SXSLT: Manipulation Language for XML

Oleg Kiselyov[1] and Shriram Krishnamurthi[2]

[1] FNMOC `oleg@okmij.org`
[2] Brown University `sk@cs.brown.edu`

Abstract. The growing use of XML languages has spurred the demand for means to transform XML documents. Many XML transformation systems are, however, limited in their expressive power, and fail to provide a useful collection of combinators for several tasks.

This paper describes SXSLT, a practical, higher-order, concise, expressive and readable declarative XML transformation language. The language is a head-first rewriting system over abstract XML syntax trees, implemented as a library extension of Scheme. SXSLT features local scoping of re-writing "templates", first-class stylesheets, flexible traversal strategies, the ability to re-traverse the original or the transformed trees. The language can emulate XSLT in whole or in part, in a more coherent and expressive manner. By virtue of being a Scheme library, SXSLT is seamlessly integrated with Scheme.

We illustrate the power of SXSLT with several examples abstracted from practical projects. We and other people have used SXSLT for over two years for real-life projects, ranging from authoring of static and dynamic Web pages to content-distribution engines. Our experience and user comments show that SXSLT is expressive and easy to use. We argue that this outcome is a consequence of SXSLT providing right abstractions for XML transformations, of being higher-order, declarative and extensible.
Keywords: XML, SXML, XSLT, tree traversal, Scheme.

1 Introduction

So far, the official (W3C) XML transformation language XSLT [19] has not proven very satisfactory. The problem is not XSLT's being a functional language — it is XSLT not being functional enough [15]. XML practitioners often cite the following drawbacks of the language:

Poor syntax Large XSLT programs are so verbose that they become unreadable. XML turns out a poor notation for a programming language [2]

Missing nuts-and-bolts "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." [15]

Low order XSLT templates are not first class. A transformed document or its part cannot easily be re-transformed [15]

Closed system XSLT is designed as a closed system. Extensions to operate with other languages exist, but are notably unwieldy [5].

In this paper, we present a practical XML transformation language that is free from the above drawbacks. This language, SXSLT, is more expressive than XSLT and is also simpler and more coherent. SXSLT is a *higher-order* pure functional transformation language based on flexible tree traversals. The language is implemented as a library in a pure functional subset of Scheme. The choice of core constructions and the illustration of their expressiveness is one contribution of this paper.

SXSLT has been used in real-life, commercial, educational and government environments. One example is an aviation weather query tool [7] for operational users and small plane pilots equipped with a handheld and a wireless receiver. Another example is generating an XML manifest file required for cataloging a weather data markup format in the U.S. Department of Defense's XML repository. Manual creation of such a file is unfeasible [8]. A Census Scope web site authored with SXSLT was highlighted in *Yahoo Pick!* on May 30, 2002. The authors of that site, whom we have not met nor corresponded with, publicly commented on how "extraordinarily easy" the project was [1]. Finally, this very paper has been composed in an abstract XML syntax (SXML) and then converted to LaTeX by an appropriate SXSLT transformation. SXSLT is in the public domain and available from a SourceForge repository [16].

SXSLT is characterized by the following features:

– XSLT and SXSLT both assume that the source XML document has been parsed into a tree form. They both transform the tree as instructed by a stylesheet. The result should be pretty-printed afterwards. A transformer of a re-writing rule (a "template") in SXSLT is a ordinary Scheme procedure, often a lambda-expression.

– SXSLT is a head-first re-writing system similar to Scheme macros. This fact makes it trivial to see when a rewriting rule applies. In XSLT, understanding when a template is triggered is a complex problem, requiring evaluating XPath expressions, computing priorities and keeping track of import orders. Unlike macros, SXSLT offers: (i) wildcard re-writing rules and transformers for text strings; (ii) the choice of applicative or normal transformation orders, even within the same stylesheet; (iii) local scoping of "templates"; (iv) repeated traversals of the source document.

– Like XSLT, SXSLT can scan the source document repeatedly; the traversal order can however be controlled by the user. Unlike XSLT 1.0, the result of the traversal can itself be traversed and manipulated, by the same or a different stylesheet.

– In an XSLT stylesheet the set of effective templates can be altered at runtime only by explicitly changing modes. In SXSLT, we can amend or replace the current stylesheet indirectly (via local scoping) or directly. The latter feature is reminiscent of expansion-passing macros.

