Web Programming in Scheme with LAML

KURT NØRMARK
Department of Computer Science
Aalborg University
Denmark
(*e-mail:* normark@cs.auc.dk)

Abstract

Functional programming fits well with the use of descriptive markup in HTML and XML. There is also a good fit between S-expressions in Lisp and the means of expression in HTML and XML. These similarities are exploited in LAML which is a software package for Scheme. LAML supports exact mirrors of HTML 4.01, the three variants of XHTML 1.0, SVG 1.0, and a number of more specialized XML languages. The mirrors are all synthesized automatically from document type definitions (DTDs). Each element in a mirror is represented by a named function in Scheme. The mirror functions validate the XML document while it is generated. The validation is based on final state automata that are automatically derived from the DTD.

1 Introduction

In this paper we discuss the use of Scheme (Kelsey *et al.*, 1998) in the domain of web programming and web authoring. Our primary concern is the modelling of HTML and XML in Scheme. This topic is relevant for both static web documents and for dynamic documents generated by programs running on a web server.

LAML stands for *Lisp Abstracted Markup Language*. The key idea of LAML is to make existing and major markup languages, such as HTML 4.01, XHTML 1.0, and SVG 1.0 available as a set of Scheme functions. Using the XML-in-LAML framework, LAML supports the generation of Scheme mirror functions of any XML language defined by a DTD. The Scheme mirror functions reflect the properties and constraints of elements and attributes in the markup language.

LAML documents are written as Scheme programs. The textual content is represented as string constants. HTML and XML document fragments are written as Scheme expressions which call the mirror functions. Internally, a document is represented as an abstract syntax tree. In a LAML source document there is no lexical nor syntactical trace left of HTML or XML. As a contrast to other similar Scheme-based systems (Scribe and BRL, see section 6) LAML uses a standard Scheme reader. LAML can therefore be used with any R4RS or R5RS Scheme system which implements a small collection of well-defined, operating system related procedures and functions (such as `file-exists?` and `delete-file`.)

The primary goal of LAML is to support the creation of complex web materials in Scheme. Complex web materials resemble in many ways non-trivial programs.

The need of abstraction is a primary concern. At the fine grained level, abstraction can be supported by definition of functions that encapsulate a number of document details. At a more coarse grained level, linguistic abstraction is supported in LAML by the generation of exact mirrors of XML languages, as defined by DTDs. Programmatic means of expressions, as reflected by selection and iteration, is also important when we deal with complex web documents. As a particular aspect, LAML has been designed to make good use of higher-order list functions. This will be illustrated in section 4.

The source of a LAML web document is a Scheme program, which uses the LAML libraries, most importantly the set of HTML and/or XML mirror functions. Working on this ground, the Scheme programming language is available at any location in a web document, and at any time during the authoring process. As a pragmatic consequence, many problem solving aspects can be handled inside the document—expressed in Scheme—as opposed to a handling by external XML tools and processors. Of these reasons, we use the term *programmatic authoring* for our approach (Nørmark, 2002).

LAML uses the relatively weak typing mechanisms of Scheme. The values of HTML and XML related expressions are typed, but the LAML source program is not. As an implication, the validity of a LAML web document is checked at document generation time, when the Scheme program generates and transforms the internal AST representation of the document.

The main contribution of this work is the mirroring scheme that makes HTML and XML elements available as ordinary Scheme functions. The integrated validation of the generated documents at document generation time is an important part of the approach, because it enhances the quality of the generated web documents. The fitting of the framework to support a natural organization of document data in lists is also important.

In section 2 we discuss a couple of simple examples of complete LAML documents at the 'hello world' level. In section 3 the HTML mirror functions are explained and discussed. Section 4 contains additional examples, primarily illustrating the use of higher-order functions together with LAML. In section 5 the XML-in-LAML framework is covered. The work on LAML is related to similar work in section 6. In the remaining parts of the paper we will restrict the discussion to XML, including XHTML, but excluding older versions of HTML.

