Applications of Fold to XML Transformation

Andy Wingo

wingo@pobox.com

Abstract

The benefit of considering XML parsing as a fold operation over the tree of elements is perhaps the best-kept secret in XML. Applying common functional programming patterns to the XML problem domain yields parsers that are both more expressive and less prone to common bugs.

This paper investigates the usefulness of fold as not just a pattern for parsing, but for transformation of parsed XML as well. A new variant of fold is introduced that is shown to underly the existing patterns of transforming XML, while providing functionality heretofore possible only via second-order traversal. Real-world examples are explored, including the motivating problem for this investigation: layout of declarative documents into Scalable Vector Graphics (SVG).

1. Introduction

Let us consider a mini-language for describing slides in XML (W3C 2006). For example, the following document describes a presentation composed of one slide:

```
<slides>
 <slide>
  <title>Hi.</title>
  <para>Hello<br/>world</para>
 </slide>
</slides>
```

Slides are for giving presentations, of course, and to present this document, the author needs to transform it into a form that some program can read and display on a screen. The goal of this paper is to investigate the applicability of standard functional programming techniques to effecting XML transformations, such as those necessary to present the above slides example. Specifically, we will show that the fold operator underlies the best of the existing traversal combinators, and can fruitfully be used as a scaffolding when deriving new ones.

Section 2 reviews one existing technique for XML transformation, `pre-post-order`, along with some of its limitations. Section 3 discusses the nature of this operator's limitations, resulting in the definition of a new fold variant, `foldts*`, in section 4. Section 5 shows that `pre-post-order` can be expressed as implementations of this new fold operator. Finally, in section 6 we derive a new traversal combinator, `fold-layout`, suitable for layout of Scalar Vector Graphics (SVG).

All of the source code presented in this paper is available for download at `http://wingolog.org/pub/fold-xml.tar.gz`.


```
(slides
 (slide
  (title "Hi.")
  (para "Hello" (br) "world")))
```

Figure 1. SXML representation of the slides example document.

```
<html>
 <body>
  <div class="slide">
   <h1>Hi.</h1>
   <p>Hello<br/>world</p>
  </div>
 </body>
</html>
```

Figure 2. Possible HTML representation of the slides example

```
(html
 (body
  (div (@ (class "slide"))
       (h1 "Hi.")
       (p "Hello" (br) "world"))))
```

Figure 3. SXML representation of the HTML of Figure 2

2. SSAX, SXML, SXSLT, and `pre-post-order`

This paper takes as its base Oleg Kiselyov's work on the SSAX project, a functional XML parsing and transformation suite (Kiselyov 2001; Kiselyov et al. 2006). SSAX is most commonly used via its standard parser, which returns an S-expression representation of the source XML document. For example, the SXML representation of our slides source document can be seen in Figure 1. While this paper will focus on processing SXML with Scheme, the discussions apply generally to any functional language operating on a parsed representation of an XML source document.

As a first try in presenting our slides example, let us take advantage of the ubiquity of the web browser, using it as a viewer application. A possible transformation of the source document into HTML (W3C 2002) is shown in Figure 2. Figure 3 shows how the output HTML parses as SXML.

Transforming this Scheme representation of the slides document into HTML is a simple operation of traversing the tree depth-first, performing different replacements at each level depending on the tag. While one could write a recursive traversal-

```
`((slides . ,(lambda (tag . kids)
               `(html (body ,@kids))))
  (slide  . ,(lambda (tag . kids)
               `(div (@ (class "slide"))
                     ,@kids)))
  (title  . ,(lambda (tag . kids)
               `(h1 ,@kids)))
  (para   . ,(lambda (tag . kids)
               `(p ,@kids)))
  (br     . ,(lambda (tag . kids)
               `(br)))
  (*text* . ,(lambda (tag txt) txt)))
```

Figure 4. Slides-to-HTML `pre-post-order` bindings

and-replacement procedure for every transformation task, it is convenient to separate out the traversal process into a dedicated operator. In this way, the user only has to think about the replacement operations. The SXML set of tools provides such an operator, `pre-post-order` (Kiselyov and Krishnamurthi 2003). It is invoked as:

```
(pre-post-order bindings tree)
```

`pre-post-order` is named as it is to indicate that it can do both bottom-up (post-order) and top-down (pre-order) traversals. In these first examples, we will speak only of the post-order traversals, which replace each node of the tree with the result of applying the appropriate binding to the node.

The user of `pre-post-order` supplies a suitable set of bindings to transform the document at hand. A binding specifies a procedure ("handler") to apply to a given kind of SXML node, which should return a replacement for that node. For example, a handler for transforming `para` to `p` could look like this:

```
(lambda (tag . children)
  `(p ,@children))
```

The bindings set is simply a list of these handlers, keyed by the tag that they should apply to. In this case, suitable bindings can be seen in Figure 4. Of course, since both `pre-post-order` and the bindings are normal Scheme functions, XML tree transformation is not limited to simple substitution. One has the full expressive power of a general-purpose language.

