Transforming XML Documents using fxt

Alexandru Berlea and Helmut Seidl

University of Trier, Germany

As XML spreads to various application domains, transformation tasks on XML documents are accomplished by an ever increasing number of non-programmers. In this respect, rather than providing just a collection of basic operations via a library in a special purpose language, it is useful to provide a more intuitive, rule-based approach to XML transformation. The rule-based approach requires pattern-matching for identifying parts of the document to be processed. As XML document processing is basically a subarea of tree processing for which the functional programming style is very natural, we choose SML as implementation language. 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, successors, 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 more efficient than implementations permitted by the processing model of the popular XSLT, where navigation in the input tree can proceed in arbitrary directions. Usual transformations are specified in fxt in an intuitive, declarative way. More elaborate transformations can be flexibly achieved by the hooks provided to the full functionality of the SML programming language, as well as by the fxt's variable mechanism.

Keywords: XML transformations, XML pattern matching with regular expressions, XSLT, fxt, fxgrep, SML, functional document model, functional programming

1. Introduction

From the very beginning, the application domain of processing hierarchically structured 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 [12] originally has been designed as a superset of Lisp. The aim was to specify transformations of documents in a way which can be created and understood also by non-expert users who typically neither know nor care about execution models and implementations.

This goal, however, was only partly achieved. In the end, DSSSL has a very complicated semantics which cannot be easily understood independently of the operational behavior of a DSSSL-processor. The same holds true for the successor transformation language XSLT [30] (see [32] for an effort of formally defining the semantics of patterns in XSLT). In particular, XSLT has the following drawbacks:

- The document model: The XSLT model of a document is a tree, yet this is not the "functional" view of a tree. Instead of a structured term, the document is conceptually viewed as a collection of linked nodes in order to allow the required arbitrary navigation in this graph. 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 only by specifying tree relationships among them. For pattern matching, XSLT uses XPath [31] whose operational model is based on successive selecting and filtering sets of tree nodes. The matching of a pattern may require as many traversals as the number of steps in the pattern. On the other hand, however, it is also overly strong, by allowing arbitrary tests to be performed on the selected set of nodes, and by providing such features as indexing

of matches and arbitrary navigation in the document.

Conceptually, XML documents are textual representations of trees. Thus, XML processing mainly reduces to tree processing. Typically, modern functional programming languages have built-in support for manipulating tree-like data-structures. Several attempts have been made to integrate XML processing into functional programming [16, 33, 10]. A short overview can also be found in [21]. The direction of our proposal is somewhat different. On the one hand, we want to support XML processing for functional programmers. On the other, however, we also want to provide the non-specialist user with a tool by which transformational ideas can be expressed conveniently, i.e., without resorting to a general-purpose programming language.

In this paper we therefore present another transformation tool for XML documents. We build on the the modern, statically typed functional programming language SML [18, 1]. 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.

In the following sections we describe concepts of our tool and critically compare them with the corresponding features of XSLT, the most commonly used XML transformation language.

2. DSL Approach

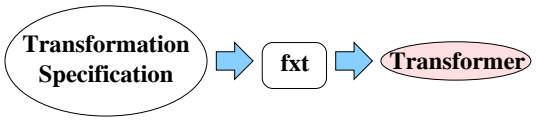

Figure 1: The fxt generator of XML Transformers.

As depicted in Figure 1, fxt (Functional XML Transformer) is in fact a generator of XML transformers. It generates and compiles SML code for the execution of a specified transformation. An fxt stylesheet (the transformation specification) is translated into an SML program which then can either be further used directly by an application programmer or compiled and used as a stand-alone application.

Clearly, this kind of architecture for a domain specific language is not new. In fact, system design and implementation here follows the *pre-processing* paradigm for DSL implementation as sketched in [6]. The advantage of such an approach is obvious: it clearly liberates us from re-implementing compiler support for standard programming language features. In particular,

- we participate in all general enhancements of compiler implementation for free;
- engineering, extending and re-engineering of our prototypical language design could very rapidly be implemented;
- embedding SML code into specifications (where needed) becomes essentially trivial.

The drawback of this approach, however, is also apparent: certain non-syntactical errors are currently catched not in the pre-processing phase, but only in the follow-up compilation phase — where the original source of mal-function is more difficult to track. This problem is partly alleviated in our system by generating comments which points back from the generated code to the corresponding fxt source locations.

