A better XML parser through functional programming

Oleg Kiselyov

Software Engineering, Naval Postgraduate School, Monterey, CA 93943
oleg@pobox.com, oleg@acm.org

Abstract This paper demonstrates how a higher-level, declarative view of XML parsing as folding over XML documents has helped to design and implement a better XML parser. By better we mean a full-featured, algorithmically optimal, pure-functional parser, which can act as a stream processor. By better we mean an efficient SAX parser that is *easy to use*, a parser that does not burden an application with the maintenance of a global state across several callbacks, a parser that eliminates classes of possible application errors.
This paper describes such better XML parser, SSAX. We demonstrate that SSAX is a better parser by comparing it with several XML parsers written in various (functional) languages, as well as with the reference XML parser Expat. In the experience of the author the declarative approach has greatly helped in the development of SSAX. We argue that the more expressive, reliable and easier to use application interface is the outcome of implementing the parsing engine as an enhanced tree fold combinator, which fully captures the control pattern of the depth-first tree traversal.
Keywords: XML parsing, traversal, tree fold, Scheme, Haskell

1 Introduction

On the surface of it, parsing of XML presents no problems. We merely need to apply yacc/lex or a similar tool to the Extended BNF grammar in the XML Recommendation. XML parsing ought to be even easier in functional languages, thanks to the development of intuitive parsing combinator libraries.

It comes as a surprise then that all but two functional-style XML parsers barely comply even with a half of the XML Recommendation [13]. None of the pure or mostly functional-style XML parsers support XML Namespaces. With the exception of FXP [10], the existing functional-style parsers cannot process XML documents in a stream-wise fashion. These parsers thus exhibit significant processing latency and are limited to documents that can fit within the available memory. The application interface of the only one functional, full-featured, stream-oriented parser FXP mirrors the API of the reference XML parser Expat [4]. The latter is notorious for its difficult and error-prone application interface.

XML is markedly more difficult to parse than it is commonly thought. It is by no means sufficient for a parser merely to follow the Extended BNF grammar of XML. Besides the grammar, the XML Recommendation [13] specifies a great number of rules (e.g., whitespace handling, attribute value normalization, entity references expansion) as well as well-formedness and validity constraint checks, which a parser must implement. Whitespace handling rules in particular require an unusually tight coupling between tokenizing and parsing.

The second peculiar aspect of XML parsing is its strong emphasis on efficiency and the convenience of the application interface. The traditional view of parsing as a transformation of a source document into an abstract syntax tree is deficient for several classes of XML applications. We should note first that the traditional approach does apply to XML, where it is called a Document Object Model (DOM) parsing. The DOM approach is a necessity for applications that repeatedly traverse and search the abstract syntax tree of a document. Other applications however scan through the document tree entirely, and only once. Such applications can potentially process an XML document as it is being read. Loading the whole document into memory as an abstract syntax tree is then inefficient both in terms of time and memory. Such applications can benefit from a lower-level, event-based model of XML parsing called a Simple Application Programming Interface for XML (SAX). A SAX parser applies user-defined actions to elements, attributes and other XML entities as they are identified. The actions can transform received elements and character data on the fly, or can incorporate them into custom data structures, including the DOM tree. Therefore, a SAX parser can always act as a DOM parser. The converse is not true.

Although the SAX XML parsing model is more general, more memory efficient and faster, SAX parsers are regarded as difficult to use: "It feels like you are trapped inside an eternal loop when writing code. You find yourself using many global variables and conditional statements" [3].

Is it possible to implement an efficient, compliant, stream-oriented XML parser with a convenient user interface that minimizes the amount of user-application state? Furthermore, can functional programming help to design and to implement such a parser?

This paper proves by construction that the answer to both questions is yes. The contribution of this paper is a SSAX parser [7], a compliant SAX XML parser that is being used in several industrial applications. SSAX is not a toy parser: it fully supports XML Namespaces, character, internal and external parsed entities, `xml:space`, attribute value normalization, processing instructions and CDATA sections. At the same time, SSAX minimizes the amount of application-specific state that has to be shared among user-supplied event handlers. SSAX makes the maintenance of an application-specific element stack unnecessary, which eliminates several classes of common bugs. SSAX is written in a pure-functional subset of Scheme. Therefore, the event handlers are referentially transparent, which makes them easier for a programmer to write and to reason about. The superior user application interface for the event-driven XML parsing is in itself a contribution of the paper. The paper demonstrates that this interface is not an accident but the outcome of a correctly chosen control abstraction, which captures the pattern on depth-first traversal of trees.

