Transformation-by-Example for XML[*]

Shriram Krishnamurthi, Kathryn E. Gray, and Paul T. Graunke

Department of Computer Science
Rice University
Houston, TX 77005-1892, USA

Abstract. XML is a language for describing markup languages for structured data. A growing number of applications that process XML documents are *transformers*, i.e., programs that convert documents between XML languages. Unfortunately, the current proposals for transformers are complex general-purpose languages, which will be unappealing as the XML user base broadens and thus decreases in technical sophistication. We have designed and implemented XT3D, a highly declarative XML specification language. It demands little more from users than a knowledge of the expected input and desired output. We illustrate the power of XT3D with several examples, including one reminiscent of polytypic programming that greatly simplifies the import of XML values into general-purpose languages.

1 XML and Transformations

XML [3] is a simplified version of the markup description language SGML. Because of XML's syntactic simplicity, it is easy to implement rudimentary XML processors and embed them in a variety of devices. As a result, a wide variety of applications are adopting XML as a data representation standard. Declarative programming languages must therefore provide support for XML. They can do better; as this paper demonstrates, ideas from declarative programming can strongly enhance the toolkit that supports XML.

Syntactically, XML has a structure similar to other SGML-style markup languages such as HTML. The difference between XML and a language like HTML is that XML really represents a *family* of languages. Concretely, XML provides two levels of specification:

- An XML *element* defines a tree-structured representation of terms. This representation is rich enough to express a wide variety of data. A sample element, which might represent information about music albums, is

```
<album title="everybody else is doing it, so why can't we?">
 <catalog><num>A043</num><fmt>CD</fmt></catalog>
 <catalog><num>BD34</num><fmt>LP</fmt></catalog>
</album>
```

[*] This work is partially supported by NSF grants CCR-9619756, CDA-9713032 and CCR-9708957, and a Texas ATP grant.

– An XML *language* is essentially a BNF grammar that lists the valid elements and states how they may nest within each other. A language thus circumscribes a subset of the universe of all possible elements. The language of music albums may, for instance, allow an *album* element to mention only the name and catalog entries of an album.

An input whose elements meet the basic syntactic requirements of XML (such as matching and properly nested element tags) is said to be *well-formed*. A well-formed element that meets the requirements of a language, or *document type definition*, is called *valid* (with respect to that definition).

This two-level syntax makes XML documents easy to process in two phases. The first phase converts an input character stream into a stream of elements, checking only for well-formedness. The second phase, which can proceed in a top-down fashion, checks each element in the stream for conformance with a language definition. In short, XML is relatively easy to parse.

XML is commonly associated with the Web. New XML languages can be used to provide more structure to information than Web markup languages like HTML offer. These benefits can also be harnessed in several other contexts, so XML is expected to see widespread use for defining syntaxes for communication standards, database entries, programming languages, and so forth. Already, user communities are defining XML languages for business data, mathematics, chemistry, etc. XML thus promises to be an important and ubiquitous vehicle for data storage and transmission.

By itself, an XML document does nothing. It represents uninterpreted data. Its value lies in processors that understand the document and use it to perform some action. The set of processors for a language is potentially unlimited, e.g.,

XML Language	Processor	Action
HTML	Web browser	renders document on screen
music albums	inventory lister	generates HTML listing
a programming language	pretty-printer	generates HTML listing
a programming language	interpreter	runs program

A surprising number of these actions involve transforming one XML language into another. Even rendering documents on screen involves transformations. The XSL standard [6] defines an XML language for "formatting objects" that provide low-level formatting control of a document's content. As the number of domains that use XML to represent their information increases, more of these actions will be XML transformations.

Recognizing the importance of transformations, the XML standards committee is defining XSLT, a language for describing transformations between XML languages. Unfortunately, XSLT is an ad hoc language with no complete, formal semantics. Worse, XSLT appears to be a fairly complex language, and seemingly simple transformations require the user to essentially write a traditional procedural program. As XML's audience grows to encompass users of decreasing technical sophistication, XSLT's complexity imposes prohibitive demands on users, and increases the likelihood of errors.

