

Fxt - A Transformation Tool for XML Documents

Alexandru **Berlea** <aberlea@psi.uni-trier.de>

Helmut **Seidl** <seidl@psi.uni-trier.de>

Abstract

XML document processing is a subarea of tree processing for which the functional programming style is very natural. A pattern matcher is necessary for identifying parts of the tree to be processed. The functional style implies a processing model in which navigation is possible only to subtrees of a tree. This restriction can be compensated by using a tree pattern matcher able to relate to ancestors as well as to siblings of a match. On top of the powerful *fxgrep* XML pattern matcher, we build *fxt*, a transformation tool for XML documents. The functional processing model that *fxt* uses, allows an implementation much more efficient than implementations permitted by the processing model of the popular XSLT, where navigation in the input tree can proceed in arbitrary directions. The *fxt* transformations are specified in an intuitive, declarative way. Flexibility is provided by hooks to the full functionality of the *SML* programming language, as well as by the *fxt*'s variable mechanism.

1. Introduction

From the very beginning, document processing of hierarchical documents has been attracted by the functional programming style of declarative specifications. So, SGML- as well as XML- syntax quite heavily resembles Lisp expressions. Also, the document querying and specification language DSSSL [ISO96] originally has been designed as a superset of Lisp: partly because there was an ISO standard of this language, but mainly because of the need of specifying transformations of documents in a way which can be understood without referring to some operational semantics.

This goal, however, was only partly achieved. In the end, DSSSL has a very complicated semantics which can not be easily understood independently of the operational behavior of a DSSSL-processor. The same holds true for the successor transformation language XSLT [W3C99a] (see [Wad99] for an effort of formally defining the semantics of patterns in XSLT). In particular, XSLT has the following drawbacks:

- The document model: XSLT does not have a "functional" view onto the document. Instead of a structured term, the document is conceptually viewed as a collection of linked nodes -- implying that arbitrary navigation in this graph is possible. Such free navigation makes it difficult to reason about meaning independently of the order of the relative navigation steps.

- The pattern language: On the one hand side, it is very weak as it basically identifies nodes in the document tree by only specifying tree relationships among them. XSLT uses for pattern matching the operational model of XPath [[W3C99b](#)] which is based on successive selecting and filtering tree nodes. The matching of a pattern may require as many traversal of the input as the number of steps in the pattern. On the other hand, however, it is also overly strong, allowing arbitrary tests to be performed on the selected set of nodes, and providing such features like indexing of matches and arbitrary navigation in the document.

In this paper we take a different approach of XML transformation. Our three main design goals are:

- to provide an as declarative specification of the intended transformation as possible;
- to provide only primitives, in particular for pattern matching, which can be implemented efficiently;
- to allow maximal flexibility by fully integrating an external programming interface.

We build on the modern statically typed functional programming language Standard ML of New Jersey ([SML](#)). The choice of the implementation language is justified by the fact that [SML](#) is known to be good at tree processing in general, which is to what XML processing mainly reduces, as XML documents are conceptually textual representations of trees. Furthermore, [SML](#) already provides support for at least basic pattern matching. We define Functional Document Model ([FDM](#)) -- a fully functional document model. Based on that, we critically revise the rule-based approach of XSLT. In particular, we restrict navigation to accesses of sub-documents only. In order to compensate for that, we provide a flexible variable mechanism to collect arbitrary intermediate values. At the same time, we enhance the pattern matching capability by employing Functional XML Grep ([fxgrep](#)). [fxgrep](#) [[Neu00](#)] [[Fxx01](#)] is an XML querying tool written in [SML](#). [fxgrep](#) patterns are similar to XSLT patterns (excluding the operators for selecting nodes other than descendants) but additionally allow to specify conditions on the structure of a match as well as on its left and right context. Although very expressive, [fxgrep](#) patterns can be implemented by a very efficient automata-based matcher whose complexity is independent of the structure of the patterns. This is in clear contrast at least to current implementations of XSLT pattern matching (see [Section 5](#)) where the run-times heavily depend on the patterns. Finally, building on [FDM](#), we provide a programming interface which allows interactions of transformations with arbitrary [SML](#) code.

2. Processing Model

Functional XML Transformer ([fxt](#)) is a generator of XML transformers. It generates and compiles [SML](#) code for the execution of a specified transformation. The generated transformation code can be subsequently used to transform XML input documents. The intermediately generated [SML](#) code can be also used separately, e.g. in other [SML](#) programs.

The processing model of the XML transformations that we consider in this paper is that of "recursive transformers". A recursive transformer is specified by a set of rules. Each rule consists

of a *match pattern*, identifying sub-documents from the input to which the rule apply and a corresponding action specifying how to transform these sub-documents. An action constructs a piece of XML content, typically by using parts of the matching sub-document and by selecting further sub-documents from the input and recursively applying the transformation on them. The sub-documents for recursive processing may be selected using *select patterns* in the context of the sub-document on which the transformation is being currently applied (the current sub-document).

The result of the transformation is given by executing the action associated with the first match pattern in the specification which is matched by the root of the input document. When the transformation is to be applied on sub-documents selected for recursive processing, the selected sub-documents are processed according to the match pattern that they match in the initial input tree. Note thus, that match patterns, in contrast to select patterns, always refer to the initial input tree. One can think of such a transformation as consisting of two phases. In the first phase, the pattern matching phase, a pattern matcher annotates each node of the input tree with the corresponding matched patterns. In the second phase, the transformation phase, the annotations are used as a guide as for what actions are to be taken.