2 An initial example

Figure 1 shows a 'hello world' example to illustrate the composition of a complete LAML document. The first line loads the fundamental LAML software; The second line loads the XHTML 1.0 transitional mirror library; Then follows a `write-html` clause, which contains a `(html ...)` expression. The expression uses the XHTML mirror functions `html`, `head`, `title`, `body`, `p`, and `a` which correspond to the similarly named elements in XHTML. The first parameter of `write-html` controls the kind of rendering (`raw` as opposed to pretty printed) and the use of a document `prolog` in terms of an XML declaration and document type definition.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-xhtml1.0-transitional-validating")

(write-html '(raw prolog)
 (html 'xmlns "http://www.w3.org/1999/xhtml"
  (head (title "Hello World"))
  (body (p "Hello" (a 'href "http://www.w3c.org/" "W3C")))))

(end-laml)
```

Fig. 1: *A LAML 'Hello World document'.*

Most LAML documents introduce a number of document abstractions. Even in relative simple web documents there are many good uses of functional abstractions. This is illustrated by elaborating the example from figure 1 to that of figure 2. Many useful abstractions are related to attribute values, such as the function `w3c-url` that abstracts the prefix part of the W3C URL. Others are related to sets of attributes, such as `html-props` and `body-props`.

It is also useful to introduce content-related abstractions. As an example, the document in figure 2 implements and uses the function `indent-pixels`. The function is implemented in terms of an HTML table. The function `author-signature`, which is intended to be defined in a LAML startup file called `.laml`, returns the author's name, affiliation, and email address.

We will here make two observations about content-related abstractions in LAML. First, ordinary positional parameters do not fit well with the parameter conventions of the HTML mirror functions. Therefore it may be attractive to use a more HTML-like parameter profile of `indent-pixels` and similar functions. We show how this can be done in section 4. Second, if a coherent collection of content abstractions is necessary, it is often useful to implement this collection as a new XML language in LAML. We discuss this in section 5.

3 XHTML mirror functions

The XHTML mirror functions are designed with the goal that element instances in web documents should have straightforward and easily recognizable counterparts in Scheme. As examples, the XML clauses

```
<tag1 a1 = "v1" ... am = "vm">contents</tag1>
<tag2 a1 = "v1" ... am = "vm" />
```

correspond to the Scheme expressions

```
(tag1 'a1 "v1" ... 'am "vm" "contents")
(tag2 'a1 "v1" ... 'am "vm")
```

Expressions like these are evaluated to instances of ASTs, which in turn are represented as tagged list structures. AST nodes hold information about the element name, the element contents, the attributes, and the XML language being used. An

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-xhtml1.0-transitional-validating")
(lib-load "xhtml1.0-convenience.scm")

(define html-props (list 'xmlns "http://www.w3.org/1999/xhtml"))

(define body-props
   (list 'bgcolor (rgb-color-encoding white) 'text (rgb-color-encoding black)
         'link (rgb-color-encoding blue) 'vlink (rgb-color-encoding blue)))

(define (w3c-url suffix) (string-append "http://www.w3c.org/" suffix))

(define (indent-pixels p indented-form)
  (table 'border "0"
    (tr (td 'width (as-string p))
        (td 'width "*" indented-form))))

(write-html '(raw prolog)
 (let ((ttl "A simple page"))
  (html html-props
   (head (title ttl))
   (body  body-props
     (h1 ttl)
     (indent-pixels 50
      (p "The" (a 'href (w3c-url "") "W3C")
          "web site has information about"
          (a 'href (w3c-url "MarkUp/") "HTML") _ ","
          (a 'href (w3c-url "XML/") "XML") _ ","
          "and many other web technologies."))
     (author-signature)   ))))

(end-laml)
```

Fig. 2: *A simple LAML web document with a number of abstractions.*

AST is typically transformed to a lower level representation (such as an HTML AST), rendered as a strings, or rendered directly to an open output port. In both rendering situations we avoid excessive string concatenation in order to reduce the amount of garbage collection of string parts.

A mirror function accepts AST values, character reference values (tagged lists like the ASTs) as well as strings, symbols, booleans, and lists. At run time, the type of the actual parameter values are used to control the interpretation of the parameters, prior to the building of an AST. The flexibility of Lisp, as a contrast to the rigidness of statically typed functional languages, is crucial for our approach. The mirror functions obey the following parameter passing rules:

- ***Rule 1.*** A symbol represents an attribute name. Symbols of the form `css:`a refers to the a attribute in CSS. A symbol must be followed by a string that plays the role of the attribute's value.
- ***Rule 2.*** A string which does not follow a symbol is an *element content item*. Character reference values as well as AST values returned by mirror functions are also element content items.
- ***Rule 3.*** All element content items are implicitly separated by white space.