We believe the combination of the above features is unique and powerful. We must stress that these features are not heaped up indiscriminately, but are induced by a compact core of tree traversal combinators and first class transformers. In its essence XSLT, too, implements a very simple mechanism of iterating over the document and of dispatching based on predicates. This simplicity is however covered by layers of extensions, such as modes and priorities. The examples in later sections of the paper show that SXSLT's technique of implicit and explicit switching of stylesheets subsumes the modes. In SXSLT, transformation templates become first class, which expands the expressiveness to a large degree. A user of SXSLT can easily re-transform the source or the output trees and hence "invoke a template" or an entire stylesheet, without any need for additional syntactic sugar.

SXSLT is seamlessly integrated with Scheme: re-writing transformers are ordinary Scheme functions, which may invoke other Scheme procedures. Traversal combinators are also Scheme functions. This integration of SXSLT with the mature general purpose programming language is especially important in the context of Web services, which require a deeper relation between manipulating the incoming XML document and performing the requested service.

We must stress that SXSLT transformation stylesheets are ordinary Scheme data structures: S-expressions. The ability of a transformation language to manipulate its own transformation rules was considered important by the designers of XSLT. The XML notation was meant to enable such a reflection. XSLT however fell short: matching, selections, string and arithmetic operations are expressed in XPath, in a syntax markedly different from that of XML, and from XSLT in particular. XML notation for the rest of XSLT made the language terribly verbose. In contrast, SXSLT is both reflective and comprehensible. The fact that an SXSLT transformer is a Scheme function generally limits the reflective power. The selection criterion of a SXSLT transformation rule, however, is always open to reflection. In contrast, the non-XML syntax of an action trigger in XSLT inhibits reflection.

The bulk of this paper introduces SXSLT on three expository examples. The examples meant to exhibit the salient features of SXSLT and to compare SXSLT and XSLT in expressiveness, while avoiding the deluge of minute details present in real-world projects. This paper cannot illustrate however all SXSLT features. In particular, the use of XPath-conformant expressions in SXSLT will not be covered. All the examples were inspired by XSLT samples found in literature. The first example gives a general feel for SXSLT and exhibits local scoping of re-writing transformers. The example in Section 3 highlights the higher order of SXSLT and its treatment of XML Namespaces. We show how the higher functional order of SXSLT has made a context-sensitive transformation concise. Section 4 elaborates an example of a multi-stage transformation with several stylesheets. We emphasize the ability to invoke stylesheets and to emulate XSLT's modes. The final two sections discuss the related work and conclude. Appendix A formally specifies SXML and SXSLT and shows an implementation of the SXSLT engine.

2 Example 1: Context-Sensitive Character Substitutions

The present section gives a general illustration of SXSLT. We describe a traversal combinator pre-post-order, first-class transformers, wildcard re-writing rules, and local scoping of the rules.

As the illustrating example we chose the conversion of XML documents into LaTeX for further typesetting and publishing. The example is of clear practical relevance: indeed, we use its full version to produce this paper. The conversion has to preserve and appropriately translate the marked-up structure. We should also translate between character encodings: XML and LaTeX have different sets of "bad" characters and different rules for escaping them. Furthermore, the escaping rules depend on the context. More precisely, we want to transform `<tag>text</tag>` into a LaTeX environment `\begin{tag}text\end{tag}`, `<p>text</p>` into `text` followed by an empty line, and `<br/>` into \\. The `text`, the content of an XML element, may contain characters such as `$`, `%`, and `{` that must generally be escaped in LaTeX , the `verbatim` markup being an exception.

A simplified version of our example, without the context dependency of character encoding rules, was elaborated by Moertel [15]. We deliberately built on his example so to compare our SXSLT solution with the XSLT code in his article. Even though Moertel solves a notably simpler problem, his XSLT code is voluminous.

The first step of an XML transformation is parsing a source document into an abstract syntax tree (AST). SXSLT uses an abstract syntax called SXML [10], which is also a realization of the XML Information set in the form of Lisp's S-expressions. An input XML document to translate into LaTeX will look in SXML similar to the following:

```
(document (quotation "citation line 1" (br) "another line"))
```

An SXML expression is indeed a tree of proper nodes and text strings. Proper SXML nodes are S-expressions formed from a *tag* (a Scheme symbol) followed by the list of child nodes. The SXML specification [10] describes in great detail the syntax of SXML nodes: element, attribute, namespace declaration, processing instruction, and other nodes. We can use any XML parser to convert the source document into this format.