The processing model described so far is the model that both XSLT and [fxt](#) use. The particular processing model of a recursive transformer is mainly given by the kind of match and select patterns it provides. Here are important differences between XSLT and [fxt](#). While in XSLT the trees that an action may select for further processing may be ancestors, siblings or descendants of the current node, [fxt](#) potentially selects only descendants of the current node. An [fxt](#) transformation proceeds thus strictly top-down over the hierarchical structure of the input document. The need to relate some node to its ancestors, or to previous or following nodes in the input document is fulfilled in [fxt](#) by the expressivity of its match and select patterns. Powerful contextual conditions, including regular conditions to be fulfilled by the path to a match, as well as regular conditions on the siblings of the match or of the nodes lying on the path to the match, can be specified. Besides, a strictly top-down recursive transformation has the advantage of being clearer and more efficient.

As an example of a recursive transformation, consider the following specification of an [fxt](#) transformation, which given an XML document, produces a list of titles of the sections in the document:

```
<fxt:spec>
  <fxt:pat> /* </fxt:pat>
    <ul>
      <fxt:apply/>
    </ul>

  <fxt:pat> //section/title/" "</fxt:pat>
    <li>
      <fxt:current/>
    </li>

  <fxt:pat> default </fxt:pat>
    <fxt:apply/>
```

```
</fxt:spec>
```

An [fxt](#) transformation is specified by an XML document having `fxt:spec` as document element type. The `fxt:spec` element can contain one or more `fxt:pat` elements, each of them specifying a match pattern as its content. A `pattern default` can be used for matching any sub-document in an XML document.

Every `fxt:pat` element is followed by a sequence of [fxt](#) actions specifying the result to be produced by the transformation for the sub-documents of the XML input document which match the pattern. The sequence of actions consists of the sequence of sub-documents between the triggering `fxt:pat` element and the next `fxt:pat` element or the end of the specification. The result of an [fxt](#) action is a sequence of XML sub-documents. The result of a sequence of [fxt](#) actions is the concatenation of the results of the [fxt](#) actions in the sequence. A more formal description of the syntax of the specification of an [fxt](#) transformation can be found in [Section 8](#) in form of a pseudo DTD.

The first pattern in the example above, `/*`, matches the topmost element of the document to be transformed. The corresponding action specifies that the result must be an element of type `ul`, whose content is given by the `fxt:apply` action. The result of this action is the sequence of sub-documents obtained by applying the transformation recursively to the content of the current sub-document (which is here the topmost element). The second rule says `,` that whenever text is found inside the `title` element of a `section`, a new `li` element should be created whose content is the matched text. The currently matched sub-document (which is here the matched text) is returned by the action `fxt:current`. The rule for the default pattern says, that the transformation should otherwise simply proceed to the sequence of sub-documents in the content of the current sub-document.

3. Pattern Matching

A recursive transformer must use a pattern language to express its match and select patterns. While XSLT uses XPath, [fxt](#) uses [fxgrep](#) as its pattern language. The capabilities of [fxgrep](#) and implementation details may be found in [[Neu00](#)]. The syntax of [fxgrep](#) patterns is similar to the abbreviated syntax of XPath.

The [fxgrep](#) pattern matching is based on a regular engine. This allows [fxgrep](#) patterns used in [fxt](#) to specify powerful conditions on the nodes that lie on the path from the root of the considered tree to the target:

- *Structural conditions* for a node may be specified as regular conditions over tree patterns. The children of a node to which a structural constraint refers must then be such that they fulfill the regular condition. Structural conditions are given between brackets following the node to which they refer. For example, the pattern `a[b*c[d*]]` is fulfilled by an `a` element that has one or more `b` children followed by a `c` child that itself has only `d` children.
- *Contextual conditions*

- Vertical contextual conditions can be used to specify properties of paths in document trees as regular expressions. For example, $(a/)^+b$ identifies a b node, where each ancestors (at least one) is an a node.
- Horizontal contextual conditions may be also specified as regular expressions over the siblings of a node. A contextual constraint consists of two regular conditions ℓ and r , given as $[\ell\#r]$ following a node pattern. Suppose that the node pattern is followed by a tree pattern. Then the child of the node that matches the node pattern in which the matches of the following tree pattern are found must be such that its left siblings structurally match ℓ and its right siblings structurally match r . Here are a few examples that illustrate the use of contextual conditions:
 - $b[c\#d^*]/a$ matches in a tree with type b the a child whose left siblings are all of type c and right siblings are all of type d .
 - $//*[\#_]/a$ matches all the a elements that are the first child of their fathers.
 - $//*[b\#]/a$ matches a elements that are the last child of their fathers if preceded only by b siblings.

Despite their similar syntax, the operational models of XPath and [fxgrep](#) are completely different. [fxgrep](#) locates matches in at most two passes. In the first pass, a right to left traversal of the input tree determines candidates for matches as nodes where structure and right context match. In the second pass, a left to right traversals identifies from the candidates, those matches for which the left context also matches.

In contrast, in XPath, matches are located in a number of successive steps. Each such location step selects in turn nodes which find themselves in a specified relationship with the nodes selected by the previous step. The nodes selected by a location step may be subsequently filtered via predicates using arbitrary XPath expressions.

