XML Transformations in Scheme with LAML - a Minimalistic Approach

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.auc.dk

ABSTRACT

Transformation of XML documents is often supported by special purpose languages that make use of pattern matching and replacement. Many XML programmers need to learn and understand one of these languages for transformation purposes. In this paper we contrast and compare special purpose, pattern matching solutions with transformation programs written in a general purpose, functional programming language. We use Scheme and the LAML libraries for XML transformations. Our approach is minimalistic in the sense that only a small vocabulary is needed to carry out the transformations. Overall, we have found that the Scheme/LAML solution is doing well in a direct comparison with other solutions. We have also found that the integrated validation of both the source and target documents makes it easier to write trustworthy transformations. A DTD-guided pruning of document traversals is also a contribution of the research.

Categories and Subject Descriptors

D.1.1 [**Functional Programming**]: Scheme, XML

1. INTRODUCTION

XML [4] is a meta language, which is used to define *high-level markup languages* for a variety of different purposes. As for other high-level languages (such as high-level programming languages) there is a need to transform XML documents[1] to *lower-level languages*. Often, but not necessarily, the target language is another XML language, such as XHTML [5].

Transformation of XML documents is of interest to the programming language community in general [24], and to the functional programming community in particular. A Haskell combinator library represent early work in the area [25]. Much of the work in Scheme has been oriented towards improving the pattern matching and replacement framework of XSLT [12, 10].

[1]We use the term "XML document" for documents written in a language defined by XML, seen as a meta language.

In this paper we describe our experience with XML transformations in Scheme with LAML [21, 19]. We use a direct programming approach, in which the source document is analyzed and taken apart by means of a few selectors and traversal functions. These functions work on an tree-structured internal documentation representation (ASTs). The target document is constructed with use of the mirror functions (see section 2.2) of the XML target language. Both tasks are supported by the fact that only valid input documents can be dealt with, and only valid output documents can be constructed. The LAML approach is *minimalistic* in the sense that it primarily uses simple and plain Scheme functionality for XML document decomposition. The mirror functions are used for document synthesis.

This paper presents the LAML approach to XML transformation. The examples of the paper are all taken from similar work [7, 25, 12] which expose the merits of a number of other transformation techniques. In that way it is possible to make direct comparisons of the LAML approach and similar approaches. In the paper we also describe our own practical experiences with the XML transformation framework in LAML.

The LAML approach to XML transformation represents a pure and 100% Scheme solution, centered around the functional subset of the language. It is not necessary to understand a new special purpose pattern matching-replacement apparatus to program XML transformations in LAML. We hypothesize that this makes it easy and straightforward for competent Scheme programmers to carry out XML transformations using LAML.

In section 2 we will first briefly present some relevant background information on XML transformation in relation to programming languages. We also present some overall qualities of LAML. Next, in section 3, follows the main part of the paper, namely the example-based discussion of XML transformations. In section 4 we describe our experience with the implementation of several different XML languages in LAML. The conclusions of our work are summarized in section 5.

2. BACKGROUND

In this section we will first discuss XML transformations in relation to special purpose and general purpose programming languages. Next, we will introduce some overall aspects of LAML.

2.1 XML transformations

Transformation of XML documents is frequently done by means of XSLT [3]. XSLT is an XML language which is based on pattern matching and replacement. Seen as a programming language, XSLT is a *special purpose language* within the functional paradigm. Although XSLT is Turing-complete, it is often used together with a general purpose programming language for more specialized transformation tasks. XSLT works well in many contexts, but critical remarks are also encountered [14]. The main objections against XSLT are the syntactic verbosity, the lack of key features from the functional paradigm, and the lack of some general purpose programming mechanisms and concepts.

As a captivating aspect of XSLT, the transformation program and the source documents are written in the same linguistic framework, namely XML. This is a *mono-lingual* situation which we call *markup subsumption* [18], because the programming notation is subsumed in the markup language. To make this approach operational, an XSLT interpreter is needed. As an alternative to using XSLT, an existing general purpose programming language can be used to carry out the program transformations. In this situation, no special interpreter is needed. If we stick to a mono-lingual setup, we should also formulate the XML documents in that particular programming language. This is called *program subsumption* [18], because the markup notation will have to be subsumed in the programming language. With this, both XML authoring and XML transformation take place on the ground of the programming language. The use of Scheme with LAML is a concrete example of this approach.

In a *multi-lingual* situation, the transformations are written in a programming language, and the source documents are written in XML. This is a more common situation than the program hosted, mono-lingual solution introduced above. Many different programming languages have been used for transformation of XML documents. As a general characteristics of a multi-lingual setup, fragments of XML documents and fragments of the programming language will be mixed. It can be argued that the mixing of fragments from different syntactic traditions is confusing.

2.2 LAML

LAML is a set of Scheme libraries and tools related to web work. LAML is the host of the Scheme Elucidator [15], LENO [16], SchemeDoc, and other tools. LAML is intended to work with all R4RS or R5RS Scheme systems [8], on any platform, and in any operating system. This is a contrast to a number of similar systems such as Scribe [22] and BRL [13] which use non-standard Scheme readers. These systems are bound to particular Scheme processors.

