Extreme Markup Languages 2004

Montréal, Québec
August 2-6, 2004

A Simple Proof for the Turing-Completeness of XSLT and XQuery

Stephan Kepser
University of Tübingen

Abstract

The World Wide Web Consortium recommends both XSLT and XQuery as query languages for XML documents. XSLT, originally designed to transform XML into XSL-FO, 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. Both languages are known to be Turing-complete. We provide here a very simple proof of Turing-completeness of XSLT and XQuery by coding μ-recursive functions thereby showing that Turing-completeness is a consequence of a few basic and fundamental features of both languages.

A Conference of IDEAlliance
Extreme Markup 2004 Languages®

A Simple Proof for the Turing-Completeness of XSLT and XQuery

Table of Contents

Introduction....................................1
μ-Recursive Functions....................................2
Some XSLT....................................3
Coding μ-Recursive Functions in XSLT....................................4
    Basic Functions....................................5
    Composition....................................6
    Primitive Recursion....................................7
    μ-Recursion....................................9
    A Complete Style Sheet....................................10
Correctness....................................10
Coding μ-Recursive Functions in XSLT Using Stylesheet Functions....................................12
Coding μ-Recursive Functions in XQuery....................................14
Conclusion....................................16
Acknowledgements....................................17
Bibliography....................................17
The Author....................................18

A Conference of IDEAlliance
Extreme Markup Languages®
2004

A Simple Proof for the Turing-Completeness of XSLT and XQuery

Stephan Kepser

§ Introduction

The World Wide Web Consortium (W3C) recommends both XSLT and XQuery as query languages for XML documents. Query languages in this sense comprise two components. The first component identifies relevant substructures of the documents to be queried. For XSLT and XQuery, XPath forms the core of this component. The second component allows to arrange the matches and to provide structured output. This distinction is a mostly conceptual one. In practical query languages, these components are often not clearly separated. An example is the FLWOR-expression of XQuery, which unites matching and arranging tasks.

XSLT (X Style sheet Language Transformations [XSLT 1.0][XSLT 2.0]) is the recommendation of the W3C for an XML [XML] style sheet language. The original primary role of XSLT was to allow users to write transformations of XML to XSL-FO, thus describing the presentation of XML documents. Nowadays, XSLT allows for arbitrary transformation from one XML document to another, and many people use it 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.

Turing-completeness is a statement on the expressive power of a query language, a programming language, or an arbitrary computational model. It states that everything that can be computed with a Turing machine can be computed in that language or computational model. Since computation with Turing machines is the most powerful model of computation known to exist, Turing-completeness of a language basically means that everything that can be computed at all can be computed with implementations of the language. For programming languages, this is a desirable property, because is states that the language does not restrict the user in what he or she may wish to express or compute with the language. For a query language, the situation is more complicated, because there is a price to be paid for the high expressive power, and this price is that it is possible to write queries which require enormous amounts of time to evaluate or even queries that cannot be evaluated at all. Since XSLT is a general language for the transformation of XML documents, it can be seen as a programming language as much as a query language. Therefore it is a sensible approach not to restrict the type of transformations expressible in XSLT. In other words, Turing-completeness is a desirable property, because users should be able to decide freely which types of transformations they want to use.

Turing-completeness of XSLT seems to be a folklore result in the XSLT programmers community. There is even an implementation of Turing-machines in XSLT [TMML] available. Thus we certainly do not claim the result to be new. The advantage of the present proof is that it is short, simple, and clear. In opposite to a Turing-machine implementation it can be understood and checked easily. It also shows that there are not many XSLT features needed to prove Turing-completeness. It is not the case that one needs remote complicated features or sophistication. Basically all it takes is recursion and very basic arithmetics (simple addition). Amongst other things this means that one cannot just remove a rarely used feature from XSLT in the hope to obtain a subset of XSLT which is to be situated in a lower complexity class.

Previous work on the expressive power of XSLT was mostly concerned with fragments of XSLT. Neven et al. [Bex] [Neven] 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 [XQuery 1.0] as a query language for human use. XQuery is based on XPath [XPath 1.0][XPath 2.0], 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. XQuery grew out of Quilt [Quilt], an XML query language that was deliberately chosen to be Turing-complete. Hence XQuery is also Turing-complete. To show this in a simple way we provide a coding of μ-recursive functions along the same lines as we do it for XSLT. This coding is a lot simpler to perform in the case of XQuery, because (unrestricted) recursion is part of the language; it need not be emulated.

