Compiling XSLT 2.0 into XQuery 1.0

Achille Fokoue, Kristoffer Rose, Jérôme Siméon, Lionel Villard
IBM T.J. Watson Research Center
P.O.Box 704, Yorktown Heights
NY 10598, USA
{achille,krisrose,simeon,villard}@us.ibm.com

ABSTRACT

As XQuery is gathering momentum as the standard query language for XML, there is a growing interest in using it as an integral part of the XML application development infrastructure. In that context, one question which is often raised is how well XQuery interoperates with other XML languages, and notably with XSLT. XQuery 1.0 [16] and XSLT 2.0 [7] share a lot in common: they share XPath 2.0 as a common sub-language and have the same expressiveness. However, they are based on fairly different programming paradigms. While XSLT has adopted a highly declarative template based approach, XQuery relies on a simpler, and more operational, functional approach.

In this paper, we present an approach to compile XSLT 2.0 into XQuery 1.0, and a working implementation of that approach. The compilation rules explain how XSLT's template-based approach can be implemented using the functional approach of XQuery and underpins the tight connection between the two languages. The resulting compiler can be used to migrate a XSLT code base to XQuery, or to enable the use of XQuery runtimes (e.g., as will soon be provided by most relational database management systems) for XSLT users. We also identify a number of areas where compatibility between the two languages could be improved. Finally, we show experiments on actual XSLT stylesheets, demonstrating the applicability of the approach in practice.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Languages, Standardization

Keywords

XSLT, XQuery, XML, Web services.

1. INTRODUCTION

As XQuery 1.0 [3] gets closer to recommendation, developers are starting to consider it as a viable alternative platform for XML application development. As a result, the question of how XQuery fits with the existing XML infrastructure becomes a crucial one. In particular, how to use XQuery together with existing XSLT-based applications is often a crucial question. In this paper we describe an approach to compile XSLT transformations into XQuery, and an implementation based on that approach. This provides a practical solution for using XQuery and XSLT jointly in a way that is both effective and efficient.

Despite of their similarities, understanding the precise relationship between XSLT and XQuery is not as easy as it seems. On the one hand, XSLT 2.0 [7] and XQuery 1.0 [3] share many characteristics. Both have XPath 2.0 [1] as a subset and are based on a common data model [5]. Both are functional languages without side-effects, and both are Turing-complete. On the other hand, XSLT and XQuery are based on fairly different designs. XSLT relies on a highly declarative template-based approach which gives the ability to easily extend existing programs or merge programs together. XQuery is based on a purely functional approach, which gives more direct control to the user but is somewhat more operational.

Since they have the same expressive power, one could argue that either XSLT or XQuery could be used for any given application. Another option would be to rely solely on the fact that XQuery and XSLT share a common data model. However, experience suggests a need for tighter coupling between those technologies. First of all, even if the languages target two different user communities, modern applications will increasingly require expertise from both. In addition, certain applications are more easily written with one language or the other. For instance, joins are very naturally expressed using XQuery's "FLWOR" expressions, while XML to HTML conversion is still often easier to write using XSLT's template-based approach. Finally, some popular systems will support only one of those two languages, but not the other. For instance, all popular database management systems [4] are planning to support XQuery, but not always XSLT, while some popular editors and libraries support XSLT but not XQuery. For all those reasons, there is a strong need to develop technology which can provide a tight coupling between the two languages.

The main contribution of this paper is an approach to compile XSLT 2.0 stylesheets into XQuery 1.0, which provides the foundations for a tight coupling between the languages. The compiler covers almost the complete XSLT 2.0 language, and we provide experiments with our current implementation that show that the approach is practical and effective. Because of space limitations, we concentrate on explaining the compilation rules for the core of the compiler, notably how to compile XSLT's template based approach to XQuery's functional approach. We also identify key problems in making the compiler complete, which often relate to specific semantic incompatibilities between the two languages. For most of those problems, practical solutions are proposed and have been implemented.

One of the strengths of the proposed approach is that the resulting compiler can be used for a variety of practical needs. It can be

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

used by XQuery developers who may want to migrate an existing code base to XQuery. It can be used by XSLT developers who may want to write applications in XSLT, while running those applications on top of an existing XQuery run-time, such as provided by relational database systems. It can also be used as a component in providing a common XQuery-XSLT infrastructure, which in turn can be used to enable the development of common optimizations, as well as the ability to call templates from XQuery expressions, or vice versa.

The key technical contributions in the paper are:

- We provide detailed compilation rules from the template-based approach of XSLT 2.0 into XQuery's functional approach. The rules are designed in order to provide the most natural compilation, so that the resulting program can easily be understood by an experienced XQuery programmer.
- Covering the complete XSLT language is difficult due to its size and complexity. We identify the fragments of the language that are the most challenging to compile into XQuery, and provide some corresponding solutions. In some cases, we identify concrete locations for which the alignment between XSLT 2.0 and XQuery 1.0 could be improved.
- This approach has been implemented in a running prototype. We describe the architecture of that prototype and provide experiments which demonstrate the feasibility of the approach. Our current prototype runs a very large fragment of a full set of XSLT conformance tests, and has been tested on a number of non-trivial stylesheets.

The paper is organized as follows. In section 2, we illustrate the compilation approach on a simple example. In Section 3, we give the translation rules for the heart of the compiler. In Section 4, we focus on the most complex detailed issues that must be addressed to support the complete language. We describe the implementation of our XSLT to XQuery compiler and present experiment results in Section 5. Finally we conclude and give some perspectives in Section 7.

2. APPROACH AND EXAMPLE

In this section, we illustrate our approach by describing the compilation of a simple XSLT stylesheet into XQuery, and use that example to explain some of the key technical challenges and how to address them.

2.1 The `recipe` example

Figure 1 shows a simple recipe stylesheet inspired by the Sarvega XSLT benchmark [11]. This stylesheet formats a single recipe XML document to HTML.

An XSLT stylesheet is composed of templates. Each template associates a pattern that matches against certain nodes to the evaluation of an expression. When the node currently being processed matches a given match pattern, its associated template is evaluated to create a fragment of the output document. For instance, the very first rule in Figure 1 matches a `recipe` element and creates an `html` element with a `body` within it. The content of the `body` element is then composed of: a `h1` header, which is obtained by applying the templates to the children `title` elements within the `recipe`, a list of ingredients, and the description for the preparation. The rest of the stylesheet contains the remaining templates for the other elements within a `recipe`.