While in [fxt](#) the select patterns can only select parts of the current tree, in XSLT they can also relate to siblings and ancestors. It is arguable however if this is necessarily needed. Here it should be noted that the functionality of all examples in the XSLT specification can be achieved by equivalent [fxt](#) transformations.

Selecting nodes that are not descendants of the current node for recursive processing may result in non-terminating loops. Moreover, we suspect that this may render the specification of a transformation unclear. More important maybe than the arguable expressiveness is the possibility of an efficient implementation. The processing model of XPath suggests to use as many traversals as the size of a pattern to find the matching patterns. The regular expression engine of [fxgrep](#) guarantees that the matches are found in at most two traversals.

Consider the following XML document:

```
<document>
  <title>Sections</title>
  <section>
```

```

<title>Section One</title>
<content>Here is section 1...</content>
</section>
<section>
  <title>Section Two</title>
  <content>Here is section 2...</content>
</section>
</document>

```

The `fx` internal tree representation of the input document is depicted in Figure 1.

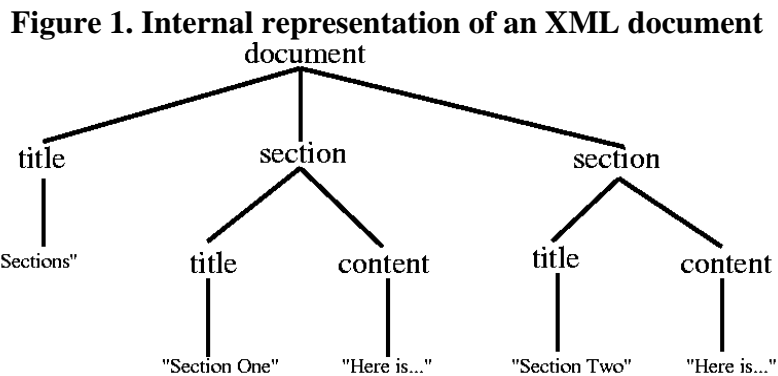
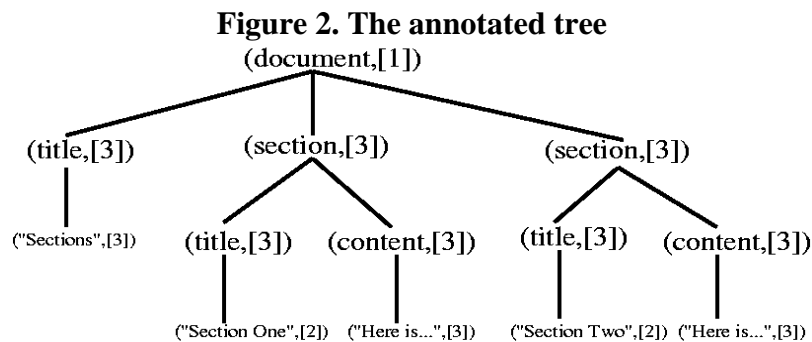


Figure 2 depicts the input tree annotated by the pattern matching phase of the `fx` transformation from the previous section. Each node is annotated with the list of patterns that it has matched. 1, 2, 3 refer to the first, second and third pattern, respectively. The root of the tree for example has only matched the first pattern. The text content of a `title` within a `section` matches only the second pattern. Nodes that have matched neither the first nor the second pattern are annotated with a list containing 3, the number of the default pattern.



4. Transforming

In the transformation phase itself, the transformation proceeds top-down over the annotated input document. At a sub-document in the hierarchy, the sequence of actions corresponding to the first matched pattern is executed.

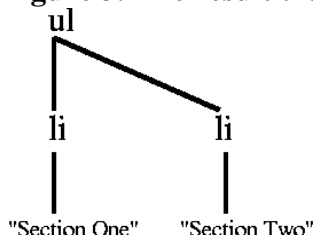
The XML syntax allows for a natural way of specifying parts of the output, particularly when these are simple pieces of XML content. Elements not prefixed by `fx`: appearing in actions

are exactly reproduced in the output of the transformation. Elements prefixed by `fx` have semantics defined by `fxt` as for example copying parts of the input, recursive application or generating XML nodes.

Consider the transformation and the XML input document from the previous sections. The transformation proceeds over the tree annotated in the pattern matching phase depicted in [Figure 2](#) as it follows. It starts at the root node which has matched the first pattern. The corresponding action creates an `ul` element whose content is given by the `fxt:apply` action. `fxt:apply` concatenates the results of recursively applying the transformation on the children of the current tree. As the children of the root node have all matched the default pattern, the application of the transformation on each of them recursively descend to the children if any available, as specified by the `fxt:apply` action for the default pattern. When the transformation arrives at the text node of a title within a section, matching the second pattern in the specification, the transformation returns an `li` element whose content is the current text. The result tree of the transformation is depicted in [Figure 3](#).

The recursive application of a transformation can proceed to the children of the current node, as with the `fxt:apply` element above. Furthermore, the nodes to be further processed can be as well explicitly selected using an `fxgrep` pattern.

Figure 3. The result tree



4.1. The Functional Document Model

`fxt` provides means by which `SML` code can be embedded in the generated transformation. `SML`, however, offers no special support for XML processing. A collection of `SML` types and functions for manipulating XML documents is provided in `fxt` via `FDM` (the Functional Document Model) a functional-styled library.

