Web Programming in Scheme - the LAML approach

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 (Lisp Abstracted Markup Language) which is a software package for Scheme. LAML supports exact mirrors of different versions of HTML. In the mirrors each HTML element is represented by a named Scheme function. The mirror functions guarantee that the generated HTML code is valid. LAML has been used for both server side CGI programming and programmatic authoring of non-trivial static web materials. The programmatic LAML author can use the power of functional programming for the production of everyday web documents. Equally important, it is straightforward to define domain-specific web languages in Scheme syntax which parallel the advantages of XML.

1 Introduction

In this paper we discuss the use of Scheme (Kelsey *et al.*, 1998) in the domain of web programming. Web programming covers both the WWW server side, the client side (browsers), and tools that generate web contents. We will primarily report on experience with Scheme programming in the domain of tools that generate static web contents, but we will also touch on CGI programming of web servers.

Almost any non-trivial web development effort involves some kind of programming side by side with use of a markup language (HTML or XML). In many contexts the source documents are written as a mix of fragments from a markup language and an imperative programming language. Such mixed source documents typically represent a clash between languages from two different cultures: The SGML culture (Bradley, 1997) and the culture of imperative programming. The main reason behind this state of affairs is a desire to separate the authoring of web contents from programming, not least because only relatively few web developers master both areas. We see three major problems with this mixed approach:

1. **The borderline problem**. Mixing markup and program fragments in a single document creates borderlines between two linguistic universes which cannot smoothly interact with each other. As a concrete example, server pages

in the style of ASP, PHP, and JSP make it difficult to apply the principle of abstraction as stated by Tennent (Tennent, 1981). (See Meijer's and van Velzen's discussion (Meijer & van Velzen, 2001)[Section 2.2] for a convincing argument and an example of this problem).

2. **The aesthetic problem**. The mixing of two languages in a single document gives a confusing impression, and it almost certainly eliminates any remaining rest of elegance in the source document.
3. **The imperative programming problem.** There is an evident misfit between imperative program fragments and fragments of HTML or XML that use *descriptive markup* (Coombs *et al.*, 1987).

The LAML approach contributes with solutions to all three problems. First, we eliminate direct use of the markup language in web documents by mirroring the elements of the markup language in the abstractions of a programming language. It implies that the markup aspects are made available through the abstractions of the programming language. With this, we provide for use of only a single language - the programming language. This eliminates the borderline problem mentioned above. As a consequence, any programmatic means of expression can be used together with even the finest details of the markup language. If the mirror is complete and accurate, this approach ensures that the full expressiveness of the markup language is kept intact at the programmatic level.

Second, we propose the use of a functional programming language instead of using an imperative language. The declarative style of functional programming fits well with the use of descriptive markup, which today dominates earlier use of procedural markup (which is more akin to commands in the imperative style). In the same vein, the nesting of markup elements has a natural counterpart in nested expressions, but it runs counter to the use of imperative commands, which cannot be nested in a similarly direct fashion.

By using the LAML approach, we maximize the utilization of programmatic means in web documents. As the best illustration, we write static documents directly in Scheme. Simple documents use almost exclusively the HTML mirror functions, whereas more complicated web documents draw on the advantages of programmatic solutions, such as conditional branching, organization of data in lists with accompanying iterations, and definition of abstractions with the goal of mastering the complexity of the document.

As it appears, we have gone for an inclusion of the markup language in the programming language by means of *mirroring* (to be explained in section 2.1.) If we compare markup languages with programming languages on the ground of computational power, the *weak language* has been mirrored in the *strong language*. We have rejected the idea of mixing the two languages due to the problems listed above. But yet a third relationship would be possible, namely an inclusion of programming capabilities in the markup language. With this, programmatic solutions can be expressed in the markup language without resorting to solutions, where pieces of foreign program fragments pollute the web document. We are aware of a few of attempts in this direction (Glickstein, 1999; Krishnamurthi *et al.*, 2000; Nicol,

2000)—all somehow related to Scheme. From a comparison of these with LAML it is safe to conclude that it is easier to subsume the markup language as abstractions in a functional programming language than the other way around.

Although Scheme has strong roots in the functional programming paradigm, Scheme is not a pure functional programming language. In our work on LAML, including the LAML-based tools and applications, we strive for solutions in the functional programming style. However, we have had at least two reasons to deviate from this course:

- **Adaptions to the surrounding imperative world.** Web applications are part of a context in which mutable state is a fact of live. Therefore it is not possible to ignore imperative solutions entirely in LAML-based applications.
- **Imperative patching of a functional program.** In some situations, a change of a functional program will be unreasonably complicated on functional ground, but straightforward if you introduce an imperative patch. In a number of the major LAML application, this is the main reason of the multiparadigmatic style found in these programs.