The template based approach of XSLT is similar to a functional approach in the sense that templates operate without side effects

```
<xsl:stylesheet>
 <xsl:template match="recipe">
  <html>
   <body>
    <h1><xsl:apply-templates select="title"/></h1>
    <ul><xsl:apply-templates select="ingredient"/></ul>
    <xsl:apply-templates select="preparation"/>
   </body>
  </html>
 </xsl:template>
 <xsl:template match="ingredient">
  <xsl:param name="num" select="count(ingredient)"/>
  <li><xsl:value-of select="@name"/></li>
  <ul>
   <xsl:apply-templates
     select="ingredient[position() le $num]">
       <xsl:with-param name="num" select="$num - 1"/>
   </xsl:apply-templates></ul>
  <xsl:apply-templates select="preparation"/>
 </xsl:template>
 <xsl:template match="preparation">
  <ol><xsl:apply-templates select="step"/></ol>
 </xsl:template>
 <xsl:template match="step">
  <li><xsl:value-of select="text()|node()"/></li>
 </xsl:template>
</xsl:stylesheet>
```

Figure 1: Recipes stylesheet.

over their input parameters. However, there are also some important differences. Notably, templates are not called explicitly within the stylesheet, instead, the `xsl:apply-templates` expression is applying the whole set of templates on the selected nodes. The actual template being triggered is decided using a set of rules specified as part of the semantics of XSLT. In case of conflicts, XSLT provides a resolution mechanism based on template priority that always selects a unique template. On top of this built-in semantics, the user can partially control how the templates are triggered using the notion of *mode*. It can associate a given template to a mode, and calls the `xsl:apply-templates` expression with a particular mode.

2.2 Compilation approach

The close relationship between XSLT and XQuery makes some of the compilation easy. Notably XQuery and XSLT share XPath 2.0 as a subset.[1] In addition, XPath 2.0 expressions are used only in specific locations within a stylesheet, which facilitates their identification during compilation into XQuery. Note that the reverse translation would be more difficult because XQuery can arbitrarily compose XPath expressions with other kinds of expressions. On first approximation, compiling an XPath 2.0 expression to XQuery 1.0 is essentially applying the identity function. As we will see, this is not entirely true, since some care is needed to make sure the resulting XPath expression will operate over the proper input context. Nonetheless, the principle applies, which facilitates the translation, and makes the resulting XQuery easier to read and edit for a programmer.

Dealing with the rule-based execution model of XSLT is the main challenge that must be tackled when compiling stylesheets to XQuery. First, although `xsl:apply-templates` may resemble a function call, its semantics does not correspond to explicit function calls, but instead relies on a kind of dynamic dispatch based on pattern matching, template priority, import precedence, and modes. Second, the notions of pattern matching and implicit context item at each point of the evaluation of a stylesheet do not exist in XQuery. Third, template parameters, as opposed to XQuery

[1]Note that we do not consider here the compilation of XSLT 1.0, which would require the treatment of backward compatibility issues with XPath 1.0 [1].

function parameters , may be optional. In this section, we focus on how our compilation addresses these three issues by translating the `xsl:template` and `xsl:apply-templates` instructions in the example of Figure 1.

Fortunately, with the proper care, the template-based approach of XSLT can be implemented using XQuery user-defined functions. The main idea here is to create an explicit function for each template, and to replace each `xsl:apply-templates` instruction by an XQuery function call to the generated XQuery function performing the proper explicit dynamic dispatch.

For each kind of XSLT components, we apply the following compilation principles:

- XQuery variables are used to model the XSLT context.
- Relative XPath expressions that implicitly depend on the current context item, position or size are translated into equivalent absolute expressions (prefix by either a function call or a variable) that do not depend on the implicit context.
- XSLT match patterns are translated into an equivalent combination of standard XPath expressions with conditionals.
- XSLT templates definitions are compiled into XQuery user-defined functions.
- `xsl:apply-templates` are compiled into function calls to a generated XQuery function which consists of a combination of XQuery's conditional expressions to model XSLT's dynamic dispatch, and calls to the appropriate XQuery function for the corresponding templates.

2.3 Step by step translation

In the rest of the section, we illustrate each of those principles on concrete examples extracted from the `recipe` stylesheet. In what follows, we will use the namespace prefix `t2q` for variables and functions used by our compiler.

Context and relative path expressions

XSLT uses a notion of context to implicitly pass parameters between templates during the evaluation. XQuery also supports a notion of context. However, that context cannot be bound explicitly. In order to deal with that issue, and also avoid possible wrong interaction between the XSLT context and the XQuery context, we use explicit variables to model the XSLT context. Those variables are `$t2q:dot` for the context item, `$t2q:pos` for the context position, and `$t2q:last` for the context size.

Each relative path expression within the original stylesheet must be prefixed by the appropriate bindings to the context variables. For example, the relative path expression `description` is translated into `$t2q:dot/description`. How the input context is passed to the path expression must pay attention to the actual way that expression is constructed. For instance, the translation for the expression `count(ingredient/ingredient)` is the slightly more complex:

```
count($t2q:dot/ingredient/ingredient)
```

Here, the input parameter is passed on the path within the function call.

Match patterns

The notion of match pattern does not exist in XQuery. Therefore the compiler translates match patterns into equivalent XPath expressions by *reversing* the pattern. A node matches a pattern if it belongs to the list of nodes that this pattern can select.

For instance, the match pattern `recipe` is translated into the path expression `exist(self::recipe)`, which returns true iff the input node is an element `recipe`.