Another popular solution to extending functional languages to specific domains is to enrich them with a library of combinators. Combinators are higher-order functions, uniformly defined such that they can be flexible combined with another. A small core of functions are defined in terms of which all the functionality required by the specific domain should be possible to express. A number of combinators libraries exist for XML processing [33], parsing [24], pretty-printing [11], computer music, and many others (see [23] for a discussion on this approach). One advantage of the combinator approach in general is that a number of algebraic laws of combinators can be derived, which could be used for code optimizations. However, even though a set of operators can be defined to improve the readability, the syntax remains limited to the syntax of the implementation language, which in particular is not suitable for describing XML content. Furthermore, writing

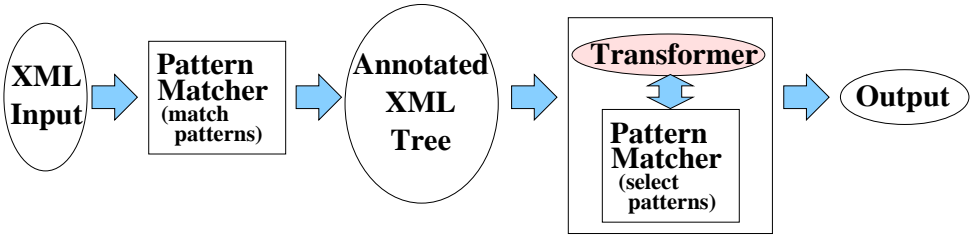

Figure 2: Phases of an XML Transformers.

transformations in terms of combinators leads itself to not especially readable code, making the approach prohibitive for users from outside the functional community. We therefore think that, at least for XML processing, a library of combinators alone is not satisfactory.

3. Processing Model

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* (the *where*), identifying sub-documents from the input to which the rule applies and a corresponding action specifying how to transform these sub-documents (the *what*). 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 pattern*s 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 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 here, that match patterns always refer to the initial input tree. One can therefore 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.