XML documents can contain any legal Unicode character. `SML` supports only 8-bit characters and has no notion of Unicode. Therefore a Unicode library is provided containing types for the Unicode characters and strings, along with basic functions for manipulating them, as well as conversion functions from and to `SML` strings. The `FDM` makes use of the Unicode library for representing XML strings.

The `FDM` is specified as an `SML` signature, the `SML` way of providing controlled access over a collection of types, values and functions. The types `Tree` and `Forest` are provided as abstractions of XML sub-documents and of sequences of XML sub-documents, respectively. Constituting parts of a document tree are XML elements, text and processing instructions. The generated transformation has access to the tree representation via the current node which represents the

sub-document being currently transformed and which can be referred to as `current`. Functions are provided for testing the type or content of a node, for accessing its constitutive parts and for constructing different node types. Further functionality is available for transforming a sequence of trees (a forest), as for example for mapping, successive composition (folding), filtering, sorting or outputting trees or forests. The functional concept of higher-order functions makes it possible to elegantly obtain complex processing from combining basic functions provided by the [FDM](#).

Consider the following excerpt from the [FDM](#) interface:

```
signature FDM =
  sig
    type Tree
    type Forest = Tree vector

    ...

    val hasElementType : Unicode.Vector -> Tree -> bool
    val hasTextContent : Tree -> bool
    val filterFirst : (Tree -> bool) -> Forest -> Tree
  end
```

`filterFirst` is a function which expects a predicate over trees as its first argument and a forest as the second argument. It returns the first tree in the forest satisfying the predicate.

Consider the call:

```
filterFirst hasTextContent
```

where `hasTextContent` tests whether a node has plain text content. Then a function is returned which takes a forest and returns the first tree in the forest having only text content.

In the call:

```
filterFirst (hasElementType (String2Vector "alfa"))
```

`(String2Vector "alfa")` returns the Unicode string `alfa`. The application of `hasElementType` on this Unicode string returns a predicate testing whether a tree has the specified type. The application of `filterFirst` returns thus a function which takes a forest and returns the first tree in the forest having element type `alfa`.

XML sub-documents can be generated using [SML](#) code which is provided as [SML](#)-values within attributes of some [fxt](#) actions. Such actions include creation of XML text, processing instructions and elements as well as actions for attribute manipulation.

The following is the specification of a transformation that replaces every element name containing more than six characters by its first six characters:

```
<fxt:spec>
  <fxt:pat>/**</fxt:pat>
  <fxt:tag
```

```

nameExp= '
  let
    val name = getElementType current
  in
    if Vector.length name > 6 then
      Vector.extract (name,0,5)
    else
      name
    end' >
<fxt:apply/>
</fxt:tag>
</fxt:spec>

```

The `fxt:tag` outputs an element whose name is given as the value of an attribute `nameExp`, which must be an [SML](#) expression evaluating to a Unicode string. `getElementType current` returns the name of the current element. The content of the output element is given by the content of the `fxt:tag` element. If, as above, a `default` pattern is not specified, a default action is considered added for trees which do not match any of the specified patterns. This default action copies the root of the current tree and recursively applies the transformation on its children, or outputs them if they are leaf-nodes.

4.2. Variables

Elaborate transformations can be generated, by specifying different rules for different sub-documents and by using recursion. However, by this mechanism, at every step, the transformations only sees the tree rooted at the current node. In order to deal at some point during a transformation with information which is located at nodes that have been processed earlier we introduce [fxt](#) variables.

[fxt](#) variables have global visibility. [fxt](#) does not provide local variables, e.g. for processing several nested occurrences of some element type. Instead, global variables are organized as stacks meaning that we can perform push and pop operations on variables. A push can be seen as introducing a variable scope, whereas a pop means that the scope of the local variable is left. We decided for stacked globals instead of local variables as this enhances flexibility in scoping.

The variables may be of any [SML](#) built-in or user-defined type and also of one of the [FDM](#) types `Tree` or `Forest`. [fxt](#) actions are provided for pushing, popping, setting or outputting the values on the stacks. Special actions are provided for the comfortable use of `Tree` and `Forest` variables.

The following specification generates a transformation that given an XML document containing imbricated `li` elements (list items), adds before every `li` an integer representing the number of the list item on its imbrication level:

```

<fxt:spec>
  <fxt:global name="i" type="int"/>
  <fxt:push name="i" val="0"/>

  <fxt:pat>//li</fxt:pat>
  <fxt:get name="i"/>:

```

```

    <fxt:copyTag>
      <fxt:set name="i" val='(get i)+1' />
      <fxt:push name="i" val="0" />
      <fxt:apply />
      <fxt:pop name="i" />
    </fxt:copyTag>

  </fxt:spec>

```

A global stack `i` containing integers is declared and a zero is pushed onto the stack. Whenever an `li` element is transformed, the value on top of the stack is output using `fxt:get`. The `li` tag is copied, and before proceeding with the transformation to the sons, the value on top of the stack, representing the counter for the current imbrication level is incremented and a new value of zero is pushed onto the stack for the next imbrication level. The value is popped off the stack after the sons are transformed.