One subtlety is that to obtain the right semantics without negatively impacting performance, the patterns need to be reversed. For instance, a pattern: `recipe/title` has to be reversed into a path expression of the following form:

```
exist(self::title[parent::recipe])
```

The more complex XPath expression

```
people/person[@name="John Doe"]//phone
```

is reversed into

```
exist(self::phone[
      ancestor::person[@name="John Doe"]/
                                parent::people])
```

The detailed translation of patterns can be somewhat involved in some cases. Attribute patterns must not be translated into an attribute axis as it would not match an input node of type attribute, but return attributes of that input note. Using the self axis would not work either, since it would only select the current node if it is an element. Therefore, `@name` is translated into the more complex

```
exist((.)[.  instance of attribute("name")])
```

Similarly, the pattern `@*:name` is translated into the more explicit

```
exist((.)[(.  instance of attribute())
             and (local-name(.)  eq "name")])
```

Finally, special attention must be paid to the translation of patterns containing steps containing position predicates. For example,

```
people/person[2]/address
```

is matched by the address of the second person. The simple translation

```
self::address[parent::person[2]/parent::people]
```

would be wrong, as it would not match any elements (because `parent::person[2]` would not select any elements). A correct translation must recover the position by going up then down the tree, as follows:

```
exist(self::address[parent::person[
  parent::node()/person[2]=.]/parent::people])
```

Templates

Templates are translated into equivalent XQuery functions. The signature of these functions includes the context node, the context position, the context size and the list of parameters declared in the template. For example, the following template

```
<xsl:template match="collection/description">
   <xsl:value-of select="text()"/>
</xsl:template>
```

is translated to

```
declare function t2q:template1(
  $t2q:dot as node(),
  $t2q:pos as xs:integer,
  $t2q:last as xs:integer,
  $t2q:mode as xs:string)
```

```
{
 (text {string-join(
    for $t2q:d in data($t2q:dot/child::text())
    return ($t2q:d cast as xs:string),' ')})
} ;
```

Template application

Dealing with `xsl:apply-templates` is the most complex part of the translation. The evaluation of the `xsl:apply-templates` instructions consists of first evaluating the XPath selection associated to it and second looking for a template that matches the selected nodes. All templates with the same mode attached to the `xsl:apply-templates` instruction are considered. Whenever several templates match the same node, then the winner is the one with the highest priority. Basically, the translation consists of the following steps:

- Definition of the XQuery `applyTemplates` function which implement the processing described above, i.e., looking for the correct template function to call.
- The translation of each `xsl:apply-templates` instruction is an XQuery function call to a generic `applyTemplates` function, this for each node selected by the XPath selection associated with the `xsl:apply-templates` instruction.

The `applyTemplates` function can be broken down in two main pieces: template ordering and parameter binding. We first describe these two pieces and then we put them together.

Dealing with priority

The search for the template to instantiate depends on the template's priority and mode. Templates are ordered according to their import precedence and priority. The latter is either specified by the user through the `priority` attribute on the template or computed by analyzing the syntax of the template's pattern [7, §6.4]. Templates are picked up according first to their import precedence and then their priority, both statically known.

Dealing with parameters binding

An important mismatch between XQuery function calls and XSLT's `xsl:apply-templates` is that the latter can be called with implicit parameters. For example, the evaluation of the instruction

```
<xsl:apply-templates select="ingredient"/>
```

may pass the default value of parameter `"num"` implicitly to the evaluation of the `"ingredient"` template. Default function parameters do not exist in XQuery. Therefore, when invoking the generated XQuery `applyTemplates` function, parameters must be fully bound, either by using the value specified in connection with the `xsl:apply-templates` instruction (via `xsl:with-param`) or by using the special generated variable `$UNDEFINED` to indicate that the default value of the parameter should be used. For example

```
<xsl:apply-templates select="ingredient"/>
```

without an explicit parameter in the recipe template is translated to

```
let $t2q:sequence := $t2q:dot/child::ingredient
return
 let $t2q:last := count($t2q:sequence)
 return
  for $t2q:dot at $t2q:pos in $t2q:sequence
  return
   t2q:applyTemplates($t2q:dot, $t2q:pos,
                      $t2q:last,'#default',
                      $t2q:UNDEFINED)
```

whereas

```
<xsl:apply-templates
  select="ingredient[position() le $num]">
 <xsl:with-param name="num" select="$num - 1"/>
</xsl:apply-templates>
```

in the ingredient template is translated to

```
let $t2q:sequence
 := $t2q:dot/child::ingredient[position() le $num]
return
 let $t2q:last := count($t2q:sequence)
 return
  for $t2q:dot at $t2q:pos in $t2q:sequence
  return
   t2q:applyTemplates($t2q:dot, $t2q:pos,
                      $t2q:last,'#default',
                      $num - 1)
```

Dealing with `xsl:apply-templates`

In addition to context information, the signature of the generated XQuery `applyTemplates` function has as many parameters as there are template parameters with distinct names. The position of each of these additional parameters uniquely identify the template parameter name it represents. Thus the `applyTemplates` function has all information required to call the appropriate template function with all its parameters bound. The following generated XQuery fragment illustrates this:

```
declare function t2q:applyTemplates(
  $t2q:dot as node(),
  $t2q:pos as xs:integer,
  $t2q:last as xs:integer,
  $t2q:mode as xs:string,
  $t2q:param0)
{
   (: ... :)
   t2q:template3(
    $t2q:dot,
    $t2q:pos,
    $t2q:last,
    $t2q:mode,
      typeswitch($t2q:param0)
        case $t2q:a as comment() return (
               if (($t2q:a is $t2q:UNDEFINED))
         then count(($t2q:dot/child::ingredient))
         else $t2q:param0)
       default return $t2q:param0)
      (: ... :)
}
```

Notice the test needed to figure out whether the default parameter value should be used.

Finally we can outline the function `applyTemplates`, which is defined as follows (in pseudo-code):

```
declare function applyTemplates(
   $dot as node()?,
   $pos as xs:integer,
   $last as xs:integer,
   $mode as xs:string,
   $param1,
     ...,
   $paramN)
{
 if ($mode = mode template 1
     and fn:exists(select template 1))
 then template1
  ($dot, $pos, $last, $mode,
   typeswitch($param1)
```

```
   case $a as comment() return
    if ($a is $UNDEFINED)
    then default value param 1 template 1
    else $param1
   default return $t2q:param1,
    typeswitch($paramN)
    case $a as comment() return
     if ($a is $UNDEFINED)
     then default value param N template N
     else $paramN
    default return $paramN
  ) (: end of template1 function call :)
  ...
 else if ($mode = mode template N
          and exists(select template N))
 then templaten($dot, $pos, $last, $mode, ...)
 else
  builtInApplyTemplates($dot, $pos, $last, $mode,
                        $param1,..., $paramN)
}
```

where `buildInApplyTemplates` is a function that calls XSLT built-in templates. The `applyTemplates` function takes as parameters the current context node, the current context position, the current context size and a list of `N` parameters of type `item()*`, namely `$param1`,..., `$paramN` where `N` is the number of distinct names of parameters defined by templates of the style sheet. All parameters names of a template are thus mapped into positional names, from 1 to `N`.