I have been using `pre-post-order` for a few years now, and am very happy with it. However a new problem motivated the writing of this paper: the transformation of declarative documents like our `<slides>` example into SVG, an XML vector graphics format (W3C 2003). For example, one way to render the slides example into SVG would be the following:

```
<svg width="1024" height="768">
 <g><text x="96" y="216"
          font-size="64px">
   <tspan x="96" y="216">Hello</tspan>
   <tspan x="96" y="280">world</tspan>
 </text></g>
</svg>
```

The most striking observation that results from this output, besides the profusion of seemingly-arbitrary numbers, is that the strings "Hello" and "world" need to be rendered differently. Recall our text transformer from Figure 4:

```
`(*text* . ,(lambda (tag txt) txt))
```

When `pre-post-order` encounters a string leaf in the SXML tree, it will call the procedure in the above binding with the string as `txt`[1]. There are no other parameters to the transformation, so it is impossible for the text handler to render two strings differently[2].

```
`((para
   . ,(lambda (tag . procs)
        `(text
          (@ (x "96") (y "216"))
          ,@(if (null? procs) '()
                (apply (car procs)
                       96  ; initial x
                       216 ; initial y
                       (cdr procs))))))
  (*text*
   . ,(lambda (_ txt)
        ; second-order: return a closure
        (lambda (x y . procs)
          (cons
           `(tspan (@ (x ,x) (y ,y))
                   ,txt)
           (if (null? procs) '()
               (apply (car procs)
                      x
                      (+ y 64) ; increment y
                      (cdr procs))))))))
```

Figure 5. Second-order `pre-post-order` stylesheet for paragraph layout

3. Context-dependent transformations

Our problem is that the standard XML transformation combinator, `pre-post-order`, does not appear to allow context-dependent transformations. This is not precisely the case; as suggested in Kiselyov and Krishnamurthi 2003, we can obtain the desired effect by performing a second-order traversal. Instead of returning values directly, we make the `*text*` handler return a function, and make the `para` handler apply those functions to each other in order.

Figure 5 gives bindings that could work to transform `para` into SVG. In this example, the text handler increments the y position by 64 at every line[3]. The result of processing the `para` element is still a first-order value because it starts the initial function application.

It is insufficient, however, to perform a partial second-order transformation. Layout of multiple paragraphs in a slide creates the need for paragraphs to be laid out differently, e.g. with different initial x, y positions. The `para` handler itself would need to return a closure. Likewise, a desire to print the slide number on each slide creates the need for the slide handler to know its number: a context-dependent transformation.

Clearly this solution lacks the grace of the stylesheet in Figure 4. With second-order transformations, we reintroduce traversal into the processing problem. In addition, we needlessly allocate a tree of closures as an intermediate step to producing the output tree[4].

Fortunately there is another way to approach this problem, for which we turn to Kiselyov's XML parsing paper (Kiselyov 2001). First we realize that `pre-post-order` is a multithreaded transfor-

[1] The `tag` will always be `'*text*`, an oddity of `pre-post-order`'s interface.

[2] Impossible, that is, unless we introduce imperative elements to the text transformer, with all the potential for bugs that that would entail.

[3] For simplicity we are laying out separate text fragments on separate lines. Layout of actual paragraphs would in general require font metrics and a full unicode support library.

[4] I am given to understand that this is not the case in lazy languages, such as Haskell.

```
(define (atom? x)
  (not (pair? x)))
(define (foldt fup fhere tree)
  (if (atom? tree)
      (fhere tree)
      (fup (map (lambda (kid)
                  (foldt fup fhere kid))
                tree))))
```

Figure 6. `foldt`

```
(define (assq-ref alist key default)
  (cond ((assq key alist) => cdr)
        (else default)))
(define (post-order/foldt bindings tree)
  (let* ((err (lambda args
                (error "no binding available")))
         (default (assq-ref bindings
                            '*default* err)))
    (define (fup kids)
      (apply
       (assq-ref bindings (car kids) default)
       kids))
    (define (fhere txt)
      ((assq-ref bindings '*text* err)
       '*text* txt))
    (foldt fup fhere tree)))
```

Figure 7. `post-order` in terms of `foldt`

mation, where the result of transforming one node is independent of its siblings.

That is to say, `post-order` can be implemented with the standard tree fold operator, `foldt`, shown in Figure 6. `foldt` requires the user to supply two hooks: one for processing leaves, and one for bottom-up processing of branches. The former corresponds to our `*text*` handler from Figure 4, and the latter corresponds to all other nodes (`para`, `slide`, etc.). A `post-order` based on `foldt` will have `fup` and `fdown` perform their replacements based on handlers obtained from the bindings set. Figure 7 shows a definition of `post-order/foldt`, already a powerful tool capable of transforming our slides example into HTML.

The "multi-threadedness" of `foldt` comes from its use of `map`, which does not specify the order in which its input list is mapped. `map` could process its list from multiple threads of execution with no change to its semantics. In order for identical siblings to be processed differently, we need to perform a single-threaded traversal, threading a monadic seed through all operations. The need for this extra seed argument indicates that we need a different fold algorithm, the more general `foldts`, as presented in Kiselyov 2001.

Figure 8 shows the implementation of `foldts`. Note that the `fhere` of `foldts` has an additional argument, the seed at that point in the traversal. We can make that seed carry layout information in addition to the transformed document, propagating such information as slide number and current x, y position across the transformation.

However, just as one would prefer to write transformations with `post-order` rather than `foldt`, it would be good to write a higher-order data-driven combinator to wrap `foldts` in a more usable interface. The following section examines the requirements for such an operator.

```
(define (fold proc seed list)
  (if (null? list)
      seed
      (fold proc (proc (car list) seed)
            (cdr list))))
(define (foldts fdown fup fhere seed tree)
  (if (atom? tree)
      (fhere seed tree)
      (fup seed
           (fold (lambda (kid kseed)
                   (foldts fdown fup fhere
                           kseed kid))
                 (fdown seed tree)
                 tree)
           tree)))
```