The key design principle of SSAX was a view of an XML document as an n-ary tree laid out in a depth-first order. XML parsing is then a tree traversal. We review the topic of functional-style tree traversals in Section 2. We will concentrate on efficiency and on capturing the pattern of such traversals in a higher-order combinator, `foldts`. In Section 3 we describe the SSAX parser, which is an implementation of `foldts` with the tree in question being an XML document. Section 4 demonstrates on several concrete examples that the SSAX parser is indeed efficient, easier to use and less error-prone, compared to other SAX parsers, in particular the reference XML parser Expat and its pure functional analogue FXP. We conclude in Section 5 that functional programming is intuitive and helpful not only for processing XML but for parsing it as well.

2 Depth-first traversals of trees

We start with a very simple example of a functional-style depth-first tree traversal and gradually extend it to improve efficiency and to abstract the pattern of the traversal. Although the SSAX parser has been implemented in Scheme, this section will use Haskell notation. The latter is more succinct; furthermore, it is more convenient for direct comparison with important papers on tree folding [5][6], which use Haskell notation.

Our trees are represented by the datatype

```
data Tree = Leaf String | Nd [Tree]
```

Given such a tree, we turn to our first problem of concatenating strings attached to all leaves, in their depth-first traversal order. If we view our trees as realizations of an XML information set [14], our first problem becomes that of computing a *string-value* for the root node of the information set .

The obvious solution to the problem

```
str_value1:: Tree -> String
str_value1 (Leaf str) = str
str_value1 (Nd kids) = foldr (++) "" (map str_value1 kids)
```

where

```
foldr:: (a->b->b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

although elegant, is deficient. Indeed, let us apply `str_value1` to a full binary tree of depth k whose leaves are one-character strings (2^k leaves total). Executing `str_value1` then requires $k2^k$ character-moving operations and produces $(k-1)2^k$ garbage characters. The algorithm can be improved by noting that we do not have to concatenate the strings eagerly. Instead, we can accumulate strings in a list and join them after the traversal.

```
str_value2:: Tree -> String
str_value2 = concat . str_value2'

str_value2' (Leaf str) = [str]
str_value2' (Nd kids) = concat (map str_value2' kids)
```

This halves the amount of garbage and the number of character movements. However, appending two lists of size 2^i takes 2^i operations. The algorithm still has the time complexity of $O(k2^k)$; it still produces $k2^{k-1} + 2^{k+1}$ list cells of garbage. The best solution is to build a list of strings in the reverse order – with the reversal and concatenation at the very end:

```
str_value3:: Tree -> String
str_value3 = concat . reverse . (str_value3' [])

str_value3' seed (Leaf str) = str : seed
str_value3' seed (Nd kids) = foldl str_value3' seed kids
```

where

```
foldl:: (a->b->a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Some language systems offer a string-concatenate-reverse function, which halves the amount of the produced garbage. The running time of `str_value3` is linear in the size of the tree. The amount of garbage – while unavoidable – grows only linearly with the size of the tree. The function `str_value3` differs from `str_value1` and `str_value2` in another aspect. The actions at children nodes of the same node are no longer independent. The actions are threaded through the seed argument and must be performed in order. The independence of actions in `str_value1` and `str_value2` manifested itself in the presence of `map`, which is absent in `str_value3`.

We now turn to the next example – computing a digest of a tree. We want to traverse a tree depth-first and to compute an MD5 hash of all encountered nodes and leaf values. A hash function is generally non-associative. Therefore, we have no choice but to use a stateful traversal similar to that of `str_value3`.

```
md5Init:: MD5Context
md5Update:: String -> MD5Context -> MD5Context
md5Final:: MD5Context -> String

tree_digest:: Tree -> String

tree_digest = md5Final . (tree_digest' md5Init)
tree_digest' ctx (Leaf str) =
  md5Update "/leaf" $ md5Update str $ md5Update "leaf" ctx
```

```
tree_digest' ctx (Nd kids) =
  md5Update "/node" $ foldl tree_digest' (md5Update "node" ctx) kids
```

Can we separate the task of tree traversal and recursion from the task of transformation of a node and a state? The benefits of encapsulating common patterns of computation as higher-order operators instead of using recursion directly are well-known [11][5]. For lists, the common pattern of traversal is captured by the familiar `foldl` and `foldr` operators, which can be generalized to trees [11][6]:

```
foldt:: (String -> a) -> ([a] -> a) -> Tree -> a
foldt f g (Leaf str) = f str
foldt f g (Nd kids) = g (map (foldt f g) kids)
```