3. FROM RULE-BASED EXECUTION TO XQUERY FUNCTIONS

At the heart of our compiler is the ability to translate the rule-based execution style of XSLT into the "pure" functional XQuery approach. In this section, we formally present three kinds of translation rules that achieve this goal, following the approach described in Section 2. First, templates are mapped into XQuery function definitions. The second kind of translation rules, called XQuery Applicator Function Generators (XAFG), generates XQuery functions that encode, in XQuery, all the implicit rules for template selection, execution and conflict resolution. Finally, another set of translation rules describes how XSLT applicators (`xsl:apply-templates`, `xsl:apply-imports` and `xsl:next-match`) are converted into XQuery by invoking XQuery functions generated by the XAFG translation rules.

Notations. In this section, we formally describe the compilation from XSLT to XQuery with a set of translation rules, in the style of the XPath 2.0 and XQuery 1.0 Formal Semantics. Each translation rule takes part of an XSLT 2.0 stylesheet as input, and produces part of an XQuery expression as output. We use the following notations for the translation rules:

$$[\text{XSLT stylesheet}]_{\text{Const}} == \text{XQuery}$$

where Const denotes the translation function name.

3.1 From template definitions to XQuery functions

Template definition

```
<xsl:template
 match? = pattern
 name ? = qname
 priority? = number
 mode? = tokens
 as? = sequence-type>
 <!-- Content:
  (xsl:param*, sequence-constructor) -->
</xsl:template>
```

The constructor `xsl:template` defines a transformation rule based either on a name (when the attribute `name` is specified) and/or on a source document (when the attribute `match` is specified).

Translation rules

Templates with `match` attribute can be statically and completely ordered according to their import precedence as defined in [7, §6.4] and their priority (either explicitly specified, or, if absent, computed by analysing the syntax of their match pattern as specified in [7, §6.4]). In the remainder of this paper, we assume that templates with match attribute have been sorted according to their import precedence and their priority. ($\texttt{template}_1$, ..., $\texttt{template}_n$) denotes the sorted list of templates with match attributes in the input stylesheet. The translation rule of the i^{th} template is as follows:

```
[<xsl:template match='pattern' priority='number'
mode='token1...tokent' as='type'>
  xsl:param1...xsl:paramn
  sequence-constructor
 </xsl:template>]Const
==
declare function t2q:templatei (
 $t2q:dot as node(),
 $t2q:pos as xs:integer,
 $t2q:last as xs:integer,
 $t2q:mode as xs:token,
 [xsl:param1]Const,...,[xsl:paramn]Const) as type
{
 [sequence-constructor]Const
}
```

The information required to instantiate a template must be passed as parameters to the generated XQuery function. The current focus is specified by the parameters `$dot`, for the current context node, `$pos`, for the current context position, and `$last`, for the current context size. The `$mode` parameter indicates the mode in which the template is being instantiated. In XSLT, modes allow the processing of a node many times. In XSLT 1.0, the mode in which a given template is instantiated is always statically known, but this does no longer hold in XSLT 2.0 where the following is valid:

```
<xsl:template match='slide' mode='#all'>
 <xsl:apply-templates mode='#current'/>
</xsl:template>
```

`#all` denotes all possible modes; `#current` denotes the current template mode. In general, it is no longer possible to statically reduce the list of templates that can be applied based upon the mode attribute. Thus, by default, all templates need to be considered and the current mode passed as argument of the generated XQuery functions corresponding to XSLT templates.

Finally, template parameters are translated by extracting from their definition their name and type as follows (note that tunnel parameters are not supported yet, see section 4):

```
[<xsl:param
  name = qname
  select? = expression
  as ? = sequence-type
  required? = "yes" | "no">
  <!-- Content: sequence-constructor -->
</xsl:param>]Const
   ==
 qname as sequence-type
```

If as attribute is not specified, `sequence-type` is replaced by `item()*`.

Templates that do not specify a match attribute are translated into XQuery functions in a similar manner, and their invocation, through `xsl:call-template`, simply corresponds to a XQuery function call to the appropriate generated function:

```
[<xsl:template name='qname'
mode='token1 ... tokenN' as='type'>
xsl:param1...xsl:paramn
sequence-constructor
</xsl:template>]Const
==
declare function [.]fctname($dot as node(),
 $t2q:pos as xs:integer,
 $t2q:last as xs:integer,
 $t2q:mode as xs:token,
  [xsl:param1]Const,...,[xsl:paramn]Const,
 $t2q:impPrec as xs:integer,
 $t2q:priority as xs:integer) as type
{
 [sequence-constructor]Const
}
```

where `'.'` represents the template constructor being translated and $[]_{\texttt{fctname}}$ is a function that generates unique XQuery function names for a given XSLT instruction.

$$[\textit{instruction}]_{\text{fctname}} = \text{concat}(\text{"}\textit{instruction}\text{"}, \text{" \_"}, \$\text{decl\_order})$$

Note that the two additional parameters of the generated function are used to explicitly passed the import precedence and priority of the current template through a `xsl:call-template`. Note also that as explained in Section 2.3, relative XPath expressions are translated into equivalent expressions where the context is explicit.

3.2 Capturing applicator logic in XQuery

Generating an XQuery expression that corresponds to the body of a given XSLT templates is just the first step toward translating XSLT stylesheets. Once the logic for selection and invocation with the right parameter values is in place, the final XQuery function must be created. This section defines a set of translation rules that generate, for each applicator (`xsl:apply-templates` , `xsl:apply-imports` and `xsl:next-match`), an XQuery function which explicitly captures the selection, conflict resolution and invocation logic.