- ***Rule 4.*** A boolean *false* value in between element content items (normally bound to the underscore variable) suppresses white space.
- ***Rule 5.*** A list of symbols, strings, booleans, character references, and ASTs is processed recursively, and the resulting content items, attributes and white space markers are spliced with the surrounding list of parameters.

The following LAML expression illustrates the parameter passing rules.

```
(p "The" (a 'href "http://www.w3c.org" "WWW") "Consortium" _ ".")
```

The value of the expression is an AST, which can be printed as

```
(ast "p" ("The" #t (ast "a" ("WWW") (href "http://www.w3c.org")
           double xhtml10-transitional)
           #t "Consortium" ".")   () double xhtml10-transitional)
```

A boolean *true* value in the AST explicitly represents white space. The AST can be rendered as the following HTML clause:

```
<p>The <a href = "http://www.w3c.org">WWW</a> Consortium.</p>"
```

Notice that the mutual order of element content items and attributes is arbitrary as long as Rule 1 is adhered to. Thus, `(a 'href "http://www.w3c.org" "A" "B" "C")` and `(a "A" 'href "http://www.w3c.org" "B" "C")` are equivalent expressions. The rationale behind Rule 3 is to support white space in between content constituents (the typical case) without use of additional, explicit markup elements.

In Lisp it is often convenient to represent document fragments as nested lists. This is the rationale behind Rule 5. As an example, the expression

```
(let ((attributes (list 'start "3" 'compact "compact"))
      (contents (map li (list "one" "two" "three"))))
  (ol 'id "demo" contents (li "final") attributes))
```

which can be rendered as

```
<ol id = "demo" start = "3" compact = "compact">
  <li>one</li> <li>two</li> <li>three</li> <li>final</li>
</ol>
```

shows that both an attribute list and a content fragment list can be passed to the `ol` mirror function.

The XHTML mirror functions validate the generated document at the time the LAML expressions are evaluated. The validation is done relative to the underlying DTD. Both the document composition and the attributes are checked. The document composition must be in accord with the element content models, which taken together represent a context free grammar of the XML language; The attributes are checked for attribute existence, presence of required attributes, attribute types, and avoidance of attribute duplication. In case of validation problems, warnings are issued. If the author wants to, a validation failure may also lead to a fatal error. Additional details of the document validation framework is discussed in section 5.

The validation of the document against the DTD would be in vain if the textual content or an attribute value of a document is allowed to contain the character '<'

or '>' (or a double quote character in an attribute value). Instead of prohibiting these characters in the textual contents of LAML document we translate them to their similar HTML character references, such as `<`. The translation is carried out by means of a systematic mapping of every character in the textual contents and in attribute values. We also use the mapping to translate national characters, such as the Danish 'æ', 'ø', and 'å', to the corresponding HTML character references.

4 Examples with higher-order functions

There are many good uses of higher-order functions in relation to the XHTML mirror functions. As the first application, we will see how an HTML table can be made by combining the `table`, `tr`, and `td` mirror functions. In many contexts we find it natural to represent tables as list of rows, where each row is a list of elements:

```
(define sample-table '(("Row" "no." "1") ("Row" "no." "2")))
```

The following expression generates an XHTML table of `sample-table`

```
(table (map (compose tr (map td)) sample-table) 'border "1")
```

The table is rendered as

```
<table border = "1"> <tr><td>Row</td> <td>no.</td> <td>1</td></tr>
                     <tr><td>Row</td> <td>no.</td> <td>2</td></tr>
</table>
```

Above, it is assumed that `map` is curried (done by the LAML function `curry-generalized`). The function `compose` composes two or more functions to a single function.