Just like XSLT, SXSLT turns an AST of the source document into an AST of the result. The latter, when written out, is the transformed (in our case, LaTeX) document. Figure 1 presents the translation function and auxiliary procedures. An auxiliary procedure `make-char-quotator`, a higher-order generator of character-replacement functions, is elided. That code is available from the SXSLT distribution. The transformation task is carried out by a traversal combinator pre-post-order, which is invoked with the source SXML expression and a transformation stylesheet. The stylesheet is a list of re-writing rules, which associate SXML node tags (as keys) with the corresponding transformation procedures. We often call the latter handlers. Recall that an SXML node tag is the head of an S-expression that represents the node. The stylesheet is therefore not unlike a macro environment of Scheme, which associates syntactic keywords with macro transformers. Both SXSLT and macro transformers are ordinary Scheme procedures, which often use quasiquotation. Given a source S-expression, the pre-post-order combinator visits its nodes, looks up node tags in the supplied stylesheet and applies the corresponding transformers to the nodes. Therefore, pre-post-order is similar to Scheme's macro-expander. There are important differences however. By default, the combinator pre-post-order traverses the source expression in post-order. Given an SXML node `(tag child-node ...)` the combinator first visits `child-node`s, then looks up a re-writing rule for the `tag` in the stylesheet, and applies the corresponding handler to the *transformed children*. Such a re-writing regimen is appropriate for our task. The combinator pre-post-order can carry out other transformation strategies, which will be demonstrated in the later sections. The complete specification of the pre-post-order function and its implementation are given in Appendix A.

Figure 1 shows that the SXML->LaTeX translation is indeed rather uniform. We turn text strings into escaped text strings and proper SXML nodes into the corresponding LaTeX environments. SXSLT stylesheets may contain wildcard rules `*text*` and `*default*`, which facilitate such a generic processing. A `*default*` re-writing rule applies when no other rule does. A `*text*` transformer handles SXML character data nodes.

We have to make an exception for `verbatim` nodes in the source SXML document. Such nodes are translated into an `alltt` LaTeX environment to set off literal text. The `alltt` environment has a different set of characters to be escaped. Therefore, we have to alter our normal handling of text strings. The pre-post-order traversal combinator supports such a switching of node handlers, through scoping. An association between a node tag and its handler in a stylesheet may optionally include a "local stylesheet," which extends the current stylesheet during the traversal of node's children. The rule for a `verbatim` node, Fig. 1, takes advantage of this facility. Therefore, whenever pre-post-order visits a `verbatim` node, it shadows the default text transformer with the local one. The latter is responsible for special encoding of character data within the node. Altering the transformation environment through scoping appears more intuitive than that through XSLT modes.

The present paper is a production version of our example. The master file [11] contains the source of this paper in SXML and the stylesheet to transform the source into the LaTeX format.

3 Example 2: Context-Sensitive Term Generation

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: "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." [15]

```
(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 here
         ,(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))))
     )))

; 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))
```

Fig. 1. The SXML->TeX converter and auxiliary functions. `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.

The example in this section emphasizes how higher-order transformers in SXSLT elegantly solve a class of frequently arising problems: transforming an XML document so that different occurrences of an element yield different results depending on that element's ancestors, siblings, or descendants. We also highlight the treatment of XML Namespaces. Our example is an extended version of generating rudimentary English phrases from a database. The original problem was used in [13] to demonstrate XT3D: 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. 2. 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.

```
<db:purchase
  xmlns:db='http://internal.com/db'>
 <html:p
   xmlns:html=
     'http://www.w3c.org/HTML/'>
  4 tinkers</html:p>
</db:purchase>
```

⟹

```
<out:text
  xmlns:out='http://internal.com/out'
  xmlns:acc=
    'http://internal.com/accounting'
  acc:count='0'>4 tinkers</out:text>
```

```
<db:purchase
  xmlns:db='http://internal.com/db'
  xmlns:html=
    'http://www.w3c.org/HTML/'>
   <html:p>4 tinkers</html:p>
   <html:p>5 tailors</html:p>
   <html:p>2 soldiers</html:p>
   <html:p>1 spy</html:p>
</db:purchase>
```

⟹

```
<out:text
  xmlns:out='http://internal.com/out'
  xmlns:acc=
    'http://internal.com/accounting'
  acc:count='3'>4 tinkers,
    <out:text>5 tailors,
      <out:text>2 soldiers and
        <out:text>1 spy</out:text>
      </out:text>
    </out:text>
</out:text>
```

Fig. 2. Sample XML purchase records and the corresponding generated phrases.