Due to the minimalistic operating system support in Scheme, it is necessary to implement a few so-called *compatibility functions* in order to make use of LAML in a particular Scheme system. Compatibility functions exist for MzScheme, Guile, SCM, and SISC on Solaris, Linux and Windows (but not in all combinations.)

```
(load (string-append laml-dir "laml.scm"))
(lib-load "xml-in-laml/xml-in-laml.scm")
(load (in-startup-directory "basic-soccer-style.scm"))
(load (in-startup-directory "soccer-mirror.scm"))

(results 'group "A"
 (match
   (date "10-Jun-1998")
   (team 'score "2" "Brazil")
   (team 'score "1" "Scotland")
 )
 (match
   (date "10-Jun-1998")
   (team 'score "2" "Marocco")
   (team 'score "2" "Norway")
 )
 (match
   (date "16-Jun-1998")
   (team 'score "1" "Scotland")
   (team 'score "1" "Norway")
 )
 (match
   (date "16-Jun-1998")
   (team 'score "3" "Brazil")
   (team 'score "0" "Marocco")
 )
 (match
   (date "23-Jun-1998")
   (team 'score "1" "Brazil")
   (team 'score "2" "Norway")
 )
 (match
   (date "23-Jun-1998")
   (team 'score "0" "Scotland")
   (team 'score "3" "Marocco")
 )
)
```

Figure 1: A LAML soccer document.

The most central part of LAML is called XML-in-LAML. XML-in-LAML supports *authoring* [17] and *programming* with XML in the context of a Scheme program. As we will see in section 3, XML-in-LAML allows generation of a set of Scheme functions from an XML Document Type Definition (DTD.) Each XML element is represented as a named *mirror function* in Scheme. The Scheme definitions, which are common for all XML-in-LAML languages, are located in a shared LAML library. In the current version of LAML, mirror functions cannot be generated from XML Schemas [6].

An XML document is written as a Scheme expression that uses the mirror functions as constructors. The parameter passing conventions of the mirror functions allow flexible composition of XML-in-LAML expressions. (Examples are given in section 3.) In addition, an XML-in-LAML expression checks that only valid documents are generated. The mirror functions work on an internal tree structured document representation (ASTs), not on text. The internal representation can be textually linearized, and pretty printed if desired. Mirrors of major markup languages (different versions of XHTML, and SVG) are pre-generated in LAML.

In the rest of this paper "LAML" will be used in the meaning of "XML-in-LAML". We will explain the necessary LAML details during the discussions of the examples in section 3.

3. EXAMPLES

In this section we will illustrate the LAML approach to XML transformation by a number of examples. All of the examples stem from other papers which expose special purpose XML transformation languages.

3.1 Example 1: Soccer tournaments

In an XSLT tutorial [7] Michael Kay introduces an XML language for soccer tournament results, and two different HTML targeted transformations are discussed. The first example from Kay's paper is discussed below. The second example is available from the accompanying web page of this paper (the URL is given at the end of section 5.)

Figure 1 shows the Scheme/LAML version of the sample soccer document from Kay's paper. The functions that are called in the `results` expression are all mirror functions. Take as an example the expression `(team 'score "2" "Brazil")` which illustrates the handling of XML attributes and textual content in LAML. The dynamic type of the parameters and the mutual positions of the parameters are both significant for the interpretation of the expression. A name of an attribute, such as `score`, is represented as a symbol. The text string following the attribute name, such as `"2"`, represents an attribute value. Text strings which do not succeed a symbol, such as `"Brazil"`, are part of the textual content of the enclosing element.

In order to deal with the example in LAML we need an XML document type definition (XML DTD) of the soccer language. In this case the DTD is brief and simple:

```
<!ENTITY % Number "CDATA">
<!ELEMENT results (match)*>
<!ATTLIST results  group   CDATA  #REQUIRED>
<!ELEMENT match (date, team, team)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT team  (#PCDATA)>
<!ATTLIST team  score   %Number; #REQUIRED>
```

The DTD is parsed and the parsed DTD is given as input to a tool which generates a mirror of the XML soccer language. The generated mirror is represented as a Scheme source program, which can be loaded as shown in line 4 of Figure 1. Both of the tools that are involved are written in Scheme, and both are part of the LAML software package [19]. The mirror of the XML soccer language defines the Scheme functions `results`, `match`, `date`, and `team`. When applied together, as in Figure 1, these function validate the document according to the grammatical rules in the XML DTD. The validation is done by deterministic finite state automata (using Algorithm 3.5 of [1]) for all elements with element contents (in the sense of [4].)

Figure 2[2] shows Kay's XSLT transformation (to the left) and the similar transformation written in Scheme with LAML (to the right.) This particular transformation outputs a straightforward HTML paraphrasing of the XML document. The HTML document is shown in Figure 2 of Kay's paper, and it is also available on the web page that accompanies this paper. The XSLT document illustrates the basic use of the template constructs, and the extraction of attributes using `xsl:value-of`.

[2]The XSLT program is a direct, but re-indented copy of listing 2 from [7].

The Scheme transformation should be understood in the context of the document in Figure 1. The third line of Figure 1 loads the Scheme transformation functions in Figure 2. Following that the soccer mirror functions are loaded. As part of the transformations in Figure 2 the target mirror is loaded; in this case it is XHTML 1.0 transitional. The procedure `results!` is a so-called *action procedure* of the `results` element, and it serves as the top level control point of the transformation. The syntax tree of the soccer document is passed to `results!`. The procedure builds the HTML document, and it arranges that the document is written to a file using the LAML procedure `write-html`. The procedure `results!` calls the function `present-matches`, which is responsible for the most interesting part of the transformation.