For a comparison, XSLT variables are lexically scoped, can have either global or local visibility and can be bound to values of any type returned by XSLT expressions. They never change their value, being essentially named constants which may enhance the readability of the code. The XSLT variables are thus not used as means of doing calculations. Calculations can be however done via parameters with which recursive application of transformations may be provided. This kind of calculation is however less flexible than that provided by the `fxt`'s variables.

4.3. Keys

Some XML documents, like for example an XML representation of a graph, have an inherent cross-reference structure that can not be represented directly by the XML document tree. Attributes of type `ID` or `IDREF` may be used in XML in order to provide an XML document with some cross-reference structure. To deal with such documents, `fxt` provides keys, which are a generalization of XML ID-s. The key mechanism allows to collect sub-documents from the initial document in a table which can be accessed via keys.

Consider the following XML input document:

```

<graph>
  <node id="1">Trier</node>
  <node id="2">Bonn</node>
  <edge from="1" to="2" />
</graph>

```

The following specifies a transformation which lists the connections between the cities:

```

<fxt:spec>
  <fxt:key name='cities' select='//node' key='id' />

  <fxt:pat> //edge </fxt:pat>
  There is a way from:
    <fxt:copyKey name='cities' key='from' />
  to:
    <fxt:copyKey name='cities' key='to' />

```

```
<fxt:pat>default</fxt:pat>
  <fxt:apply/>
</fxt:spec>
```

A table named `cities` is declared, where elements of type `node` residing everywhere in the document are stored with keys given by their `id` attribute value. Anytime an `edge` is seen the source and the destination are retrieved from the table, using as key the value of the attribute `from` and `to` respectively. The document above is transformed to:

```
There is a way from:
  <node id='1'>Trier</node>
to:
  <node id='2'>Bonn</node>
```

When storing or retrieving, values for keys are not restricted to attribute values. They can be also computed by arbitrary [SML](#) expressions using the tree being stored or the tree being currently transformed, respectively.

[fxt](#) offers the possibility to apply the transformation to the sub-documents associated with a key. Furthermore, it is possible to store instead of a whole tree, some arbitrary Unicode string typically obtained from processing that tree.

Finally, let us note that [fxt](#) offers a number of further features for:

- conditional processing, allowing to specify that a sequence of actions is to be considered only if some condition is fulfilled.
- attribute insertion, deletion and replacement.
- using [SML](#) code for inserting processing instructions with computed content as well as arbitrary Unicode content.
- sorting and filtering of forests generated during the transformation.
- using command line arguments within transformations.

For details we refer the interested reader to the online documentation <http://www.informatik.uni-trier.de/~aberlea/Fxt/> [[Fxt01](#)].

5. Comparing with XSLT Implementations

In order to assess the time performance of [fxt](#) we compared it with two XSLT processors. We chose the Xalan processors which are part of the Apache XML Project [[Xal01](#)]. These were the Xalan Java 2 and Xalan C++ 1.2 implementations.

We considered the following benchmark transformations:

1. Birds

- XML Input (10 KB): Description of classes of birds
 - Output: Plain text file presenting the information in the input in an indented manner
2. GCA Paper
 - XML Input (60 KB): The current paper conforming to the DTD for GCA XML Conference Proceedings
 - Output: An HTML layout of the paper
 3. Lines
 - XML Input (200 KB): Shakespeare's "All's Well That Ends Well" play [[Bos99](#)]
 - Output: The collection of all the lines in the play
 4. Baseball
 - XML Input (600 KB): Baseball statistics [[Har99](#)]
 - Output: HTML tables containing processed information about baseball players.
 5. T1 and T2
 - XML Input (200 KB): Shakespeare's play
 - Output: The collection of matches of complex patterns

The benchmarks were executed under Linux (Kernel 2.4.9) on a AMD Athlon 800 MHz processor. The JVM used to run the benchmarks with the Java XSLT implementation was the Sun JVM implementation Java version 1.3.

The times listed are measured for a single run of the corresponding transformation. The total times for [fxt](#) include thus the time needed to generate and compile [SML](#) code to achieve the transformation. The total times for Xalan also take into account time needed for processing the stylesheets. It is likely that a stylesheet source is used to transform multiple XML sources. In this case the stylesheet need to be processed only once before the first transformation. It is therefore sensible to individually list the time taken by the processing of the stylesheets.

As the time spent for parsing the XML input is significant, it is individually listed in the results. [fxt](#) uses the Functional XML Parser ([fxp](#)) XML parser, written also in [SML](#) [[Fxp01](#)]. [fxp](#) proved to be faster then the xml4j Java XML parser from IBM [[IBM99](#)] for smaller documents and up to twice time slower as the input document size increases [[Neu00](#)]. The same relation of [fxp](#) and the Java XML parser of Xalan can also be observed from our results.

Furthermore, times for startup for the JVM and for the [SML](#) runtime-system were also not considered.

The patterns used in the first four transformations are very simple. For them `fxt` proved to be in general faster than Xalan Java and able to keep up with Xalan C++. [Table 1](#) shows the results for the first four transformations.