Definition

The evaluation of an applicator instruction is performed in three steps. First a selector XPath selects a sequence of nodes; then, for each selected node, the template that is best matched by the node is selected; finally, the selected template or a built-in template, if no user-defined templates are matched by the selected node, is invoked. The applicators only differ by the set of considered templates in the second step.

For simplicity of the presentation, we only focus here on one applicator, `xsl:apply-templates`. In appendices B.3 and B.2, we briefly present minor adjustments to the translation rules to handle `xsl:apply-imports` and `xsl:next-match`.

The `xsl:apply-templates` content model is:

```
<xsl:apply-templates
 select? = expression
 mode? = token>
 <!-- Content:
 (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

Applicator translation rules

We give the translation rules to generate an XQuery function (referred to as an XQuery applicator function) that implements an applicator selection, along with the proper conflict resolution and invocation logic.

An XQuery applicator function takes as parameters the current context node, the current context position, the current context size, the current mode, and a list of p parameters of type `item()*` ($\texttt{param}_1, \ldots, \texttt{param}_p$) where p is the number of distinct names of XSLT template parameters defined by templates with `match` attribute in the input stylesheet. Basically, a parameter name of a template is mapped to a position from 1 to p by the $[]_{\texttt{ParamName2Pos}}$ function. The function $[]_{\texttt{Pos2ParamName}}$ is the inverse function of $[]_{\texttt{ParamName2Pos}}$.

The translation rule that generates an XQuery applicator function takes as input the sequence of templates with `match` attribute in the input stylesheet sorted according to their import precedence and priority. The body of an XQuery apply-templates function consists of a large nested if-then-else expression that selects the first template whose mode matches the `$t2q:mode` parameter and whose match pattern is matched by the node referenced by the `$t2q:dot` parameter:

```
[(template1, ..., templaten)]
==
declare function t2q:applyTemplates(
$t2q:dot as node()?,
$t2q:pos as xs:integer,
$t2q:last as xs:integer,
$t2q:mode as xs:string,
$t2q:param1 as item()*,
...,
$t2q:paramp as item()*)
as item()*
{
 if ($t2q:mode = [template1]mode
  and exists([template1]toSelect))
 then [template1]invoke
 ...
 else if ($t2q:mode=[templaten]mode
  and exists([templaten]toSelect)) )
 then [templaten]invoke
 else
  t2q:builtInApplyTemplates($t2q:dot,$t2q:pos,
   $t2q:last,$t2q:mode,
   $t2q:param1,...,$t2q:paramp)
};
```

where the rule $[]_{\texttt{mode}}$ generates a sequence of tokens corresponding to modes specified by a given template. In appendix B.1, we formally define `t2q:builtInApplyTemplates()` function, which encodes all built-in templates. This translation rule does not attempt to detect nodes matching two or more templates of the highest import precedence and priority. This could trivially be done by adding additional tests.

The $[]_{\texttt{toselect}}$ function is used in the previous rule to translate the notion of match pattern, which does not exist in XQuery. The semantics of a match pattern is specified in XSLT 2.0 [7, §5.5.3] by the specification of a translation rule from a match pattern into a valid XPath expression. However, the evaluation of the generated XPath expression is inefficient because it does not simply test the pattern on a given candidate node, but it first evaluates an absolute XPath expression containing at least one descendant axis, and then tests whether the candidate is in the resulting sequence. Our efficient translation based on reversing pattern is illustrated in section 2.3 and is formally described in appendix A.

To complete our translation, we formally define the $[]_{\texttt{invoke}}$

rule. It generates a function call to the XQuery function corresponding to a given XSLT template. Unlike XSLT that allows optional template parameters, in XQuery all function parameters are mandatory. Therefore, when invoking an XQuery function, all its parameters must be explicitly bound. To handle XSLT optional parameters, we must be able to 1) detect that an XSLT applicator has been called without explicitly specifying the value of an optional parameter, and 2) call the XQuery template function with the default value of the missing parameter. The former goal is reached by always binding unspecified parameters (i.e. parameters not present in the list of `xsl:with-param` of the considered XSLT applicator) of an XQuery applicator function call to the special global variable `$t2q:UNDEFINED`. The $[]_{\texttt{invoke}}$ translation rule, specified below, achieves the latter objective (the input template is assumed to be at the i^{th} position in the sorted list of templates) .

```
[<xsl:template name='qname'
mode='token1 ... tokent' as='type'>
<xsl:param name='name1' .../>
...
<xsl:param name='namek' .../>
sequence-constructor
</xsl:template>]invoke
(with k <= p)
==
t2q:templatei(
$t2q:dot,
$t2q:pos,
$t2q:last,
$t2q:mode,
typeswitch($t2q:param[name1]ParamName2Pos)
   case $t2q:a as comment()
    return
      if ($t2q:a is $t2q:UNDEFINED)
    then [xsl:param1]defaultValue
    else $t2q:param[name1]ParamName2Pos
   default
    return $t2q:param[name1]ParamName2Pos,
...,
typeswitch($t2q:param[namek]ParamName2Pos)
   case $t2q:a as comment()
    return
    if ($t2q:a is $t2q:UNDEFINED)
    then [xsl:paramk]defaultValue
    else $t2q:param[namek]ParamName2Pos
   default
    return $t2q:param[namek]ParamName2Pos)
```

where $[]_{\texttt{defaultValue}}$ rule generates the default value of a given `xsl:param` and `$t2q:UNDEFINED` is defined as:

```
declare variable $t2q:UNDEFINED as comment()
{comment {
undeclared variable used
for node identity test
}};

[<xsl:param name='name' select='expr'/>]defaultValue
==
[expr]Expr
```

3.3 Invoking XQuery applicator functions

After formalizing the translation rules that generate XQuery applicator functions, we are now ready to present rules that generate, for each instance of a XSLT applicator, a function call to the appropriate XQuery applicator function.

These rules are as follows :