In the rest of the paper we will first—in section 2—describe how the markup language is made available as an "HTML mirror" in Scheme. In section 3 we discuss how to use the HTML mirror functions together with higher-order Scheme functions. In section 4 we discuss the role and the potential of abstraction with the goal of obtaining source documents at a higher level. This includes the definition of new domain specific Scheme-based languages. Section 5 gives a brief overview of LAML, regarded as a system of document styles, tools, and environmental support. The paper is finalized by an overview of similar work and a conclusion.

All the examples of the paper are available on-line (Nørmark, 2002) in the formats of LAML source documents, HTML target files, and as 'verbatim HTML files' (which allow the interested readers to view the details of the generated HTML documents).

2 Markup language mirroring

We use Scheme as the document source language for web pages and web sites. In other words, the source of a web page, or a set of interlinked web pages, is a Scheme program. Figure 1 shows an example of a web document written in Scheme by means of the HTML mirror functions. Notice that typical LAML documents would include other Scheme aspects. One of the main points of bringing HTML into a functional programming language is to use the potential of abstraction, such that the document can be handled at a higher level. We will return to this in section 4.

2.1 Basic mirroring

The markup language, such as HTML, is made available in the programming language by means of *mirroring*. Formally, a mirror μ maps each element of the markup language to a function in the programming language.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-html4.01-transitional-validating")

(write-html '(pp)
 (html
  (head
    (title "WEB Programming in Scheme - the LAML approach")
  )
  (body
    (h1 "WEB Programming in Scheme - the LAML approach")

    (p "The paper"
      (a 'href "http://www.cs.auc.dk/~normark/laml/papers/jfp.pdf"
         (em "WEB Programming in Scheme - the LAML approach"))
      "is written for people who are interested in functional programming."
      "The" (a 'href "abstract.html" "abstract") "is available as a separate page.")

    (p "The paper contains the following sections:")

    (ol
      (li "Introduction")
      (li "Markup language mirroring")
      (li "Programming with the HTML mirror functions")
      (li "Raising the level of abstraction")
      (li "LAML overview")
      (li "Reflections and similar work")
      (li "Conclusions"))

    (p "There exists other papers about LAML, such as:")

    (ul
     (li (a 'href "http://www.cs.auc.dk/~normark/laml/papers/www2002/p296-normark.html"
           "Programmatic WWW authoring using Scheme and LAML")))

    (p "Kurt Normark" (br) "normark@cs.auc.dk" (br)
       (a 'href "http://www.cs.auc.dk/~normark" "http://www.cs.auc.dk/~normark")))))
```

Fig. 1: *A sample web document written in Scheme with use of LAML.*

μ: Markup Element $\rightarrow$ Scheme Function

We have chosen to pre-apply μ on every element in the markup language, hereby creating a relative large number of Scheme functions (91 for HTML4.01 transitional, and 77 for XHTML1.0 strict) each of which we bind to a variable of the same name as the markup element. (In HTML this causes a single name clash, between the `map` element and the essential Scheme procedure `map`. The clash is handled by prefixing the mirror of the `map` element with "`html:`".) As an alternative we could generate the mirror functions on demand, and avoid the name bindings, but we find that this would blur the lexical and syntactical similarity between a LAML document and an HTML/XML document.

The following shows a sample application of μ on the HTML `a` anchor element:

μ (*the HTML* `a` *element*) = [`(generate-html-function "a" 'double)`]

The higher-order function `generate-html-function` generates the mirror function based on the tag name and the fact that it is an element with both start and end tag.

The parameter profiles of the generated functions, such as the functions `a`, `img`, and `p`, are chosen as close as possible to the counterparts in the markup language, with a few convenient generalizations and extensions. Basically and intuitively, the Scheme form

```
(tag 'a1 "v1" ... 'am "vm" contents)
```

corresponds to the HTML fragment

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

In the Scheme form `'a1` ... `'am` are symbols and `"v1"` ... `"vm"` are strings. The `contents` constituent represents zero, one or more contents elements in terms of strings or activations of mirror functions. The actual correspondence is richer and slightly more complicated, as reflected by the rules described below.

The generated mirror functions each returns a value, which we below will consider as a string. In the most recent version of the HTML mirror the mirror functions return abstract syntax trees (represented as nested lists) which eventually will to be transformed to strings. The function `render` performs this transformation.

The HTML mirror functions obey the following rules:

- ***Rule 1.*** An attribute name is a symbol in Scheme, which must be followed by a string that plays the role as the attribute's value.

- ***Rule 2.*** Parameters which do not follow a symbol are content elements (strings or instances of elements).

- ***Rule 3.*** All content elements are implicitly separated by white space.

With these rules, we see that the expression