To address this problem, we have designed and implemented XT3D, a transformation system for XML. XT3D is itself an XML language, so users do not need to learn a new surface syntax. The principal advantage of XT3D over XSLT is that it provides an extremely simple, declarative language for describing transformations over XML elements. Specifically, an XT3D specification contains little more than outlines of the expected input and the desired output of the transformation. Thus we anticipate that even users with minimal technical skills can use XT3D.

We hope the ideas of this paper will broaden the discussion on XML transformation languages. In particular, this paper includes several examples of transformations that can be implemented conveniently in XT3D, including some reminiscent of polytypic programming. We believe these examples can serve as part of a benchmark suite for evaluating transformation languages.

The rest of this paper is organized as follows. Section 2 briefly describes XML's syntax and language descriptions. Section 3 explains XT3D through a series of examples, and section 4 describes an extension and some pragmatic considerations. Section 5 describes how XT3D can automate the embedding of XML data in a general-purpose language. Section 6 describes some details about our implementation and its current status. The last two sections discuss related work and offer concluding remarks.

2 Background

XML documents consist of *elements*. The outermost element in the sample presented in section 1 describes an album. An album has one *attribute*, a title. Its *content* is a sequence of catalog entries. Each catalog entry contains a number and a format element. All elements must be properly nested, with elements represented by matching opening and closing tags, e.g., <*album*> and </*album*>. *Empty* elements, which are elements with no contents (but possibly with attributes), use only one tag, which is closed with />, e.g., <*empty*/>.

XML users define *languages* via an XML SCHEMA [1], which is essentially a BNF description. A schema that validates the sample XML element of section 1 is shown in figure 1. The elements in a document can specify attributes in any order, whereas content must follow the order described by the schema. The *minOccur* attribute specifies the minimum length required of a sequence of the referred element.

As the example suggests, XML SCHEMA is itself an XML language. Schemas are being proposed as an alternative to traditional markup specifications, inherited from SGML, called DTDs. Unlike schemas, DTDs are not XML languages. Section 5 exploits the fact that schemas are XML documents.

3 Transformation by Example (by Example)

We call the style of transformations employed by XT3D "transformation by example" by analogy to the work of Kohlbecker and Wand [13]. An XT3D trans-

```
<schema>
 <elementType name="album">
  <sequence minOccur="0">
   <elementTypeRef name="catalog"/>
  </sequence>
  <attrDecl name="title"/>
 </elementType>

 <elementType name="num">
  <datatypeRef name="string"/>
 </elementType>

 <elementType name="catalog">
  <sequence>
   <elementTypeRef name="num"/>
   <elementTypeRef name="fmt"/>
  </sequence>
 </elementType>

 <elementType name="fmt">
  <datatypeRef name="string"/>
 </elementType>
</schema>
```

Fig. 1. Sample Schema

formation consists of pairs of patterns representing the expected source and the desired output. These patterns, which are parameterized over pattern variables, are in the source and destination XML languages, respectively.

The pattern-matcher works by comparing each element in the input tree against the collection of defined patterns. An element matches a pattern if the element has the same structure as the pattern, with pattern variables in the pattern matching an arbitrary element in the actual input. The pattern matcher binds pattern variables to the corresponding parts of the inputs. It then generates an output element, substituting pattern variables with the sub-terms bound in the input. This new element is then expanded. This process continues until the input cannot be transformed further.

Figure 2 presents a sample XT3D transformation that processes elements that conform to the schema of section 2. The element *xt3d-transformation* introduces a new set of transformations. The *xt3d-macro* element represents a single transformation in this set. The *xt3d-clause* element delimits the two boxed terms, which constitute an input- and an output-pattern pair. Pattern variables are introduced as *xt3d-bind* elements in the input pattern, and *xt3d-use* elements in the output. An element's attributes (as opposed to its content) are matched by the *xt3d-attr* elements of the *xt3d-attributes* element.[1] If an attribute is specified literally in the input pattern, the pattern matcher checks that the element contains that attribute, with the value bound in the pattern.