```
[<xsl:apply-templates select='expr' mode='mode'>
xsl:with-param*
</xsl:apply-templates>]Const
==
let $t2q:sequence := [expr]Expr return
let $t2q:inner-last := count($t2q:sequence)
return
  for $t2q:inner-dot
  at $t2q:inner-pos in $t2q:sequence
    return
    t2q:applyTemplates(
      $t2q:inner-dot,$t2q:inner-pos,
      $t2q:inner-last,
      'mode',
      [1]ParamValue(xsl:with-param*),
       ...,
      [p]ParamValue(xsl:with-param*))
```

where $[]_{\texttt{ParamValue(xsl:with-param*)}}$ returns, for a given position (recall that a position uniquely identifies a parameter name), its value specified in `xsl:with-param` list if it exists; otherwise it returns the global variable `t2q:UNDEFINED`. Note that if the mode is `#current`, then the translation is the same except that `'mode'` is replaced by `$mode`. Formally, $[]_{\texttt{ParamValue(xsl:with-param*)}}$ is defined as follows:

```
[ <xsl:with-param name='name'/> ]name = "name"

[k]ParamValue(xsl:with-param*)
==
if ([ xsl:with-param1 ]name = [k]Pos2ParamName)
then [ xsl:with-param1 ]Const
else
...
if ([ xsl:with-paramn ]name = [k]Position2ParamName)
then [ xsl:with-paramn ]Const
else t2q:UNDEFINED
```

Appendices B.3 and B.2 show how these rules can be modified to translate template instantiations with `xsl:apply-import` and `xsl:next-match`.

4. INTEROPERABILITY AND ISSUES

Our translation from XSLT to XQuery highlights the differences between XSLT and XQuery. In this section we summarize our experience with some of these.

Tunnelling parameters. XSLT 2.0 allows template parameters declared with the special `tunnel='yes'` attribute to "pass through" to all templates that are called while the parameter binding is in effect *dynamically*, even through templates that do not declare the parameter, including templates that are called from other modules [7, §10.1.2].

Since the collection of tunneling parameters is known statically for a complete stylesheet (including all the imported modules), this can be implemented by adding as parameters the list of the current value of all potentially tunneled parameters to all the functions generated for all templates. Unfortunately the need for the complete stylesheet prevents separate compilation of imported stylesheets.

An alternative is to add a single "dynamic environment" parameter to each template which contains a representation (in XML) of the currently bound tunneled parameters. This, however, requires translating parameter access into XPath expressions that select a fragment of the dynamic environment object.

Dynamic sort specification. The `xsl:sort` instruction in XSLT 2.0 permits that several of the aspects of sorting are specified using attribute value templates (AVT) computed dynamically

at runtime. The XQuery 1.0 `order by` construction does not allow this, and does not allow dynamic composition of sort modes. This means that `xsl:sort` has to be compiled into code that tests the values of the AVT attributes and at runtime branches to a `for` expression with the appropriate `order by` sort specification. Only one attribute cannot be supported in this way: setting the collation for sorting to a computed value cannot be translated into XQuery.

White-space stripping. Through the `xsl:strip-space` and `xsl:preserve-space` declarations [7, §4.3], XSLT 2.0 allows declaring *per element name* whether white-space from the input document should be preserved or not. This functionality is not available to XQuery.

Serialization. XSLT 2.0 gives very detailed control over how the generated output is serialized through the `xsl:output` and `xsl:character-map` declarations. Most of this serialization control is not available in XQuery and thus these cannot be fully supported by our translation.

Finally, since the `xsl:disable-output-escaping` attribute from XSLT 1.0 is optional in XSLT 2.0 and has not been considered for the translation.

5. EXPERIMENTAL EVALUATION

The translation rules presented earlier have allowed the implementation of a XSLT 2.0 compiler into XQuery. The purpose of this section is to describe briefly our implementation and report on our first experiments with that implementation.

Implementation. Most of the translation rules have been implemented in Java. The main instructions that have not been implemented include `xsl:sort`, `xsl:for-each-group`, `xsl:key` and `xsl:number`. Moreover issues presented in the previous section haven't been implemented.

The compiler architecture relies on a three stage processing. During the first stage, the XSLT stylesheet is parsed using a standard SAX parser. The second stage consists of applying translation rules that can be treated on-the-fly, when receiving events. This is notably the case for the `xsl:value-of` or `xsl:if` instructions. The instructions that cannot be treated on the fly are those which require the full list of templates, for instance `xsl:apply-templates` or `xsl:next-match`. Those instructions are kept in memory and translated during the third stage.

The main advantage of such architecture is to keep the memory consumption very low during compilation time while being fast. Indeed experiments show a relatively low resources consumption even for relatively big stylesheets like docbook [14].

Experiments. We have performed two kinds of experiments: on conformance and on performance. The purpose of the conformance experiments is to evaluate the correctness of the compiler. The compiler has been run over the Xalan conformance testsuite [15] which contains 1686 XSLT 1.0 test cases (it is the largest component of the OASIS suite [10, 12]). The generated queries are then processed using Saxon 8.0 [6]. The compiler is able to compile and run properly 55% of the test cases. Most of test cases fail either because they include an unimplemented instruction (more than 25%) or because Saxon crashes or produces a wrong result (more than 10%). We expect that as more mature XQuery implementations emerge, the number of tests passed will increase accordingly.

The goal of the second experiment is to compare the performance of the evaluation of queries produced by the compiler against the original XSLT stylesheet. We have run several XSLT transformations from the Sarvega [11] benchmark and their XQuery equivalents using Saxon 8.0. One advantage of using Saxon is that it executes both programs using the same internal runtime. Therefore it allows a fair comparison in particular because optimizations will be applied on the same instruction set. However it is worth noticing that Saxon currently provides better optimizations for XSLT than for XQuery.