Unlike the functions `str_value1` and `str_value2`, the efficient `str_value3'` cannot be expressed via `foldt` in a simple way because the actions at branches are dependent on the history of the traversal and cannot be simply mapped to children nodes. Such functions are often distinguished [11] by an extra parameter, which acts as a an accumulator or a continuation: (cf. `str_value2'` with `str_value3'` above). Such functions can be written as second-order folds [11], which return procedures as results. In our example:

```
str_value31 tree = concat $ reverse $ (str_value31' tree [])
   where
      str_value31' = foldt (\str seed -> str : seed)
          (\new_kids seed -> foldl (flip ($)) seed new_kids)
```

This representation requires higher-order features of the language and often not as efficient because `(str_value31' tree)` creates as many closures as there are nodes in the `tree`. The closures are then applied to `[]`, which generates the final result. In strict languages such as ML or Scheme (used in the following sections), closure creation is relatively expensive.

To make "mapping" of an accumulating function to a tree efficient, we introduce a more general control operator:

```
foldts :: (a->a) -> (a->a->a) -> (a->[Char]->a) -> a-> Tree-> a
foldts fdown fup fhere seed (Leaf str) = fhere seed str
foldts fdown fup fhere seed (Nd kids) =
     fup seed $ foldl (foldts fdown fup fhere) (fdown seed) kids
```

A user instantiates `foldts` with *three* actions; for comparison, `foldr` requires only one action and `foldt` needs two. The three `foldts` actions are threaded via a `seed` parameter, which maintains the local state. An action accepts a seed as one of its arguments and returns a new seed as the result. The action `fhere` is applied to a leaf of the tree. The action `fdown` is invoked when a non-leaf node is entered and before any of the node's children are visited. The `fdown` action has to generate a seed to be passed to the first visited child of the node. The action fup is invoked after all children of a node have been seen. The action is a function of two seeds: the first seed is the local state at the moment the traversal process enters the branch rooted at the current node. The second seed is the result of visiting all child branches. The action fup is to produce a seed that is taken to be the state of the traversal after the process leaves the current branch.

The two previously considered examples – computation of a string value and of a digest for a tree – can easily be written with foldts:

```
str_value32 = concat . reverse . (str_value32' [])
  where
        str_value32' = foldts id (\_->id) (flip (:))
```

In this example, the seed is the list of leaf values accumulated in the reverse order. The fhere action prepends the value of the visited leaf to the list. The actions fdown and fup are trivial: they merely propagate the seed.

```
tree_digest2 = md5Final . (foldts fd fu fh md5Init)
  where fh ctx str = md5Update "/leaf" $  md5Update str $
                                          md5Update "leaf" ctx
        fd ctx = md5Update "node" ctx
        fu _ ctx = md5Update "/node" ctx
```

The computation of the tree digest is no more complex. The seed is the MD5 context. The fdown and fup actions mark the fact of entering and exiting a non-leaf node. This example clearly demonstrates that consuming node values and updating the local state are separated from the task of traversing the tree and recurring into its branches. This separation makes operations on tree nodes simpler to write and to comprehend.

3 XML parsing as tree traversal

The enhanced tree fold, foldts, has more than theoretical interest. The foldts combinator is literally at the core of the pure functional XML parser SSAX. To see how foldts applies to XML parsing, we note that an XML document with familiar angular brackets is a concrete representation of a tree laid out in a depth-first order. Elements, processing instructions, CDATA sections and character data are the nodes of such a tree. The latter three are always the leaf nodes. Attributes are collections of named values attached to element nodes. Since element nodes can be non-terminal nodes, the moments the traversal enters and leaves an element node must be specifically marked, respectively as the start and the end tags. XML parsing then is a depth-first traversal of an XML document regarded as a tree. XML parsing is a pre-post-order, down-and-up traversal as it invokes user actions when the traversal process enters a node and again when the process has visited all child branches and is about to leave the node.

Just like the `foldts`, the SSAX framework captures the pattern of the XML document traversal (i.e., parsing). To be more precise, the framework carries out such parsing chores as tokenizing, XML namespace resolution and the namespace context propagation, the whitespace mode propagation, the expansion of character and parsed entity references, attribute value normalization, maintaining the traversal order. The user can therefore concentrate on the meaningful work – what to do at encountered nodes.