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. XQuery is strictly typed. XSLT is more liberal about typing. A stylesheet may be strictly typed. But types can often also be left out. In that case the programmer relies on the built-in type conversion mechanisms of XSLT. So, perhaps apart from typing issues, XSLT is more machine oriented while XQuery is more suitable for human use.

This paper is organised as follows. We start with an explanation and the formal definition of μ-recursive functions. Section “Some XSLT” contains a short overview over those XSLT constructs that are needed for the coding of μrecursive functions. The coding itself is provided in Section “Coding μ-Recursive Functions in XSLT”. Since it is not simple to see that this coding is correct, we provide an extensive argument that it is in the next section (“Correctness”). Section “Coding μ-Recursive Functions in XSLT Using Stylesheet Functions” shows that there is a much simpler coding available if one uses XSLT's stylesheet functions. In Section “Coding μ-Recursive Functions in XQuery”, we give a coding of μ-recursive functions in XQuery.

§ μ-Recursive Functions

A Turing machine is an abstract concept of defining computations [Lewis]. Its components are an infinite tape, a finite set of states, and a finite set of instructions. At any moment, the machine is in one of the states. The tape is organised in cells; the machine has a read-write head that can read one cell from the tape or write a symbol onto a cell. A computation step involves changing the state of the machine, writing a symbol to the tape and moving the read-write head of the machine one cell to the left or to the right, depending on the instruction set. The exact details do not matter here. What is important, though, is that this relatively simple concept of computation has proven to be very powerful and universal. Many different concepts of computation have been proposed. All of them were found to be equivalent to Turing machines or less powerful. It is therefore assumed that every “effective computation” or “algorithm” can be programmed to run on a Turing machine. This assumption is known as Church's thesis.

Amongst the many equivalent definition of Turing-complete computations there exists a proposal by Kleene [Kleene] using so-called μ-*recursive functions* for computations on natural numbers. Our definition of μ-recursive functions follows the exposition by Lewis and Papadimitriou [Lewis]. μ-recursive functions are inductively defined. The base forms a set of functions that always return the number 0, a set of functions that select a component out of a tuple, and the successor function, which adds 1 to any given natural number. These basic functions can be combined to give more complex functions using three different schemata. The simplest one is composition of functions. Primitive recursion, the second schema, is the counterpart of inductive definitions on natural numbers. It allows to define the value of a function for some number n based on the value this function returns for its predecessor n-1. The recursion is based in the function value for the number 0. The third schema is μ-recursion. It provides an unbounded search mechanism. The formal definition for μ-recursive functions follows.

Definition 1

The *basic* functions are the following:

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

The *composition* is defined as follows: Let $k, l \geq 0$, let $g : \mathbb{N}^k \rightarrow \mathbb{N}$ be a k-ary function, and let $h_1, \dots, h_k$ be l-ary functions. Then the composition of g with $h_1, \dots, h_k$ is the l-ary function $f(n_1, \dots, n_l) = g(h_1(n_1, \dots, n_l), \dots, h_k(n_1, \dots, n_l))$.

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

- $f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k)$,
- $f(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m))$

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 $f(n_1, \ldots, n_k) =$

- the least m such that $g(n_1, \ldots, n_k, m) = 0$
- 0 otherwise

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

It is well known that μ-recursion is required to define all Turing-computable functions. μ-recursion is the only way to provide the equivalent of unbounded search. For a discusion of this issue including an equivalence proof to Turing machines, see, e.g., [Lewis]. 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.

§ Some 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 [XSLT 1.0][XSLT 2.0], 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 provides the 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 are defined in XPath [XPath 1.0] [XPath 2.0], the standard for expressions in XSLT.