Figure 8. `fold` and `foldts`

$$
\begin{array}{lcl}
\textit{bindings} & ::= & (\textit{binding}...) \\
\textit{binding} & ::= & (\textit{tag}\ \textit{bindings}\ .\ \textit{handler}) \\
 & | & (\textit{tag}\ *\texttt{preorder}*\ .\ \textit{handler}) \\
 & | & (\textit{tag}\ *\texttt{macro}*\ .\ \textit{handler}) \\
 & | & (*\texttt{default}*\ .\ \textit{handler}) \\
 & | & (*\texttt{text}*\ .\ \textit{handler}) \\
\textit{tag} & ::= & \textit{symbol} \\
\textit{handler} & ::= & \textit{tree} \rightarrow \textit{tree}
\end{array}
$$

Figure 9. `pre-post-order` bindings syntax

4. `foldts*-values`, a new fold operator

As a first step to making a replacement for `pre-post-order` based on fold, we should be sure that `pre-post-order` itself can be expressed in terms of `foldts`. Besides the post-order template application that we have already seen, `pre-post-order` offers two more features:

1. Two variants of pre-order traversal, `*preorder*` and `*macro*`, differing in that the latter implicitly implicitly re-runs the result of pre-order traversal through the transformer. Lisp hackers will note the similarity to defmacro expansion.
2. Context-sensitive bindings. For example, although we did not write it this way, the `*text*` handler we defined could only apply within a `para` element.

A full bindings grammar is given in Figure 9.

Let us first consider `*preorder*` and `*macro*` processing. The underlying desire under these two processing methods is for a user of `pre-post-order` to modify the tree being traversed, during traversal (pre-order). Such tree modification cannot be performed in `foldts`, so we have to create a modified version of that combinator. We can implement pre-order tree munging by hooking into the pre-order function `fdown`, making it return a possibly-modified subtree in addition to a new seed value.

This new fold variant, `foldts*`, is shown in Figure 10. `foldts*` is `foldts`, but altered so that `fdown` returns two values. In this way we satisfy our requirement to support `*preorder*` and `*macro*` processing.

Our implementation of `foldts*` is complicated by Scheme's clunky handling of multiple values. Languages with first-order tuples would express this pattern more elegantly (Hutton 1999).

Likewise, the plan to implement context-sensitive bindings is more straightforward than its implementation. We need to pass around more information in the seed, not just the transformed

```
(define (foldts* fdown fup fhere seed tree)
  (if (atom? tree)
      (fhere seed tree)
      (call-with-values
          (lambda () (fdown seed tree))
        (lambda (kseed tree)
          (fup seed
               (fold (lambda (kid kseed)
                       (foldts* fdown fup fhere
                                kseed kid))
                     kseed
                     tree)
               tree)))))
```

Figure 10. `foldts*`

```
(define (fold-values proc list . seeds)
  (if (null? list)
      (apply values seeds)
      (call-with-values
          (lambda ()
            (apply proc (car list) seeds))
        (lambda seeds
          (apply fold-values proc (cdr list)
                 seeds)))))
```

Figure 11. `fold-values`

tree. `fdown` would then augment the seed to hold any specialized bindings, and `fup` and `fhere` would take their bindings from the seed instead of from the evaluation environment. Finally, `fup` strips the custom bindings off the seed, leaving the rest of the traversal with the original bindings.

In principle, making a multi-valued seed does not require further modifications to the `foldts*` algorithm. One simply constructs the seed as a record or other compound type. With Scheme, however, lack of tuples makes it much more convenient to define a `foldts*-values` that takes as many seeds as the user wants, passing them all around to the functions `fdown`, `fup`, and `fhere`.

We begin by defining a variant of the normal `fold` that takes multiple seeds, in Figure 11. Note that the *list* argument has been moved ahead so that all seeds are at the end of the argument list. We then build on `fold-values` to define `foldts*-values` in Figure 12.

The casual observer might be forgiven for missing the elegance of `foldts*-values`. To be explicit, it has three chief virtues:

Generality. `foldts*-values` builds on the generality of `foldts`. In the single-list case it reduces to `fold`. It adds the ability to perform pre-order rewrites of the tree being traversed.

Concision. `fdown`, `fup`, and `fhere` implementations are straightforward and non-recursive. Multiple seeds get destructured into the argument list, and multiple value returns are made via the standard first-order `values` function. Processing is completely separated from traversal.

Efficiency. As with `foldts`, traversal is strictly O(n) in the number of nodes visited. No intermediate garbage need be generated.

The last point about garbage generation might not be readily apparent. It is true that an implementation of `foldts*-values` for any number of seeds will produce garbage, without advanced type inferencing. However we can avoid the calls to both `apply` and

```
(define (foldts*-values fdown fup fhere
                        tree . seeds)
  (if (atom? tree)
      (apply fhere tree seeds)
      (call-with-values
          (lambda () (apply fdown tree seeds))
        (lambda (tree . kseeds)
          (call-with-values
              (lambda ()
                (apply fold-values
                       (lambda (tree . seeds)
                         (apply foldts*-values
                                fdown fup fhere
                                tree seeds))
                       tree kseeds))
            (lambda kseeds
              (apply fup tree
                     (append seeds kseeds))))))))
```

Figure 12. `foldts*-values`

`append` via macro-expanding `foldts*-values` for a specific number of values. In that way, most Scheme implementations will avoid heap allocation altogether.

5. `pre-post-order` in terms of `foldts*-values`

As a first example, let us augment our earlier implementation of `post-order` from Figure 7 to support context-sensitive bindings. Context-sensitive bindings are not supported in Figure 7's `post-order/foldt` because `foldt` only provides hooks for bottom-up rewrites, and the bindings should be modified top-down, in a manner similar to lexical scoping.