The function `present-matches` traverses the soccer tournament document using the LAML function `find-asts`. The activation of the function `find-asts` in Figure 2 returns the list of all `match` constructs. The last parameter of `find-asts` is an optional transformation function, which is mapped on the list of located subtrees (in the actual case, `match` elements.) Thus, the bulk of the transformation work is done in `(lambda (match-ast) ...)`. In a simple way the `team` elements and their `score` attributes are extracted and bound to local names in the `let*` name binding form. In the body of `let*` the HTML fragment is formed, using the mirror functions of XHTML. Notice that the lambda expression returns a list of HTML elements. In general, the mirror functions systematically unfold lists of attributes and lists of content. The automatic unfolding has turned out to be extremely convenient, as it eliminates the need of explicit list flattening in LAML documents and programs.

In order to use LAML for XML transformation tasks the programmer will have to master the Scheme programming language and the basic LAML conventions explained above. It is also necessary to learn the XML AST functions that extract information from an XML document. In the example we use `ast-attribute`, `ast-subtree`, and `ast-text`. The `ast-attribute` function extracts the value of an attribute from an AST. The function `ast-subtree` returns a given subtree of an AST, and `ast-text` aggregates the textual contents of an AST. A few other such functions and predicates exist. The most advanced function used in the example is `find-asts` which is used for both tree traversal and transformation. We use this function extensively in our XML transformations (together with a sibling function `find-first-ast` that only returns the first match.)

As an additional overhead, the mirror of the soccer XML language has to be created, and as illustrated above the underlying DTD needs to be written. As an important bonus of this work, each of the generated mirror functions validate the source document. The LAML framework generates

```
<xsl:transform
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">
<xsl:template match="results">
 <html>
 <head><title>
   Results of Group <xsl:value-of select="@group"/>
 </title></head>
 <body><h1>
   Results of Group <xsl:value-of select="@group"/>
 </h1>
 <xsl:apply-templates/>
 </body></html>
</xsl:template>
<xsl:template match="match">
 <h2>
  <xsl:value-of select="team[1]"/> versus
  <xsl:value-of select="team[2]"/>
 </h2>
 <p>Played on <xsl:value-of select="date"/></p>
 <p>Result:
   <xsl:value-of select="team[1] "/>
   <xsl:value-of select="team[1]/@score"/>,
   <xsl:value-of select="team[2] "/>
   <xsl:value-of select="team[2]/@score"/>
 </p>
</xsl:template>
</xsl:transform>
```

```
(lib-load
 "xml-in-laml/mirrors/xhtml10-transitional-mirror.scm")

(define (results! soccer-ast)
 (let* ((ttl (list "Result of group"
                   (ast-attribute soccer-ast 'group))))
  (write-html '(pp)
   (html (head (title ttl))
         (body (h1 ttl) (present-matches soccer-ast))))))

(define (present-matches soccer-ast)
 (find-asts soccer-ast "match"
  (lambda (match-ast)
   (let* ((team1 (ast-subtree match-ast 'team 1))
          (team1-score (ast-attribute team1 'score))
          (team2 (ast-subtree match-ast 'team 2))
          (team2-score (ast-attribute team2 'score))
          (dt (ast-subtree match-ast 'date)))
     (list
      (h2 (ast-text team1) "versus"
          (ast-text team2))
      (p "Played on" (ast-text dt))
      (p "Result:"
         (ast-text team1) team1-score _ ","
         (ast-text team2) team2-score)))))))
```

Figure 2: The XSLT transformation (left) and the Scheme/LAML transformation (right).

the underlying finite state automata automatically from the DTD. The XHTML target document, as returned by the transformation functions, is validated in a similar way.

The integrated validation of the source document makes the programming of the transformation easier, because the programmer can safely assume, as a precondition, that the input is without grammatical errors. Thus, there is no reason for the programmer to include any kind of error handling functionality that take care of malformed input. Similarly, the integrated validation of the target document (HTML in this case) guaranties that only grammatically valid output can be returned. Thus, the programmer will get a warning (or a fatal error, if desired) in case invalid XHTML fragments are generated by the transformation program.

3.2 Example 2: LP/CD albums

This example is based on an early paper (from 1999) by Wallace and Runciman on transformation of XML documents in Haskell [25]. In the paper two different processing techniques are addressed. The first is based on generic combinators which work on a tree-structured internal document representation, independent of grammatical information from a DTD. This work is called HaXml. The second is a type-based approach, in which the XML elements of a DTD give rise to new Haskell types. With this, it is the goal to guarantee validity of the involved documents via an extended type checking of the processing program.

As a concrete example, the paper defines an LP/CD album DTD and it defines a transformation of an album XML document to HTML. The transformation does a partial HTML paraphrasing of the album properties. The example illustrates the approach based on generic combinators. The paper does not give a concrete example of the type-based approach. Other researches, most notable Thiemann [23], have since then refined the type-based approach substantially.

We will now compare the LAML work in Scheme with the Haskell work of Wallace and Runciman. The LAML approach relies on DTD information, but the document validation is done at run time. In contrast, the majority of the research on XML in relation to Haskell focuses on static validation of the processing program, seen as an extended type check of the programs that process XML documents. In our opinion, these differences are natural seen in the light of traditions and preferences of the Lisp community and the functional programming community.

In this section we will illustrate, compare, and discuss the LP/CD album transformation example given by Wallace and Runciman. We do that by creating a mirror of the album DTD[3] in Scheme and by developing a transformation program in Scheme similar to the Haskell script. Our main objective is to compare the use of the Haskell combinators in [25] with LAML.

