

XML Transformations as Valuations of XML

Oleg Kiselyov, Northrop Grumman IT, oleg@okmij.org
Shriram Krishnamurthi, Brown University, sk@cs.brown.edu

May 31, 2002

Abstract

This paper presents an unconventional view of XML. We consider an XML document not as a data structure but as a term to evaluate. The evaluation of a term is tantamount to an XML transformation. The benefits of this perspective are two-fold. First, it helps us exploit the fund of knowledge about evaluation strategies, such as calling regimen. Second, we can directly reuse existing evaluators, instead of (effectively) interpreting the transformation language. Additionally, we can transform XML into non-XML languages.

The presentation in this document is driven by a series of illustrative examples. Some examples, which are not pretty, stress the ugly practical details of real-world XML use. Indeed, our tool suite, SXML, has seen use in several concrete commercial and government applications, ranging from authoring static Web pages to implementing complex Web services.

Keywords: XML, SXML, XSLT, evaluation, Scheme.

1 Introduction

The robustness of current and future programming standards and tools for XML will depend on the strength of their foundations in core programming technologies. So far, the official (W3C) XML transformation language XSLT [13] has not proven very satisfactory. For instance, the article [9] and the related discussion point out serious expressiveness and performance problems in XSLT, in addition to the terrible verbosity of the language. Adam Bosworth, vice president for engineering of the leading Web Services vendor BEA Systems Inc., has succinctly summarized the charge against XSLT:

I noticed two new categories of books: books about programming with Java and XML and books about XSLT. In both cases, whole forests are dying to compensate for the XML community's one great failure the lack of a decent programming model for manipulating XML.

...

We need a language that can natively support XML as a data type and yet can gracefully integrate with the world of objects (Java or otherwise) and can take advantage of the self-describing nature of XML by supporting querying of its own variables. This language as used by humans will look like a programming language, not an XML grammar. This is the language we will use to convert from one XML format to another.

...

I contend that until we have a language that natively understands the data structures inherent in XML and enables optimized algorithms for processing it, we will not have real XML programming. [1]

In contrast, we treat an XML document as a program to evaluate. This approach immediately raises questions about evaluation orders and environments, which we resolve in this paper. Our results will appear routine to programming language researchers, but this is intentional: even the basic lessons of functional programming have yet to cross the gulf to XML. As a case study [9] says,

The XSLT designers created a language that, while rich in domain-specific functionality, lacked much of the basic functionality necessary to make it genuinely suited for its intended purpose. While the designers probably left "generic" functionality out of the spec on the grounds that XSLT was never intended to be a

general-purpose programming language, they failed to realize that even simple document transformations often require a little nuts-and-bolts programming. Leaving out the nuts and bolts made XSLT a half-broken language.

Transformation-as-evaluation may appear baffling at first. Normally, we expect programs to produce predictable, usually identical results for identical inputs. In contrast, given an XML document, different contexts require distinct transformations. This apparent contradiction is, however, superficial. Even closed programs are not closed terms: they have implicit dependencies on, e.g., compile-time constants such as `MAXINT`, run-time constants such as the current working directory, command-line arguments and OS services. Similarly, we use environments to give meaning to the tags of a document. The environment effectively forms a stylesheet, and different stylesheets specify different XML transformations.

This paper shows that functional programming languages more than satisfy Bosworth's criteria. They are mature programming languages that natively support XML data structures and their efficient manipulation and transformation. Most of their implementations integrate well with Java, I/O and other OS facilities. Indeed, they can go further and treat XML data structures as program terms. In this document, we use Scheme for this purpose for three reasons:

- the availability of our tool frameworks, which have already been implemented in Scheme;
- the ready representation of XML terms as s-expressions;
- the ability to conveniently view a data structure as program text, which we exploit.

It is, however, not too difficult to port these ideas to other advanced programming languages.

The bulk of this paper consists of representative uses of our toolkit, SXML. We portray the use of SXML on three examples we have drawn from the literature, personal experience, and from discussions with experts. We elide most of the semantic details in the presentation because these should be quite evident to programming language researchers. Instead, we hope that these examples can be viewed as constituents of a microbenchmark that other transformation authors can use to evaluate their tools and compare against ours.