Logically, therefore, the strategy for implementing context-sensitive bindings via `foldts*-values` focuses on the `fdown` procedure. Furthermore, given that we already have to traverse the bindings list in `fdown` in order to find the lexical bindings, we can make `fdown` note the post-order handler at the same time, passing it in the seed to be called later in `fup`[5]. This approach will prove advantageous when it comes time to implement the `*preorder*` and `*macro*` processing for `pre-post-order`.

We will have three seeds passed around in `post-order`: the bindings, the handler that `fup` will call, and the return value that we are building up. Figure 13 shows the implementation.

A number of observations come to mind after seeing the `post-order` of Figure 13. On the positive side, almost all of the code in the example is concerned with the problem at hand, with only some "boilerplate" at the end due to Scheme's multiple-value handling. None of the code deals with traversal.

On the negative side, the implementation does make some garbage. It calls `reverse` in `fup` because the result of the bottom-up transformation is `cons`'d up from left to right; that is, because we are using `fold-left` instead of `fold-right`.

More grave is the new implementation's verbosity. It is less readable than `post-order/foldt`, and less readable than the canonical implementation that uses explicit recursion (not shown in this paper). Hutton suggests that most recursive algorithms suffer a loss of readability when expressed as fold operations, but then goes on to state that they gain by virtue of being easier to reason about (Hutton 1999). While it is true that I feel a subjective comfort in the correctness of a fold-based traversal operator, I will have to leave more formal proofs to others. From the perspective of this paper,

[5] The post-order handler might be considered to be the continuation of the bottom-up rewrite for that subtree.

```
(define (post-order bindings tree)
  (define (err . args)
    (error "no binding available" args))
  (define (fdown tree bindings pcont ret)
    (let ((tail (assq-ref bindings (car tree)
                          #f)))
      (cond
       ((not tail)   ; default binding
        (let ((default (assq-ref bindings
                                 '*default* err)))
          (values tree bindings default '())))
       ((pair? tail) ; lexical bindings
                     ; with handler
        (let ((new-bindings (append (car tail)
                                    bindings))
              (cont (cdr tail)))
          (values tree new-bindings cont '())))
       (else         ; handler found
        (values tree bindings tail '())))))
  (define (fup tree bindings cont ret
               kbindings kcont kret)
    (values bindings cont
            (cons (apply kcont (reverse kret))
                  ret)))
  (define (fhere tree bindings cont ret)
    (define (tcont x)
      (if (symbol? x)
          x          ; pass tags through
          ((or (assq-ref bindings '*text* #f)
               (assq-ref bindings '*default* err))
           '*text* x)))
    (values bindings cont
            (cons (tcont tree) ret)))
  (call-with-values
      (lambda ()
        (foldts*-values fdown fup fhere tree
                        bindings #f '()))
    (lambda (bindings cont ret)
      (car ret))))
```

Figure 13. `post-order` in terms of `foldts*-values`, with support for context-sensitive bindings

the important property of this version of `post-order` is that it is extensible: there is a clear mechanism by which we might thread a monadic seed through the traversal.

Indeed, the only part of the `post-order` implementation that needs to change to implement `*preorder*` and `*macro*` processing is the `fdown` handler. A suitable redefinition of `fdown` is given in Figure 14 . There is some inelegance in the `*preorder*` case, because `fup` expects the kids to be in reverse order, but otherwise the extension is straightforward.

6. SVG layout with `foldts*-values`

Finally, we are ready to deal with the motivating problem for this paper, functional programming techniques for SVG layout. Layout is a single-threaded operation: every text and graphic object takes up some part of the "page", meaning that as the layout proceeds, there is a constantly-growing region of space that is used up and should not be allocated to subsequent elements. For example, two subsequent text lines should not print on top of each other. We will need to thread some representation of this already-allocated space through the entire layout operation.

```
(define (fdown tree bindings pcont ret)
  (let ((tail (assq-ref bindings (car tree)
                        #f)))
    (cond
     ((not tail)
      (let ((default (assq-ref bindings
                               '*default* err)))
        (values tree bindings default '())))
     ((pair? tail)
      (let ((cont (cdr tail)))
        (case (car tail)
         ((*preorder*)
          (values '() bindings
                  (lambda x (reverse x))
                  (apply cont tree)))
         ((*macro*)
          (fdown (apply cont tree) bindings
                 pcont ret))
         (else
          (let ((new-bindings (append (car tail)
                                      bindings)))
            (values tree new-bindings cont
                    '()))))))
     (else
      (values tree bindings tail '())))))
```

Figure 14. A replacement `fdown` to turn Figure 13 into `pre-post-order`

In general, for two-dimensional layout, the already-allocated space is a set of polygons and half-planes in Cartesian space. Each element to be layed out on the page would have to determine which space is available, and choose a location in that space in which to render itself. This paper takes the drastically simplified view that these polygons are rectangular, contiguous and stretch for the width of the output medium. Furthermore each element may reserve some amount of horizontal space on the left-hand side, for left-to-right locales. In this way we can conflate the representation of the layout with the algorithm to choose a rendering space by representing the layout as a constantly-advancing x, y pen position, as in a typewriter[6].

While it would be possible at this point to define a derivate of `pre-post-order` that threads the x, y pen position through the traversal and handler procedures, it would be distasteful to hard-code a line height of 64 pixels into the `*text*` handlers as we did in Figure 5. It would be useful to define a layout operator of more general utility, like `pre-post-order`. To this effect, consider that layout of objects is always parameterized in a number of ways: line spacing, font size, margins, etc. To be of maximum utility, our operator should provide these parameters to the layout handlers. In addition, it would be useful for these parameters to be hierarchical in the sense of the web's Cascading Style Sheets (CSS), whereby each level of the document tree can override certain parameters (W3C 1999a). One might wish to render a certain paragraph in a different font, for example.