The LAML higher-order function `xml-modify-element` is able to bind attributes (and content items as well) to fixed values in a mirror function. As an example, the following expression returns a specialized `a` (anchor) function in which the `target` and the `title` attributes have fixed values:

```
(xml-modify-element a 'target "main" 'title "Goes to the main window")
```

It is sometimes useful to convert a function with ordinary positional parameter passing to functions with *LAML mirror function parameter passing* (as defined by the five rules in section 3). As an example, we defined the function `indent-pixels` in figure 2 to take two parameters, namely the indentation and a single element instance. Instead of the expression

```
(indent-pixels 50 (div (p "First par.") (p "Second par.")))
```

we want to introduce attributes and content items, such as

```
(new-indent-pixels 'indentation "50" (p "First par.") (p "Second par."))
```

With the new parameter profile we can pass an arbitrary number of content items to `new-indent-pixels` without aggregating them with `div`. The function `xml--in-laml-parametrization` generates the new version of the indentation function from the existing one:

```
(define new-indent-pixels
 (xml-in-laml-parametrization indent-pixels
  (lambda (contents attributes)
    (list (get-prop 'indentation attributes) (div contents)))
  (required-implied-attributes '(indentation) '())))
```

The second parameter of `xml-in-laml-parametrization` is a function which is demanded to return the parameter list to `indent-pixels` given the content items and the attribute property list. The third parameter of `xml-in-laml-parametrization` is supposed to validate the the contents and the attributes. Above, we use the function `required-implied-attributes` which returns a predicate that ensures the presence of the `indentation` attribute, and that no other attributes are passed. A function similar to `xml-in-laml-parametrization` allows us to make ad hoc abstractions on top of existing XML mirror functions.

5 Synthesis of XML mirror functions

The mirror functions of an XML language can be synthesized from the XML document type definition (DTD) of the language. LAML supports a DTD parser, which delivers a list representation of the DTD, in which all entity instances (textual macro applications) are unfolded. The list representation of the DTD is used as input to the LAML mirror generation tool, which creates a Scheme source file with the mirror functions of XML elements. The XHTML mirrors described in section 3, as well as a mirror of SVG, have been produced by these tools.

As an important aspect, the mirror functions validate XML documents at document generation time. The validation of the attributes has already been explained in section 3. LAML defines a validation procedure for each mirror function. The validation procedure checks the context free correctness of a construct relative to the *content specifications* of the XML DTD. In case of validation problems, an error message is printed. We have emphasized the production of straightforward and easily understandable error messages. The content specifications are regular expressions. As examples of content specifications, the `table` and the `body` elements in XHTML 1.0 are constrained by the following (slightly abbreviated) regular expressions:

```
(caption?, (col*|colgroup*), thead?, tfoot?, (tbody+|tr+))
(#PCDATA | a | abbr | acronym | address | applet | b | basefont | ...)*
```

The LAML mirror synthesizer generates deterministic final state automata for those of the elements that have *element content* (such as `table`) whereas the elements with *mixed content* (such as `body`) are validated by simpler means. The automata are implemented from Algorithm 3.5 of (Aho *et al.*, 1986). The automata are represented as lists and vectors in Scheme, and they are embedded directly and compactly in the validation procedures. Automaton compactness is important to keep down the software loading times. The automaton validation functions are fast due to use of binary search for the transitions.

In XHTML 1.0 strict/transitional/frameset there are 6/2/2 automata with more

than 40 transitions. In SVG 1.0 there are 27 automata with more than 40 transitions. The largest of the XHMTL and SVG automata has 40 states and 1600 transitions, and it occupies appr. 16 Kbytes in the Scheme mirror source file. The large automata occurs in the cases where choices among many XML constructs appear in elements with *element content*. The size of the Scheme source file of the SVG mirror is 403 Kbytes. All the mirror source files of XHTML are less than 170 Kbytes.

In addition to the mirrors of XHTML and SVG we have defined a number of other XML languages, each of which can be seen as linguistic abstractions in contrast to definition of a set of individual functional abstractions. The mirror functions of all the XML languages are part of a framework which we call XML-in-LAML. As a central aspect of XML-in-LAML, a set of library functions are shared among all XML languages in LAML. The shared XML-in-LAML library supports the internal AST document format, higher-order functions for AST traversal and transformation, textual rendering functions, content and attribute validation functions, and some bookkeeping functionality which allows fragments from two or more different XML languages to coexist in a single document.