Figure 3 shows an LAML CD album document, which follows the DTD from [25]. Of space reason, our example is shorter than the example given by Wallace and Runciman. Figure 4 shows the result of the transformation in HTML, as produced by LAML. (The XML declaration and the document type declaration have been elided). Finally, Figure 5 shows the Haskell transformation program and the Scheme transformation program.

[3]We do not show the DTD in this paper. The DTD is available in both [25] (on paper and on the web), and from the LAML home page [19].

```
(load (string-append laml-dir "laml.scm"))
(lib-load "xml-in-laml/xml-in-laml.scm")
(lib-load "color.scm")
(load (in-startup-directory "music-album-transf.scm"))
(load (in-startup-directory "music-album-mirror.scm"))

(set! xml-check-error laml-error)
(define album:title (music-album 'title))

(album
 (album:title "Graceland")
 (artist "Paul Simon")
 (recordingdate 'date "1986")
 (coverart 'style "art")

 (catalogno 'label "Warner Bros." 'number "925 447-2")

 (personnel
  (player 'name "Paul Simon" 'instrument "vocal")
  (player 'name "Baghiti Khumalo" 'instrument "bass")
 )

 (tracks
  (track 'title "The boy in the bubble"
         'credit "Forere Motlobeloa" 'timing "3m59s")
  (track 'title "Graceland"
         'credit "Paul Simon" 'timing "4m48s"))

 (notes 'author "normark"
  "Merges a South African style."
  (albumref 'link "http://www.classicalmusicreview.com"
  "See classical music review") _ ".")
)
```

Figure 3: A LAML CD album document.

The key processing functions in the Haskell transformation program (the left part of Figure 5[4]) are called *content filters*. A content filter function is applied on a document fragment, and it returns a list of document fragments. A contents filter can be used both to select document constituents and to create new contents. The content filter functions can be combined in many interesting ways, using one of numerous *filter combinator* functions introduced in the paper. As an example from Figure 5, the composite content filter `keep /> tag "title" /> txt` use the filter combinator `/>` to combine the content filters `keep`, `tag "title"`, and `txt`. When the composed filter is applied to the root document it returns the text of the title element. The functions called `hhead`, `htitle`, `hbody`, etc. are similar to the HTML mirror functions in LAML. The top-level control point is the application of `processXmlWith`, which activates the `albumf` content filter. In this function the outer parts of the HTML target document is constructed in a straightforward way, and the `notesf` and the `summaryf` contents filters are applied to deal with subtransformations.

The Scheme transformation program (the right part of Figure 5) is activated via the `album!` action procedure of the `album` element in Figure 3. Some explicit document decomposition is done first, hereby binding the local names `title-ast`, `artist-ast`, `dt-ast`, and `notes-ast`. In the body of the `let*` form the HTML document is aggregated using the HTML mirror functions and the two auxiliary

[4]The Haskell program is a verbatim copy of Figure 4 from [25].

```
<html>
 <head><title>Paul Simon: Graceland</title></head>
 <body bgcolor="#ffffff">
  <center><h1>Graceland</h1></center>
   <h2>Notes</h2>
   <p>
    Merges a South African style.
    <a href="http://www.classicalmusicreview.com">
     See classical music review
    </a>.
   </p>
   <table border="1">
    <tr><td>Album title</td> <td>Graceland</td></tr>
    <tr><td>Artist</td> <td>Paul Simon</td></tr>
    <tr><td>Recording date</td> <td>1986</td></tr>
    <tr>
     <td valign="top">Catalog numbers</td>
     <td>
      <ul><li>1. Warner Bros. 925 447-2 (CD)</li></ul>
     </td>
    </tr>
   </table>
 </body>
</html>
```

Figure 4: The transformed CD album document.

functions `present-notes` and `present-summary`. These auxiliary functions correspond to the `notesf` and `summaryf` content filters in the Haskell program.

In the function `present-notes` the function `transform-ast-list` is used to transform `trackref` fragments to emphasized text, and `albumref` fragments to anchored links. In general, `transform-ast-list` takes a list of AST fragments and a list of *transformation specifications* as parameters; each transformation specification is a list of an AST predicate and an AST transformation function. `transform-ast-list` applies a particular transformation on a document fragment when the corresponding predicate, as applied on the AST, holds. Constituents that do not satisfy any of the predicates are returned unchanged. (The latter is an important fact for understanding the example.) The function `laml-source-prepare` is an LAML technicality related to the handling of white space.