The empty element <*xt3d-···*/> denotes an ellipsis, which affects pattern-matching. It follows a ("head") pattern in a sequence and matches a source

[1] This is the one point where XT3D departs from a pure "by example" syntax. We are forced to invent notation because XML attribute values can only be strings. Given a string value for an attribute, XT3D cannot determine whether the user means it to be a pattern variable (to be bound for use in the output) or a literal (to be checked for presence in the input). We rejected both embedding a special-purpose language in attribute value strings, and deviating from XML syntax.

```
<xt3d-transformation>
 <xt3d-macro>
  <xt3d-clause>
   <xt3d-input>
    <album>
     <xt3d-attributes>
      <xt3d-attr name="title">
       <xt3d-bind name="name"/>
      </xt3d-attr>
     </xt3d-attributes>
     <catalog>
      <num><xt3d-bind name="num"/>
      </num>
      <fmt><xt3d-bind name="fmt"/>
      </fmt>
     </catalog><xt3d-.../>
    </album>
   </xt3d-input>
   <xt3d-output>
    <ul>
     <li><xt3d-use name="name"/>
     </li>
     <li>
      <b><xt3d-use name="num"/>
      </b>
      <i><xt3d-use name="fmt"/>
      </i>
     </li><xt3d-.../>
    </ul>
   </xt3d-output>
  </xt3d-clause>
 </xt3d-macro>
</xt3d-transformation>
```

Fig. 2. Sample XT3D Transformation

sequence of zero or more instances of the head pattern. It binds each pattern variable in the head pattern to a sequence. This sequence consists of the subterms, in order, of the terms in the source sequence that correspond to the pattern variable's position in the head pattern. Ellipses can be nested to arbitrary depth. Each nesting level introduces a nested sequence in the binding of a pattern variable.[2]

This transformation converts the album element of section 2 into

```
<ul>
 <li>everybody else is doing it, so why can't we?</li>
 <li><b>A043</b><i>CD</i></li>
 <li><b>BD34</b><i>LP</i></li>
</ul>
```

The following sub-pattern of the transformation's expected source uses ellipses:

```
<catalog>
 <num><xt3d-bind name="num"/></num>
 <fmt><xt3d-bind name="fmt"/></fmt>
</catalog><xt3d-.../>
```

[2] Hence the name XT3D, which stands for "XML Transformations with Three Dots".

In the sample album entry, this pattern matches against

```
<catalog><num>A043</num><fmt>CD</fmt></catalog>
<catalog><num>BD34</num><fmt>LP</fmt></catalog>
```

binding *form* to the sequence "CD" followed by "LP", and *number* to "A043" followed by "BD34". The output pattern pairs numbers with formats:

```
<li>
  <b><xt3d-use name="num"/>, </b>
  <i><xt3d-use name="fmt"/></i>
</li><xt3d-.../>
```

As a second example, consider a variant on the first that lists the numbers and formats separately. We only need change the output rule:

```
<xt3d-output>
  <ul>
    <li><xt3d-use name="name"/></li>
    <li><b><xt3d-use name="num"/></b> <xt3d-.../></li>
    <li><i><xt3d-use name="fmt"/></i> <xt3d-.../></li>
  </ul>
</xt3d-output>
```

In short, this transformation is a simple version of a table transposition. (By using nested ellipses, we can handle arbitrary numbers of rows and columns.)

These examples distill the essence of similar ones presented in the XSLT document [4] and the paper by Wallace and Runciman [18]. The XT3D specifications do not involve list processing combinators or conventional procedural programming. We therefore believe XT3D will be especially helpful for users who have little or no formal programming experience.

The third example illustrates a transformation that may be useful in software that generates rudimentary English phrases from databases. Given a database of purchase records, it uses the transformations of figure 3 to generate a purchase summary such that multiple purchases are separated by commas, except for the last two, which are separated by the word "and". Two examples of the input and generated terms are

```
<purchase>
  <p>4 tinkers</p>      ⟹      <text>4 tinkers</text>
</purchase>
```