As before, the first step of the transformation is parsing of an XML document into an abstract syntax tree form, SXML. The sample input and generated terms from Fig. 2 have the SXML form shown on Fig. 3.

The figure demonstrates that SXML tags are *extended* XML names, made of a local name and a Namespace URIs parts [20]. SXML tags are also Scheme symbols, with a space-efficient internal representation as mere references into a symbol table. A user may still wish to assign a short *namespace-id* for a long Namespace URI. We have indeed done so on Fig. 3: we assigned the namespace-id `h` for the HTML namespace URI. The namespace-ids are not to be confused with XML Namespace prefixes. The former uniquely identify Namespace URIs, and are chosen by the author of the stylesheet rather than by the author of the source

```
(http://internal.com/db:purchase
  (h:p "4 tinkers"))                ==> (out:text (@ (acc:count 0)) "4 tinkers"))
(http://internal.com/db:purchase        (out:text (@ (acc:count 3))
 (h:p "4 tinkers")                         "4 tinkers,"
 (h:p "5 tailors")                  ==>    (out:text "5 tailors,"
 (h:p "2 soldiers")                           (out:text "2 soldiers and"
 (h:p "1 spy"))                                  (out:text "1 spy"))))
```

Fig. 3. The sample SXML terms before and after the purchase-records-to-phrases translation.

document. The subject of namespaces in SXML is thoroughly discussed in [10]. We must emphasize that both simple and extended XML names are represented in SXML as atomic Scheme symbols. They only differ in appearance: symbols representing extended XML names are spelt with a colon character.

Figure 4 presents the translation function, `convert-db`. The latter takes an SXML expression, such as the one in the left column of Fig. 3, and yields the corresponding expression in the right column. The translation function is again an invocation of the pre-post-order traversal combinator. The combinator receives the source SXML expression and the translation stylesheet with three transformation rules. Of a particular interest is the rule for a `h:p` element. Its transformer is of a higher order: it rewrites an `h:p` element into a procedure. The latter takes a `count` argument and the variable number of arguments that must be procedures of the same kind. The transformer for the `purchase` element starts the domino effect of applying these procedures to the list of their followers. This sequence of (implicitly context-passing) applications is the reason the transformation of a `h:p` element turns out to be dependent on the element's siblings.

The stylesheet code on Fig. 4 is complete. It is instructive to compare the stylesheet with the equivalent XSLT code ([13], 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.

To obtain the final result as the one shown in the right-hand column of Fig. 2, we need to pretty-print the transformed tree. Curiously, SXML-to-XML conversion is also a SXSLT transformation task. The SXML specification (itself authored in SXML) and the SXSLT distribution give several examples of this straightforward transformation.

4 Example 3: Recursively Reorganizing Punctuation

Our third example, which highlights recursive, reflexive transformations with several stylesheets, 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 here

```
(define (convert-db doc)
  (pre-post-order doc
   ; the conversion stylesheet
   `((h:p . ,(lambda (tag str)
       (lambda (count . args)  ; count can be #f:
        (let ((att-list        ;   don't generate the attr
               (if count `((@ (acc:count ,count))) '())))
          (match-case args
           (() `(out:text ,@att-list ,str))
           ((?arg)
            `(out:text ,@att-list
               ,(string-append str " and")
               ,(arg #f)))
           ((?arg . ?rest)
            `(out:text ,@att-list ,(string-append str ",")
                   ,(apply arg (cons #f rest)))))))))

     (http://internal.com/db:purchase . ,(lambda (tag . procs)
        (if (null? procs) '()
            (apply (car procs)
                   (cons (length (cdr procs)) (cdr procs))))))
     (*text* . ,(lambda (trigger str) str)))))
```

Fig. 4. SXSLT transformation that implements translation on Fig. 3.

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 [3].