Let this new combinator be named `fold-layout`. It will invoke `foldts*-values` with the three seeds of `pre-post-order`: the bindings, post-handler, and the return value. Additionally, it will have two extra seeds: the currently allocated space (the "current layout"), and the parameters.

[6] Readers seeking actual layout algorithms are probably now disappointed. I prefer to think of it as leaving open areas of investigation.

```
(define (fold-layout bindings params layout tree)
  (define (err . args)
    (error "no binding available" args))
  (define (fdown ...) ...)
  (define (fup ...) ...)
  (define (fhere ...) ...)
  (call-with-values
      (lambda ()
        (foldts*-values
         fdown fup fhere tree
         bindings #f params layout '())
    (lambda (bindings cont params layout ret)
      (values (car ret) layout))))
```

Figure 15. Skeleton of `fold-layout`

```
(cartouche (@ (line-color "red")
              (text-height 56))
           (para "Warning: Smoking Kills"))
```

Figure 16. Example source document for `fold-layout` transformation

```
(g (rect (@ (fill "none") (stroke "red")
            (stroke-width "4")
            (width "660") (height "120.0")
            (x "0") (y "0")
            (ry "20")))
   (text (@ (xml:space "preserve")
            (font-size "56")
            (font-family "Georgia")
            (x "32")
            (y "88"))
         (tspan (@ (x "32") (y "88"))
                "Warning: Smoking Kills")))
```

Figure 17. Result of transforming Figure 16 into an SVG fragment

`fold-layout` itself will take four arguments: the set of bindings, the initial parameters, the initial layout, and the SXML document. It will return two values, the transformed document and the final layout.[7] Figure 15 gives the skeleton of `fold-layout`, leaving us the task of defining `fdown`, `fup`, and `fhere`, along with determining the representation of the bindings, params, and layout seeds.

In the end we want a usable operator; that is, an operator whose capabilities are adequate for its anticipated uses. We have already mentioned some of the operator's desired capabilities, but to keep us honest we should focus concretely on an example document. Let us use as an example the SXML from Figure 16. The intention is to transform the source document into SVG that has a paragraph inside a box with rounded corners. The intended output document is given in Figure 17, and an annotated rendering of the output SVG is shown in Figure 18.

When traversing the source document, we have a number of well-defined places that we can hook into to modify the seeds. The

[7] The final layout might be useful to pass to another invocation of `fold-layout`.

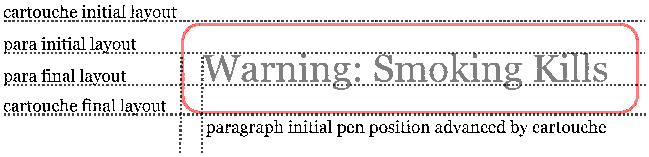

Figure 18. Annotated rendering of Figure 17

bindings	::=	(*binding*...)
binding	::=	(*tag handlerpair*...)
	|	(∗`default`∗ . *posthandler*)
	|	(∗`text`∗ . *texthandler*)
tag	::=	*symbol*
handlerpair	::=	(`pre` − `layout` . *prelayouthandler*)
	|	(`post` . *posthandler*)
	|	(`pre` . *preorderhandler*)
	|	(`macro` . *macrohandler*)
	|	(`bindings` . *bindings*)

Figure 19. Partial definition of `fold-layout` bindings syntax. The "handler" types are functions.

following subsections simulate the traversal, noting the changes that the seeds should undergo at each point[8].

6.1 `fdown cartouche`

Here we enter the processing of the `cartouche`. The layout seed (pen position) is what was passed into `fold-layout` as its initial value, and is represented in Figure 18 by the topmost horizontal line. Because we want to make some space between the red outline box and the paragraph, we should return a layout advanced by the cartouche's top padding, and indented in the x direction by the amount of left padding.

The cartouche's top padding itself should be taken from the parameters, augmented by any parameters set on the `cartouche` element itself. We can take the parameters from the XML attributes, represented in SXML as a subelement with tag `@`, which must follow the initial tag. In this case, the specific parameters are the `line-color` and `text-height` attributes. Since the parameters are lexically scoped, we can use a pattern similar to that of simple Scheme interpreters, representing parameters as a list of association lists. At each descent into a new SXML node, we cons the new parameters onto the list. Lookup proceeds left-to-right in the parameters list, stopping at the first alist in which a parameter is found.

Regarding the bindings, we will need potentially more than one binding. For example, we will see that in the cartouche document we need a post-order binding to render the box, in addition to a pre-layout binding to advance the pen. Here we break compatibility with `pre-post-order` and define the bindings as an alist of tagged alists, with exceptions for `*default*` and `*text*`. A partial definition of the bindings syntax is given in Figure 19.

Finally, the `fdown` handler can do pre-order rewrites of the tree. In my experience, this is useful when constructing domain-specific SXML dialects, and so should be supported in `fold-layout`. Fortunately we already implemented it for `pre-post-order`. While we are doing the rewrite, given that we already process the at-

[8] The seeds do not change in the imperative sense, of course; they change in the sense that the old values are thrown away, and the rest of the traversal proceeds with the values returned by the `fdown`, `fup`, and `fhere` handlers, as appropriate.

```
(define (cartouche-pre-layout tree params layout)
  (let ((x (layout-x layout))
        (y (layout-y layout)))
    (let-params params (margin-left margin-top)
      (make-layout (+ x margin-left)
                   (+ y margin-top)))))
```

Figure 20. Example pre-layout handler for `cartouche`

tributes as parameters, we can dispense with recursing into `@` altogether.