The design of SXML has evolved through extensive application. (In keeping with the best of traditions, we have used SXML to convert this very paper from XML to \LaTeX .) SXML has seen use in several commercial and government sector applications, including a large meteorology service. Most recently, Today's Yahoo! Pick, a popular site in the Yahoo! domain, switched to using SXML running atop PLT Scheme to generate their HTML from XML data. This suggests a vacuum in XML technology not being met by existing XML technology, and also an increasing willingness to experiment with ideas that are extremely traditional in the languages community, but rather more novel to the wider world.

2 Example 1: Context-Sensitive Term Generation

In this section we show the integration of XML data structures and Scheme through an unusual design criterion: SXML was designed so that its textual representation is a well-formed Scheme program. We can literally compile or interpret an abstract syntax tree of an XML document, given the bindings of tag names to appropriate procedures. This evaluation is tantamount to an SXML transformation. This in turn lets us put programming language evaluation to work directly, rather than indirectly through (implicit) interpreters, for XML transformation.

In practice, we often need to transform an XML document so that different occurrences of an element yield different results depending on that element's ancestors, siblings, or descendants. For instance, consider the following example, extended from one presented to demonstrate XT3D [7]: generating rudimentary English phrases from a database. Given a database of purchase records, the goal is to create 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 given on Fig. 1. The transformation involves the restructuring of the original markup and the insertion of item delimiters: commas and "and". We also add to the top-level `text` element an attribute `count` with the count of the descendant `text` elements.

The first step of an XML transformation is parsing a source document into an abstract syntax tree (AST). In this paper we use an abstract syntax based on Lisp's S-expressions [5]. We can use any XML parser to convert an

<pre> <purchase> <p>4 tinkers</p> </purchase> <purchase> <p>4 tinkers</p> <p>5 tailors</p> <p>2 soldiers</p> <p>1 spy</p> </purchase> </pre>	\Rightarrow	<pre> <text count='0'>4 tinkers</text> <text count='3'>4 tinkers, <text>5 tailors, <text>2 soldiers and <text>1 spy</text> </text> </text> </pre>
---	---------------	--

Figure 1: Sample XML purchase records and the corresponding generated phrases.

XML document into this format. The sample input and generated terms from Fig. 1 will have the following form in SXML:

<pre> (purchase (p "4 tinkers")) (purchase (p "4 tinkers") (p "5 tailors") (p "2 soldiers") (p "1 spy")) </pre>	\Rightarrow	<pre> (text (@ (count 0)) "4 tinkers") (text (@ (count 3)) "4 tinkers," (text "5 tailors," (text "2 soldiers and" (text "1 spy")))) </pre>
---	---------------	--

Figure 2: The sample SXML terms before and after the purchase-records-to-phrases translation.

Notice that the S-expressions in Figure 2 have the form of executable Scheme code, and can hence be evaluated. The expressions are not closed terms, however: they contain the free identifiers `purchase` and `p`. The expressions must therefore be evaluated in an environment that binds these identifiers to appropriate procedures.

Figure 3 presents a sample environment. With these bindings, evaluating the SXML code in the left column of Fig. 2 gives the result in the right column. The code on Fig. 3 is complete. It is instructive to compare transformations implemented by the procedures `p` and `purchase` with the equivalent XSLT code ([7], Fig. 4). The latter code (which does not generate the attribute `count`) occupies the better half of a page in a two-column layout and contains 31 elements, with numerous selections and mathematical operations.

SXML transformations convert one SXML document into another. We have already seen that we can rely on an XML parser to turn XML into SXML. Hence to complete our design of XML transformations, we need to convert an SXML S-expression into the corresponding XML document. Naturally, in SXML we treat conversion to XML also as a transformation task. At first blush this seems slightly tricky due to the presence of attributes, which have no distinguishing tag. In SXML, however, attributes have a distinguishing immediate parent, `@`. This makes it possible to exploit SXML for this task.

Figure 4 shows a simplified converter to XML syntax for the output of the database transformation. We associate attribute names with procedures that print attributes, and we bind an identifier `@` to a function that collects them in a uniquely-tagged list. The list is passed as an argument to a procedure responsible for formatting an XML element. This procedure looks up the attribute list by its characteristic tag and pulls the attributes into the start-tag markup.