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 (`<em>` 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 '(#\. #\, #\? #\! #\; #\:))
```

Again we start by parsing the sample document into SXML (using the SSAX parser [9]):

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

Our pull-in 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 that 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 of the node.

While the description of the algorithm is fairly simple, it calls for a pre-order traversal. This is because nodes must examine their neighbors before they examine their children, and must modify the traversal of their children based on what they find adjacent. Fig. 5 presents the classification stylesheet and Fig. 6 that for transformation. Both stylesheets instruct the pre-post-order traversal combinator to navigate pre-order. This is the most appropriate strategy since the pull-in algorithm appears to require a hybrid traversal strategy based on breadth-first search.

The classification stylesheet is the only place that specifies, in a clear declarative manner, which elements can or cannot accept punctuation. The transformation stylesheet is generic. We should point out that the the `*default*` handler on Fig. 6 first transforms element children with a different, classification, stylesheet. The handler then re-processes that transformation's result. This fact highlights the advantages of SXSLT over XSLT. Because a node handler may recursively invoke the traversal function and pass it an arbitrary stylesheet, there is no need to introduce modes and to clutter the syntax and the semantics of the transformation language. Because the traversal combinator is a ordinary function, we can easily capture the result of a stylesheet transformation for further manipulation. Such an operation in XSLT 1.0 is immensely unwieldy [15].

The transformed SXML tree 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 [4]. DSSSL did not, however, survive the switch from SGML to XML. One of the major differences between SXML transformations and DSSSL are more flexible traversal strategies of SXSLT and flexible switching of stylesheets.

XT3D, introduced in [13], 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.

One common way to specify graph traversals and transformations is by combining node accessors by combinators. Wallace and Runciman [17] [18] have

```
; 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)))))
     )
    ))
```

Fig. 5. 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 that can accept the punctuation.

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

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

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

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

    (*default*
     *preorder*
     . ,(lambda (tag . elems)
           (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)
          )))))
```

Fig. 6. The transformation stylesheet.

developed a combinator library 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, the combinator approach is less intuitive for specifying generic transformations with exceptions (i.e., where the set of excepted nodes depends on the traversal context).

XDuce [6] 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" [14], 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 described SXSLT, an XML manipulation language. The language is a Scheme library that realizes a head-first re-writing system over trees representing XML Information set. SXSLT has been used in practice for over two years by us and other users. The examples elaborated in this paper have been abstracted from real-life projects. In our and other users' experience, SXSLT is expressive and easy to use. The elaborated examples indicate that such a positive experience is the consequence of SXSLT being a domain-specific declarative layer over a general-purpose functional language. The layer implements right abstractions and patterns of XML transformations: (i) abstract XML syntax trees, (ii) pre- and post-order traversals of abstract syntax trees, (iii) transformation environments (stylesheets), (iv) dispatch to re-writing transformers based on node names, (v) wildcard transformation rules, (vi) local scoping of re-writing rules. The base language, Scheme, gives us mature syntax, first-class and higher-order transformers and traversal combinators, extensibility, and "nuts-and-bolts."

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, many aspects of SXSLT can be expressed in a powerful macro system such as McMicMac [12], which commutes with program-processing tools such as type inference engines; this is one useful path for recovering type information.

SXML transformations often make multiple selections from one or more XML documents. It is possible to perform such queries with an SXSLT stylesheet that rewrites the document tree into the one that contains the desired selection. Using transformations to implement queries is inefficient, however, and reflects a mismatch of abstractions. For these situations, our library includes a query language called SXPath [16]. SXPath is, of course, a reflection of the W3C's XPath query language [21] into Scheme. Comparing SXPath with the Consortium's more advanced declarative language, XQuery [22], is a task for future work.

Acknowledgments

We would like to thank Philip Wadler for comments and invaluable advice, and the anonymous reviewers for many helpful suggestions. This work has been supported in part by the National Science Foundation under grants ESI-0010064 and ITR-0218973, the National Research Council Research Associateship Program, Naval Postgraduate School, and the Fleet Numerical Meteorology and Oceanography Center.