Table 1. Transformation times (seconds)

Application	Size	Transformer	Parsing	Processing stylesheet	Transforming	Total
Birds	10 KB	Xalan C++	0	0.010	0.010	0.020
		fxt	0.022	0.578	0.006	0.692
		Xalan Java	0.144	1.233	0.416	2.310
GCA Paper	60 KB	Xalan C++	0.120	0.030	0.050	0.200
		fxt	0.120	2.559	0.200	3.186
		Xalan Java	1.250	1.358	0.691	3.985
Lines	209 KB	Xalan C++	0.200	0.010	0.140	0.350
		fxt	0.469	0.479	0.191	1.486
		Xalan Java	0.738	1.202	1.319	3.917
Baseball	655 KB	Xalan C++	0.760	0.030	0.870	1.660
		fxt	1.699	2.758	1.469	7.266
		Xalan Java	0.928	1.326	3.060	6.299

T1 and T2 contain more elaborate but completely meaningful patterns. A play is a sequence of `acts`, each of them containing a sequence of `scenes`. A `scene` has a sequence of `speeches`, each of them containing a `speaker` and a sequence of `lines` containing plain text. T1 collects all the lines in the speeches of `Lafeu` appearing in scenes where `Bertram` is also present. Given the DTD for Shakespeare plays we have used, these lines are matched by the XPath pattern:

```
SCENE[ //SPEAKER="BERTRAM" ] /
  SPEECH[ SPEAKER="LAFEU" ] /LINE
```

The corresponding `fxgrep` pattern is:

```
//SCENE[_ ( //SPEAKER/ "BERTRAM" ) _] /
  SPEECH[_ ( SPEAKER/ "LAFEU" ) _] /LINE/ " "
```

T2 collects all the speeches in scenes containing a line having the word "husband" and being in an act containing a line having the word "abundance". The XPath and the functionally equivalent `fxgrep` patterns that were used are:

```
ACT[ //LINE[ contains( ., "abundance" ) ] ] /
  SCENE[ //LINE[ contains( ., "husband" ) ] ] /SPEECH
```

and respectively:

```
//ACT[_ ( //LINE/ "abundance" ) _] /
  SCENE[_ ( //LINE/ "husband" ) _] /SPEECH
```

We suppose that the processing model in XPath imposes for these transformation multiple tree traversals in the XSLT implementations incurring high processing time. The processing times for T1 and T2 are presented in Table 2. Considering the times spent for transforming, [fxt](#), performs significantly faster even than the C++ XSLT implementation.

Table 2. Transformation times for more elaborate patterns(seconds)

Application	Transformer	Parsing	Processing stylesheet	Transforming	Total
T1	Xalan C++	0.200	0.010	0.780	0.990
	fxt	0.427	0.480	0.271	1.376
	Xalan Java	0.774	1.210	1.326	3.812
T2	Xalan C++	0.200	0.010	7.500	7.710
	fxt	0.432	0.475	0.365	1.485
	Xalan Java	0.767	1.211	5.657	8.131

For T1 and T2 we also considered the dependency of the transformation time on the size of the input document. The input document was augmented by duplicating the `ACTS` of the play, which is doubling the breadth of the input tree. The expected effect on both the [fxt](#) and the XSLT transformations is that of doubling of the transformations steps. The size-time dependency showed to be indeed linear.

6. Related Systems

XDuce [[HP00](#)] is a statically typed language for tree transformation specialized on XML processing. Values in XDuce are XML documents and can be specified either in the XML syntax or in a native syntax. XDuce extends the type system of a language as ML with the possibility of defining regular expression types, as a way of describing structure in XML documents. The ML pattern matching is correspondingly extended to allow for matching values of regular expression types. Beyond the ML pattern matching, this allows for matching sequences of trees of variable lengths. Besides this, however, there is no further support for specifying the context and the structure of a match. It is also for example not possible to match trees which found themselves at a variable depth, as permitted by the `//` specifier in [fxgrep](#) and XPath.

HaXML [[WR99](#)] presents two approaches of using modern functional languages in XML processing. The first one uses a generic structure of XML documents as base for the design of a library of combinators for generic processing, including selection, generation and transformation of trees. The selection of matching nodes is achieved by a filtering procedure consisting of composing a sequence of filters, being thus similar with the processing model of XPath. For selecting nodes for recursive processing a series of combinators are introduced allowing for selecting the topmost matching elements, the bottom-most or all the matches. The syntax proposed is mainly the Haskell syntax which is not well-suited for describing XML fragments. In the second approach DTDs are translated to data types in functional languages. The advantage is that the static type-checking of the language can be then used for the validation of XML documents. This approach can be taken only when DTDs of the input and the output are known and incurs costs for translating DTDs into the data-types by which these are represented. Fur-

thermore, no generic support for XML processing can be provided in this approach. A language using the DTDs-to-types approach is XMLambda [MS00], a small functional language which has XML documents as its basic data-types.

Term rewriting languages are another area of tree processing, usually applied in program transformation. Programs are represented as terms, built from variables, constant and function symbols. A transformation must specify a number of rules. The form of the rules is similar with that of fxt or XSLT rules. The left hand side is a term pattern, while the right hand side is a term template. The template is instantiated when the rule is applied on a term matching the left hand side. The application of the rule is controlled by a strategy. Ordinary term rewriting languages use a standard, fixed strategy. In term rewriting languages like ELAN [PBHKR00] or Stratego [Vis99a] [Vis99b] [Vis01] the user can provide his own strategies. In Stratego, a strategy is an operation that transforms a term into another term or fails. Basic building blocks in strategies are matching terms, building terms and variable bindings. More complex strategies can be obtained by using strategy operators. These can be divided into operators for sequential programming and operators for term traversals. Rules are abbreviations that allow to conveniently specify basic strategies. The separation of matching and construction of terms from the building of scopes for variable bindings allows for a pattern matching more expressive than that of functional programming languages like ML (where pattern matching is done by simultaneously recognizing structure and binding variables to sub-terms). For example, a pattern match can be passed on to a local strategy to match sub-terms at a variable depth in the subject term. Another feature of Stratego is the possibility of expressing patterns that describe recursive structure. On the one hand, the pattern language of fxgrep does not provide variable bindings in patterns, and can not express recursive patterns. On the other, matching at an arbitrary depth, via the operator // is only one of the contextual condition that can be expressed in fxgrep. There is furthermore no special support for XML processing.