Given the definitions in Fig. 4, `(eval (purchase (p "4 tinkers")))` transforms the SXML expression into a tree of string fragments. Writing out the latter in-order creates an XML document `<text count="0">4 tinkers</text>`. SXML intentionally builds a tree and performs a single in-order traversal to create the document to avoid the intermediate concatenation of strings, which can be quadratic in the size of the resulting document and proves to be a problem in practice for large documents.

3 Example 2: Context-Sensitive Character Substitutions

A domain-specific language should by definition offer convenient abstractions for domain-specific data and operations. The need for basic arithmetic, string processing and i/o is also usually fairly obvious. It is far less clear that the

```

(define (p str)
  (lambda (count . args) ; count can be #f: don't generate the attr
    (let ((att-list
          (if count '((@ (count ,count))) '())))
      (cond
        ((null? args) '(text ,@att-list ,str))
        ((null? (cdr args))
         '(text ,@att-list
               ,(string-append str " and")
               ,(car args) #f)))
        (else
         '(text ,@att-list ,(string-append str ",")
                       ,(apply (car args) (cons #f (cdr args))))))))))

(define (purchase . procs)
  (if (null? procs) '()
      (apply (car procs) (cons (length (cdr procs)) (cdr procs)))))

```

Figure 3: Bindings to XML "tags" p and purchase that effect the transformation on Fig. 2.

```

(define (entag tag)
  (lambda (elems)
    (cond
      ((match-tree '((@ . _attrs) . _children) elems '()) =>
       (match-bind (attrs children)
                   (list #\< tag attrs #\> children "</" tag #\>)))
      (else
       (list #\< tag #\> elems "</" tag #\>))))))

(define (enattr attr-key)
  (lambda (value)
    (list #\space attr-key "=" value #\")))

(define (@ . elems) (cons '@ elems))
(define text (entag "text"))
(define count (enattr "count"))

```

Figure 4: A binding environment for translating SXML expressions on Fig. 2 into XML.

language will need abstractions beyond those of its specialized domain.

XSLT offers good tools for XML transformations and acceptable string processing facilities. Leaving out seemingly luxurious features such as higher-order functions, however, has proven to cripple the language. Moertel [9] demonstrates the unsoundness of this decision by showing the need for higher-order functions even in fairly simple transformation problems. He describes the excruciating contortions needed to emulate higher-order functions in XSLT:

The really bad thing is that the designers of XSLT made the language free of side effects but then failed to include fundamental support for basic functional programming idioms. Without such support, many trivial tasks become hell. Our text-substitution problem is a good example. [9]

For illustration and comparison, we elaborate on the example of that paper [9]. The example is of clear practical relevance: converting XML documents into L^AT_EX for further typesetting and publishing. (Indeed, we use the full version of this example to produce this paper.) This conversion needs to preserve and appropriately translate the marked-up structure. We should also translate between character encodings: XML and L^AT_EX have different sets of "bad" characters and different rules for escaping them.

Moertel applied the encoding rules uniformly. However, this is an over-simplification. The example in the present section makes the encoding dependent on the context. More precisely, we want to transform `<tag>text</tag>` into a L^AT_EX environment `\begin{tag}text\end{tag}`, `<p>text</p>` into `text` followed by the empty line, and `
` into `\\`. The `text`, the content of an XML element, may contain characters such as `$`, `%`, and `{` that must be escaped in L^AT_EX, but these change in the body of a `verbatim` element. The reader might find it edifying to compare our solution with the XSLT solution given by Moertel [9].

As usual, we parse the XML document into an `s-expression` which we then evaluate. Figure 5 defines auxiliary functions, in particular, `make-char-quotator`. This is a higher-order function that makes a character-replacement function; the implementation shown here is a redaction of the production code.

Attempting to use the standard Scheme evaluator for this document, however, is not trivial. The problem arises because for the majority of tags (corresponding to L^AT_EX environments), our transformer does nothing special. In all these cases, we want to apply a standard transformation strategy. This translates into having a function for every single one of these tags, all performing the default action. Keeping track of these functions is inconvenient, yet the alternative would be to use a different embedding into Scheme, which would likely make other applications more complex.

To mediate between these two undesirable propositions, SXML offers a collection of built-in traversal strategies. The most common of these is pre-post-order (writing which is left as an exercise to the curious reader; the impatient reader can peek on the Web [11]), which we employ for L^AT_EX conversion. By default, pre-post-order conducts a post-order traversal, which corresponds to call-by-value evaluation of the tree. The traversal consumes an SXML document and a stylesheet (which corresponds to an environment) in which to perform the evaluation.