Templates are XSLT's way of expressing procedures.

```
<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 subroutine in a program code. After completion of the called template execution of the calling template is continued, as is standard for procedure calls.

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. It may be intersting to note that in XPath 2.0 integers – and therefore natural numbers – are at least theoretically of arbitrary size (see [XML Schema]).

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. In other words, the leftmost element of the string is the top of the stack. To get the rest of the stack, we use *substing-after*.

§ Coding μ-Recursive Functions in XSLT

The subsequent coding is based on the following assumption:

- There is no way to define functions in XSLT.

A function in this sense is a procedure or subroutine that can return an atomic value such as a string or an integer value to the calling routine. This assumption is true for XSLT 1.0. For XSLT 2.0, the assumption is no longer true. The ability to use functions in the coding of μ-recursive functions simplifies this coding a lot. The details are discussed in Section “Coding μ-Recursive Functions in XSLT Using Stylesheet Functions”. Since it is the aim of the present paper to show that Turing-completeness is a consequence of a few key features of XSLT, we still believe it to be justified to assume there to be no functions, because they are not part of the original core of XSLT.

Thus the key problem of computing with templates is the (assumed) lack of the ability to return a value from a subcomputation. How can one pass on the result of a subcomputation? We propose to recode the control flow for templates and to pass on results of prior computations by means of stacks. Here is a simple example. Suppose template A contains a call to template B. We divide template A into two pieces. The first part contains everything up to and including the call of B. The second part forms a new template A' that contains everything following the call of B. Now at the end of the template B we do not simply return to A, rather we insert a call to A'. If both B and A' are equipped with a stack of results computed so far as a parameter, we can transfer knowledge of previous computation results from A to B and B to A' by passing on this stack in each template call. If we want a generic coding of B, one in which template B may be called from arbitrary templates, we do not know which template to call at the end of the computation of B. A solution to this problem consists in the use of another stack, a stack for names of templates to be called to continue computation. If there is such a stack as a parameter of all templates, then template A could “tell” template B to continue with A' by simply pushing the name A' on that stack. At the end of template B, the top of the stack would contain the name of the template to be called next, namely A'.

Thus 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. This stack is the *call stack*. 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. This stack is the *value stack*. These two stacks will be parameters of all templates, being passed on from one template to the next.

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 $\text{zero}_k(n_1, \ldots, n_k)$ 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>
```

The core of this code is `concat('0/',$value-stack)`, which pushes 0 onto the value stack. All the rest is parameter and continuation point handling. The parameter block is there to receive the parameters n_1, … , n_k of the calling function, although they are not needed for “computing” the value of the function. The last two lines in the parameter block receive the call stack and the value stack. The next part is already the recursive call to the next template. To determine which one that is we just pop it from the call stack.

Please note that the `k` serves as an external parameter in the name `zero-k` or the parameter name `n-k`. That is to say, in a actual instantiation the `k` is replaced by the natural number the parameter represents. An example would be `zero-5` and `n-5` for $k = 5$.

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>
```

Here again, `k` and `j` serve as external parameters to the template and variable names. An instantiated template name could be, e.g., `pi-5-2`. These external parameters `k`, `l`, and `j` will be used in most subsequent codings.

Coding of the successor function succ(n):

```
<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 $f(n_1, \dots, n_l) = g(h_1(n_1, \dots, n_l), \dots, h_k(n_1, \dots, n_l))$ by a series of templates:

```
<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-s-1/',$call-stack)"/>
    <xsl:with-param name="value-stack"
         select="concat($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-s-j;">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rv" select="substring-before($value-stack,'/')"/>
  <xsl:variable name="vs-1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs-1,'/')"/>
  <xsl:variable name="vs-2" select="substring-after($vs-1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs-2,'/')"/>
  <xsl:variable name="vs-3" select="substring-after($vs-2,'/')"/>
  …
  <xsl:variable name="n-l" select="substring-before($vs-l,'/')"/>
  <xsl:variable name="vs-l+1" select="substring-after($vs-l,'/')"/>
  <xsl:variable name="vss-1" select="concat($rv,'/',$vs-l+1)"/>
  <xsl:variable name="vss-2"
       select="concat($n-1,'/',$n-2,'/',… ,'/',$n-l,'/',$vss-1)"/>
  <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/',$call-stack)"/>
    <xsl:with-param name="value-stack" select="$vss-2"/>
  </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 $n_1, \dots, n_l$ 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($value-stack,’/’)"/>
  <xsl:variable name="vs-1" select="substring-after($value-stack,’/’)"/>
  <xsl:variable name="n-1" select="substring-before($vs-1,’/’)"/>
  <xsl:variable name="vs-2" select="substring-after($vs-1,’/’)"/>
  <xsl:variable name="n-2" select="substring-before($vs-2,’/’)"/>
  <xsl:variable name="vs-3" select="substring-after($vs-2,’/’)"/>
  …
  <xsl:variable name="n-l" select="substring-before($vs-l,'/')"/>
  <xsl:variable name="vs-l+1" select="substring-after($vs-l,'/')"/>
  <xsl:variable name="vss" select="concat($rv,'/',$vs-l+1)"/>
  <xsl:call-template name="h-k">
    <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-k/',$call-stack)"/>
    <xsl:with-param name="value-stack" select="$vss"/>
  </xsl:call-template>
</xsl:template>
```