The figure 2 shows a summary of the experiments made on the recipe and MathML transformations. Each transformation has been applied over different input document sizes. The figure shows that the XSLT transformations and the XQuery queries execute in O(n), n being the size of the input document. This is a very promising result demonstrating that our compiler doesn't change the algorithm complexity. It is in part due to our efficient translation of match pattern: replacing our reverse pattern approach by the naive translation defined in [7, §5.5.3] results in a nonlinear behavior (the performance degradation is such that, in the MathML example, Saxon is unable to process even a 10K document). However, the figure 2 also shows that the queries perform worse than the original stylesheet by a constant factor. This loss of performance is mainly located in the execution of the instruction `xsl:apply-templates`. Indeed, Saxon provides an aggressive optimization for looking up the right template to instantiate (using hashtables) whereas the generated queries tests sequentially which template function to execute.

6. RELATED WORK

At the language level, XSLT 2.0 [7] and XQuery 1.0 [3] are defined in close collaboration by the W3C XSLT and XML Query working groups. At the infrastructure level, there has not been enough work on how to make both languages interoperate. To the best of our knowledge, SAXON [6] is the only public implementation that supports both XQuery and XSLT. Although it is likely that SAXON reuses as much infrastructure to support the two languages as possible, there is little information available that describes how this is achieved. In [9], Moerkotte describes an implementation of XSLT on top of a database management system. That approach relies on compiling XSLT into a database algebra. This makes the approach more difficult to apply in the various application contexts we have been considering, where looser coupling is useful. In addition, the way his approach covers the complete XSLT language, notably how the specific details of its semantics are handled, is not fully specified. In comparison, our approach is to rely on the common features between XQuery and XSLT in order to provide a lightweight, user-friendly, yet complete, implementation.

From a theoretical standpoint, several papers have studied the semantics of XSLT. In [13], Wadler proposes a denotational semantics for XSLT patterns. The approach we have described here is a more efficient variant of the same semantics, relying on the notion of pattern reversal. In [8, 2], Bex, Maneth and Neven propose a precise semantics for a fairly large fragment of the XSLT language, based on tree grammars. Part of that semantics can be used to implement the template-based approach we described. However, it is not complete, instead trying to identify a fragment of XSLT with good complexity properties.

7. CONCLUSION

In this paper, we have presented a general method for translating the highly declarative rule-based approach of XSLT into the purely functional XQuery approach, leading the way to closer integration between the two languages. Our initial experiments have shown the feasibility of this approach by confirming that the evaluations of

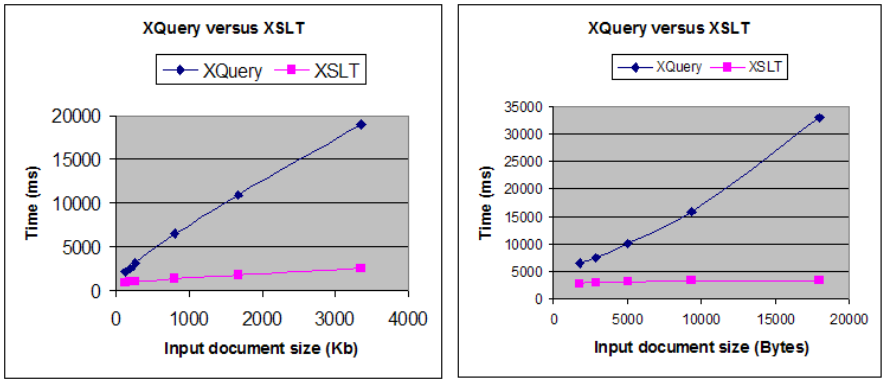

Figure 2: Performance comparison between XSLT and XQuery (left: recipes, right: mathml)

an XSLT transformation and its generated XQuery share the same algorithmic complexity.

However, a naive evaluation of the generated XQuery exhibits a performance penalty of up to factor 7 compared to the initial XSLT transform. Addressing this performance degradation constitutes an interesting challenge. We plan to investigate how a combination of context sensitive flow analysis and function specialization can be applied on the generated XQuery to statically reduce the list of considered templates in the if-then-else expression in the XQuery applicator functions.

This work has highlighted some important differences between the two languages. Most of them seem justified by the different design rationales, but we could not find reasonable grounds to account for the following differences, and, therefore, we suggest that the two working groups adopt a common solution:

- difference in white-space processing. Unlike XQuery, XSLT allows a finer control on white space processing,
- although the two languages share the same serialization specification, unlike XSLT, XQuery does not define a processor independent mechanism to specify serialization attributes.

8. REFERENCES

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0. W3c working draft, World Wide Web Consortium, July 2004. http://www.w3.org/TR/2004/WD-xpath20-20040723.

[2] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of xslt. In *Computational Logic 2000*, pages 1137–1151, London, UK, July 2000.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3c working draft, World Wide Web Consortium, July 2004. http://www.w3.org/TR/2004/WD-xquery-20040723.

[4] DB2. DB2 XML extender. http://www-306.ibm.com/software /data/db2/extenders/xmlext/.

[5] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model. W3c working draft, World Wide Web Consortium, July 2004. http://www.w3.org/TR/2004/ WD-xpath-datamodel-20040723.

[6] M. Kay. SAXON 8.0. SAXONICA.com. `http://www.saxonica.com/`.

[7] M. Kay. XSL transformations (XSLT) version 2.0. W3c working draft, World Wide Web Consortium, Nov. 2003. http://www.w3.org/TR/2003/WD-xslt20-20031112.

[8] S. Maneth and F. Neven. Structured document transformations based on XSL. In *Proceedings of International Workshop on Database Programming Languages*, pages 80–98, Kinloch Rannoch, Scotland, Sept. 1999.

[9] G. Moerkotte. Incorporating XSL processing into database engines. In *VLDB*, pages 107–118, Hong Kong, China, Sept. 2002.

[10] Oasis test suite for XSLT 1.0. http://www.oasis-open.org /committees /tc_home.php?wg_abbrev=xslt.

[11] Sarvega XSLT benchmark study and test suite. http://www.sarvega.com /xslt-benchmark.php.

[12] L. VanVleet, G. K. Holman, and D. Marston. Oasis XSLT/XPath conformance committee procedures and deliverables. http://www.w3.org/2001/01/qa-ws/pp /ken-holman-oasis /xsltconf.htm.

[13] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Languages*, Philadelphia, PA, June 2001.

[14] N. Walsh. The docbook document type. Committee specification, Oasis, July 2002.

[15] Xalan XSLT/XPath conformance test suite. http://xml.apache.org /xalan-j /downloads.html.

[16] XQuery 1.0: An XML query language. W3C Working Draft, Apr. 2002.

APPENDIX

A. MATCH PATTERN

$[]_{\text{toselect}}$ performs the translation of a pattern into the expression for which an existence test will be performed (by fn:exist()). The first step in the translation consists of obtaining the Equivalent Expression (EE) defined in XSLT 2.0 [7, §5.5.3]. The EE is an XPath expression whose first step may have attribute-or-top and child-or-top as axis. The translation rule of an EE is as follows (EPS denotes a step in the EE defined in [7, §5.5.3]) :

$$[\text{EPS}]_{\text{toselect}} = \$t2q{:}dot/(.)[[\text{EPS}]_{\text{match}}]$$

$$[\text{EPS}_0/.../\text{EPS}_n]_{\text{toselect}} = \$t2q{:}dot/(.)[[\text{EPS}_n]_{\text{match}}][[[\text{EPS}_n]_{\text{axis}}]_{\text{inv}}{::}\text{node}()[[\text{EPS}_{n-1}]_{\text{match}}]/.../[[\text{EPS}_1]_{\text{axis}}]_{\text{inv}}{::}\text{node}()[[\text{EPS}_0]_{\text{match}}]]$$