Figure 6 presents the use of pre-post-order. At each node name, the traversal queries the stylesheet for a handler corresponding to the visited tag; the corresponding handler extends the current stylesheet to evaluate children. The administrative node names `*text*` and `*default*` indicate the operation to perform on regular text and otherwise uninterpreted tags, respectively. Note the use of Scheme's quasiquote mechanism, which means the handlers for the tags are regular Scheme procedural values. This saves us the burden of writing an entire interpreter just to obtain this special behavior.

The SXML->LaTeX translation is performed by a function `generate-TEX` (Fig. 6). The function takes an SXML expression and passes it to the pre-post-order evaluator, along with the transformation stylesheet. We must stress the extensive use of higher-order functions in the stylesheet. Higher-order functions are vital to the implementation of the evaluator, as well as to the implementation of the domain-specific SXML->LaTeX code (the stylesheet and the character-encoding rules).

4 Example 3: Recursively Reorganizing Punctuation

Our third example was suggested by David Durand, a member of Brown University's Scholarly Technology Group (STG). The STG has done extensive work on real-world SGML and XML documents with rich markup; the problem

```

(define (make-char-quotator char-encoding)
  (let ((bad-chars (map car char-encoding)))

    ; Check to see if str contains one of the characters in charset,
    ; from the position i onward. If so, return that character's index.
    ; otherwise, return #f
    (define (index-cset str i charset)
      (let loop ((i i))
        (and (< i (string-length str))
              (if (memv (string-ref str i) charset) i
                  (loop (+ 1 i))))))

    ; The body of the function
    (lambda (str)
      (let ((bad-pos (index-cset str 0 bad-chars)))
        (if (not bad-pos) str ; str had all good chars
            (let loop ((from 0) (to bad-pos))
              (cond
                ((>= from (string-length str)) '())
                ((not to)
                 (cons (substring str from (string-length str)) '()))
                (else
                 (let ((quoted-char
                       (cdr (assv (string-ref str to) char-encoding)))
                       (new-to
                        (index-cset str (+ 1 to) bad-chars)))
                   (if (< from to)
                       (cons
                        (substring str from to)
                        (cons quoted-char (loop (+ 1 to) new-to)))
                       (cons quoted-char (loop (+ 1 to) new-to))))))))))

    ))

; Quotator for bad characters in the document's body
(define string->goodTeX
  (make-char-quotator
   '(("#\# . "\\#" (#\$ . "\\$") (#% . "\\%") (#& . "\\&")
      (#~ . "\\textasciitilde{~}") (#\_ . "\\_" (#\\~ . "\\~")
      (#\\ . "\\backslash$") (#{ . "\\{" (#} . "\\}")))))

; Place the 'body' within the LaTeX environment named 'tex-env-name'
(define (in-tex-env tex-env-name options body)
  (list "\\begin{" tex-env-name "}" options nl
        body
        "\\end{" tex-env-name "}" nl))

```

Figure 5: Auxiliary functions for the XML->LaTeX converter. `nl` is a string of the newline character. The `make-char-quotator` procedure takes a list of (`char . string`) pairs and returns a quotation procedure. The latter checks to see if its argument string contains any instance of a character that needs to be encoded. If the argument string is "clean", it is returned unchanged. Otherwise, the quotation procedure will return a list of string fragments. The input string will be broken at the places where the special characters occur. The special character will be replaced by the corresponding encoding string.

```

(define (generate-TEX Content)

  (pre-post-order Content
    ; The stylesheet, the initial environment.
    '(
      (*default* . ,(lambda (tag . elems)
                      (in-tex-env tag '() elems)))

      (*text* . ,(lambda (trigger str)
                   (if (string? str) (string->goodTeX str) str)))

      (p
        . ,(lambda (tag . elems)
              (list elems nl nl)))

      (br
        . ,(lambda (tag)
              (list "\\ \\ ")))

      (verbatim ; set off pieces of code: one or several lines
        ((*text* . ; Different quotation rules apply within a verbatim block
          ,(let ((string->goodTeX-in-verbatim
                 (make-char-quotator
                   '( (#\~ . "\\textasciitilde{ }"
                      (#\{ . "\\{ "
                      (#\} . "\\} "
                      (#\ . "{\\begin{math}\\backslash\\end{math}}")))))
                (lambda (trigger str)
                  (if (string? str) (string->goodTeX-in-verbatim str) str)))
          ))
        . ,(lambda (tag . lines)
              (in-tex-env "alltt" '()
                (map (lambda (line) (list " " " line nl))
                  lines))))
    ))
)