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>
```

Here, the fxt:pat elements contain the match patterns of the rules, whereas the action parts span from their trigger pattern to the following fxt:pat element (or the end). In the given example, the topmost element of every document to be transformed matches the first pattern, i.e. /*. The corresponding action specifies that the result must be an element of type ul, whose content is given by recursively transforming the content of 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 rule for the default pattern says, that otherwise, the transformation should simply proceed to the sub-documents in the content of the current sub-document.

The processing model of recursive transformation as described so far is used both by XSLT and fxt and seems quite natural for basic XML processing applications. The main differences between XSLT and fxt consist in the classes of match and select patterns which are supported.

4. Match Patterns

For matching the trigger patterns of rules, the XML transformation language XSLT relies on the pattern language XPath [31]. The key idea of XPath is to use a directory tree like analogy to document trees and use specifications of *paths* to locate subtrees. So, e.g., a/b/c specifies a path which starts in an a element, proceeds to an immediate descendent element b and from there to the ultimate target element c. Clearly, paths are not very expressive, in particular if the only iteration operator allowing for *deep* matching is "//" (arbitrary descendent). Therefore, in order to enhance expressiveness, XPath also allows non-regular features like checking the number of a match and applying an arbitrary predicate to it.

We take a completely different approach here. In our perspective, non-regular features are difficult to understand (and also difficult to implement efficiently). Therefore, these should be avoided in the pattern language. Thus, we provide support for regular matching only, i.e., for patterns which are related to regular forest languages. Also, we see no point in unnecessarily restricting the patterns to the very weak XPath core language. Instead, we build on the pattern matching facilities as provided by the functional XML querying tool fxgrep [7]. In principle, fxgrep admits *arbitrary regular forest grammars* as patterns [20, 19]. These are implemented by one or two deterministic pushdown forest automata. The key implementation trick then is, not to compute the transition tables of the automata in advance — but only "on demand" as transitions are used during the matching process. Thus, the total number of transitions which are actually computed is at most linear in the size of the input document (in practice, we found that typically only few of all possible transitions are computed). The complexity of pattern matching therefore does not depend on the structural complexity of patterns. Details about the capabilities and the implementation of fxgrep can be found in [19].

We doubt, however, that the power of regular forest grammars could be exploited by a non-expert fxgrep or fxt user. Therefore, fxgrep and hence also fxt provides a pattern language whose syntax is similar in spirit to the abbreviated syntax of XPath but additionally allows more precise specifications of paths as well as of left and right context of paths.

- *Structural conditions* for a node may be specified as regular expressions over tree patterns. The children of a node to which a structural constraint refers must then be such that they fulfill the regular expression. Structural conditions on content 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. Opposed to the simple path expressions of XPath, fxgrep provides full regular expressions for the vertical context. For example, (a/)+b identifies a b node, where each ancestor (at least one) is an a node. This kind of deep matching clearly exceeds the expressiveness of XPath.
  - Horizontal contextual conditions may also be specified as regular expressions over the siblings of a node. A contextual constraint consists of two regular conditions l and r, given as [l#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 horizontal contextual conditions:
    - b[c*#d*]/a matches in a tree with type b the a children such that its left siblings are all of type c and its right siblings are all of type d.
    - //*[#_]/a matches all the a elements that are the first child of their fathers. The $*$ node test is fulfilled by all element types. _ denotes an arbitrary sequence of nodes.
    - //*[b*#]/a matches a elements that are the last child of their fathers if preceded only by b elements. Note that the first star here is the wild-card

node test, while the second is the operator for regular expressions.

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 the operational model of 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. For example a[@b="x"]//c is (conceptually) evaluated as follows. First, all a elements are selected. From these, the [@b="x"] predicate filters those having a b attribute with value x. // selects further all the descendants of the remaining a elements. Finally, the nodes with type c are filtered from the nodes selected by the previous step.

5. Select Patterns

While processing a node from the input document, a recursive transformation can produce XML content by recursively transforming a set of selected nodes. These nodes are selected through a pattern matching process — which, however, now does not (necessarily) start at the root node of the input document but determines the matches relative to the node being currently processed. The patterns for this "dynamic" matching process are called *select patterns*. For selection, the transformation language XSLT again relies on XPath patterns. Beyond the navigation operators, however, which can be used in match patterns, select patterns also may go *upward* in the document tree to ancestors of the current node.

In our perspective such *arbitrary* navigation in the document is likely to cause confusion about the intended meaning of a transformation and error-prone as the user easily gets conceptually lost. It is for this reason that fxt allows exactly the same patterns for selection as for matching — implying that every fxt transformation proceeds strictly top-down over the hierarchical structure of the input document.

There are, however, transformations which cannot be achieved by simply proceeding top-down and exclusively selecting descendants of the current node for recursive processing. Consider, for example, an input document in which consecutive repetitions of the same XML content are avoided by using a special place-holder denoting previously defined content. Now, a transformation should process the XML input and replace the definition as well as the place-holder elements by the result of processing this XML content. Suppose the XML content whose repetition is avoided is enclosed in an element def whereas the place-holder is the element lastdef. In XSLT, such a transformation could be implemented by using selection of non-descendants nodes (in this case the next preceding def element) as depicted in the transformation below.

This kind of backtracking obviously cannot be achieved by using the fxt selection mechanism. It turns out, however, that this transformation

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:template match="def">
    <DEF><xsl:apply-templates/></DEF>
  </xsl:template>

  <xsl:template match="lastdef">
    <xsl:apply-templates select="preceding::def[1]"/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy><xsl:apply-templates/></xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

can still be achieved in fxt through variables and even — at least to our taste — much more elegantly and efficiently.

6. The Transformation Language

An fxt transformation itself is specified by an XML document, as seen in the example in Section 3. The fxt specification must have fxt:spec as document element type and essentially consists of a sequence of rules. Each rule starts with an fxt:pat element containing the triggering match pattern which is followed by the corresponding sequence of actions.[1] Patterns must be specified in the pattern language of fxgrep. A pattern default can be used for matching any sub-document in an XML document. If the current sub-document matches the specified pattern of a rule, the corresponding action should be taken. In case patterns of more than one rule match, the earlier defined rule takes precedence. Here, we deviate from the XSLT philosophy where the "most specific" pattern succeeds. In our opinion, our rule is not only much easier to implement but also much easier to understand.

When designing a specification language for the action part, we follow the approach chosen in XSLT as well as in XMλ [16]: by default, elements together with their attributes constitute parts of the output. Calls to transformation primitives as well as potentially embedded foreign language code must be escaped. In our case, we use the prefix fxt: for identifying fxt actions. Basic actions are provided by fxt for:

- Copying parts from the matching sub-document like element tags (fxt:copyTag), attributes (fxt:copyAttributes) or content (fxt:copyContent);
- Inserting new XML content like element tags (fxt:tag), attributes (fxt:attribute, fxt:replaceAttribute, etc.), text (fxt:text), and others. The output to be produced by these elements is specified as values for special attributes;
- Recursive application of the transformation: the element fxt:apply outputs the sequence of sub-documents obtained by applying the transformation recursively to the children of the current node. The nodes to be further processed can be explicitly selected through an fxgrep pattern given as value for a select attribute.

Consider the transformation from Section 3 with the list of section titles from an XML document. When running on the input 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 transformation proceeds as follows. It starts at the root node which has matched the trigger pattern of the first rule. 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 descends 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 text being currently matched. The result tree of the transformation is depicted in Figure 3.

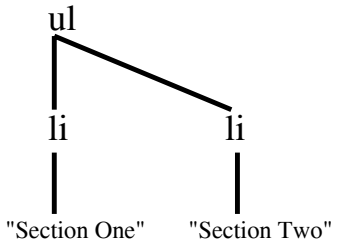

Figure 3: The result tree.

The XML syntax of the fxt specification language allows for a natural way of expressing the output of a transformation. The nesting structure of the output as well as the XML elements which appear literally in the output can both be directly recognized in the fxt specification.

[1] A more formal description of the syntax of the specification of an fxt transformation can be found in Appendix A in form of a pseudo DTD.

This convenience should be contrasted to approaches which are based on XML libraries for general purpose programming languages like, e.g., HaXML [33] for Haskell, where non-trivial specifications soon become incomprehensible.

7. Variables and Tables

Complex transformations can be generated, by specifying different rules for different sub-documents and by using recursion. The limitation of such transformations, however, is that at every step, the transformer only has access to the sub-document currently under consideration. In order to access information obtained earlier during the transformation, it would be nice to have the possibility of simply storing certain data for later use. Therefore, we decided to include a notion of global variables into fxt.

As a simple example of the use of fxt variables, recall the XSLT specification of the transformation presented in Section 5. When processing a lastdef element, the XSLT transformation needed to search backward for the last occuring def element and to recursively transform it. Opposed to that, fxt remembers the result of processing the last def element in a forest variable. This value then just has to be looked up when transforming subsequent lastdef elements:

```
<fxt:spec>
  <fxt:global name="res" type="Forest"/>
  <fxt:push   name="res" val="emptyForest"/>

  <fxt:pat>//def</fxt:pat>
    <fxt:setForest name="res">
      <DEF><fxt:apply/></DEF>
    </fxt:setForest>
    <fxt:get name='res'/>

  <fxt:pat>//lastdef</fxt:pat>
    <fxt:get name='res'/>

</fxt:spec>
```

Processing hierarchically nested elements incurs the need for introducing scopes for variables. One way of doing so is to allow local variables which are visible just during processing a specific XML element. Re-structuring transformations, however, asks for more flexible scoping rules. Therefore, we decided to organize every variable as a *stack* — meaning that we support push and pop operations on variables. A push operation introduces a new scope whereas a pop leaves this scope again. So, the single fxt:push element in our example above creates the first instance of the variable and initializes it to the empty forest.

Besides for pushing and popping, fxt actions are provided for setting or outputting the values of the topmost elements. Special actions like fxt:setForest 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:inc name="i"/>
        <fxt:push name="i" val="0"/>
        <fxt:apply/>
        <fxt:pop name="i"/>
      </fxt:copyTag>

</fxt:spec>
```

The global variable i is declared of type int. Whenever an li element is transformed, the last value of i is output using fxt:get, and the li tag is copied to the output. Before proceeding to the transformation of the sons, the current value of i is incremented, and a new level of imbrication is opened. This is achieved by means of the fxt:push element which introduces a new instance of the variable i which is initialized with 0. After processing the contents of the li element, the last instance of i is popped again using fxt:pop.

It should be noted that XSLT also provides some notion of variables. XSLT variables, however, 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, therefore being merely named constants.

Some XML documents like, e.g., XML representations of graphs, have an inherent cross-reference structure which cannot be represented directly in 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. In order to deal with

such documents, fxt provides keys, which are a generalization of XML IDs. The key mechanism allows to collect sub-documents from the initial document in a table which later can be accessed via keys.

As an example, consider the following XML input document:

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

A specification of a transformation which lists the connections between the cities could then look as follows:

```
<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>
```

The fxt:key element declares a table with keyed access called cities. Elements of type node residing everywhere in the document are stored therein with keys given by their id attribute values. Whenever an edge is seen, the source and the destination of the edge are retrieved from the table, using as key the value of the attribute from and to respectively. The document above is therefore transformed into:

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

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 which, typically, is obtained from processing that tree.

8. Interfacing with SML

We expect the features provided by fxt to be rich enough to satisfy the basic needs of a non-expert user of our system. But obviously, not all potential uses, in particular of XML processing specialists are foreseeable. It is for such advanced and elaborated applications, that we have embedded into fxt an escape mechanism into the full programming language SML. In order to do so in a clean way, an interface is needed which abstracts from the implementation details of documents.

There are two main types of interfaces: tree-based and event-based. Through the event-based interface the application has a serialized view of the document. Each syntactical component of the document triggers an event. The application can register a handler for each type of events. The document can thus only be inspected once in a pre-determined order. One example of such an interface is the programming *hooks* as provided by the functional XML parser fxp [19, 8] or the SAX interface for object-oriented languages [15].