At the heart of SSAX is a function `SSAX:make-parser`, which takes user-supplied node action procedures (also called content handlers) and instantiates the corresponding XML parser. Similarly to `foldts`, `SSAX:make-parser` requires three mandatory handlers: `new-level-seed`, `finish-element`, and `char-data-handler`. These handlers closely correspond to the procedural parameters `fdown`, `fup` and `fhere` passed to `foldts`. The output of `SSAX:make-parser` is a procedure `Port -> Seed -> Seed`. The first argument is a port from which to read the input XML document. The port is treated throughout the SSAX framework as if it were a "unique" parameter, using the terminology from the programming language Clean. The second argument to the parser is the initial value of the application state, the seed. The parser returns the final value of the seed, the result of a tree-induced composition of the user-supplied handlers.

`SSAX:make-parser` also accepts a number of optional handlers, which will be called when the parser encounters a processing instruction, a document type declaration, or the root element. If the optional handlers are omitted, the instantiated parser will be non-validating. `SSAX:make-parser` is actually a macro, which *integrates* the handlers into the generated parser code. We can regard `SSAX:make-parser` as a staged parser.

The semantics of `SSAX:make-parser` is the same as that of `foldts`. Both traverse a tree in a depth-first order and invoke handlers at "interesting points". Besides the traversal state `SSAX:make-parser` also maintains the list of active entities and the namespace context. The user handlers of `SSAX:make-parser` are also more complex, receiving as additional arguments the name of the current element and its attributes.

In the following section we consider several typical instantiations of `SSAX:make-parser`, with the goal of estimating SSAX complexity and comparing it with other XML parsers. The comparison will demonstrate the benefits of modeling the SSAX parser after `foldts`.

4 SSAX examples and comparisons

4.1 The complexity of SSAX parsing

The first example of using SSAX is untagging. This is a common "XML to text" translation that removes all markup from a well-formed XML document. We should point out that this is the same example as the one discussed in Section 2. Indeed, untagging is precisely determining the string-value of an XML document tree. The example in Section 2 operated on trees represented as linked data structures in memory. In this section a tree is an XML document itself. In both cases, we traverse the tree and accumulate all character data as we encounter them. As Section 2 explained, it is beneficial to accumulate the character data in a list in reverse order and join them at the very end.

The procedure to remove markup from an XML document is shown below. This is an instantiation of the SSAX parser with three handlers: `new-level-seed`

```
(define (remove-markup xml-port)
  ; Accumulate the text values of leaves in a seed, in reverse order
  (let ((result
          ((SSAX:make-parser
             NEW-LEVEL-SEED
             (lambda (elem-gi attributes namespaces expected-content seed)
               seed)

             FINISH-ELEMENT
             (lambda (elem-gi attributes namespaces parent-seed seed) seed)

             CHAR-DATA-HANDLER
             (lambda (string1 string2 seed)
               (let* ((seed (cons string1 seed)))
                 (if (string-null? string2) seed (cons string2 seed))))
           )
           xml-port '())))
     (string-concatenate-reverse result)
    ))
```

and `finish-element` merely propagate the seed, while `char-data-handler` adds the character data to the list. Since the optional document-type and root element handlers are omitted, the remove-markup parser is non-validating. The pieces of character data are passed to `char-data-handler` in two string arguments for efficiency. The similarity of the remove-markup code with `str_value32` of Section 2 is striking. We must note however that `str_value32` relied on `foldts`, which traversed a linked structure of type `Tree` in memory. The remove-markup procedure on the other hand parses an XML document, which it reads from a given input port. When this port contains a document such as the one on Fig. 1, the procedure yields a string "01234567". To verify that remove-markup, just

```
<node><node><node><leaf>0</leaf><leaf>1</leaf></node>
            <node><leaf>2</leaf><leaf>3</leaf></node></node>
      <node><node><leaf>4</leaf><leaf>5</leaf></node>
            <node><leaf>6</leaf><leaf>7</leaf></node></node></node>
```

Figure 1. A full binary tree as an XML document

as `str_value32`, runs in time and space that grows only linearly with the size of XML documents, we applied the procedure to documents such as the one on Fig. 1 of increasing depth. We ran all benchmarks on a Pentium III Xeon 500 MHz computer with 128 MB of main memory and FreeBSD 4.0-RELEASE operating system. The benchmark Scheme code was compiled by a Gambit-C 3.0 compiler. Figure 2 shows the result. The SSAX parser indeed has the linear space and time complexities. This is the *experimental* result, obtained by measuring the performance of the full-scale XML parser.