and

```
                             <text>4 tinkers,
<purchase>                     <text>5 tailors,
  <p>4 tinkers</p>               <text>2 soldiers and
  <p>5 tailors</p>                 <text>1 spy</text>
  <p>2 soldiers</p>     ⟹        </text>
  <p>1 spy</p>                 </text>
</purchase>                  </text>
```

```
<xt3d-macro>
 <xt3d-clause>
  <xt3d-input>
   <purchase>
    <p><xt3d-bind name="i"/></p>
   </purchase>
  </xt3d-input>
  <xt3d-output>
   <text><xt3d-use name="i"/>
   </text>
  </xt3d-output>
 </xt3d-clause>

 <xt3d-clause>
  <xt3d-input>
   <purchase>
    <p><xt3d-bind name="i"/></p>
    <p><xt3d-bind name="i2"/></p>
   </purchase>
  </xt3d-input>
  <xt3d-output>
   <text><xt3d-use name="i"/> and
     <purchase>
       <p><xt3d-use name="i2"/>
       </p>
     </purchase></text>
  </xt3d-output>
 </xt3d-clause>
```

```
 <xt3d-clause>
  <xt3d-input>
   <purchase>
    <p><xt3d-bind name="i"/>
    </p>
    <xt3d-bind name="rst"/>
    <xt3d-···/>
   </purchase>
  </xt3d-input>
  <xt3d-output>
   <text><xt3d-use name="i"/>,
    <purchase>
     <xt3d-use name="rst"/>
     <xt3d-···/>
    </purchase></text>
  </xt3d-output>
 </xt3d-clause>
</xt3d-macro>
```

Fig. 3. Phrases from Databases in XT3D

Figure 4 presents what we believe is the equivalent transformation in XSLT. Though an XML language, XSLT encodes a special-purpose programming language in attribute strings. This language performs numerous actions such as boolean tests, mathematical operations, selections of attributes and content, and so on.[3] We believe this kind of encoding violates the spirit of XML, since it uses a flat representation for structured data (in this case, the special-purpose programs).

4 Beyond Macros

The alert reader will have noticed that our transformation language is essentially identical to the pattern-matching notation used for specifying Scheme

[3] Technically, the example is in XSLT's "abbreviated syntax", but the full syntax has the same flavor.

```
<xsl:template match="purchase">
  <xsl:choose>
    <xsl:when test="last()=1">
      <xsl:text>
        <xsl:apply-templates/>
      </xsl:text>
    </xsl:when>
    <xsl:when test="last()=2">
      <xsl:text>
        <xsl:apply-template select="p(1)"/>
        and
        <xsl:apply-template select="p(2)"/>
      </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template name="lrgr">
        <xsl:with-param name="len"
                        select="last()"/>
        <xsl:with-param name="curr"
                        select="1"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

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

```
<xsl:template name="lrgr">
  <xsl:param name="len"/>
  <xsl:param name="curr"/>
  <xsl:choose>
    <xsl:when
      test="curr &lt; (len -2)">
      <xsl:text>
        <xsl:apply-template
          select="p(curr)"/>,
        <xsl:call-template name="lrgr">
          <xsl:with-param
            name="len" select="len"/>
          <xsl:with-param
            name="curr"
            select="curr+1"/>
        </xsl:call-template>
      </xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>
        <xsl:apply-template
          select="p(curr)"/>
        and
        <xsl:apply-template
          select="p(last)"/>
      </xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Fig. 4. Phrases from Databases in XSLT

macros [11]. The last example above, for instance, is a straightforward extension to the traditional **and** macro used in Scheme implementations.

Macros work by repeated expansion. The macro processor scans the source term for the outermost use of a macro. It expands this term and recurs on the generated source. Expansion stops when there are no macro uses left. While this is convenient for simple specifications, it denies the user control of what to expand. It also presumes that the generated terms will be in the same language as the source, preventing transformations from one language to another.