While event-based interfaces are well-suited for one-pass applications, tree-based interfaces also support applications that need multiple passes over the input. In the tree-based case, an abstract data type is specified. The most commonly used interface is the Document Object Model (DOM) [29]. Though claiming to be designed for any programming language, the DOM is committed to the object-oriented paradigm: it defines class interfaces for accessing XML documents. Also, it views the document tree as a graph within which arbitrary navigation is possible.

8.1. The Functional Document Model

A different interface is necessary for a functional style of XML processing. One such collection of types and useful functions for Haskell is HaXML [33]. For SML and specific use with fxt, we define the interface FDM (Functional Document Model) and provide an implementation for it.

One practical problem of SML here is that XML documents may contain any legal Unicode character [25], while SML supports only 8-bit char-

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

    (* testing type and content *)
    val isElement      : Tree -> bool
    val isText         : Tree -> bool
    val hasElementType : Unicode.Vector -> Tree -> bool
    val hasTextContent : Tree -> bool
    val hasAttribute   : Unicode.Vector -> Tree -> bool
    ...

    (* constructing node types *)
    val element        : Unicode.Vector -> Attribute list -> Tree vector -> Tree
    val text           : Unicode.Vector -> Tree
    ...

    (* accesing constitutive parts *)
    val getElementType : Tree -> Unicode.Vector
    val getTextContent : Tree -> Unicode.Vector
    val getAttribute   : Unicode.Vector -> Tree -> Unicode.Vector
    ...

    (* transforming forests *)
    val map            : (Tree -> 'a) -> Forest -> 'a vector
    val foldl          : (Tree * 'a -> 'a) -> 'a -> Forest -> 'a
    val deleteAll      : (Tree -> bool) -> Forest -> Forest
    val deleteFirstN   : int -> (Tree -> bool) -> Forest -> int * Forest
    val filterFirst    : (Tree -> bool) -> Forest -> Tree
    ...

    (* outputting *)
    val putTree        : Tree -> string -> string option -> unit
    val putForest      : Forest -> string -> string option -> unit