```

Figure 6: The SXML->TeX converter.

here is abstracted from one of their applications. Durand's example requires a transformer that moves punctuation inside a tag:

```
Click <a href='url'>here</a>! ==> Click <a href='url'>here!</a>
```

Even in this simple formulation, an XSLT solution is extremely unwieldy [2].

This transformation, to be truly useful, should account for several important particular cases. For example, we should not move the punctuation inside an XML element in the following context:

```
<br></br>;scheme comment
```

Furthermore, we may need to exempt an element from receiving adjacent punctuation. For instance, in

```
<p>For more details, see the paper <cite>Krishnamurthi2001</cite>.</p>
```

The content of the `cite` element is typically a bibliographic key, so it is not appropriate to add any punctuation to it. Finally, we do want to move punctuation recursively: given

```
<p>This <strong><em>needs punctuating</em></strong>!</p>
```

we would like to see the exclamation mark inside the innermost appropriate element (`` in our case).

Suppose we begin with the following sample document:

```
<div><p>some text <strong><em>needs punctuating</em></strong>!</p><br></br>.
This is a <cite>citation</cite>. Move <a href='url'>period around</a>.text</div>
```

We assume that `cite` elements are barriers to inward punctuation movement. We define the punctuation character set as:

```
(define punctuation '(#\ . #\, #\? #\! #\; #\:))
```

The sample document, once parsed (using the SSAX parser [4]), reads

```
(*TOP* (div (p "some text " (strong (em "needs punctuating")) "!")
  (br)
  ".\nThis is a "
  (cite "citation")
  ". Move "
  (a (@ (href "url")) "period around")
  ".text"))
```

At this point, we are ready to evaluate the document to perform the transformation. The transformation works in two alternating phases. The first phase is a classification, which identifies the punctuation and the nodes that can receive punctuation. The identified nodes are wrapped in annotating elements. That is, if an SXML node is a string and that string starts with a punctuation character, we break the string and return the punctuation and the remainder wrapped into an identifying SXML element: (`*move-me* punctuation-char-string orig-string-with-punctuation-removed`). Other string SXML nodes are returned unannotated. If an SXML node is an element node, we check whether the node has no children or has been blacklisted from receiving punctuation. If so, we return the node unchanged. Otherwise, we annotate it by wrapping it in an SXML element (`*can-accept* original-node`).

The second phase, which performs the actual transformation, examines the result of the classification. When we see a (`*can-accept* original-node`) immediately followed by a (`*move-me* punctuation-char-string string`) node, we move the punctuation inside the `original-node`. We then apply the algorithm recursively to the children.

While the description of the algorithm is fairly simple, it is difficult to implement using the standard Scheme evaluator or the post-order traversal shown in section 3. This is because nodes must examine their neighbors before

```

; The identity function for use in a SXSLT stylesheet
(define sxslt-id (lambda x x))

(define classify-ss
  '((*text*
    . ,(lambda (trigger str)
      (cond
        ((equal? str "") str)
        ((memv (string-ref str 0) punctuation)
         (list '*move-me*
               (string (string-ref str 0))
               (substring str 1 (string-length str))))
        (else str))))

; the following nodes never accept the punctuation
(@ *preorder* . ,sxslt-id) ; an attribute node
(cite *preorder* . ,sxslt-id)

(*default*
 *preorder*
 . ,(lambda (tag . elems)
      (cond
        ((null? elems) (cons tag elems)) ; no children, won't accept
        ((match-tree '(@ . _) elems '()) ; no children but attributes
         (cons tag elems)) ; ... won't accept
        (else
         (list '*can-accept*
               (cons tag elems))))))
)
))