The `fdown` handler has a lot of work to do. It can potentially modify all of the seeds (bindings, params, layout, return value, post-layout handler), in addition to the tree itself. To make a usable interface for the user of `fold-layout`, we will need to divide this functionality into separate handlers with sensible defaults. This paper's approach to parameter and bindings specialization can be coded directly into `fdown`. The pre-layout pen advancement depends on the current parameters, the tree being processed, and the current layout; as such the $prelayouthandler$ type of Figure 19 is $tree \rightarrow parameters \rightarrow layout \rightarrow layout$, in Haskell notation[9]. Pre-order and macro rewrites can be treated the same as in `pre-post-order`, with both types $preorderhandler$ and $macrohandler$ equivalent to $tree \rightarrow tree$. Figure 20 shows a suitable pre-layout handler for this example, for suitable definitions of `make-layout`, `layout-x`, and `layout-y`. Also, Figure 20 uses without definition the macro `let-params`, which binds lexical variables from the parameters list.

6.2 `fdown para`

The `para` element presents no special problems. We have already discussed the generic nature of parameter and bindings specialization, and of pre-order rewrites. The one thing that one might want to do to paragraphs in pre-order would be to advance the pen to account for any margins or padding, which would take place in the `pre-layout` handler.

6.3 `fhere *text*`

Here we render our first element. Recall the desired output from Figure 17:

```
(tspan (@ (x "32") (y "88"))
       "Warning: Smoking Kills)))
```

In SVG, the x and y positions of text are relative to the baseline, which in latin scripts corresponds roughly to the bottom of the characters. We then take our current layout, increment its y by the current font-size as taken from the parameters, and render the `tspan` at the unmodified x and the new y. The result of the text handler is then the `tspan` output, as well as a new layout advanced by the current linespacing. That is to say, that the type $texthandler$ is equivalent to $string \rightarrow parameters \rightarrow layout \rightarrow (tree, layout)$[10]. An implementation of the text handler is defined in Figure 21.

6.4 `fup para`

To define the post-order handler for `para`, we again look to the desired SVG output from Figure 17:

```
(text (@ (xml:space "preserve")
         (font-size "56")
         (font-family "Georgia")
         (x "32")
         (y "88"))
      ...)
```

[9] This notation indicates that the last type is the return type, and all others are the argument types.

[10] The parenthesized $(tree, layout)$ indicates that the procedure returns two values.

```
(define (make-text-x params layout)
  (layout-x layout))
(define (make-text-y params layout)
  (let-params params (text-height)
    (+ text-height (layout-y layout))))
(define (layout-advance-text-line params layout)
  (let-params params (text-height line-spacing)
    (make-layout (layout-x layout)
                 (+ (* text-height line-spacing)
                    (layout-y layout)))))
(define (text-handler text params layout)
  (values
   (layout-advance-text-line params layout)
   `(tspan
     (@ (x ,(number->string
             (make-text-x params layout)))
        (y ,(number->string
             (make-text-y params layout))))
     ,text)))
```

Figure 21. Example text handler for SVG layout

```
(define (p-post tag params old-layout layout kids)
  (values
   layout
   (let-params params (text-height font-family)
     `(text
       (@ (xml:space "preserve")
          (font-size ,(number->string text-height))
          (font-family ,font-family)
          (x ,(number->string
               (make-text-x params old-layout)))
          (y ,(number->string
               (make-text-y params old-layout))))
       ,@kids))))
```

Figure 22. Example post-layout handler for `para`

The SVG text model allows multiple spans of text (`tspans`) to be regarded as a single element, so as to allow the human user to select lines in a paragraph with a mouse, among other applications. This works well enough for our idea of paragraphs. We wrap the child `tspan` elements in a `text` element, whose x and y are identical to those of the first `tspan`. In addition, the `para` constructs a `style` attribute for the output SVG based on which parameters were specialized when going into the `para` element. Finally, as in the text handler, the post-order handler will return an advanced pen position, accounting for the layout of its kids and any padding that the `para` chooses to add.

We are free to choose a convenient type for $posthandler$, for which convenient formulation is $tag \rightarrow parameters \rightarrow layout \rightarrow layout \rightarrow (tree...) \rightarrow (layout, tree)$. The two layouts correspond to the layout of the parent and the layout after processing the child trees, respectively. The last argument is a list of children, not a tree by itself because it omits the tag and attributes. Figure 22 shows a simple post-layout handler that could work for `para`.

```
(define (cartouche-post tag params old-layout
                        layout kids)
  (let ((oldx (layout-x old-layout))
        (oldy (layout-y old-layout))
        (newy (layout-y layout)))
    (let-params params (margin-bottom stroke-width
                        line-color page-width)
      (values
       (make-layout oldx (+ newy margin-bottom))
       `(g (rect
            (@ (fill "none") (stroke ,line-color)
               (stroke-width ,(number->string
                               stroke-width))
               (width ,(number->string
                        (- page-width (* 2 oldx))))
               (height ,(number->string
                         (- newy oldy)))
               (x ,(number->string oldx))
               (y ,(number->string oldy))
               (ry "20"))) ; rounded corners
           ,@kids)))))
```

Figure 23. Example post-layout handler for `cartouche`

6.5 `fup cartouche`

Finally, when performing post-layout on the `cartouche` node, we will want to draw a rectangle and advance the pen. Because the post-layout handler can return only one node, we wrap the rectangle and whatever other kids the cartouche has (in this case, the paragraph) in a generic SVG grouping container, `g`. The exact dimensions of the rectangle will depend on the page width, the size of the paragraph, and the internal padding of the cartouche. A possible implementation is shown in Figure 23.