7. Conclusions

In this paper we have presented fxt, a transformation language for XML documents. fxt combines the computational power of the SML functional language with the expressivity and the efficiency of the pattern matching provided by fxgrep.

To our experience, the expressivity of our pattern language together with the additional features of fxt such as stackable variables and a flexible programming interface more than compensate for fxt's restrictions on navigation through the input document.

This restriction, however, allowed us to provide an elegant and understandable specification language for document transformation whose implementation is more efficient than comparable implementations of XSLT.

For further information and updates we refer the interested reader to the fxt's online documentation <http://www.informatik.uni-trier.de/~aberlea/Fxt/> [Fxt01].

8. Appendix: syntax of an fxt specification

```
<!ENTITY % ACTION
"#PCDATA
|fxt:addAttribute|fxt:apply|fxt:applyKey|fxt:attribute
|fxt:copyAttributes|fxt:copyContent|fxt:copyKey|fxt:copyTag
|fxt:copyTagAddAttribute|fxt:copyTagApply|fxt:copyTagDeleteAttribute
|fxt:copyTagReplaceAttribute|fxt:copyType|fxt:cr|fxt:current
|fxt:currentText|fxt:deleteAttribute|fxt:getTableItems|fxt:ht
|fxt:if|fxt:literate|fxt:pi|fxt:pop|fxt:push|fxt:pushForest
|fxt:replaceAttribute|fxt:set|fxt:setForest|fxt:sml|fxt:sp
|fxt:switch|fxt:tag|fxt:text|ANY">

<!ENTITY % DECLARATION "fxt:arg|fxt:global|fxt:key|fxt:open|fxt:push
|fxt:pop|fxt:set|fxt:table">

<!ENTITY % RULE "fxt:pat, (%ACTION;)*">

<!ELEMENT fxt:spec ((%DECLARATION;)*,(%RULE;)*>
<!ELEMENT fxt:pat (#PCDATA)>
<!ELEMENT
(fxt:addAttribute|fxt:apply|fxt:applyKey|fxt:arg
|fxt:attribute|fxt:copyAttributes|fxt:copyContent
|fxt:copyKey|fxt:copyTagApply|fxt:cr|fxt:current
|fxt:currentText|fxt:deleteAttribute|fxt:getTableItems
|fxt:ht|fxt:pi|fxt:pop|fxt:push|fxt:replaceAttribute
|fxt:set|fxt:sml|fxt:sp|fxt:table|fxt:text)
EMPTY>

<!ELEMENT
(fxt:case|fxt:copyTag|fxt:copyTagAddAttribute
|fxt:copyTagDeleteAttribute|fxt:copyTagReplaceAttribute
|fxt:copyType|fxt:default|fxt:if
|fxt:pushForest|fxt:setForest|fxt:tag)
((%ACTION;)*>

<!ELEMENT fxt:literate (#PCDATA)>

<!ELEMENT fxt:switch (fxt:case*,fxt:default)>

<!ATTLIST fxt:addAttribute
(name|nameExp) NMTOKEN #REQUIRED
(val|valExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:apply
test NMTOKEN #IMPLIED
(select|selectExp) NMTOKEN #IMPLIED>
<!ATTLIST fxt:applyKey
name NMTOKEN #REQUIRED
(key|keyExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:arg name NMTOKEN #REQUIRED>
<!ATTLIST fxt:attribute name NMTOKEN #REQUIRED>
<!ATTLIST fxt:case test NMTOKEN #REQUIRED>
<!ATTLIST fxt:copyTagAddAttribute
(name|nameExp) NMTOKEN #REQUIRED
(val|valExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:copyTagDeleteAttribute
(name|nameExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:copyTagReplaceAttribute
(name|nameExp) NMTOKEN #REQUIRED
(val|valExp) NMTOKEN #REQUIRED>
```

```

<!ATTLIST fxt:deleteAttribute
  (name|nameExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:getTableItems
  name NMTOKEN #REQUIRED
  (key|keyExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:global
  name NMTOKEN #REQUIRED
  type NMTOKEN #REQUIRED
  toForest NMTOKEN #IMPLIED>
<!ATTLIST fxt:if test NMTOKEN #REQUIRED>
<!ATTLIST fxt:key
  name NMTOKEN #REQUIRED
  select NMTOKEN #REQUIRED
  (key|keyExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:open
  structure NMTOKEN #REQUIRED
  file NMTOKEN #IMPLIED>
<!ATTLIST fxt:pi
  procesor NMTOKEN #REQUIRED
  data NMTOKEN #REQUIRED>
<!ATTLIST fxt:pop name NMTOKEN #REQUIRED>
<!ATTLIST fxt:push
  name NMTOKEN #REQUIRED
  val NMTOKEN #REQUIRED>
<!ATTLIST fxt:pushForest name NMTOKEN #REQUIRED>
<!ATTLIST fxt:replaceAttribute
  (name|nameExp) NMTOKEN #REQUIRED
  (val|valExp) NMTOKEN #REQUIRED>
<!ATTLIST fxt:set
  name NMTOKEN #REQUIRED
  val NMTOKEN #REQUIRED>
<!ATTLIST fxt:setForest name NMTOKEN #REQUIRED>
<!ATTLIST fxt:sml code NMTOKEN #REQUIRED>
<!ATTLIST fxt:table
  name NMTOKEN #REQUIRED
  select NMTOKEN #REQUIRED
  (key|keyExp) NMTOKEN #REQUIRED
  item NMTOKEN #REQUIRED>
<!ATTLIST fxt:tag nameExp NMTOKEN #REQUIRED>
<!ATTLIST fxt:text code NMTOKEN #REQUIRED>