And finally the template to call g :

```
<xsl:template name="f-s-k">
  <xsl:param name="call-stack"/>
  <xsl:param name="value-stack"/>
  <xsl:variable name="rv-k" select="substring-before($value-stack,’/’)"/>
  <xsl:variable name="vs-1" select="substring-after($value-stack,’/’)"/>
  <xsl:variable name="rv-k-1" select="substring-before($vs-1,’/’)"/>
  <xsl:variable name="vs-2" select="substring-after($vs-1,’/’)"/>
  …
  <xsl:variable name="rv-1" select="substring-before($vs-k-1,'/')"/>
  <xsl:variable name="vs-k" select="substring-after($vs-k-1,'/')"/>
  <xsl:call-template name="g">
    <xsl:with-param name="n-1"/ select="$rv-1"/>
    …
    <xsl:with-param name="n-k" select="$rv-k"/>
    <xsl:with-param name="call-stack" select="$call-stack"/>
    <xsl:with-param name="value-stack" select="$vs-k"/>
  </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-s-1`' 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

- $f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k)$,
- $f(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m))$

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($value-stack,'/')"/>
  <xsl:variable name="vs-1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs-1,'/')"/>
  <xsl:variable name="vs-2" select="substring-after($vs-1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs-2,'/')"/>
  <xsl:variable name="vs-3" select="substring-after($vs-2,'/')"/>
  …
  <xsl:variable name="n-k" select="substring-before($vs-k,'/')"/>
  <xsl:variable name="vs-k+1" select="substring-after($vs-k,'/')"/>
  <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 elements 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 $f(n_1, \dots , n_k) =$

- the least *m* such that $g(n_1, \dots , n_k , m) = 0$, if such an *m* exists,
- 0 otherwise.

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($value-stack,'/')"/>
  <xsl:variable name="vs-1" select="substring-after($value-stack,'/')"/>
  <xsl:variable name="n-1" select="substring-before($vs-1,'/')"/>
  <xsl:variable name="vs-2" select="substring-after($vs-1,'/')"/>
  <xsl:variable name="n-2" select="substring-before($vs-2,'/')"/>
  <xsl:variable name="vs-3" select="substring-after($vs-2,'/')"/>
  …
  <xsl:variable name="n-k" select="substring-before($vs-k,'/')"/>
  <xsl:variable name="vs-k+1" select="substring-after($vs-k,'/')"/>
  <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 $n_1, \dots , n_k , m$ 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 $n_1, \dots , n_k , m$, 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 $g(n_1, \dots , n_k , m)$, $n_1, \dots , n_k , m$. These are popped from the stack and stored in local variables. If $g(n_1, \dots , n_k , m) \neq 0$, we call `f` recursively with parameters $n_1, \dots , n_k , m +1$, the call stack and the value stack to loop on. If $g(n_1, \dots , n_k , m) = 0$ 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 $g(n_1, \dots , n_k , m)$ 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 $f(m_1, m_2, \dots , m_k)$ 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="m-1"/>
    <xsl:with-param name="n-2" select="m-2"/>
    …
    <xsl:with-param name="n-3" select="m-k"/>
    <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>
</xsl:text>
</xsl:template>
```

The `<xsl:output method=… />` 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, passing 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.

§ Correctness

Proposition 1 Let *f* be a *k* -ary μ-recursive function and $n_1, \dots , n_k \in \mathbb{N}$. After calling the XSLT-coding of *f*, the top of the value stack is the value of $f(n_1, \dots , n_k)$.

Proof

Since μ-recursive functions are defined by structural recursion, the structure of the proof follows this recursion. The base case are of course the basic functions.

If f is a basic function, then $f(n_1, \dots , n_k)$ 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:

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

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

If f is defined by composition as $f(n_1, \dots , n_k) = g(h_1(n_1, \dots , n_k), \dots , h_l (n_1, \dots , n_k))$, then the template for f calls the template for h_1 with parameters $n_1, \dots , n_k$ pushing `f-s-1` on the call stack and the parameters $n_1, \dots , n_k$ on the value stack. By hypothesis we can assume that at the end of the call to h_1 the value $h_1(n_1, \dots , n_k)$ will be on top of the value stack followed by $n_1, \dots , n_k$. Computation will continue with the template `f-s-1`.