In case that two different XML-in-LAML languages have identically named elements, there will be a clash of mirror function names in Scheme. XML solves this problem by means of *name spaces* which disambiguate the two names by means of a unique prefix. In LAML, we have introduced the concept of a *language map*. A language map for a given XML language maps an element name to the corresponding Scheme mirror function. Take as an example the following:

`(xhtml10-strict 'title)` ⇒ *the* `title` *mirror function in XHTML 1.0 strict*

If no ambiguity is present, a mirror function can be accessed via a simple name. In case of ambiguity, a warning is issued at document generation time, and the mirror function should be accessed via the language map. At mirror generation time, we check that no mirror function collides with names of R4RS Scheme functions.

6 Related work

We will restrict the discussion of related work to similar work done in Scheme, and to work in the area of other functional programming languages.

BRL is a language designed for server-side, database connected web applications (Lewis, 2000). BRL allows evaluation of Scheme program fragments within an HTML document. The Scheme fragments are nested in square brackets. As an alternative understanding, a BRL document can be seen as a non-standard Scheme program, in which strings are surrounded by 'reverse square brackets', such as `]a string[`. As a contrast, a LAML document is a standard Scheme program which avoids the mixing of XML markup and Scheme fragments.

Scribe is a Scheme-based system for authoring of web pages, and in particular technical documents (Serrano & Gallesio, 2002). Like BRL, Scribe is based on a non-standard Scheme reader, which introduces a new bracketed syntax for (lists of) strings with inspiration from Scheme quasiquotation. The string `[a ,(bold`

`"bold") string]` serves as an example. Scribe defines a particular document language (in the style of LaTeX) and because of that Scribe is able to generate output in different formats, such as HTML, PS, PDF and others. Like LAML, Scribe uses an internal AST document representation. Scribe uses the internal document representation for document introspection in interesting ways.

Latte (Glickstein, 1999) is mixture of the Latex text formatting system and Scheme, at least at the conceptual level. In Latte, the author uses a Latex-like markup style. Most interesting, however, Latte supports a Scheme-like language in TeX syntax.

The PLT Scheme group has developed XT3D for 'XML transformation by example' (Krishnamurthi *et al.*, 2000) with inspiration from the R5RS Scheme macro facility. As part of this work it is possible to generate Scheme builder functions from XML Schemas. In LAML the similar mirror functions are generated from XML DTDs. Internally, the PLT tools represent XML documents as list structures, which are called x-expressions. These are similar to the AST structures used in LAML. Currently, LAML only support straightforward AST traversal and transformation functions, whereas the PLT work relies on a much more elaborate framework based on pattern matching and replacement.

Kiselyov (2002) defines an XML format in Scheme called SXML. An SXML clause is an S-expression (a list data structure), whereas a LAML clause is a Scheme expression which refers to named XML mirror functions. Both formats are intended for authoring purposes. Conceptually, however, the SXML format is similar to a LAML AST. In a recent paper, Kiselyov and Krishnamurthi (2003) describe a Scheme counterpart to the W3C XSLT transformation framework, which they call SXSLT. SXSLT works on SXML structures. It is argued that SXSLT is superior to XSLT, and that it is more adequate for 'power users' than XT3D.

Wallace and Runciman (1999) discuss two different representations of XML documents in Haskell. One is based on a generic tree representation of XML documents; The other is based on typed document fragments, where the DTD gives rise to a number of algebraic type definitions in Haskell. The driving force behind the second approach is validation of XML documents via static type checking of the Haskell XML programs.

Meijer and colleagues have in a number of papers dealt with aspects of web programming using Haskell. In the first of these a Haskell framework for CGI programming is presented (Meijer, 2000). In a second paper, Meijer and Shields (2000) define a new language called XMλ which is indented for generation of dynamic XML documents. XMλ is based on the point of view that programmatic XML expressions, in which the textual content is written and passed as quoted strings, is intractable. Therefore XMλ deals with verbatim XML documents, expressed in a language similar to Haskell, in which program fragments are escaped. In comparison, LAML is based on programmatic notation, and the textual contents is passed as string constants.