Our implementation of XT3D is built atop McMicMac [15], a sophisticated macro system for Scheme. The McMicMac macro expander provides a complement to macros called *micros*. In a micro, all terms in the output pattern are left unexpanded unless the micro's author explicitly chooses to expand them. Thus programmers can construct terms in a destination language that is distinct from the source language, while recursively processing source sub-terms within the

```
<xt3d-micro>
 <xt3d-clause>
  <xt3d-input>
   <doc>
    <chapter>
     <p><xt3d-bind name="text"/>
     </p><xt3d-.../>
    </chapter>
   </doc>
  </xt3d-input>
  <xt3d-output>
   <fo:basic-page-sequence>
    <fo:simple-page-master/>
    <fo:queue>

    <xt3d-expand><xt3d-pattern>
     <chapter>
      <p><xt3d-use name="text"/>
      </p><xt3d-.../>
     </chapter>
    </xt3d-pattern></xt3d-expand>

    </fo:queue>
   </fo:basic-page-sequence>
  </xt3d-output>
 </xt3d-clause>
</xt3d-micro>
```

Fig. 5. Sample XT3D Micro

partially constructed output. In XT3D, the primitive *xt3d-expand* triggers recursive expansion, while *xt3d-pattern* expands its contained pattern in the pattern environment generated from the input pattern. Everything outside these terms in the output is assumed to be in the target language, and is therefore neither macro- nor pattern-expanded.

Figure 5 presents an XT3D micro. It converts a document language into XSL formatting objects. The salient portion is the boxed term in the output pattern. Everything outside the box is treated literally, and is thus immune to the expansion rules of the source language. (In principle, therefore, the macros in the preceding section should really have been micros. They work by accident, relying on the lack of overlap between the source and target languages.) In the boxed term, *xt3d-expand* further expands the body, which is still in the source language. The body can be an arbitrarily complex pattern that includes *xt3d-*$\cdots$.

For example, an input and its expansion after one step are

```
                          <fo:basic-page-sequence>
<doc>                      <fo:simple-page-master/>
 <chapter>                 <fo:queue>
  <p>Veni.</p>
  <p>Vidi.</p>      ⟹      <chapter>
  <p>Vici.</p>               <p>Veni.</p> <p>Vidi.</p> <p>Vici.</p>
 </chapter>                </chapter>
</doc>
                           </fo:queue>
                          </fo:basic-page-sequence>
```

The boxed term in the expansion represents the element that is about to be expanded again. The surrounding elements are in the target language, so they are not subject to the rules of expansion for the document description language.

5 Processing XML Data in General-Purpose Languages

In this section, we outline how we can import XML data into a general-purpose language (in our case, Scheme) for processing. We accomplish this using several hundred lines of XT3D transformations. As a case study, this illustrates:

1. how to automate typeful embeddings of XML SCHEMAs into programming languages,
2. how XT3D facilitates this embedding, and,
3. the XML support we have built for Scheme.

This library therefore accomplishes similar ends as Wallace and Runciman's system [18], but does so as a consequence of our primitives rather than as an end in itself.

The library consists of three families of XT3D transformers, each of which converts (restricted) XML SCHEMAs into Scheme programs:

1. The first phase converts XML SCHEMAs into corresponding Scheme structure definitions (which introduce new types) using MzScheme's **define-struct** facility [8].
2. The second phase generates a family of *builders*, one per element type. A builder is a Scheme procedure that consumes an XML element of an expected type, validates it, and produces an instance of the structure corresponding to that type (defined in the first phase).
3. The third phase generates *walker* generators. Walkers are procedures that consume instances of the structures defined in the first phase. The walker traverses each field of the structure using the walker for the type of that field. It then combines the results from these traversals using a procedure that the programmer supplies to the walker generator. This relieves the programmer of having to know the names of the fields or access them explicitly.

These transformations therefore automate the creation of validators, and enable a programmer to process XML data in a type-driven manner.

The preceding text discusses XT3D generating Scheme code, but Scheme is not an XML language, and XT3D can only generate XML terms. We have therefore defined an XML language, XScheme, to represent Scheme programs. Our XML library enriches our Scheme implementation with a reader that accepts XScheme programs in addition to those written in traditional Scheme syntax.