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 $h_j (n_1, \dots , n_k)$ is on top of the value stack followed by the parameters $n_1, \dots , n_k$. The template `f-s-j` pops all those values from the value stack and stores them in local variables. It pushes $h_j (n_1, \dots , n_k)$ back onto the value stack and thereafter also pushes $n_1, \dots , n_k$ 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 $n_1, \dots , n_k$ 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 $h_{l-1}(n_1, \dots , n_k)$ is on top of the value stack followed by the parameters $n_1, \dots , n_k$. The template `f-s-l-1` pops all those values from the value stack and stores them in local variables. It pushes $h_{l-1}(n_1, \dots, n_k)$ back onto the value stack and calls the template for h_l with parameters $n_1, \dots, n_k$ 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 $h_l(n_1, \dots, n_k)$, $h_{l-1}(n_1, \dots, n_k)$, $h_{l-2}(n_1, \dots, n_k)$, … , $h_1(n_1, \dots, n_k)$. 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 $g(h_1(n_1, \dots, n_k), \dots, h_l(n_1, \dots, n_k))$.

This completes the proof.

Completeness of the translation is simply given by the fact that we provide a translation that follows the structural definition of μ-recursive functions.

It may be astonishing that the only output of the coding is a single natural number whereas normally XSLT produces an XML document. Why is there no need to show that XSLT can produce arbitrary XML documents? The answer is that we have done that, but we have done that in an indirect fashion. The argument follows the main idea in the proof that μ-recursive functions themselves are Turing-complete. The key idea of how to emulate arbitrary Turing machine computations with recursive arithmetic functions is Gödelisation, i.e., the (injective) coding of arbitrary strings by natural numbers. The arithmetics available allows one to pull the intended “meaning” of a string out of its numerical coding. Hence, manipulation of stings can thereafter be emulated by computations on natural numbers. The case is similar for XSLT. The fact that we can perform arbitrary computations on natural numbers and output the resulting number means we can produce arbitrary XML documents. In other words, we output XML documents, but not as clear text, but rather in a numerical coding. For practical purposes this coding would have to be made explicit. But from a theoretical point of view there is no difference between an output in clear text and an output as a coding by natural numbers as long as one can be computed from the other.

§ Coding μ-Recursive Functions in XSLT Using Stylesheet Functions

In opposite to the assumption in Section “Coding μ-Recursive Functions in XSLT”, there is a way to define functions in XSLT 2.0. Actually there are two. Firstly, templates can return atomic values such as integer values. And secondly, one can define stylesheet functions that can be used at each place where an XPath function can be used. Since the option to return a value from a template call is a bit unusual, we provide a simple example on how to do this.

```
<xsl:template name="f">
  <xsl:param name="i"/>
  <xsl:value-of select="2*$i"/>
</xsl:template>

<xsl:template name="g">
  <xsl:variable name="j" as="xs:integer">
    <xsl:call-template name="f">
      <xsl:with-param name="i" select="3"/>
    </xsl:call-template>
  </xsl:variable>

  <!-- now, $j contains the value of f(3), namely 6 -->
</xsl:template>
```

The method works as follows. Template `f` produces a temporary tree (see [XSLT 2.0], Section 9.4) consisting of a single element node with name 2*$i. This is not the same as the numeric value 2*$i. This temporary tree is the content of the variable `j` in template `g`, but the content is converted to an integer. The conversion is possible since the tree consists of a single element with an integer as name of the element. Therefore variable `j` is bound to the integer value of f(3). Using this method, μ-recursive functions can be coded without using stacks and without complicated arrangements of continuation points for subsequent computations. But there is an even simpler way. If one uses stylesheet functions, the coding can be done straight forwardly.

Stylesheet functions are functions defined by the user or programmer inside a stylesheet (see [XSLT 2.0], Section 10.3). They can be used at every place where an XPath function can be used. Hence they extend the range of XPath functions. Definition of parameters and variables as well as the control flow elements are shared with template definitions. We will use the typing facilities that XSLT offers to indicate clearly that we do computations on natural numbers. All parameters and return value are of this datatype. It is specifed as *nonNegativeInteger* in [XML Schema].

In the following, we use the abbreviation `Nat` for the datatype `xs:nonNegativeInteger` to enhance readability. Since all stylesheet functions must have a prefixed name, we choose the prefix `mu:` for μ-recursive functions.