6.6 Implementation of `fdown`, `fup`, `fhere`

Having fleshed out all of the data structures of `fold-layout`, we are ready for the somewhat complicated implementations of the `fdown`, `fup`, and `fhere` handlers that `fold-layout` will pass to `foldts*-values`.

We begin with `fdown`, which is the most difficult handler. In this case it will be even more complicated than its analog in `pre-post-order`, as here there is an additional pre-order handler (`pre-layout`), in addition to a different approach to SXML attributes. Specifically, as mentioned in section 6.1, we will avoid recursing into the SXML `@` node, instead treating them as parameters.

Figure 24 shows the implementation of `fdown` for `fold-layout`. There are four cases that it handles, shown in the final `cond` block. The first is the degenerate case in which the node's tag is not in the bindings set, in which the params and layout are passed unchanged. The `macro` and `pre` cases are very similar to their equivalents in `pre-post-order`. Finally, there is the case in which we have to look up any custom bindings, `post` handler, and `pre-layout` handler from the bindings set. Because we skip past the tag and attributes, the `cont-with-tag` closure is created to reintroduce the tag to the `post` handler.

Thankfully, `fup` and `fhere` are simple, shown in Figures 25 and 26. With these definitions, `fold-layout` is complete. Figure 27 shows an example invocation of `fold-layout` that could be used to transform the document from Figure 16.

When developing `fold-layout`, we coded in the capability for `pre` and `macro` handlers, but showed no example of their use. While further practical use of `fold-layout` will make their utility more apparent, I imagine that `macro` handlers will be useful

```
(define (fdown tree bindings pcont
               params layout ret)
  (define (fdown-helper new-bindings
                        new-layout cont)
    (let ((cont-with-tag
           (lambda args
             (apply cont (car tree) args)))
          (bindings
           (if new-bindings
               (append new-bindings bindings)
               bindings)))
      (cond
       ((null? (cdr tree))
        (values
         '() bindings cont-with-tag
         (cons '() params) new-layout '()))
       ((and (pair? (cadr tree))
             (eq? (caadr tree) '@))
        (let ((params (cons (cdadr tree) params)))
          (values
           (cddr tree) bindings cont-with-tag
           params new-layout '())))
       (else
        (values
         (cdr tree) bindings cont-with-tag
         (cons '() params) new-layout '())))))
  (define (no-bindings)
    (fdown-helper
     #f layout
     (assq-ref bindings '*default* err)))
  (define (macro macro-handler)
    (fdown (apply macro-handler tree)
           bindings pcont params layout ret))
  (define (pre pre-handler)
    (values '() bindings
            (lambda (params layout
                     old-layout kids)
              (values layout (reverse kids)))
            params layout
            (apply pre-handler tree)))
  (define (have-bindings tag-bindings)
    (fdown-helper
     (assq-ref tag-bindings 'bindings #f)
     ((assq-ref tag-bindings 'pre-layout
                (lambda (tag params layout)
                  layout))
      tree params layout)
     (assq-ref tag-bindings 'post
               (assq-ref bindings
                         '*default* err))))
  (let ((tag-bindings (assq-ref bindings
                                (car tree)
                                #f)))
    (cond
     ((not tag-bindings)
      (no-bindings))
     ((assq-ref tag-bindings 'macro #f)
      => macro)
     ((assq-ref tag-bindings 'pre #f)
      => pre)
     (else (have-bindings tag-bindings)))))
```

Figure 24. `fdown` implementation for `fold-layout`

```
(define (fup tree bindings cont params layout ret
             kbindings kcont kparams klayout kret)
  (call-with-values
      (lambda ()
        (kcont kparams layout klayout
               (reverse kret)))
    (lambda (klayout kret)
      (values bindings cont params klayout
              (cons kret ret)))))
```

Figure 25. `fup` implementation for `fold-layout`

```
(define (fhere tree bindings cont params
               layout ret)
  (call-with-values
      (lambda ()
        ((assq-ref bindings '*text* err)
         tree params layout))
    (lambda (tlayout tret)
      (values bindings cont params tlayout
              (cons tret ret)))))
```

Figure 26. `fhere` implementation for `fold-layout`

```
(define *cartouche-stylesheet*
  `((para
     (post . ,p-post))
    (cartouche
     (pre-layout . ,cartouche-pre-layout)
     (post . ,cartouche-post))
    (*text* . ,text-handler)))
(define *default-params*
  '((margin-left 32) (margin-right 32)
    (margin-top 32) (margin-bottom 32)
    (line-spacing 1.0)
    (font-family "Georgia")
    (stroke-width 4)
    (line-color "blue")
    (text-height 64)
    (page-width 660)))
(define (cartouche->svg doc)
  (fold-layout doc *cartouche-stylesheet*
               *default-params*
               (make-layout 0 0)))
```

Figure 27. Example function, wrapping `fold-layout` to transform the slides SXML of Figure 16 into the SVG of Figure 17

mostly as abbreviations, and `pre` handlers as ways of drawing pre-generated graphic elements onto the canvas without updating the layout, at least in the SVG case. `pre` handlers can be seen as a kind of built-in escape valve, inserting fragments into the output without further processing.

7. Future Directions and Applications

Several directions of future work are apparent. I have used `foldts*` as a basis for `fold-layout` because it offers the possibility of single-threaded traversal, but fold based combinators have other interesting properties as well. As mentioned in section 5, investigating the applicability of proof procedures to XML traversal might yield fruit, academic or otherwise.