Thiemann describes a modelling of XML and HTML in Haskell (Thiemann, 2002). Thiemann is able to synthesize Haskell combinator libraries from XML and HTML DTDs. The synthesized libraries validate Haskell XML and HTML expres-

sions statically by means of Haskell type checking. Although a fully validating HTML library is provided for, Thiemann finds that a partial validation is adequate for practical purposes. As a contrast, we have found that a comprehensive XML/HTML validation of statically generated documents is both important and worthwhile.

Hanus (2001) describes a functional/logical web programming framework for the language called Curry. This work is based on a straightforward modelling of HTML as Curry data structures.

7 Conclusions

LAML is designed for authoring of complex web pages and web sites using Scheme. A LAML document is a Scheme program, which uses a normal Scheme reader, and which can be processed by use of most Scheme systems.

We have found that programmatic authoring using LAML is convenient and powerful for Scheme programmers. The rules of the XML mirror functions have been developed through a number of LAML generations. We have used LAML extensively over the last five years, primarily for static processing of XML-in-LAML documents.

The XML-in-LAML framework has been used to bring in support of major and existing XML markup languages, such as XHTML and SVG. We have also used XML-in-LAML for definition and support of new XML languages in LAML, primarily in the educational domain. The comprehensive validation of XML-in-LAML documents is seen as a valuable asset, because any deviation from the document standard is identified when the document is processed. The fully automatic generation of the validation procedures is a major step forward, compared to earlier versions of the system which required some manual programming efforts to produce the validation predicates.

Although LAML is relatively mature, there are still areas where more work needs to be done. The transformation of documents from one XML-in-LAML language to another represents one such area. Pretty printed XML rendering, and LAML's XML parser, are two other areas where more work is needed.

LAML is available as free software from the LAML homepage (Nørmark, 1999).

References

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. (1986). *Compilers - principles, techniques and tools*. Addison-Wesley.

Glickstein, Bob. (1999). *Latte—the language for transforming text*. http://www.latte.org/.

Hanus, Michael. (2001). High-level server side web scripting in Curry. *Pages 76–92 of:* Ramakrishnan, I.V. (ed), *PADL 2001*. LNCS 1990. Springer Verlag.

Kelsey, Richard, Clinger, William, & Rees, Jonathan. (1998). Revised^5 report on the algorithmic language Scheme. *Higher-order and symbolic computation*, **11**(1), 7–105.

Kiselyov, Oleg. 2002 (August). *SXML*. http://okmij.org/ftp/Scheme/SXML.html.

Kiselyov, Oleg, & Krishnamurthi, Shriram. (2003). SXSLT: Manipulation language for

XML. *Pages 256–272 of:* Dahl, V., & Wadler, P. (eds), *PADL 2003*. LNCS 2562. Springer Verlag.

Krishnamurthi, Shriram, Gray, Kathryn E., & Graunke, Paul T. (2000). Transformation-by-example for XML. *Pages 249–262 of:* Pontelli, E., & Costa, V. Santos (eds), *PADL 2000*. LNCS 1753. Springer Verlag.

Lewis, Bruce R. 2000 (October). *BRL—a database-oriented language to embed in HTML and other markup*. http://brl.sourceforge.net/.

Meijer, Erik. (2000). Server side web scripting in Haskell. *Journal of functional programming*, **10**(1), 1–18.

Meijer, Erik, & Sheilds, Mark. (2000). *Xmλ - a functional language for constructing and manipulating XML documents*. Submitted to USENIX Annual Technical Conference 2000. Available via http://www.cse.ogi.edu/~mbs/pub/xmlambda/.

Nørmark, Kurt. (1999). *The LAML home page*. `http://www.cs.auc.dk/~normark/laml/`.

Nørmark, Kurt. 2002 (May). Programmatic WWW authoring using Scheme and LAML. *The proceedings of the eleventh international world wide web conference - the web engineering track*. ISBN 1-880672-20-0. Available from http://www2002.org/CDROM/-alternate/296/.

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

Thiemann, Peter. (2002). A typed representation for HTML and XML documents in Haskell. *Journal of functional programming*, **12**(5), 435–468.

Wallace, Malcolm, & Runciman, Colin. (1999). Haskell and XML: generic combinators or type-based translation? *Pages 148–159 of: Proceedings of the fourth acm sigplan international conference on functional programming*. ACM Press. Published in Sigplan Notices vol 34 number 9.