```

Figure 7: A stylesheet for the pre-post-order function to convert an SXML tree into an annotated tree. The latter identifies strings of punctuation characters and nodes which can accept the punctuation.

```

(define transform-ss
  '(
    (*text* . ,(lambda (trigger str) str))

    (@ *preorder* . ,sxmlt-id) ; Don't mess with the attributes

    (*move-me* . ,(lambda (trigger str1 str2)
      (string-append str1 str2))) ; remove the wrapper
    *preorder*

    (*can-accept* . ,(lambda (trigger node)
      (pre-post-order node transform-ss))) ; remove the wrapper and transform
    *preorder* ; the node (recursively)

    (*default* . ,(lambda (tag . elems)
      *preorder*
      (cons tag
        (pre-post-order
          (let loop ((classified (pre-post-order elems classify-ss)))
            (cond
              ((null? classified) classified)
              ; check to see if a *can-accept* node is followed
              ; by a *move-me* node
              ((match-tree '(*can-accept* _node)
                (*move-me* _punctuation _str)
                . _rest)
               classified
               '())
              =>
              (match-bind (node punctuation str rest)
                (cons (append node (list punctuation))
                  (cons str (loop rest))))))
            (else
              (cons (car classified) (loop (cdr classified))))
            ))
          transform-ss)
      ))))

```

Figure 8: The transformation stylesheet.

they examine their children, and must modify the traversal of their children based on what they find adjacent. Fig. 7 presents the classification stylesheet and Fig. 8 that for transformation. Both stylesheets instruct the pre-post-order traversal abstraction to navigate pre-order, resulting in a evaluation strategy akin to call-by-name. This is the most appropriate strategy since the pull-in algorithm appears to require a hybrid traversal strategy based on breadth-first search.

The transformed SXML tree now looks as follows:

```
(div (p "some text "
      (strong (em "needs punctuating" "!") "")
      "")
      (br)
      ".\nThis is a "
      (cite "citation")
      ". Move "
      (a (@ (href "url")) "period around" ".")
      "text")
```

which, converted to XML, reads as

```
<div><p>some text <strong><em>needs punctuating!</em></strong></p><br/>.
This is a <cite>citation</cite>. Move <a href="url">period around.</a>text</div>
```

5 Related work

The most venerable related work on applying functional programming to markup is probably DSSSL, a Document Style Semantics and Specification Language. DSSSL did not, however, survive the switch from SGML to XML. One of the major differences between SXML transformations and DSSSL (besides the more flexible traversal strategies of SXML) is that DSSSL treats the SGML tree as an abstract data type rather than a program, effectively requiring an interpretation step.

There are many other ways to author, convert, or query XML documents, relying either on template expansions, or embeddings of XML/HTML into or around a host language: Microsoft's Active Server Pages (ASP), Java Server Pages (JSP), Hypertext Preprocessor (PHP), etc. They all, however, introduce a mini-language with a peculiar, non-XML-like syntax and ad-hoc substitution semantics. At the surface, XML transformation language XSLT is itself an instance of XML. However, queries, selections, string and arithmetic operations are expressed in XPath, in a syntax markedly different from that of XML. This causes a linguistic gap, which makes XSLT scripts more difficult to manipulate computationally.

Representing an XML/HTML document as code that upon execution will generate the document is also a rather popular approach. CGI.pm in Perl, LAML in Scheme [10], htmllib in Tcl, or Element Construction Set in Java are a few examples. Of these, LAML is the closest in spirit to SXML, and was one of the first to notice a similarity between a marked-up document and the structure of an expression to construct the document. LAML does not, however, treat markup documents themselves as programs, nor does it consider traversal strategies other than that resulting from regular Scheme evaluation.

XT3D, introduced in [7], is a declarative XML transformation language based on transformation-by-example. This language is intentionally very different from XSLT: XT3D is far more intuitive and designed for inexperienced users. In contrast, the present paper is oriented towards power users, who need all the expressive and transformation power of W3C standards and even more. XT3D gains its insight from Scheme's high-level macros; we, on the other hand, draw inspiration from Scheme's eval.

One common way to specify graph traversals and transformations is by combining node accessors by combinators. Wallace and Runciman [12] have developed a library of high-level combinators that serves as an extensible domain-specific language for transforming a subset of XML. SXML supports all of XML, including processing instructions and XML Namespaces. In addition, compared to the evaluation approach, the combinator approach is less intuitive for specifying generic transformations with exceptions (i.e., where the set of excepted nodes depends on the traversal's context).