Basic functions

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

```
<xsl:function name="mu:zero-k" as="Nat">
  <xsl:param name="n-1" as="Nat"/>
  …
  <xsl:param name="n-k" as="Nat"/>
  0
</xsl:function>
```

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

```
<xsl:function name="mu:pi-k-j" as="Nat">
  <xsl:param name="n-1" as="Nat"/>
  …
  <xsl:param name="n-k" as="Nat"/>
  $n-j
</xsl:function>
```

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

```
<xsl:function name="mu:succ" as="Nat">
  <xsl:param name="n" as="Nat"/>
  <xsl:value-of select="$n + 1"/>
</xsl:function>
```

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:function name="mu:f" as="Nat">
  <xsl:param name="n-1" as="Nat"/>
  …
  <xsl:param name="n-l" as="Nat"/>
  <xsl:value-of select="mu:g(mu:h-1($n-1,… ,$n-l), … , mu:h-k($n-1,… ,$n-l))"/>
</xsl:function>
```

Primitive recursion

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

- $f(n_1, \ldots , n_k , 0) = g(n_1, \ldots , n_k)$,
- $f(n_1, \ldots , n_k , m +1) = h(n_1, \ldots , n_k , m, f(n_1, \ldots , n_k , m))$

We code f as follows:

```
<xsl:function name="mu:f" as="Nat">
  <xsl:param name="n-1" as="Nat"/>
  …
  <xsl:param name="n-k" as="Nat"/>
  <xsl:param name="m" as="Nat"/>
  <xsl:choose>
    <when test="$m = 0">
      <xsl:value-of select="mu:g($n-1, … ,$n-k)"/>
    </xsl:when>
    <xsl:otherwise>
     <xsl:value-of select="mu:h($n-1, … ,$n-k,$m - 1, mu:f($n-1, … ,$n-k,$m - 1))"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

μ-recursion

For μ-recursion, let $k \geq 0$ and f defined as $f(n_1, \dots , n_k) =$

- the least m such that $g(n_1, \dots , n_k , m) = 0$, if such an m exists,
- 0 otherwise.

We code *f* as follows:

```

<xsl:function name="mu:f" as="Nat">
  <xsl:param name="n-1" as="Nat"/>
  …
  <xsl:param name="n-k" as="Nat"/>
  <xsl:value-of select="mu:mu-f($n-1, … , $n-k, 0)"/>
</xsl:function>

<xsl:function name="mu:mu-f" as="Nat">
  <xsl:param name="n-1" as="Nat"/>
  …
  <xsl:param name="n-k" as="Nat"/>
  <xsl:param name="m" as="Nat"/>
  <xsl:choose>
    <when test="mu:g($n-1, … ,$n-k, $m)  = 0">
      <xsl:value-of select="$m"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="mu:mu-f($n-1, … ,$n-k,$m + 1)"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:function>
```

The stylesheet function `mu:f` is a wrapping function. It just adds one more argument, the one on which we minimise, and calls `mu:mu-f`, which is the function that really computes the smallest null (if it exists). The wrapper is necessary because there cannot be any optional arguments in a stylesheet function definition.

§ Coding μ-Recursive Functions in XQuery

The following coding of μ-recursive functions shows that XQuery [XQuery 1.0] 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 without functions. 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 [XML Schema].

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 consists 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.

Again we use the abbreviation `Nat` for the datatype `xs:nonNegativeInteger` to enhance readability.

Basic functions

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

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

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

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

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

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

Composition

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

- $f(n_1, \dots , n_k , 0) = g(n_1, \dots , n_k),$
- $f(n_1, \dots , n_k , m +1) = h(n_1, \dots , n_k , m, f(n_1, \dots , n_k , m))$

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 $f(n_1, \dots , n_k) =$

- the least m such that $g(n_1, \dots , n_k , m) = 0$, if such an m exists,
- 0 otherwise.

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.

The codings of μ-recursive functions in XQuery and in XSLT using stylesheet functions are obviously very similar. This is a consequence of the fact that the structural means used in both codings are similar. After all, we code functions (μ-recursive ones) using functions. Existing differences can mostly be attributed to the differences in syntax. The only exception may perhaps be that the `if-then-else` construct of XQuery is a little bit simpler than the `xsl:choose` construct of XSLT.