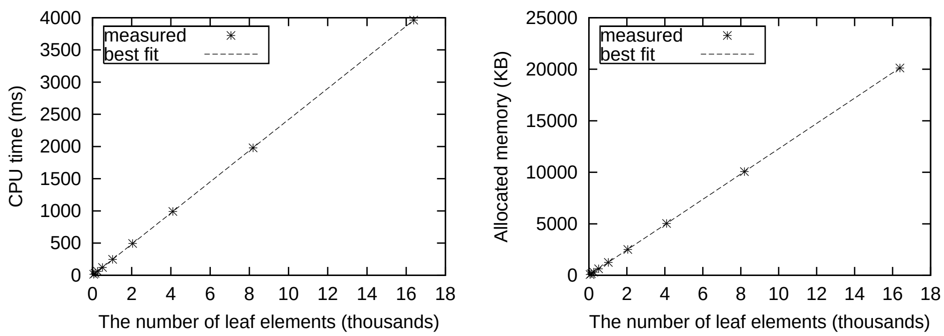

Figure 2. Performance of the SSAX parser for documents of the form given on Fig. 1. The CPU time and the cumulative amount of allocated memory are plotted as functions of the number of leaf elements in the input XML document. Most of the allocated memory was garbage-collected.

4.2 SSAX and Expat

No discussion of XML parsing can avoid Expat, which is *the* reference XML parser, written in C by James Clark [4]. Expat is a SAX, i.e., a stream-oriented parser. As the user passes it chunks of the input XML document, Expat identifies elements, character data or other entities and invokes the appropriate handler (if the user has registered one). The size of the chunks is controlled by the user; chunks can range from one byte to the whole XML document.

A tutorial article about Expat [2] explains well how Expat is supposed to be used. A user application most certainly has to have "a good stack mechanism in order to keep track of current context... The things you're likely to want to keep on a stack are the currently opened element and it's attributes. You push this information onto the stack in the start handler and you pop it off in the end handler." As an illustration, the Expat tutorial discusses a sample application `outline.c`, which prints an element outline for an XML document, indenting child elements to distinguish them from the parent element that contains them. In this case, the stack is represented by a global variable Depth, which controls the amount of indenting white space to print before the element name. The variable is incremented in a user-supplied start-element handler, and is decremented in the end-element handler. A simplified code for the two user handlers is given on Fig. 3.

```
int Depth;
void start(void *data, const char *el, const char **attr) {
 int i;
 for (i = 0; i < Depth; i++)
      printf("  ");
 printf("%s\n", el);
 Depth++; }

void end(void *data, const char *el) { Depth--; }

int main(void) {
   XML_Parser p = XML_ParserCreate(NULL);
   XML_SetElementHandler(p, start, end); /* register the callbacks */
   /* invoke XML_Parse() passing the buffer with the XML document
     or a part of it */ }
```

Figure 3. A simplified code for the `outline.c` application, using Expat.

It is instructive to compare the Expat application `outline.c` with the corresponding SSAX application, whose complete code is given on Fig. 4.

In the Expat application, the maintenance of the application state, the Depth, is split across two separate handlers. This fact increases the possibility of an error. The `ssax-outline` application on the other hand has no global variables or other application-specific stack to maintain. Unlike the Expat handlers, SSAX handlers can be largely decoupled and thus easily written and understood.

The function `ssax-outline` also illustrates the benefit of the SAX XML parsing mode. The function prints element names as they are identified and accumulates no data. It can therefore process documents of arbitrary size – far bigger than the amount of available memory. The function `ssax-outline` is a true stream processor, with low memory requirements and low latency.

To compare the performance of SSAX and Expat, we ran several benchmarks. We need to discuss first the difference in input modes of the two parsers. An application that uses Expat is responsible for reading an XML stream by blocks and passing the blocks to Expat, specifically noting the last block. Expat requires the calling application be able to determine the end of the XML document stream *before* parsing the stream. If an application can do that, it can read the stream by large blocks. An application can potentially load the whole document into memory and pass this single block to Expat. Expat uses shared substrings

```
(define (ssax-outline xml-port)
  ((SSAX:make-parser
     NEW-LEVEL-SEED
     (lambda (elem-gi attributes namespaces expected-content seed)
       (display seed)                ; indent the element name
       (display elem-gi) (newline)   ; print the name of the element
       (string-append "  " seed))    ; advance the indent level

     FINISH-ELEMENT                  ; restore the indent level
     (lambda (elem-gi attributes namespaces parent-seed seed) parent-seed)

     CHAR-DATA-HANDLER
     (lambda (string1 string2 seed) seed)
   )
   xml-port ""))
```