  end
```

acters and has no notion of Unicode. Therefore a Unicode library is provided inside FDM declaring 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 types Tree and Forest are provided as abstractions of XML sub-documents and of sequences of XML sub-documents, respectively. Functions are provided for testing the type or content of a node, for accessing its constitutive parts and for constructing different node types. Basic functionality is supplied for transforming sequences of trees (forests), like, e.g., 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 depicted excerpt from the FDM interface above.

The names of the FDM functions are mostly self-explaining. filterFirst,for example, 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 to 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.

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

8.2. Embedding SML Code into Transformations

SML code is embedded into an fxt specification via attributes of fxt actions. The values for these attributes are SML expressions. Their evaluation can provide XML content to be used in the output, predicates for filtering XML forests, or even functions for application onto document components.

The stylesheat presented above is the specification of a transformation that replaces every element name containing more than six characters by its first six characters.

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. The reserved FDM name current always refers to the current sub-document. Thus, 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 text nodes.

9. Typeful Transformations

Currently, fxt transformations are "un-typed" in the sense that no guarantee is given that the newly created documents conform to some specific document type. In fact, it is possible to produce even non-XML output, and we have extensively made use of this feature in order to create the HTML pages of our tools' online documentations.

There are, however, situations perceivable where guarantees on the relationship between the document types of input and output are indeed desirable. Several suggestions in this direction have been recently made. One approach is to build type safety into the transformation language itself. This approach is pursued by the language proposal XMλ which sketches an enhancement of the Haskell type-checking mechanism guaranteeing conformance of element forests to content models [16]. Based on finite tree automata, Hosoya and Pierce have developed a type system for the small tree transformation language XDuce [10]. They provide an exponential-time algorithm for statically inferring the types of all program expressions. Their language, however, similar to XMλ, provides only very restricted forms of pattern matching. Neither does it deal with attributes or dynamic modification of element tags.

Another approach is to abstract the transformational model into a formal device for which general typing algorithms can be provided. This is advocated by Suciu [17]. Suciu proposes the formal concept of a "k-pebble transducer" and proves a decidability result. The best currently known upper complexity bound of the decision procedure, however, is way beyond what is feasible in practice. Also, it is not clear yet whether this model can deal with fxt-type pattern matching as well — not to speak about dynamic modification of element tags. Yet another approach is suggested by Lämmel and Lohmann [14]. They propose to trigger the transformation on the document level by a reorganization of the document type itself. It is an interesting topic of future research in how far this idea can be integrated into a general XML transformation language.

10. Traversals and Strategies

fxt transformations proceed in several traversals over the input document. Up to two traversals are used for determining the matches of the triggering match patterns of the rules. Possibly, further traversals allow to fill tables for random access to selected document parts. Finally, the main traversal produces the output. This seems to suffice for most standard document processing needs.

We think, however, that in future applications, more complicated traversal organization might be needed. In particular, it could become desirable to compose several transformations of different kinds. Experiences with calculi for for such compositions have been collected in the area of term rewriting languages. 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. Transformations are also specified through rules, where the form of rules is similar to 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* which in a certain sense corresponds to one of our traversals. Typically, term rewriting uses a standard, fixed strategy. In term rewriting languages like Stratego [27][26][28], however, the user can provide his/her 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 subterms). 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. ELAN [3] is another term rewriting language using strategies. One of its features is the support for associative-commutative pattern matching. Like in Stratego, there is no special support for XML processing.

11. Comparing fxt with XSLT Implementations

In order to assess the time performance of fxt we compared it with three XSLT processors. These were the Xalan processors (part of the Apache XML Project [22]) Xalan Java 2 and Xalan C++ 1.2 and the Saxon 6.4.4 processor written in Java also [13].

We considered the following benchmark transformations:

- Birds
  - XML Input (10 KB): Description of classes of birds
  - Output: Plain text file presenting the information in the input in an indented manner
- GCA Paper
  - XML Input (60 KB): An fxt presentation conforming to the DTD for GCA XML Conference Proceedings [5]
  - Output: An HTML layout of the paper
- Lines
  - XML Input (200 KB): Shakespeare's "All's Well That Ends Well" play [4]
  - Output: The collection of all the lines in the play
- Baseball
  - XML Input (600 KB): Baseball statistics [9]
  - Output: HTML tables containing processed information about baseball players.
- T1 and T2
  - XML Input (200 KB): Shakespeare's play as above
  - Output: The collection of matches of complex patterns

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
		Saxon	0.086	0.692	0.196	1.075
		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
		Saxon	0.538	0.761	0.369	1.770
		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
		Saxon	0.398	0.640	0.495	1.633
		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
		Saxon	0.590	0.731	0.851	2.272
		Xalan Java	0.928	1.326	3.060	6.299

Table 1: Transformation times (seconds).

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 implementations 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 the XSLT processors also take into account the time needed for processing the stylesheets. It is likely that one stylesheet source is used to transform multiple XML sources. In this case the stylesheet needs 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. Times for startup for the JVM and for the SML runtime-system were also not considered.

The patterns used in the first three transformations are very simple. For them fxt proved to be in general faster than the Java processors and able to keep up with Xalan C++. Table 1 shows the results for the first three transformations.

T_1 and T_2 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. T_1 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/""
```

Note the differences in the patterns above due to the fact that the structural conditions within the brackets are expressed in XPath as XPath predicates, while in fxgrep they are regular conditions to be fulfilled by the sons of the preceding node. Thus, the first XPath predicate above is a comparison of the node-set selected by .//SPEAKER with the string BERTRAM, and is true if the string content of some SPEAKER descendant of the SCENE element is equal to BERTRAM. The corresponding structural constraint in fxgrep tells that the children of the SCENE element must consists of an arbitrary sequence of trees (as specified by the _ wildcard), followed by a node containing a descendant SPEAKER containing the text BERTRAM, followed by an arbitrary sequence of trees.

Application	Transformer	Parsing	Processing stylesheet	Transforming	Total
T_1	Xalan C++	0.200	0.010	0.780	0.990
	fxt	0.427	0.480	0.271	1.376
	Saxon	0.401	0.681	1.213	2.395
	Xalan Java	0.774	1.210	1.326	3.812
T_2	Xalan C++	0.200	0.010	7.500	7.710
	fxt	0.432	0.475	0.365	1.485
	Saxon	0.405	0.672	3.839	5.020
	Xalan Java	0.767	1.211	5.657	8.131

Table 2: Transformation times when using elaborate patterns (seconds).

T_2 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 XPath implementation in the XSLT processors reflect more or less the processing model of XPath imposing multiple tree traversals which incur a lot of processing time. The processing times for T_1 and T_2 are presented in Table 2. fxt performs significantly faster even than the C++ XSLT implementation.

For T_1 and T_2 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 transformation steps. The size-time dependency showed to be indeed linear.

12. Conclusions

In this paper we have presented fxt, a transformation language for XML documents. fxt combines the efficiency of the pattern matching provided by fxgrep with a simple, yet powerful set of transformation primitives and a clear programming interface to the functional programming language SML. Besides the features we have mentioned here, fxt offers a number of further features which allow for more convenient and expressive specifications. Among these are:

- 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 to the online documentation [2].