In the function `present-summary` the function `find-asts` is used to locate and transform `catalogno` elements. We have already introduced the basic properties of the tree traversal function `find-asts` in section 3.1. As an important additional property, the expression `(find-asts album-ast "catalogno")` limits the search to those part of the XML document in which there is hope to locate `catalogno` constituents. From the DTD it can easily be established that no `catalogno` can appear within `title`, `artist`, `recordingdate`, `coverart`, `track`, or `notes` subdocuments. This is based on the assumption that the `album` document is valid before the transformation is initiated. In LAML, this is known to be the case. During the generation of the album mirror a simple analysis of the XML language is done. During this analysis, knowledge about possible direct and indirect constituents, and possible direct and indirect attributes, are collected for all the elements.

```
module Main where
import Xml
main =
  processXmlWith (albumf `o` deep (tag "album"))
albumf =
  html
    [ hhead
      [ htitle
        [ txt `o` children `o` tag "artist"
              `o` children `o` tag "album"
        , literal ": "
        , keep /> tag "title" /> txt
        ]
      ]
    , hbody [("bgcolor",("white"!))]
      [ hcenter
          [ h1 [ keep /> tag "title" /> txt ] ]
      , h2 [ ("Notes"!) ]
      , hpara [ notesf `o` (keep /> tag "notes") ]
      , summaryf
      ]
    ]
notesf =
  foldXml (txt            ?> keep           :>
           tag "trackref" ?> replaceTag "EM" :>
           tag "albumref" ?> mkLink          :>
           children)
summaryf =
  htable [("BORDER",("1"!))]
    [ hrow [ hcol [ ("Album title"!) ]
           , hcol [ keep /> tag "title" /> txt ]
           ]
    , hrow [ hcol [ ("Artist"!) ]
           , hcol [ keep /> tag "artist" /> txt ]
           ]
    , hrow [ hcol [ ("Recording date"!) ]
           , hcol [ keep />
                        tag "recordingdate" /> txt ]
           ]
    , hrow [ hcola [ ("VALIGN",("top"!)) ]
                   [ ("Catalog numbers"!) ]
           , hcol
             [ hlist
               [ catno `oo`
                  numbered (deep (tag "catalogno"))
               ]
             ]
           ]
    ]
catno n =
  mkElem "LI"
    [ ((show n++". ")!),  ("label"?),  ("number"?)
    , (" ("!),  ("format"?),  (")"!) ]
mkLink =
  mkElemAttr "A" [ ("HREF",("link"?)) ]
    [ children ]
```

```
(lib-load
 "xml-in-laml/mirrors/xhtml10-transitional-mirror.scm")
(define html:title (xhtml10-transitional 'title))

(define (album! album-ast)
 (let* ((title-ast (ast-subtree album-ast "title"))
        (artist-ast (ast-subtree album-ast "artist"))
        (dt-ast (ast-subtree album-ast "recordingdate"))
        (notes-ast (ast-subtree album-ast "notes")))
  (write-html '(pp prolog)
   (html
    (head (html:title (ast-text artist-ast)_":"
                      (ast-text title-ast)))
    (body 'bgcolor (rgb-color-encoding white)
     (center (h1 (ast-text title-ast)))
     (h2 "Notes")
     (present-notes notes-ast)
     (present-summary album-ast title-ast artist-ast dt-ast)
)))))

(define (present-notes notes-ast)
  (p (laml-source-prepare
      (transform-ast-list (ast-subtrees notes-ast)
       (list (ast-of-type? 'element-name "trackref")
             (lambda (tf) (em (ast-text tf))))
       (list (ast-of-type? 'element-name "albumref")
             mklink)))))

(define (present-summary album-ast title-ast artist-ast dt-ast)
 (let* ((catalogno-asts (find-asts album-ast "catalogno"))
        (numbers (number-interval 1 (length catalogno-asts))))
  (table 'border "1"
   (tr (td "Album title") (td (ast-text title-ast)))
   (tr (td "Artist") (td (ast-text artist-ast)))
   (tr (td "Recording date")
       (td (if dt-ast (ast-attribute dt-ast 'date) "-")))
   (tr (td 'valign "top" "Catalog numbers")
       (td (ul (map catno catalogno-asts numbers)))))))

(define (catno cat-ast n)
 (let ((fmt (ast-attribute cat-ast 'format #f)))
  (li (as-string n) _ "." (ast-attribute cat-ast 'label)
      (ast-attribute cat-ast 'number)
      (if fmt (list "("_ fmt _")") '()))))

(define (mklink aref-ast)
  (a 'href (ast-attribute aref-ast 'link)
     (ast-text aref-ast)))
```

Figure 5: The Haskell transformation (left) and the Scheme/LAML transformation (right).

Thus, for instance, it is established that a `tracks` element only can contain a `track` element, and that the only attributes of a `tracks` element are `credit`, `timing`, and `title`. This knowledge is (at mirror generation time) arranged in a so-called *XML navigation structure*, which can be searched efficiently during the transformation process (binary search in sorted vectors.)

It should be noticed that the album XML language and HTML both contain an element called `title`. Thus, use of the non-qualified name "`title`" is ambiguous in the transformation program. In LAML a so-called *language map* is used to disambiguate the use of the `title` element. In most other XML contexts, XML name spaces [2] are used to deal with this problem. The LAML language map relates an element in a given XML language to the corresponding mirror function. Thus, the expression `(music-album 'title)` refers to the album title mirror function, and `(xhtml10-transitional 'title)` refers to XHTML's `title` mirror function. For each application of a mirror function, LAML checks that the simple name is non-ambiguous relative to the working set of LAML languages. An error occurs in case of ambiguity.

In several respects the Haskell and the Scheme solutions in Figure 5 are similar. Both make use of a relatively large set of mirror functions from the target language (HTML), and both contain a lot of expressions that select parts of the album document. The Haskell program is a rather sophisticated functional program which makes use of a lot of higher-order functions, most notably the filter combinators. The Scheme program uses mainly "lower-order functions", such as the functions that extract various information from the internal document structure (`ast-subtree`, `ast-text`, and `ast-attribute`.) Only two higher-order functions are used, namely `transform-ast-list` and `find-asts` (both explained above.) Some of these differences can be ascribed to different traditions and styles in the use of the two languages. Others have to do with the intended complexity of the solutions. We hypothesize that the learning curve of the Haskell framework is steeper than that of the LAML framework. We find that the Haskell solution is relatively complex, due the use of a many different content filters and combinators. The Scheme solution is "more down to earth" and it uses a smaller and a more basic set of functions. This is intended as a consequence of the minimalistic approach we are striving for.