XDuce [3] employs a powerful notion of regular expression types to capture XML values. It employs a powerful notion of subtyping, as well as union types, to support the rich set of patterns that programmers might write to match against evolving and semi-structured data. Its pattern-matching machinery is also strictly more powerful than that of languages like ML, while providing the benefits of static typing. In contrast to SXML, however, XDuce effectively forces programmers to write transformations as interpreters. Furthermore, the current XDuce language appears to be rather limited, without support for higher-order functions and hence traversal strategies, which are critical in many realistic transformation contexts.

It is possible to specify traversals more concisely than we do. The Demeter project revolves around a notion of "traversal strategies" [8], whereby a programmer writes a regular expression-based path through an object graph, and Demeter generates the appropriate traversal code. While this is extremely useful for processing XML trees (and even graphs), Demeter's programming language support remains weak. It currently runs atop C++ and Java, which have limited means to cleanly express higher-order behavior. Worse, Demeter currently computes entirely through side-effects, making it a poor fit for many XML transformation tasks, which have a highly functional flavor.

6 Conclusion and Future Work

This paper has proposed and illustrated a programming model for manipulating XML. The model is based on evaluation of an XML document as if it were a program. This program has free variables (the "tags"), whose meaning is supplied by an environment: a stylesheet. Different stylesheets define different valuation functions, that is, different XML transformations. The evaluation-based approach helps us exploit higher-order traversal strategies that reflect standard call-by-value and call-by-name evaluation orders. These are invaluable in some applications, as we show by example.

We identify several directions for future work. First, we would like to improve the expressiveness of the system. The current traversals do not automatically pass the context (as fold does). While we have not needed this for our applications so far, their value is becoming increasingly clear. We therefore hope to put more emphasis on building stronger traversal strategies. Second, we have not discussed transformations that more closely resemble database queries followed by rewriting; for such transformations, it remains open to combine our approach with a database query optimizer. Finally, when we use Scheme's EVAL instead of an explicit traversal strategy, we naturally limit the ability to reason about types inference. However, many aspects of SXML can be expressed in a powerful macro system such as McMicMac [6], which commutes with program-processing tools such as type inference engines; this is one useful path for recovering type information.

Acknowledgment

This work has been supported in part by the National Science Foundation, the National Research Council Research Associateship Program, Naval Postgraduate School, and the Fleet Numerical Meteorology and Oceanography Center.

References

- [1] Adam Bosworth. Programming Paradox. XML Magazine, February 2002. http://www.fawcette.com/xmlmag/2002_02/magazine/departments/endtag/
- [2] David Durand, private communication. May 22, 2002.
- [3] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67-80. January 2001.
- [4] Oleg Kiselyov. A better XML parser through functional programming. LNCS 2257, pp. 209-224. Springer-Verlag, January 2002.
- [5] Oleg Kiselyov. SXML Specification. Revision 2.1. March 1, 2002. <http://pobox.com/~oleg/ftp/Scheme/SXML.html>
- [6] Shriram Krishnamurthi, Matthias Felleisen and Bruce F. Duba. From Macros to Reusable Generative Programming. LNCS 1799, pp. 105-120. Springer-Verlag, September 1999.

- [7] Shriram Krishnamurthi, Gray, K.E. and Graunke, P.T. Transformation-by-Example for XML. Practical Aspects of Declarative Languages, 2000.
- [8] Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA. July 1997.
- [9] T. Moertel, XSLT, Perl, Haskell, & a word on language design. Posted on kuro5hin.org on January 15, 2002. <http://www.kuro5hin.org/story/2002/1/15/1562/95011>
- [10] Kurt Normark. Programming World Wide Web Pages in Scheme. ACM SIGPLAN Notices, vol. 34, No. 12 - December 1999, pp. 37-46. <http://www.cs.auc.dk/~normark/1aml/>
- [11] S-exp-based XML parsing/query/conversion. <http://ssax.sourceforge.net/>
- [12] Malcolm Wallace, and P. Runciman. Haskell and XML: generic combinators or type-based translation? Proc. the fourth ACM SIGPLAN international conference on Functional programming, 1999, pp. 148 -159.
- [13] World Wide Web Consortium. XSL Transformations (XSLT). Version 1.0. W3C Recommendation November 16, 1999. <http://www.w3.org/TR/xslt>
- [14] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation. October 6, 2000. <http://www.w3.org/TR/REC-xml>