Figure 4. The complete code for the outline application, using SSAX. The seed describes the depth of an element relative to the root of the tree. To be more precise, the seed is the string of space characters to output to indent the current element.

extensively, and therefore is specifically optimized for such a scenario. If we take a document from a (tcp) pipe, it may be impossible to tell offhand when to stop reading. Furthermore, if we unwittingly try to read a character past the logical end of stream, we may become deadlocked. SSAX reads ahead by no more than one character, and only when the parser is positive the character to read ahead must be available. SSAX does not need to be told when the document is ended. On the contrary, SSAX will tell us when it has finished parsing a root (or other) element. SSAX can therefore safely read from pipes, can process sequences of XML documents without extra delimiters, and can handle selected parts of a document.

The performance benchmarks are based on the code to remove markup from an input XML document. This task, which was described in the previous section, simulates a typical Web service reply processing. Two input documents are XML encodings of full binary trees of depth 15 and 16. The documents are similar to the one on Fig. 1. The documents contain more markup than character data and, in addition, exhibit deeply nested elements. Overall the benchmark task is a good exercise of XML parsing engines. The first benchmark application, `string-value.c`, implements the most favorable to Expat scenario: it reads the whole document into memory, passes it to Expat and asks the parser to remove the markup. The second benchmark application, `string-value-by-one.c`, also uses Expat and also loads the whole document into memory first. The application however passes the content of that buffer to Expat one character at a time. This simulates the work of the SSAX parser. Finally, a SSAX benchmark `string-value-ssax.scm` likewise loads an XML document first, opens the memory buffer as a string port and passes the port to SSAX. The complete benchmark code is a part of the SSAX project [7].The results are presented in Table 1.

benchmark	XML-tree-depth-15	XML-tree-depth-16
string-value.c	0.105/0.016	0.213/0.022
string-value-by-one.c	0.747/0.014	1.494/0.012
string-value-ssax.scm	1.092/0.024	2.170/0.095
File size, bytes	884,723	1,769,459

Table 1. User/system times, in seconds, for running three benchmarks on two sample XML documents. The timing results were obtained from a precise virtual clock and reproduce within 3%. Platform: FreeBSD 4.0-RELEASE system, Pentium III Xeon 500 MHz, Bigloo 2.4a Scheme compiler. The numbers above reflect activities that occur entirely in memory. There was no i/o of any kind, there were no page faults.

The most notable result of the benchmarks is that a Scheme application is only 1.4 times slower than a comparable well-written C application, `string-value-by-one.c`. SSAX seems quite competitive in performance, especially keeping in mind that the parser and all of its handlers are referentially transparent. The ability to read from pipes and streams whose end is not known ahead of parsing costs performance. We do think however that the feature is worth the price. Shared substrings, present in some Scheme systems (alas not in the compiler used for benchmarking) will mitigate the trade-off.

4.3 FXP, the functional equivalent to Expat, and SSAX

The closest to SSAX XML parser is FXP [10], which is a purely functional, validating XML parser "shell" with a functional variant of the event-based interface. FXP is written in SML. Both SSAX and FXP invoke user-supplied handlers (called "hooks" in FXP) at "interesting" moments during XML parsing. The hooks receive an application state parameter and must return a possibly new state. The ways SSAX and FXP frameworks are instantiated to yield a specific XML processing application are also surprisingly similar, modulo static/dynamic typing. FXP "vitally relies" on SML's parameterized modules for customization while SSAX depends on Scheme's macros.

The most notable difference between SSAX and FXP is the interface between the parsing engine and the event handlers (hooks). SSAX is based on `foldts`, whereas the interface of FXP seems to be a pure functional analogue of Expat's application interface. The difference between the SSAX and FXP interfaces is important and instructive. A sample FXP application discussed at the end of the FXP API documentation [10] is a good example to illustrate that difference. The application converts an XML document to an abstract syntax tree form, which is not unlike the `Tree` datatype from Section 2. A SSAX distribution includes a similar function `SSAX:XML->SXML`. It is instructive to compare event handlers of the two applications. In both cases the event handlers are pure functional; they receive from the parsing engine recognized pieces of markup or character data and accumulate them in a parse tree.