```

Bibliography

[Bos99] Jon Bosak. *The Complete Plays of Shakespeare, Marked up in XML*. XML Sample Files, 1999. Available online at <http://metalab.unc.edu/xml/examples/shakespeare>.

[Fxp01] Andreas Neumann. *fxgrep 1.4.5*. Source Code, 2001. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxgrep>.

[Fxp01] Andreas Neumann. *fxp 1.4.4*. Source Code, 2001. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxp>

[Fxt01] Alexandru Berlea. *fxt 2.0*. Online Documentation, 2001. Available online at <http://www.informatik.uni-trier.de/~aberlea/Fxt>.

[Har99] Eliotte R. Harold. *1998 Baseball Statistics*. XML Sample Files, 1999. Available online at <http://metalab.unc.edu/xml/examples/1998validstats.xml>.

[HP00] Haruo Hosoya and Benjamin C. Pierce. *XDuce: A Typed XML Processing Language*. In Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pages 226-244, May 2000.

[IBM99] IBM AlphaWorks. *XML Parser for Java*. Software Documentation, August 1999. Available online at <http://www.alphaworks.ibm.com/formula/xml>.

[ISO96] International Organization for Standardization. *Information technology -- Processing Languages -- Document Style Semantics and Specification Language (DSSSL)*. Ref. No. ISO/IEC 10179:1996(E), 1996.

[MS00] Erik Meijer and Mark Shields. *XM : A Functional Language for Constructing and Manipulating XML Documents*. Submitted to USENIX 2000 Technical Conference, 2000.

[Neu00] Andreas Neumann. *Parsing and Querying XML Documents in SML*. Ph.D. Thesis, University of Trier, Trier, 2000. Available online at <http://www.informatik.uni-trier.de/~neumann/Papers/thesis.ps.gz>.

[PBHKR00] Peter Borovansky, Claude and Helene Kirchner, Pierre-Etienne Moreau and Christophe Ringeissen. *An overview of ELAN*. In Electronic Notes in Theoretical Computer Science, volume 15, Elsevier Science Publishers, 2000.

[Vis99a] Eelco Visser. *The Stratego Reference Manual*. Technical report, Institute of Information and Computing Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[Vis99b] Eelco Visser. *Strategic Pattern Matching*. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of Lecture Notes in Computer Science, pages 30-44, Trento, Italy, July 1999. Springer Verlag.

[Vis01] Eelco Visser. *Stratego: A Language for Program Transformation Based on Rewriting Strategies*. To appear in A. Middeldorp (editor), *Rewriting Techniques and Applications (RTA'01)*, Utrecht, The Netherlands. Springer-Verlag, 2001.

[W3C99a] James Clark, editor. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at <http://www.w3.org/TR/xslt>.

[W3C99b] James Clark and Steve DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, World Wide Web Consortium, November 1999. Available online at <http://www.w3.org/TR/xpath>.

[Wad99] Philip Wadler. *A formal semantics of patterns in XSLT*. Markup Technologies, Philadelphia, to appear December 1999. Available online at <http://cm.bell-labs.com/cm/cs/who/wadler/papers/xsl-semantics/xsl-semantics.ps>.

[WR99] Malcolm Wallace and Colin Runciman. *Haskell and XML: Generic Document Processing Combinators vs. Type-Based Translation*. In Proceedings of the International Conference on Functional Programming, Paris, September 1999.

[Xal01] Apache XML Project. *Xalan*. Software Documentation, 2001. Available online at <http://xml.apache.org>

Glossary

FDM	Functional Document Model
fxgrep	Functional XML Grep
fxp	Functional XML Parser
fxt	Functional XML Transformer
SML	Standard ML of New Jersey

Biography

Alexandru **Berlea**

Research Assistant
University of Trier
Department of Computer Science
Trier
Germany
Email: aberlea@psi.uni-trier.de

Alexandru Berlea is a Ph.D. Student working with Helmut Seidl at the University of Trier, Germany. He has graduated in 1999 from the Computer Science and Engineering Department of the Politehnica University of Bucharest, Romania.

Helmut **Seidl**

Professor
University of Trier
Department of Computer Science
Trier
Germany
Email: seidl@psi.uni-trier.de

Helmut Seidl is a Professor with the University of Trier, Germany.