3.3 Example 3: Purchases

The example in this section stems from the Scheme-related literature on XML transformations. Originally it comes from a transformation-by-example paper [12] which develops a language called XT3D. Like XSLT, XT3D is an XML language; XT3D is intended for novice XML users, as a simple and clean alternative to XSLT. In contrast, SXSLT [10, 11] is a more advanced higher-order transformation language, made available as a Scheme library, and in part developed by the same people who created XT3D. In this section we will contrast SXSLT and LAML.

The Scheme/LAML version of the XML source document from the SXSLT paper [10] is shown Figure 6. The document is based on an extremely simple XML DTD:

```
<!ELEMENT purchase (p*)>
<!ELEMENT p (#PCDATA)>
```

```
(load (string-append laml-dir "laml.scm"))
(lib-load "xml-in-laml/xml-in-laml.scm")
(load (in-startup-directory "purchase-transformation.scm"))
(load (in-startup-directory "purchases-mirror.scm"))

(purchase
  (p "4 thinkers")
  (p "5 tailors")
  (p "2 soldiers")
  (p "1 spy")
)
```

Figure 6: A LAML purchase document.

The transformation summarizes the `p` element in the `purchase` context as `text` elements in a more ordinary textual context. The main virtue of the example is to show how to deal with a context sensitive transformation. In concrete terms, the transformed `p` elements are separated by commas. The two last elements, however, are separated by "and". Also, the number of non top-level `text` elements are counted and represented in the `count` attribute of the outer `text` element. The DTD of the target language is as simple as that of the source language:

```
<!ELEMENT text (#PCDATA | text)>
<!ATTLIST text count CDATA #IMPLIED>
```

The result of the transformation initiated in Figure 6 is the following XML document:

```
<text count="3">
   4 thinkers,
   <text>
      5 tailors,
      <text>
         2 soldiers
         and
         <text>1 spy</text>
      </text>
   </text>
</text>
```

The SXSLT transformation program and the LAML transformation program are shown in Figure 7[5]. The SXML document is passed to the `convert-db` function, which initiates the pattern matching process via the `pre-post-order` function. As the second argument, the `pre-post-order` function takes a "transformation stylesheet", which is an association list of element names and transformation functions.

The details of the SXSLT program are rather contrived. In order to implement the context sensitive transformation of the `p` elements, the transformer of the `p` elements (the first case) returns a function `(lambda (count . args) ...)`.

[5]The SXSLT program is a direct, but re-indented copy of Figure 4 from [10].

```
(define (convert-db doc)
  (pre-post-order doc
  ; the conversion stylesheet
  `((h:p . ,(lambda (tag str)
    (lambda (count . args)    ; count can be #f:
      (let ((att-list         ; dont generate the attr
             (if count `((@ (acc:count ,count))) ())))
        (match-case args
          (() `(out:text ,@att-list ,str))
          ((?arg)
           `(out:text ,@att-list
             ,(string-append str " and")
             ,(arg #f)))
          ((?arg . ?rest)
           `(out:text
              ,@att-list ,(string-append str ",")
              ,(apply arg (cons #f rest)))))))))

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

```
(load (in-startup-directory "txt-mirror.scm"))

(define (p->text p-list length add-attribute?)
 (let ((count-attr
        (if add-attribute?
            (list 'count (as-string (- length 1)))
            '())))
  (cond ((= length 0) '())
        ((= length 1)
         (text count-attr (ast-text (car p-list))))
        ((= length 2)
         (let ((p1 (car p-list))
               (p2 (cadr p-list)))
           (text count-attr (ast-text p1) "and"
                 (text (ast-text p2)))))
        ((> length 2)
         (text count-attr
           (ast-text (car p-list)) _ ","
             (p->text (cdr p-list) (- length 1) #f)))
        (else
         (laml-error "unexpected length" length)))))


(define (purchase! ast)
 (let* ((p-clauses (find-asts ast "p"))
        (p-count (length p-clauses))
       )
  (write-html '(pp) (p->text p-clauses p-count #t)
   (in-startup-directory
    (string-append (source-filename-without-extension)
                   "." "xml")))))
```

Figure 7: The SXSLT transformation (left) and the Scheme/LAML transformation (right).

Using `match-case` (a Bigloo reminiscence) the essential conversion of the `p` elements takes place in the lambda expression. The second case handles the top-level `purchase` element. The function ultimately returned by the first case is activated here: `(apply (car  procs) ....)`.

The LAML transformation is initiated by the action procedure of the top-level `purchase` element. This is the procedure called `purchase!`. It extracts the `p` elements with `find-asts`, and counts these with `length`. The function `p->text` takes care of the essential part of the transformation. This function is a recursive function with four different cases in a `cond`. The three terminating cases (length 0, 1, and 2) of the recursion in `p->text` take care of the context sensitive processing of the `p` elements.

The SXSLT program relies on a good understanding of the special purpose pattern matching framework. As a contrast, the LAML program is a much more straightforward Scheme program that uses the mirror functions of the two XML languages, a single call of the traversal function `find-asts`, and activations of `ast-text` (which extracts and aggregates the textual contents of an element.) The function `p->text` is very easy to program for a competent Scheme programmer (almost a "first timer".) We hypothesize that it is more difficult to get the details of the SXSLT program right, due to the use of the specialized `pre-post-order` transformation function, higher-order transformations (as explained above), but also due to the multi-level use of quasiquotation (first for the transformation language of `pre-post-order`, and next for the purpose of SXML document construction, which we discuss below.) In summary, the LAML program is straightforward, with only a very few functions that call for special attention. The SXSLT program is more sophisticated, slightly briefer, but much more demanding on the programmer's comprehension.

As another aspects which is worth noticing, the SXSLT program uses a list-oriented XML notation called SXML [9]. Using this notation an SXML fragment is written as a list valued expression in Scheme, typically by use of quasiquotation (backquoting.) The SXML list-notation is intended as an authoring format, but it can also be created by an XML parser. In contrast, LAML uses expressions composed of the named mirror functions, which return internal list structures (lists represented as ASTs) similar to the SXML lists. The most important advantage of the LAML approach is the validation provided by the layer of the mirror functions. Using the mirror functions, it is not possible to construct an ill-formed document, or just a "weird list" which does not make sense. Also, the mirror functions shield and protect the programmer from the details of the concrete document representation (representation independence.) As a secondary advantage, the LAML mirror functions automatically handle the details of list unfoldings. As an example, the two LAML expressions