In the FXP application, the application data – the seed, in SSAX terminology – represent the partial document tree constructed so far. As the FXP documentation describes it, the seed has two components – a stack and the content. At any point the stack holds all currently open start-tags along with a list of their left siblings. The content component accumulates the children of the current element that are known so far. In the initial state, both components are empty. Character data event handlers add the identified character data to the content of the current element. The hook for a start-tag pushes that tag together with the content of the current element onto the stack. The element started by that tag becomes the current element. The end-tag hook reverses the content of the current element, pops the tag of the current element off the stack and combines it with its content. The constructed tree is then prepended to the content of the parent element which now becomes the current element.

The code for the comparable SSAX application is given on Figure 5. The code does correspond to the description of the FXP application to a certain extent. However, `simple-XML->SXML` is notably simpler. Whereas FXP application's state is comprised of a stack and the content, `simple-XML->SXML`'s state is a regular list. The list contains the preceding siblings of the current element or a piece of character data, in reverse document order. Maintenance of FXP's stack was split across two separate hooks: the handlers for the start and the end tags. The FXP handlers have to detect possible stack underflow errors. In contrast, the handlers of `simple-XML->SXML` are relieved of any stack maintenance and error handling responsibility: The function `simple-XML->SXML` does not have any stack. As Figure 5 shows, `simple-XML->SXML` handlers hardly do anything at all. The handler `new-level-seed` is particularly trivial; `finish-element` is not more complex either. The simpler the handlers are, the easier it is to write them and to reason about them.

```
(define (simple-XML->SXML port)
  (reverse
   ((SSAX:make-parser
      NEW-LEVEL-SEED
      (lambda (elem-gi attributes namespaces expected-content seed)
        '())

      FINISH-ELEMENT
      (lambda (elem-gi attributes namespaces parent-seed seed)
        (cons (cons elem-gi (reverse seed)) parent-seed))

      CHAR-DATA-HANDLER
      (lambda (string1 string2 seed)
        (if (string-null? string2) (cons string1 seed)
            (cons* string2 string1 seed)))
     )
     port '())))
```

Figure 5. A simplified `SSAX:XML->SXML` function from the SSAX distribution.

We should point out that not only `simple-XML->SXML` lacks a stack, the SSAX parsing engine itself does not have an explicit stack of currently open XML elements. The traversal stack is implicit in activation frames of a recursive procedure `handle-start-tag` of the SSAX framework. If there is no explicit stack, there can be no stack underflow errors. Thus the comparison between FXP and SSAX indicates that the SSAX framework provides a higher level of abstraction for SAX XML parsing. This is the direct consequence of building SSAX around the `foldts` tree traversal combinator.

4.4 Other XML parsers written in functional languages

There are several other XML parsers implemented in functional languages: CL-XML (written in Common Lisp), XISO (written in Scheme), Tony (in OCaml), and HaXml (in Haskell). They are all DOM parsers. Neither of these parsers can process an XML document "on the fly," in a stream-like fashion.

Parser CL-XML [1] is the most thorough of the group. It checks all well-formedness and most of the validation constraints given in the XML Recommendation. It is the only parser among those considered in this paper (besides SSAX and Expat) that supports XML namespaces, XML whitespace handling, general entity expansion, attribute value normalization, and the proper handling of CDATA sections. CL-XML is the least functional-style parser: it is written in imperative style, with extensive reliance on global and dynamic-scope variables.

XISO [9] is a mostly-functional parser implemented in Scheme. It is a pure DOM, non-validating parser. It is does not check many of XML well-formedness constraints either. Another parser of the similar quality is Tony [8], which is written in OCaml. It is not a pure functional parser: the parsing state and a character data accumulator are mutable. Like XISO, Tony does not detect or expand entity references, does not handle CDATA sections, does not support namespaces – it does not even handle newlines in attribute values.

One component of HaXml [12], a collection of utilities for using Haskell and XML together, is an XML parser. The parsing component includes a hand-written XML lexer, which produces a token stream for the parser proper. The latter is based on a slightly extended version of the Hutton/Meijer parser combinators. The HaXml parser does not do the normalization of attribute values and does not support XML namespaces. It does not detect many well-formedness let alone validation errors. The separation between the lexing and the proper parsing stages in HaXml is a principal weakness as tokenizing an XML document heavily depends on the parsing context. The weakness manifests itself, for example, in parser's failure to handle newlines and special characters within quoted strings.