§ Conclusion

We provided a simple 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 straight-forward, because XQuery allows the definition of (recursive) functions. For XSLT, there was a little more work to be done – at least, if one assumes that there are no functions, – because we had to hand-code the continuation point of a computation following 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 opinion 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 “length” 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 call and value stack parameters around. And the coding that uses stylesheet functions is as concise as can be.

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 be coded quicker in XQuery. XQuery is strictly typed, whereas XSLT offers the options of either using a strict typing regiment or relying on built-in type conversions. For database applications, strict typing is probably the preferable choice. But for document applications, it is less clear which option to choose. The use of XML syntax for XSLT programs allows to feed these these programs as input into other XML applications or even other XSLT programs. Thus the XML syntax can be regarded as advantageous for the automatic processing of XSLT programs. On the other hand, XSLT programs are very cumbersome to read or write for humans, whereas the XQuery syntax is a lot more human user friendly.

XSLT and XQuery both use a navigational approach in the sense that their variable binding paradigm requires the querier to specify path navigations through the document. In contrast to this, there are query languages that are pattern based. Their variable binding paradigm is that of mathematical logics, i.e., a querier specifies patterns including variables. Arguably, these make complex queries easier to specify and read. Examples are the languages UnQL [UnQL] and Xcerpt [Xcerpt], which is based on logic programming concepts.

And there is finally the question about how much expressive power is required. There are arguments (see, e.g., [Bosak]) that a query language that is to be convenient for users almost naturally ends up being Turing-complete. From the point of view of automatic querying and conversion of documents or databases it is not too difficult to see this. But there is another view on this issue, which comes from experience with traditional relational databases and SQL. The complexity issues involved with Turing-complete query languages are not neglectable when documents are large. It is not clear that a Turing-complete query language is the right choice if queries that make full use of the expressive power of the language do not get answered in a suitable amount of time. The alternative, which was also chosen in the traditional database framework, is the restriction of the expressive power. Perhaps the best way to cope with this problem is to say that these alternatives may not necessarily be mutually exclusive. That is to say, there are definitely applications for Turing-complete query languages, and XSLT and XQuery serve their purpose well. But there are very likely also applications that do not demand the expressive power XSLT or XQuery offer. We therefore think it is worthwhile to analyse user demands and try to define a query language that has a restricted expressive power and thus fits into a lower complexity class. If there is a suitable amount of applications for such a language, the definition of such a language would make sense because applications could be processed considerably faster without restricting user demands.

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. I would also like to thank several anonymous referees, whose comments and suggestions helped improving this paper.

Bibliography

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

[Bosak] 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**.

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

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

[Neven] Frank Neven. On the Power of Walking for Querying Tree-Structured Data. In Lucian Popa, editor, *Proceedings PODS 2002*, pp. 77-84, 2002.

[Quilt] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In Dan Suciu and Gottfried Vossen, editors, *The World Wide Web and Databases. Third International Workshop WebDB2000*, pp. 1-25, Springer, LNCS 1997, 2000. **http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html**

[TMML] Robert Lyons. *Turing Machine Markup Language*. 2001. **http://www.unidex.com/turing/**

[UnQL] Peter Buneman, Mary Fernández, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal* 9 (2000), 76-110.

[Xcerpt] Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Proceedings of 29th Intl. Conference on Very Large Databases*, 2003.

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

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

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

[XPath 2.0] Anders Berglund, Scott Boag, Don Chamberlin, Mary Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0*. Technical report, W3C, 2003. **http://www.w3.org/TR/xpath20/**.

[XQuery 1.0] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. Technical report, W3C, 2003. Working draft, **http://www.w3.org/TR/xquery/**.

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

[XSLT 2.0] Michael Kay. *XSL Transformations (XSLT), Version 2.0*. Technical report, W3C, 2003. **http://www.w3.org/TR/xslt20/**.

The Author

Stephan Kepser
University of Tübingen, SFB 441
Nauklerstr. 35
72074
Tübingen
Germany
kepser@sfs.uni-tuebingen.de
http://tcl.sfs.uni-tuebingen.de/~kepser

Stephan Kepser is a research associate at the linguistics department at the University of Tübingen. His main interests are model theoretic and complexity theoretic properties of linguistic theories and the design and implementation of query languages for linguistic corpora and XML documents. He received his Ph.D. in computational linguistics from the University of Munich in 1998.

Extreme Markup Languages 2004
Montréal, Québec, August 2-6, 2004
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.