References

1. Abresch, B.: for those keeping score... A message on the PLT-Scheme mailing list. May 30, 2002 `http://www.cs.utah.edu/plt/mailarch/plt-scheme-2002/msg00998.html`
2. Bosworth, A.: Programming Paradox. XML Magazine, February 2002 `http://www.fawcette.com/xmlmag/2002_02/magazine/departments/endtag/`
3. Durand, D.: private communication. May 22, 2002
4. ISO/IEC. Information technology, Processing Languages, Document Style Semantics and Specification Languages (dsssl). Technical Report 10179 :1996(E), ISO (1996)
5. Fuchs, M.: SOXT: Building the XSL Family of Languages. The Eleventh International World Wide Web Conference. Proc. Alternate Paper Tracks (2002) `http://www2002.org/CDROM/alternate/417/index.html`
6. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. In The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2001) 67-80
7. Kiselyov, O.: Aviation total weather and SIGMET advisory queries. June 12, 2000 `http://zowie.metnet.navy.mil/cgi-bin/oleg/get-advisories` `http://zowie.metnet.navy.mil/~dhuff/pqa/FNMOC.html`
8. Kiselyov, O.: SXML as a normalized database. April 17, 2001 `http://pobox.com/~oleg/ftp/Scheme/xml.html#SXML-as-database`

9. Kiselyov, O.: A better XML parser through functional programming. In: Lecture Notes in Computer Science, Vol. 2257. Springer-Verlag, Berlin Heidelberg New York (2002) 209-224
10. Kiselyov, O.: SXML Specification. Revision 2.5. August 9, 2002 `http://pobox.com/~oleg/ftp/Scheme/SXML.html`
11. Kiselyov, O., Krishnamurthi, S.: SXSLT: Manipulation Language for XML. Master SXML file. `http://pobox.com/~oleg/ftp/Scheme/SXSLT.scm`
12. Krishnamurthi, S., Felleisen, M., Duba, B.F.: From Macros to Reusable Generative Programming. In: Lecture Notes in Computer Science, Vol. 1799. Springer-Verlag, Berlin Heidelberg New York (1999) 105-120
13. Krishnamurthi, S., Gray, K.E., Graunke, P.T.: Transformation-by-Example for XML. Practical Aspects of Declarative Languages (2000)
14. Lieberherr, K.L., Patt-Shamir, B.: Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA (1997)
15. Moertel, T.: 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`
16. S-exp-based XML parsing/query/conversion. `http://ssax.sourceforge.net/`
17. Wallace, M., Runciman, C.: Haskell and XML: generic combinators or type-based translation? Proc. the fourth ACM SIGPLAN international conference on Functional programming (1999) 148 -159
18. Wallace, M., Runciman, C.: HaXml - 1.07b (2002) `http://www.cs.york.ac.uk/fp/HaXml/`
19. World Wide Web Consortium. XSL Transformations (XSLT). Version 1.0. W3C Recommendation November 16, 1999 `http://www.w3.org/TR/xslt`
20. World Wide Web Consortium. Namespaces in XML. W3C Recommendation. January 14, 1999 `http://www.w3.org/TR/REC-xml-names/`
21. World Wide Web Consortium. XML Path Language (XPath). Version 1.0. W3C Recommendation. November 16, 1999 `http://www.w3.org/TR/xpath`
22. World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Working Draft. August 16, 2002 `http://www.w3.org/TR/xquery/`

Appendix A: SXSLT Specification and Implementation

SXSLT is a higher-order pure functional transformation language based on flexible tree traversals. The tree in question is usually an SXML tree [10], which is an abstract syntax tree of an XML document. In SXML, character data are represented as Scheme strings. XML markup — elements, processing instructions, attribute lists — are uniformly represented by a Scheme list. The head of such a list, which is always a Scheme identifier, names the markup. For an XML element, the corresponding SXML list starts with element's expanded name, optionally followed by lists of attributes and of effective namespaces. The rest of the SXML list is an ordered sequence of element's children — character data, processing instructions, and other elements.

Since SXML data is essentially a tree structure, the SXML grammar can be presented as a set of two mutually-recursive datatypes, `Node` and `Nodelist`, where the latter is a list of `Nodes`:

```
      ⟨Node⟩ ::= ( ⟨name⟩ . ⟨Nodelist⟩ ) | "text string"

  ⟨Nodelist⟩ ::= ( ⟨Node⟩ ⟨Node⟩* )
      ⟨name⟩ ::= ⟨LocalName⟩ | ⟨ExpandedName⟩ | @ |
                 *TOP* | *PI* | *COMMENT* | *ENTITY* |
                 *NAMESPACES* | @@
```

The SXML Specification [10] gives more detail.

