_k$ pushing `f-s1` on the call stack and the parameters $n_1,\ldots,n_k$ on the value stack. By hypothesis we can assume that at the end of the call to h_1 the value $h_1(n_1,\ldots,n_k)$ will be on top of the value stack followed by $n_1,\ldots,n_k$. Computation will continue with the template `f-s1`.
For $0 < j < l-1$ the template `f-s`j was called at the end of the call to h_j so that we can assume that $h_j(n_1,\ldots,n_k)$ is on top of the value stack followed by the parameters $n_1,\ldots,n_k$. The template `f-s`j pops all those values from the value stack and stores them in local variables. It pushes $h_j(n_1,\ldots,n_k)$ back onto the value stack and thereafter also pushes $n_1,\ldots,n_k$ onto the value stack, so that the order of the return value and the parameters is now reversed. The template calls the template for h_{j+1} with parameters $n_1,\ldots,n_k$ pushing `f-s`$j+1$ onto the call stack.
The template `f-s`$l-1$ was called at the end of the call to h_{l-1} so that we can assume that $h_{l-1}(n_1,\ldots,n_k)$ is on top of the value stack followed by the parameters $n_1,\ldots,n_k$. The template `f-s`$l-1$ pops all those values from the value stack and stores them in

local variables. It pushes $h_{l-1}(n_1,\ldots,n_k)$ back onto the value stack and calls the template for h_l with parameters $n_1,\ldots,n_k$ pushing `f-s`l onto the call stack.
The template `f-s`l was called at the end of the call to h_l so that we can assume that the top of the value stack now consists of the values $h_l(n_1,\ldots,n_k), h_{l-1}(n_1,\ldots,n_k), h_{l-2}(n_1,\ldots,n_k),\ldots,h_1(n_1,\ldots,n_k)$. The template `f-s`$l$ takes these from the value stack and uses them in the right order as parameters in the call to g. At the end of the call to g we can assume that the top of the value stack consists of $g(h_1(n_1,\ldots,n_k),\ldots,h_l(n_1,\ldots,n_k))$.
■

6 Coding μ-Recursive Functions in XQuery

The following coding of μ-recursive functions shows that XQuery [4] is also Turing-complete. XQuery is recommended by the W3C as a query language that human users can use to query XML documents. Coding μ-recursive functions in XQuery is simpler than in XSLT. Because XQuery offers full recursion, we do not need to emulate it by means of stacks. XQuery is a strongly typed language; parameters and return values of a function have types. In our case, the type for all parameters and return values is *nonNegativeInteger*. This data type is exactly the one of the natural numbers, as defined in [3].

The syntax for defining new functions in XQuery is similar to the one of the programming language Java or C. The head of a function definition has the prototypical format
`define function` fname `(`Parameters`) returns` Datatype
where fname is the name of the function, Datatype the data type of the return value of the function, and Parameters is a coma separated sequence of Datatype $Variablename pairs. The body of a function definition constists of a sequence of expressions enclosed by braces ({}). The only structure providing expression of XQuery we need is the conditional
`if (`Expr1`) then` Expr2 `else` Expr3
meaning obviously that if Expr1 evaluates to true, Expr2 is evaluated, otherwise Expr3 is evaluated. XQuery provides the function `eq` for testing equality of two numerical values.

In the following, we use the abbriviation `Nat` for the datatype `xs:nonNegativeInteger` to enhance readability.

Basic functions

Let $k \geq 0$. We code $\mathsf{zero}_k(n_1,\ldots,n_k)$ as follows:

```
define function zero-k(Nat $n-1, ..., Nat $n-k) returns Nat
{ 0 }
```

Let $k \geq j > 0$. We code $\pi_{k,j}(n_1,\ldots,n_k)$ as follows:

```
define function pi-k-j(Nat $n-1, ..., Nat $n-k ) returns Nat
{ $n-j }
```

We code succ(n) as follows:

```
define function succ(Nat $n) returns Nat
{ $n + 1 }
```

Composition

Let $k, l \geq 0$. We code the composition $f(n_1, \ldots, n_l) = g(h_1(n_1, \ldots, n_l), \ldots, h_k(n_1, \ldots, n_l))$ as follows:

```
define function f(Nat $n-1, ..., Nat $n-l) returns Nat
{
  g(h-1($n-1,...,$n-l), ..., h-k($n-1,...,$n-l))
}
```