As an example of putting these transformations to work, consider this schema of geometric shapes (which we restrict to two element types for brevity):

```
<elementType name="rectangle">
  <sequence>
    <elementTypeRef name="point"/>
  </sequence>
  <attrDecl name="wd"
            required="true"/>
  <attrDecl name="ht"
            required="true"/>
</elementType>
```

```
<elementType name="point">
  <sequence/>
  <attrDecl name="x"
            required="true"/>
  <attrDecl name="y"
            required="true"/>
</elementType>
```

A programmer can render data in this format to the screen using the following Scheme code:

```
;; walk-rectangle : Rectangle ⟶ void
(define walk-rectangle
  (gen-walk-rectangle
    (lambda (attrs center-walker)
      (let∗ ((center (center-walker))
             (center-x (car center))
             (center-y (cdr center))
             (wd (attr→num 'wd attrs))
             (ht (attr→num 'ht attrs)))
        (lambda (canvas)
          (draw-rectangle canvas
            (− center-x (/ wd 2))
            (− center-y (/ ht 2))
            wd ht))))))
```

```
;; walk-point : Point ⟶ (cons Nat Nat)
(define walk-point
  (gen-walk-point
   (lambda (attrs)
     (cons (attr→num 'x attrs)
           (attr→num 'y attrs)))))
```

The procedure *gen-walk-rectangle* creates a thunk that, when invoked, applies *walk-point* to the point in the rectangle type. It provides this thunk as the value for the parameter named *center-walker*. The XT3D transformations generate the procedure *gen-walk-rectangle* and other supporting routines. The programmer needs to write only a few lines of scaffolding to read in the data and validate them using the generated builders. Due to paucity of space, we cannot present more details here.

This example is reminiscent of *polytypic* programming [2, 10, 16]. Polytypic programs, at least in the style of PolyP [10], typically consume a type constructor and return functions that manipulate values of that type. They can, for instance, generate maps, folds and other traversals that operate over a wide variety of types. Similarly, our third phase consumes a schema, which is essentially a type declaration for the values that are generated by the builders, and produces simple traversals over this class of values.

The preceding operations could have been performed on any (restricted) schema. We have also applied them to the schema for the XScheme language itself. This generates Scheme code which programmers can employ to manipulate

XScheme documents (i.e., programs) as Scheme values. Examples of such applications include interpreters, compilers and analysis engines. Languages that lack Scheme's syntactic simplicity can use such a library to simplify the transmission and processing of programs.

6 Implementation Details and Status

XT3D is part of the evolving XML library for the MzScheme [8] implementation of Scheme. We have tested all the examples in this paper using our library. The library exploits the similarity between XML elements and Scheme s-expressions. XML elements differ from s-expressions primarily in that they consist of two distinguished parts: attributes and content. We embed XML elements into s-expressions by requiring all target s-expressions to have the form

$$(\textit{tag}\ ((\textit{attribute value})\ \ldots)\ \textit{element}\ \ldots)$$

where *tag* is the name of the element. In our implementation, the attributes are sorted alphabetically to yield a canonical representation. We refer to such s-expressions as *x-expressions*.

Our library contains a reader that transforms XML source into x-expressions. It also includes two x-expression transformers: one that converts x-expressions representing XScheme into conventional Scheme syntax, and another that transforms x-expressions representing XT3D macro and micro definitions into McMicMac declarations.

The reader that generates x-expressions also maintains source location information for each term. It uses the source locations to generate special-purpose structures rather than conventional Scheme lists. McMicMac processes these enriched structures and uses them to perform source-correlation and source-tracking [14]. These are especially useful for determining the loci of errors.

Unlike some other toolkits, our library does not provide intrinsic support for any document type definition languages. Instead, we employ transformations, such as those described in section 5, to generate validators. This is a serendipitous consequence of the tools that comprise our library. Since MzScheme offers a rich target language for embedding, we expect to handle other document type definition standards such as SOX [5] in a similar manner.

Our library will continue to grow. For instance, the transformations that generate code from schemas (section 5) place some restrictions on the content of the schema. This is partially because our implementation is still in the prototype phase, but is also allied to our proposal (under preparation) for more extensible schemas. For the same reasons, transformations do not interact properly with XML namespaces. Also, our specifications become unwieldy for elements with several optional attributes.

7 Related Work

XT3D draws on a rich pedigree of Lisp and Scheme macro systems. The most influential of these is Kohlbecker and Wand's *macro-by-example* [13]. It also exploits the source correlation feature first described by Dybvig, et al. [7] and the micros of McMicMac [15]. Kohlbecker and Wand also provided a semantics for their transformation system, which can be used to formalize XT3D's transformers.