```
  (a 'href "x" "y")
  (a att-list "y")
```

give identical ASTs, provided that the value of `att-list` is the list `(href "x")`. Using SXML, the author must write one of following expressions

```
(course-plan
 (course-intro "This is the intro to the example course.")

 (course-info
   'last-lecture-number "3"
   'language-preference "danish"
   'color-scheme "blue-grey"
   'course-title "The example course"
   'brief-course-title "TEC"
   'course-semester "The last semester"
   'brief-course-semester "LAST"
   'teacher-name "Kurt Normark"
   'course-url-prefix
      "http://www.cs.auc.dk/~normark/courses/"
   'author-home-url "http://www.cs.auc.dk/~normark/"
   'make-overview-pages "true"
   ...

   (time-list
     (time 'year "2002" 'month "10" 'day "1"
           'hour "8" 'minute "15")
     ...
   )
   ...
 )

 (lecture-plan-list ... )
)
```

Figure 8: A LAML Course Plan document.

```
  '(a (@ (href "x")) "y")
  `(a (@ ,att-list) "y")
```

whereas the expression `` `(a (@ ,@att-list) "y") `` is incorrect.

4. XML LANGUAGES IN LAML

The XML languages and the LAML transformations presented in section 3 have all been written for demonstration purposes. In this section we will briefly discuss real examples of XML languages and XML transformations in LAML.

In the early years of LAML (starting in 1999) the HTML mirror functions were hand crafted and they just produced textual markup. Thus, these early mirror functions were all string-valued. Higher-level markup was created as *functional abstractions* on top of these. As of 2003, the validating HTML mirror functions are created automatically from XML DTDs, and they generate the internal AST representation. The creation of new XML languages can be seen as *linguistic abstraction*. The "implementations" of these languages are done with the transformation functions, which we have discussed in section 3.

Most of the XML languages we have made are used for educational purposes. The oldest and most comprehensive language, called LENO [16], supports the creation of web-based LEcture NOtes. Originally, it was created as a set of functional abstractions on top of the first generation HTML mirror functions. This early version of LENO supported its own internal document representation. More recently, a LENO XML language has been created as a new XML surface syntax to the existing, older software. The most important practical advantage seems to be the ease of introducing new attributes of existing LENO elements. In the older software, such attributes were accommodated by long optional parameter lists of simple Scheme functions. The major transformation task has been to translate the LAML ASTs to the older internal document representation of LENO.

An XML language, called Course Plan, for definition of course home pages has also been created. A Course Plan LAML document (written in Scheme using the Course Plan mirror functions) is transformed to a number of HTML pages. An early version of the Course Plan systems used the first generation of the HTML mirror functions, in the same way as LENO. The current version, which is in everyday use, relies on the LAML transformation framework. In the early version, the properties of a course home page were provided as a large set of Scheme `define` forms. In the XML solution, these definitions have been substituted by a large set of `course-info` attributes and a few subelements of `course-info`. Figure 8 shows an outline of an (highly incomplete) Course Plan document.

The LAML manual facility and SchemeDoc[6] rely on an XML language as well. SchemeDoc is modeled after JavaDoc, and as such it is able to extract library interface documentation from Scheme source files, and to present this information as HTML pages. All interface documentation of the LAML libraries is generated by SchemeDoc. The LAML manual facility is also able to generate documentation of XML DTDs.

5. CONCLUSIONS

XML transformations come in two flavors: One for non-programmers and one for programmers. Our work is oriented towards programmers, more specifically Scheme programmers. The majority of the Scheme-related work on XML transformations [12, 10, 11] is related to and inspired from XSLT, and as such it is based on pattern matching and replacement. We have also touched on work from the Haskell community [25], which is oriented towards content filters and higher-order filter combinators.

Throughout the paper we have shown how to transform XML documents in LAML to XHTML, or to other XML languages. The comparisons with other approaches indicate that the direct programming technique used in LAML is competitive with the special purpose and more sophisticated techniques used in XSLT, HaXml, and SXSLT. As it appears from Figure 2, 5, and 7 we have been able to write equivalent Scheme/LAML programs of more or less the same size of the XSLT, HaXml, and SXSLT programs. We hypothesize that trained Scheme programmers will find it relatively easy to get started with XML transformation in LAML. The learning curves of HaXml and SXSLT seem much steeper.

The major advantages of XML transformation in LAML seem to be the following:

1. The transformation functions can safely assume that

[6]LAML SchemeDoc and Schematics SchemeDoc, `http://schematics.sourceforge.net/schemedoc.html`, are similar in purpose, but not related in their design nor implementation.