Primitive recursion

For primitive recursion, let $k \geq 0$ and let

$$\begin{aligned} f(n_1, \ldots, n_k, 0) &= g(n_1, \ldots, n_k), \\ f(n_1, \ldots, n_k, m+1) &= h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)) \end{aligned}$$

We code f as follows:

```
define function f(Nat $n-1, ..., Nat $n-k, Nat $m) returns Nat
{
  if ($m eq 0) then g($n-1,...,$n-k)
  else h($n-1,...,$n-k,$m - 1, f($n-1,...,$n-k,$m - 1))
}
```

μ-recursion

For μ-recursion, let $k \geq 0$ and f defined as

$$f(n_1, \ldots, n_k) = \begin{cases} \text{the least } m \text{ such that } g(n_1, \ldots, n_k, m) = 0, \\ \quad \text{if such an } m \text{ exists;} \\ 0 \text{ otherwise} \end{cases}$$

We code f as follows:

```
define function f(Nat $n-1, ..., Nat $n-k) returns Nat
{
  mu-f($n-1,...,$n-k,0)
}

define function mu-f(Nat $n-1, ..., Nat $n-k, Nat $m) returns Nat
{
  if (g($n-1,...,$n-k,$m) eq 0)
  then $m
  else mu-f($n-1,...,$n-k,$m + 1)
}
```

That μ-recursion is coded by two functions is a consequence of the fact that XQuery does not offer optional arguments. So function `f` serves as an interface function to call `mu-f`, which has one parameter more, the one on which we do minimisation.

7 Conclusion

We provided a proof for the Turing-completeness of XSLT and XQuery by coding μ-recursive functions. XPath, being a component of both, provides the arithmetics while XSLT and XQuery provide the recursion. In the case of XQuery the coding is straightforward, because XQuery allows the definition of (recursive) functions. For XSLT, there was a little more work to be done, because we had to hand-code the return from a recursive function call using a call-stack.

There is probably quite a number of ways to prove Turing-completeness of both XQuery and XSLT, just because the languages provide so many facilities. We think the proof presented for XQuery is likely to be the shortest one can find. It is a short one-to-one translation. We are also of the oppinion that it will be difficult to find a shorter proof for XSLT. Admitted we use two stacks and a two-stack machine is Turing-complete. But a complete coding of such a machine in XSLT would not be shorter than the one presented here. Most of the "lenght" of the coding is to be assigned to the fact that XML and XSLT are so very verbose, a problem that every coding faces. Apart from that, our coding is really just recursive function calls and passing caller and value stack parameters around.

Since both XSLT and XQuery are Turing-complete, they are interchangeable on a theoretical level. From a user's perspective, there are clear differences. A recursive transformation of a document is simpler to define in XSLT, while queries can quicker be coded in XQuery. The expressive power both provide seems an almost natural choice when defining a languange that is convenient for users [5]. They probably often demand expressive power and care a lot less about computability and complexity issues.

Acknowledgements

I would like to thank Uwe Mönnich, Frank Morawietz, Frank Neven, Helmut Seidl, Klaus Schulz, and Thomas Schwentick for helpful comments and interesting discussions.

References

[1] Anders Berglund, Scott Boag, Don Chamberlin, Mary Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. Technical report, W3C, 2001. `http://www.w3.org/TR/xpath20/`.

[2] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A Formal Model for an Expressive Fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[3] Paul Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Technical report, W3C, 2001. http://www.w3.org/TR/xmlschema-2/.

[4] Scott Boag, Don Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. Technical report, W3C, 2001. Working draft, http://www.w3.org/TR/xquery/.

[5] Jon Bosak. XML, Java, and the Future of the Web. Technical report, Sun Microsystems, 1997. http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm.

[6] James Clark. XSL Transformations (XSLT), Version 1.0. Technical report, W3C, 1999. http://www.w3.org/TR/xslt.

[7] James Clark and Steve DeRose. XML Path Language (XPath) 1.0. Technical report, W3C, 1999. http://www.w3.org/TR/xpath.

[8] World Wide Web Consortium. Extensible Markup Language (XML). Technical report, W3C, 1999. http://www.w3.org/XML/.

[9] Michael Kay. XSL Transformations (XSLT), Version 2.0. Technical report, W3C, 2001. http://www.w3.org/TR/xslt20/.

[10] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[11] Harry Lewis and Christos Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 2nd edition, 1998.

[12] Frank Neven. On the Power of Walking for Querying Tree-Structured Data. In Lucian Popa, editor, *Proceedings PODS 2002*, 2002.