To our experience, the expressivity of our pattern-language together with the additional features of fxt such as stackable variables 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 often outperforms comparable implementations of XSLT.

13. Acknowledgments

The development of fxt is supported by the DFG Research Foundation. We are very grateful to the referees for the CIT Journal Issue on DSLs for their valuable comments on a previous version of this article.

A The syntax of an fxt specification

```
<!-- An action can be some XML character data, one of the listed fxt
     actions or any other literal XML element -->
<!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:inc|fxt:setForest|fxt:sml|fxt:sp
   |fxt:switch|fxt:tag|fxt:text|ANY">

<!-- A declaration is specified by one of the following fxt elements -->
<!ENTITY % DECLARATION "fxt:arg|fxt:global|fxt:key|fxt:open|fxt:push
                       |fxt:pop|fxt:set|fxt:inc|fxt:table">

<!-- A rule is a pattern, specified by an fxt:pat element
     followed by an arbitrary number of actions -->
<!ENTITY % RULE "fxt:pat, (%ACTION;)*">

<!-- A specification consists of a number of declarations followed by
     an arbitrary number of rules -->
<!ELEMENT fxt:spec ((%DECLARATION;)*,(%RULE;)*)>

<!-- A pattern is specified as the XML character data content of an
     fxt:pat element -->
<!ELEMENT fxt:pat (#PCDATA)>

<!-- The folowing fxt:actions have empty content -->
<!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:inc|fxt:sml|fxt:sp|fxt:table|fxt:text)
    EMPTY>

<!-- The following fxt:actions can have as content an arbirary
     sequence of fxt actions -->
<!ELEMENT
  (fxt:case|fxt:copyTag|fxt:copyTagAddAttribute
  |fxt:copyTagDeleteAttribute|fxt:copyTagReplaceAttribute
  |fxt:copyType|fxt:default|fxt:if
  |fxt:pushForest|fxt:setForest|fxt:tag)
    ((%ACTION;)*)>

<!-- fxt:literate may have only character data content -->
<!ELEMENT fxt:literate (#PCDATA)>

<!-- fxt:switch has zero or more case branches and a branch for
     the default -->
<!ELEMENT fxt:switch (fxt:case*,fxt:default)>

<!-- Attributes of the fxt actions -->
<!ATTLIST fxt:addAttribute
            (name|nameExp) NMTOKEN #REQUIRED
            (val|valExp)  NMTOKEN #REQUIRED>
<!ATTLIST fxt:apply
            test NMTOKEN #IMPLIED
            (selectselectExp) 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 NMTKEN #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:inc
            name 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>
```

References

[1] *Standard ML of New Jersey*, Home Page, 1989–2001. Available online at `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

[2] ALEXANDRU BERLEA, *Fxt*, Online Documentation, 2001. Available online at `http://www.informatik.uni-trier.de/~ aberlea/Fxt/`.

[3] PETER BOROVANSKY, CLAUDE KIRCHNER, 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.

[4] JON BOSAK, editor, *The Complete Plays of Shakespeare, Marked up in XML*, 1999. Available online at `http://metalab.unc.edu/xml/examples/shakespeare`.

[5] ALEXANDRU BERLEA AND HELMUT SEIDL, fxt - A Transformation Language for XML Documents. In *XML Conference And Exposition 2001*, 2001.

[6] A. VAN DEURSEN, P. KLINT AND J. VISSER, Domain-Specific Languages: An Annotated Bibliography, *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[7] ANDREAS NEUMANN, *fxgrep1.4.1*, Source Code, 2001. Available online at `http://www.informatik.uni-trier.de/~aberlea/Fxgrep/`.

[8] ANDREAS NEUMANN, *fxp1.4*, Source Code, 2001. Available online at `http://www.informatik.uni-trier.de/~aberlea/Fxp/`.

[9] ELIOTTE R. HAROLD, editor, *1998 Baseball Statistics – XML Sample Files*, 1999, Available online at `http://metalab.unc.edu/xml/examples/1998validstats.xml`.

[10] HARUO HOSOYA AND BENJAMIN C. PIERCE, XDuce: A Typed XML Processing Language, In *Proceedings Of The Third International Workshop on the Web and Databases (WebDB2000), Dallas, Texas*, pages 111–116, May 2000. Available online at `http://www.cis.upenn.edu/~hahosoya/xduce/`.

[11] JOHN HUGHES, The Design of a Pretty-printing Library. In John Jeuring and Erik Meijer, editors, *Advanced Functional Programming, Tutorial text of the First international spring school on advanced functional programming techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96, Springer, Heidelberg, 1995.