the source document is valid.

2. The synthesis of the target document is safe, because it relies on mirror functions that implicitly validate the document while it is constructed.
3. The tree traversals are efficient (pruned traversal) because they are guided by XML navigation structures, which are derived from the XML DTDs.
4. Realistic transformation tasks can be done with only a handful of special purpose LAML functions. In this paper we have discussed `find-asts` and `transform-ast-list` and the AST selector functions `ast-subtree`, `ast-subtrees`, `ast-text`, and `ast-attribute`.

Some disadvantages can also be identified:

1. XML-work in LAML relies on a mirroring of XML in Scheme, which is foreign to people with a native XML orientation. Authoring of “XML documents” as well as programming of “XML transformations” is done on the basis of this mirroring.
2. Other XML representations are more complete than the XML representation in LAML. SXML [9] is undoubtedly the most complete representation of XML in Scheme today.
3. The need to create mirror functions of both the source and the target language complicates the matters. This part of the work may, in fact, be a stumbling block for newcomers to LAML.

The LAML transformation programs discussed in this paper are available from `http://www.cs.auc.dk/∼normark/-scheme/examples/xml-in-laml-transformation/`. The documentation of the LAML transformation functions, the documentation of the DTD parser, and the documentation of the mirror generation tool are all available from the LAML home page [19]. The LAML software is released as free software under the GNU general public license.

6. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] Time Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. W3C recommendation in http://www.w3.org/TR/1999/REC-xml-names-19990114/, January 1999.

[3] James Clark. XSL transformations (XSLT) version 1.0. W3C Recommendation. `http://www.w3.org/TR/xslt`, November 1999.

[4] World Wide Web Consortium. Extensible markup language (XML) 1.0, February 1998. http://www.w3.org/TR/REC-xml.

[5] World Wide Web Consortium. XHTML 1.0: The extensible hypertext markup language, January 2000. Available from http://www.w3.org/TR/xhtml1/.

[6] World Wide Web Consortium. XML Schema part 1: Structures, May 2001. Available from http://www.w3.org/TR/xmlschema-1/.

[7] Michael Kay. What kind of language is XSLT? An analysis and overview. Available from http://www-106.ibm.com/developerworks/xml/-library/x-xslt/?article=xr, February 2001.

[8] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[9] Oleg Kiselyov. SXML specification. *Sigplan Notices*, 37(6):52–58, June 2002. Also available from http://okmij.org/ftp/papers/SXML-paper.pdf.

[10] Oleg Kiselyov and Shriram Krishnamurthi. SXSLT: Manipulation language for XML. In V. Dahl and P. Wadler, editors, *PADL 2003*, LNCS 2562, pages 256–272. Springer Verlag, 2003.

[11] Oleg Kiselyov and Kirill Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. 2002. Presented on International Lisp Conference 2002 (ILC 2002). Available from http://okmij.org/ftp/papers/SXs.pdf.

[12] Shriram Krishnamurthi, Kathryn E. Gray, and Paul T. Graunke. Transformation-by-example for XML. In E. Pontelli and V. Santos Costa, editors, *PADL 2000*, LNCS 1753, pages 249–262. Springer Verlag, 2000.

[13] Bruce R. Lewis. BRL—a database-oriented language to embed in HTML and other markup, June 2003. http://brl.sourceforge.net/brl.pdf.

[14] T. Moertel. XSLT, Perl, Haskell, & a word on language design. Available at http://w.kuro5hin.org/story/2002/1/15/1562/95011, January 2002.

[15] Kurt Nørmark. Elucidative programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.

[16] Kurt Nørmark. Web based lecture notes - the LENO approach, November 2001. Available via [20].

[17] Kurt Nørmark. Programmatic WWW authoring using Scheme and LAML. In *The proceedings of the Eleventh International World Wide Web Conference - The web engineering track*, May 2002. ISBN 1-880672-20-0. Available from `http://www2002.org/CDROM/alternate/296/`.

[18] Kurt Nørmark. The duality of XML markup and programming notation. In *The proceedings of the IADIS International Conference on WWW/Internet.* IADIS, November 2003. Available via [19].

[19] Kurt Nørmark. The LAML home page, 2003. `http://www.cs.auc.dk/∼normark/laml/`.

[20] Kurt Nørmark. The LENO home page, 2003. `http://www.cs.auc.dk/∼normark/leno/`.

[21] Kurt Nørmark. Web programming in Scheme with LAML. Submitted to Journal of Functional Programming, April 2003. Available via [19].

[22] Manuel Serrano and Erick Gallesio. This is scribe! Presented at the 'Third Workshop on Scheme and Functional Programming', October 2002. http://www-sop.inria.fr/mimosa/fp/Scribe/doc/scribe.html.

[23] Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(5):435–468, July 2002.

[24] Philip Wadler, editor. *PLAN-X: Programming Language Technologies for XML*, October 2002. Available from http://www.research.avayalabs.com/-user/wadler/planx/.

[25] Malcolm Wallace and Colin Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 148–159. ACM Press, 1999. Published in Sigplan Notices vol 34 number 9. Also available from http://www.cs.york.ac.uk/fp/HaXml/icfp99.html.