The SXSLT language is implemented as a library in a pure functional subset of Scheme. At the core of this library is a function `pre-post-order`, which takes an SXML tree and and a stylesheet, and returns a transformed tree: `pre-post-order:: ⟨tree⟩ x ⟨bindings⟩ -> ⟨tree⟩`.

The stylesheet, ⟨`bindings`⟩, is a list of ⟨`binding`⟩s of the following structure:

```
           ⟨tree⟩ ::= ⟨Node⟩ | ⟨Nodelist⟩
       ⟨bindings⟩ ::= ( ⟨binding⟩ ⟨binding⟩* )
        ⟨binding⟩ ::= ( ⟨trigger-symbol⟩ *preorder* .
                      ⟨handler⟩ ) | ( ⟨trigger-symbol⟩
                      ⟨new-bindings⟩? . ⟨handler⟩ )
 ⟨trigger-symbol⟩ ::= ⟨name⟩ | *text* | *default*
   ⟨new-bindings⟩ ::= ⟨bindings⟩
```

and the ⟨`handler`⟩ is a procedure `⟨trigger-symbol⟩ x ⟨tree⟩ -> ⟨tree⟩`.

The function pre-post-order traverses a ⟨`tree`⟩, which is either a ⟨`Node`⟩ or a ⟨`Nodelist`⟩. For each ⟨`Node`⟩ of the form `( ⟨name⟩ ⟨Node⟩* )` the function looks up an association with the given ⟨`name`⟩ among its ⟨`bindings`⟩. When pre-post-order fails to find the association, it tries to locate a *`*default*`* binding. It is an error if the latter attempt fails as well.

Having found a binding, the function pre-post-order first checks whether the binding is of the form `( ⟨trigger-symbol⟩ *preorder* . ⟨handler⟩ )`. If the found binding prescribes a pre-order traversal, the handler is 'applied' to the current node. Otherwise, the pre-post-order function first calls itself recursively for each child of the current node, with ⟨`new-bindings`⟩ prepended to the current bindings. The result of these calls is passed to the ⟨`handler`⟩, along with the head of the current ⟨`Node`⟩. To be more precise, the handler is applied to the head of the current node and its processed children. The result of the handler, which should also be a ⟨`tree`⟩, becomes a ⟨`Node`⟩ of the transformed tree. If the current ⟨`Node`⟩ is a text string, a special binding with a symbol *`*text*`* is looked up.

The ⟨`new-bindings`⟩ that appears in `( ⟨trigger-symbol⟩ ⟨new-bindings⟩ . ⟨handler⟩ )` has the same form as ⟨`bindings`⟩. The ⟨`new-bindings`⟩ is a local stylesheet. It takes effect only when traversing the children of a particular node (whose name is the ⟨`trigger-symbol`⟩). The bindings of ⟨`new-bindings`⟩ override the bindings from the "parent" stylesheet.

Fig. 7 provides an implementation of `pre-post-order` that satisfies the above specification.

```
(define (pre-post-order tree bindings)
  (cond
   ((nodeset? tree)
    (map (lambda (a-tree) (pre-post-order a-tree bindings)) tree))
   ((not (pair? tree))
    (let ((trigger '*text*))
      (cond
       ((or (assq trigger bindings) (assq '*default* bindings)) =>
        (lambda (binding)
          ((if (procedure? (cdr binding)) (cdr binding) (cddr binding))
           trigger tree)))
       (else
        (error "Unknown binding for " trigger " and no default")))
      ))
   (else
    (let ((trigger (car tree)))
      (cond
       ((or (assq trigger bindings) (assq '*default* bindings)) =>
        (lambda (binding)
          (if (and (pair? (cdr binding)) (eq? '*preorder* (cadr binding)))
              (apply (cddr binding) tree)
              (apply
               (if (procedure? (cdr binding)) (cdr binding) (cddr binding))
               (cons trigger
                     (pre-post-order (cdr tree)
                                     (if (pair? (cdr binding))
                                         (append (cadr binding) bindings)
                                         bindings)))))))
       (else
        (error "Unknown binding for " trigger " and no default"))))))))
```

Fig. 7. The function `pre-post-order`: the core function of SXSLT