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

[13] MICHAEL KAY, *Saxon*, Software Documentation, 2001. Available online at `http://saxon.sourceforge.net`.

[14] RALF LÄMMEL AND WOLFGANG LOHMANN, Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, OCG, 2001.

[15] DAVID MEGGINSON ET ALIAS, editors, *SAX 1.0: The Simple API for XML*. Online Documentation, Megginson Technologies, May 1998. Available online at `http://www.megginson.com/SAX/index.html`.

[16] ERIK MEIJER AND MARK SHIELDS, XMλ: A Functional Language for Constructing and Manipulating XML Documents, (Draft), 1999.

[17] TOVA MILO, DAN SUCIU AND VICTOR VIANU, Typechecking for XML Transformerd. In *Proceedings of the ACM Symposium on Principles of Database Systems, 2000*, 2000.

[18] ROBIN MILNER, MADS TOFTE, ROBERT HARPER AND DAVID MACQUEEN, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[19] ANDREAS NEUMANN, *Parsing and Querying XML Documents in SML*, PhD thesis, University of Trier, Trier, 2000.

[20] ANDREAS NEUMANN AND HELMUT SEIDL, Locating Matches of Tree Patterns in Forests. In V. Arvind and R. Ramamujan, editors, *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145, Springer, Heidelberg 1998.

[21] BIJAN PARSIA, *Functional Programming and XML*. Online Article, XML.com, 2001. Available online at `http://www.xml.com/pub/a/2001/02/14/functional.html`.

[22] Apache XML Project, *Xalan*, Software Documentation, 2001. Available online at `http://xml.apache.org/xalan/index.html`.

[23] D. S. SWIERSTRA, P. AZERO AND J. SARAIVA, *Designing and implementing combinator languages*, In D. S. Swierstra, P. R. Henriques and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga, Portugal, 1998*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206, Springer, Heidelberg, 1998.

[24] D. S. SWIERSTRA AND L. DUPONCHEEL, Deterministic, Error-Correcting Combinator Parsers. In John Launchbury, Erik Meijer and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of Lecture Notes in Computer Science, pages 184–207, Springer, Heidelberg, 1996.

[25] The Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison Wesley Developers Press, Reading, Massachusetts, 1996.

[26] EELCO VISSER, *Strategic Pattern Matching*. In *Rewriting Techniques and Applications (RTA '99), Trento, Italy*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Springer, 1999.

[27] EELCO VISSER, *The Stratego Reference Manual*. Tehnical report, Institute of Information and Computing Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[28] EELCO VISSER, Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *To Appear in Rewriting Techniques and Applications (RTA '01), Utrecht, The Netherlands*, Springer, 2001.

[29] VIDUR APPARAO, STEVE BYRNE AND MIKE CHAMPION, et alias, editors, *Document Object Model (DOM) Level 1 Specification, Version 1.0*, W3C Recommendation, World Wide Web Consortium, October 1998. Available online at `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001`.

[30] JAMES CLARK, editor, *XSL Transformations (XSLT) Version 1.0*, W3C Proposed Recommendation, World Wide Web Consortium, November 1999. Available online at `http://www.w3.org/TR/xslt`.

[31] 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`.

[32] 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`.

[33] MALCOLM WALLACE AND COLIN RUNCIMAN, Haskell and XML: Generic combinators or type-based translation? In Peter Lee, editor, *Proceedings Of The International Conference on Functional Programming 1999, Paris, France*, pages 148–259, ACM Press, New York, Sept. 1999.

Received: July, 2001
Revised: October, 2001
Accepted: November, 2001

Contact address:
Alexandru Berlea, Helmut Seidl
Department of Computer Science
University of Trier
Germany
e-mail: {aberlea,seidl}@psi.uni-trier.de

ALEXANDRU BERLEA is a Ph. D. candidate 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 with the final thesis written at the Delft University of Technology in The Netherlands. His research interests include document processing, functional programming, tree automata and garbage collection.

HELMUT SEIDL graduated in Mathematics (1983) and received his Ph. D. degree in Computer Science (1986) from the University of Frankfurt/Main, Germany. He received his Dr. Habil. (1994) from the University of Saarbrücken and is currently full Professor at the University of Trier. His research interests include fixpoint algorithms, program analyses, tree automata, document processing and tele-teaching. He has published extensively, both in high-level conferences such as STOC, POPL, LICS, SAS, ICALP and ESOP, as well as in established journals like SIAM Journal of Computing, TCS, Science of Computer Programming, Journal of Logic Programming, Information Processing Letters, Journal on Parallel Programming and Nordic Journal of Computing.