Several functional and declarative languages provide a pattern matching notation over values. By converting XML documents into structured values in these languages, programmers can exploit the built-in pattern matchers and constructor syntax to define transformers. This would, however, force the average user to learn the base language and contend with its peculiarities. In particular, small mistakes could trigger unexpected interactions with the base language. (One way to address this shortcoming is to use language levels [15].) In a declarative vein, we have recently come across a tool named PatML [9] which generates XML documents via pattern-matching, but have not been able to evaluate it.

The leading proposal for XML transformations is currently XSLT. It is difficult to compare XSLT and XT3D since XSLT is under constant revision, and is only partially formalized [17]. There are several subtle differences in the way XSLT and XT3D match and transform elements, with each having some strengths over the other. In the final analysis, we believe both styles serve useful ends. We would ideally like to see a synthesis of these styles so that simple tasks remain easy, while complex ones require more effort. We do find it unfortunate that XSLT uses strings to represent so much of its transformation language, since this inhibits the effective use of XML transformers to construct and process XSLT specifications themselves.

8 Summary and Future Work

We have designed and implemented XT3D, a transformation system for XML. The heart of XT3D is a simple declarative language for describing transformations that saves users from the burden of needing conventional programming experience. In particular, users need to know little more than the structure of the expected inputs and desired outputs. It should therefore be especially useful as XML is used by audiences with decreasing technical sophistication. Meanwhile, experts can exploit XT3D's advanced features to write sophisticated tools such as compilers and polytypic programs. This paper presents several such examples, which we believe belong in a benchmark for XML transformation tools.

XT3D must reflect the features and norms of XML. Since XML is constantly evolving, XT3D is very much a work in progress. Future work on XT3D can take several directions. First, it must support features like namespaces. Second, it can benefit greatly from primitives such as those provided by XSLT. XT3D also suggests shortcomings in and improvements to XML, such as information to create *hygienic* [12] transformers. We expect such work will expose ideas from functional and declarative languages to much broader audiences.

Acknowledgements

The authors thank Matthias Felleisen for suggestions, support, and critiques of drafts. We also thank the anonymous referees for their comments. The first author thanks Phil Wadler for useful exchanges.

References

1. Beech, D., S. Lawrence, M. Maloney, N. Mendelsohn and H. S. Thompson. XML Schema part 1: Structures. Technical report, World Wide Web Consortium, September 1999.
2. Bellé, G., C. B. Jay and E. Moggi. Functorial ML. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 32–46, 1996.
3. Bray, T., J. Paoli and C. Sperberg-McQueen. Extensible markup language XML. Technical report, World Wide Web Consortium, Feburary 1998. Version 1.0.
4. Clark, J. XSL transformations. Technical report, World Wide Web Consortium, October 1999. Version 1.0.
5. Davidson, A., M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney and K. Schwarzhof. Schema for object-oriented XML. Technical report, World Wide Web Consortium, July 1999. Version 2.0.
6. Deach, S. Extensible Stylesheet Language XSL specification. Technical report, World Wide Web Consortium, April 1999.
7. Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
8. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
9. International Business Machines. PatML. Web document: `http://www.alphaWorks.ibm.com/formula/patml/`.
10. Jansson, P. and J. Jeuring. PolyP — a polytypic programming language extension. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, 1997.
11. Kelsey, R., W. Clinger and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.
12. Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
13. Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–84, 1987.
14. Krishnamurthi, S., Y.-D. Erlich and M. Felleisen. Expressing structural properties as language constructs. In *European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 258–272, March 1999.
15. Krishnamurthi, S., M. Felleisen and B. F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-Based Software Engineering*, number 1799 in Lecture Notes in Computer Science, pages 105–120, September 1999.
16. Meertens, L. Calculate polytypically! In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, 1996.

17. Wadler, P. A formal semantics of patterns in XSLT. In *Markup Technologies*, December 1999.
18. Wallace, M. and C. Runciman. Haskell and XML: Generic document processing combinators vs. type-based translation. In *ACM SIGPLAN International Conference on Functional Programming*, September 1999.