```
[axis::m[P]]axis=axis
[child]inv=parent
[descendant-or-self]inv=ancestor-or-self
[attribute]inv=parent
[self]inv=self
[axis::m[P]]match=([axis]inv::node()/axis::m[P]=.)
[child-or-top::m[P]]match =if (parent::node)
then (parent::node()/child::m[P]=.)
else self::m[P][not(. instance of attribute())]
[attribute-or-top::m[P]]match=if (parent::node)
then (parent::node()/attribute::m[P] = .)
else ((. instance of attribute()) and [m[P]]test)
[id(value)]match=(id(value) = .)
[root(self::node())]match=(root(self::node())=.)
[key(name,value)]match=([key(name, value)]Const=.)
[KinTest[P]]test=( . instance of KindTest)
and exist((.)[P])
[ncn:*[P]]test=(namespace-uri(.) eq ns)
and exist((.)[P]) - where ncn resolve to ns
[*:localName[P]]test=(local-name(.) eq localName)
and exist((.)[P])
[ncn:localName[P]]test=(namespace-uri(.) eq ns)
  and (local-name(.) eq localName)
  and exist((.)[P]) - where ncn resolve to ns
[*[P]]test=exist((.)[P])
```

When the pattern is a union of patterns, then the translation is the union of the translated patterns.

B. TEMPLATE INSTANTIATION

B.1 Built-in templates

The following XQuery function captures the semantics of selecting and invoking built-in templates.

```
declare function t2q:builtInApplyTemplates(
$t2q:dot as node()?,
$t2q:pos as xs:integer,
$t2q:last as xs:integer,
$t2q:mode as xs:string,
$t2q:param1 as item()*,...,
$t2q:paramp as item()*)
 as item()*
{
 if (exists($t2q:dot/self::text()
 |$t2q:dot/(.)[. instance of attribute()])
 then  string-join(
    for $t2q:d in data($t2q:dot)
    return ($t2q:d cast as xs:string),' ')
 else if (exists($t2q:dot/self::comment()
 |$t2q:dot/self::processing-instruction()))
 then ()
 else
  let $t2q:sequence := $t2q:dot/node() return
  let $t2q:inner-last := count($t2q:sequence)
  return
   for $t2q:inner-dot
    at $t2q:inner-pos in $t2q:sequence return
    t2q:applyTemplates($t2q:inner-dot,
    $t2q:inner-pos,
    $t2q:inner-last,$t2q:mode,
    $t2q:param1,...,$t2q:paramn)
};
```

B.2 xsl:next-match instruction

The translation rule generating the XQuery applicator function corresponding to xsl:next-match is defined as follows:

```
[(template1, ..., templaten)]
==
declare function t2q:applyNextMatch(
$t2q:dot as node()?,
$t2q:pos as xs:integer,
$t2q:last as xs:integer,
$t2q:mode as xs:string,
$t2q:param1 as item()*, ...,
$t2q:paramp as item()*,
$t2q:impPrec as xs:integer,
$t2q:priority as xs:double)
)
as item()*
{
 if ( [template1]priority < $t2q:priority
  and [template1]impPrec < $t2q:impPrec
  and $t2q:mode = [template1]mode
  and exists([template1]toSelect))
 then [template1]invoke
 ...
 else if ([templaten]priority < $t2q:priority
  and [templaten]impPrec < $t2q:impPrec
  and $t2q:mode=[templaten]mode
  and exists([templaten]toSelect)) )
 then [templaten]invoke
 else
  t2q:builtInApplyTemplates($t2q:dot,$t2q:pos,
   $t2q:last,$t2q:mode,
   $t2q:param1,...,$t2q:paramp)
};
```

where []priority generates the priority of a template and []impPrec its import precedence. The additional parameters (`$t2q:impPrec` and `$t2q:priority` indicate the import precedence and priority of the current template. They are used in body of the generated function to restrict the set of considered templates.

The translation rule for each xsl:next-match instruction is as follows:

```
[<xsl:next-match select='expr' mode='mode'>
xsl:with-param*
</xsl:next-match>]Const
==
let $t2q:sequence := [expr]Expr return
let $t2q:inner-last := count($t2q:sequence)
return
  for $t2q:inner-dot
  at $t2q:inner-pos in $t2q:sequence
    return
    t2q:applyNextMatch(
      $t2q:inner-dot,$t2q:inner-pos,
      $t2q:inner-last,
      'mode',
      [1]ParamValue(xsl:with-param*),...,
      [p]ParamValue(xsl:with-param*),
      [.]currentTemplateImpPrec,
      [.]currentTemplatePriority)
```

where [.currentTemplateImpPrec (resp. [.]currentTemplatePriority) generates the import precendence (resp. the priority) of the current template at the location of the invocation of xsl:next-match if it is statically known; otherwise (e.g. if xsl:next-match is invoked inside a named template), it generates the variable $t2q:impPrec (resp. $t2q:priority), which is always used to indicate the import precedence (resp. the priority) of the current template.

B.3 xsl:apply-imports instruction

The translation of `xsl:apply-imports` follows the same principles presented for `xsl:next-match`. In addition to parameters of `$t2q:applyTemplates`, the XQuery applicator function implementing `xsl:apply-imports` logic specifies a parameter that indicates the import path of the stylesheet module where the `xsl:apply-import` is invoked. This is used to restrict the list of considered templates.