The HaXml parser is a DOM parser. An XML document is first tokenized, then it is converted into a parse tree representation, which is handed over to a user application. Because Haskell is a non-strict language however, the lexer does not generate new tokens until they are required by the parser. The parser does not make a new node of the parse tree until this node is accessed in the user application code. Thus the HaXml framework *could* act similar to a SAX parser despite its multi-phase processing. This potential is not realized as HaXml eagerly loads the whole document into a string, to let the lexer backtrack or look-ahead by arbitrary amount. In contrast, SSAX never backtracks a character and never looks more than one character ahead. Therefore SSAX can handle (sequences of) documents from a TCP pipe or other stream.

5 Conclusions

In this paper we have shown an example of a principled construction of a SAX XML parser. The parser is based on a view of XML parsing as a depth-first traversing of an input document considered as a spread-out tree. We have considered the problem of efficient functional traversals of abstract trees and of capturing the pattern of recursion in a generic and expressive control structure. We have found such efficient and generic higher order operator: `foldts`. Unlike the regular tree fold, `foldts` permits space- and time-optimal accumulating tree traversals.

The `foldts` operator became the core of SSAX, a SAX parser that walks an XML document and invokes user-supplied handlers when it identified elements, processing instructions, character data and other entities. The SSAX parsing engine effectively abstracts the details of the XML document tree; the engine makes it unnecessary for user handlers to maintain their own stack of open elements; the engine reduces the amount of application state shared among the user handlers to the bare minimum. The comparison with other SAX parsers (the reference XML parser Expat and its functional analogue FXP) shows that SSAX provides a higher-level abstraction for SAX XML parsing. The user-handlers of SSAX are referentially transparent, are less error-prone to write and to reason about.

The SSAX parser is a full-featured, pure-functional, stream-oriented, algorithmically optimal SAX parser, which also makes user handlers easier to write and thus removes whole classes of possible bugs. The combination of these features distinguishes SSAX among other XML parsers. The features are the effect of the principled SSAX construction, in particularly, of the `foldts` traversal operator.

Acknowledgments I would like to thank Shriram Krishnamurthi for valuable discussions, detailed comments and suggestions. This work has been supported in part by the National Research Council Research Associateship Program, Naval Postgraduate School, and the Army Research Office under contracts 38690-MA and 40473-MA-SP.

References

1. Anderson, J.: Common Lisp support for the 'Extensible Markup Language' (CL-XML). Version 0.906, June 2, 2001. http://homepage.mac.com/james_anderson/XML/documentation/cl-xml.html

2. Cooper, C.: Using Expat. *xml.com*, September 1, 1999. http://www.xml.com/pub/a/1999/09/expat/index.html
3. Dunford, M.: DOM XML: An Alternative to Expat. http://www.phpbuilder.com/columns/matt20001228.php3
4. Expat XML Parser. Version 1.95.1, October 21, 2000. http://sourceforge.net/projects/expat
5. Hutton, G.: A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4):355-372, July 1999.
6. Gibbons, J., Jones, G.: The Under-appreciated Unfold. Proc. Intl. Conf. Functional Programming, pp. 273-279, Baltimore, Maryland, September 27-29, 1998.
7. Kiselyov, O.: Functional XML parsing framework: SAX/DOM and SXML parsers with support for XML Namespaces and validation. Version 4.9, September 5, 2001. http://pobox.com/~oleg/ftp/Scheme/xml.html#XML-parser
8. Lindig, C.: Tony - a XML Parser and Pretty Printer. Version 0.8. http://www.gaertner.de/~lindig/software/tony.html
9. van Mourik, H.: XML parser in Scheme. Version: 0.9.23, June 2, 1998. http://student.twi.tudelft.nl/~tw585306/
10. Neumann, A.: The Functional XML Parser. Version 1.4.4, October 30, 2000. http://WWW.Informatik.Uni-Trier.DE/~aberlea/Fxp/
11. Sheard T., Fegaras, L.: A fold for all seasons. Proc. Conf. on Functional Programming and Computer Architecture (FPCA'93), pp. 233-242, Copenhagen, Denmark, June 1993.
12. Wallace M., Runciman, C.: HaXml. Version 1.02 release, May 3, 2001. http://www.cs.york.ac.uk/fp/HaXml/
13. World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation October 6, 2000. http://www.w3.org/TR/REC-xml
14. World Wide Web Consortium. XML Information Set. W3C Candidate Recommendation. May 14, 2001. http://www.w3.org/TR/xml-infoset