Unmentioned in this paper is XSLT, the standard XML transformation toolkit (W3C 1999b). The SXSLT paper has already compared XSLT to `pre-post-order`, including a number of context-sensitive transformations using second-order techniques (Kiselyov and Krishnamurthi 2003). It would be interesting to revisit the context-sensitive tranformations using a `foldts*`-based combinator. Additionally, although angle-bracketed programming provokes pain, it would be instructive to see a solution to the SVG layout problem from an XSLT perspective.

The process of parameter specialization in `fold-layout`, in which the attributes of each element are prepended to the existing parameters set, is probably lacking in expressiveness. One would prefer an even closer similarity to Cascading Style Sheets (CSS) via support of external style sheets, in addition to those styles specified in the document itself (W3C 1999a). Perhaps parameters and SXML attributes are being conflated unnecessarily.

`fold-layout`, as expressed in the strict language Scheme, currently assumes that one has the entire document available in memory. One might investigate the applicability of `fold-layout`-based stream transformations, in which data is received one byte at a time over the network. One particularly interesting avenue to explore would be integration with SSAX, the fold-based functional XML parser (Kiselyov 2001). A `fold-layout` binding set could be compiled into a custom SSAX parser.

The SVG layout stylesheet developed in section 6 is simplistic to an extreme. Its concept of layout as a constantly-advancing pen position is only suitable for simple problems. It might be worth it to develop a more capable two-dimensional layout algorithm. Then there is the fact that the stylesheet does not wrap text lines; indeed, it does not even know how long a particular text fragment is. This most-difficult problem could have a solution via the use of Pango, a text layout and rendering library written in C (Taylor et al. 2007). Pango is available for Guile Scheme via the Guile-Gnome project (Wingo et al. 2006).

Alternately, instead of emitting SVG in SXML, one could make the `fold-layout` handlers draw directly, using a suitable graphics library. Cairo, in particular, offers a suitable set of graphics primitives, high quality output, multiple output surface types, and bindings to a number of Scheme implementations (Worth et al. 2007).

The real motivating problem for this paper was, as mentioned previously, layout of declarative documents into SVG. (As an output format, SVG has the advantage that it can be touched-up via the use of a canvas-based graphics editor.) A full solution to this problem would define a standard declarative SXML vocabulary for presentations, not just the small dialect defined in this paper. One would want a set of standard transformation bindings for transforming this dialect into SVG, HTML, text, and other output formats.

Very few people will find XML dialects to be an easy medium in which to draft and create documents, however. Recent years have seen the rise of more human-friendly structured languages, such as Markdown (Gruber 2007). The two approaches can meet: given a Markdown-to-XML parser, one can transform the Markdown language into the presentation vocabulary, allowing presentation generation from a Markdown source document. There is at least one web site that uses SVG for presentations, changing slides via toggling layer visibility in reaction to keypresses (Hirth 2006). One can imagine a web service: submit a document in Markdown format, and the user is mailed a presentation.

Alternately, one could pursue ways to integrate with dedicated SVG viewer programs. Extending this further, an SVG editor could offer an import-from-Markdown capability, powered by `fold-layout`.

8. Conclusions

These pages have presented several complete XML transformation tools: `post-order`, `pre-post-order`, and `fold-layout`. The latter operator proves capable of transforming a declarative XML document into an absolutely-positioned SVG graphic.

The important result of this paper is not the `fold-layout` combinator, however. The important result is that we may see the generic pure-functional fold operator as underlying many forms of XML transformation, and that if we need to derive a new combinator, it is possible to do so in a straightforward fashion, as we did in section 6. Furthermore, `foldts*`, along with its more comfortable cousin `foldts*-values`, proves to be a particularly lucid and appropriate fold variant to use as a starting point.

Acknowledgments

I would like to thank Oleg Kiselyov for his encouragement, and for valuable comments on a draft of this paper.

References

John Gruber. Markdown, 28 April 2007. URL `http://daringfireball.net/projects/markdown/`.

Jos Hirth. XSVG Slideshow, 15 August 2006. URL `http://kaioa.com/k/xsvgslide/`.

Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

Oleg Kiselyov. A better XML parser through functional programming. *Lecture Notes in Computer Science*, 2257:209, 2001.

Oleg Kiselyov and Shriram Krishnamurthi. SXSLT: Manipulation Language for XML. *Proc. 5th International Symposium on Practical Aspects of Declarative Languages, PADL '03*, 2003.

Oleg Kiselyov et al. S-exp-based XML parsing/query/conversion, 31 August 2006. URL `http://ssax.sourceforge.net/`.

Owen Taylor et al. Pango, 28 April 2007. URL `http://www.pango.org/`.

W3C. Cascading Style Sheets, Level 1. Technical report, World Wide Web Consortium (W3C), 11 January 1999a. URL `http://www.w3.org/TR/CSS1.html`.

W3C. Scalable Vector Graphics (SVG) 1.1 Specification. Technical report, World Wide Web Consortium (W3C), 14 January 2003. URL `http://www.w3.org/TR/SVG/`.

W3C. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). Technical report, World Wide Web Consortium (W3C), 1 August 2002. URL `http://www.w3.org/TR/xhtml1/`.

W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). Technical report, World Wide Web Consortium (W3C), 16 August 2006. URL `http://www.w3.org/TR/xml/`.

W3C. XSL Transformations (XSLT) Version 1.0. Technical report, World Wide Web Consortium (W3C), 16 November 1999b. URL `http://www.w3.org/TR/xslt/`.

Andy Wingo et al. Guile-Gnome, 5 November 2006. URL `http://www.gnu.org/software/guile-gnome/`.

Carl Worth et al. The cairo graphics library, 11 May 2007. URL `http://cairographics.org/`.