```
(p (a 'href "http://www.dcs.glasgow.ac.uk/jfp"
     "Journal of" (em "functional programming")))
```

will be rendered as

```
<p>
  <a href = "http://www.dcs.glasgow.ac.uk/jfp">
    Journal of <em>functional programming</em>
  </a>
</p>
```

Here and in the following we will show manually pretty printed HTML fragments. HTML pretty printing is available as an option in the latest version of the mirror functions.

The mutual order of attributes and content elements do not matter as long as rule number 1 is obeyed. Thus, the expression

```
(p (a "Journal of" 'href "http://www.dcs.glasgow.ac.uk/jfp"
      (em "functional programming")))
```

gives the same result as shown above.

The rationale behind rule number 3 (white space between strings) is to support the most typical situation without use of additional elements. In the cases where we want to suppress white space we rely on the rule:

- ***Rule 4.*** A boolean false value (which we conveniently bind to a variable named underscore _) suppresses white space at the location where the boolean value appears.

Thus, the expression

```
(p "Use" (kbd "HTML") _ ","  (kbd "XHTML") _ ","
         (kbd "XML")_ "," "or" (kbd "LAML") _ ".")
```

suppresses white space before the punctuations.

In addition we support the rule:

- ***Rule 5.*** Every place an attribute or a content element is accepted we also accept a list, the elements of which are processed recursively and spliced into the result.

Thus, the following is a legal LAML expression

```
(ul (map li (list "one" "two" "three")))
```

which is rendered as

```
<ul><li>one</li> <li>two</li> <li>three</li></ul>
```

The following expression illustrates the versatility of Rule 5:

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

The result is rendered as:

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

As it appears, both fragments of the contents and fragments of the attribute lists may be represented and passed as lists side by side with singular contents elements and attributes. More examples and additional discussion of the consequences of Rule 5 are found in section 3.

Finally, the LAML mirror of HTML treats HTML attributes and CSS attributes (Cascading Style Sheet attributes (Bos *et al.*, 1998)) uniformly, via use of the following convention:

- ***Rule 6.*** An attribute with the name "css:*a*" refers to the *a* attribute in CSS.

Inline use of CSS attributes, as opposed to use of external style sheets, is quite useful when new layers of functions are created on top of the mirror functions. As an example that depends on Rule 6, the expression

```
(em 'css:background-color "yellow" "JFP")
```

is rendered as

```
<em style = "background-color: yellow;">JFP</em>
```

Without Rule 6, we should have used the following LAML expression

```
(em 'style "background-color: yellow;" "JFP")
```

which includes CSS attribute notation within the HTML `style` attribute.

2.2 Discussion of the mirror

As illustrated in the previous section (and further discussed in section 3 and 4) the use of LAML expressions in Scheme contributes with flexible authoring of web documents. As an additional advantage, the use of the mirror functions guaranties syntactic correctness (validity). This is due to the following properties of the mirror:

1. **Use of standard elements only.** There is no risk that the LAML author uses a non-standard HTML element. The reason is that the equivalent Scheme functions of such non-standard elements do not exist. The author will be aware of such a document anomaly when the document is processed.
2. **Assurance of well-formed results.** The generated HTML document will always be well-formed. Well-formedness ensures that elements, delimited by their start and end tags, are nested properly within one another. It is impossible to generate an ill-formed document by using the mirror functions of HTML. At the Scheme source level, the problem of ill-formed documents is concealed by the use of less redundancy (no end tags).
3. **Valid use of attributes.** The author will be warned if the HTML attributes are used inappropriately in a document. A warning is issued when the Scheme program is executed (at HTML generation time). The attribute check assures that all the required attributes are present, that the no illegal attribute names are used, and (to some degree) that the type of the attribute values are as specified in the DTD. It is not yet possible to check the validity of CSS attributes, because we currently have no detailed knowledge of CSS in LAML.
4. **Valid HTML composition.** Using the most recent mirrors of HTML in Scheme, the author will be warned or stopped if an invalid HTML document is generated. The validation is done on the ground of the element content models defined by the HTML document type definition (DTD).

The validation of the document against the DTD would be in vain if the textual content of the document was allowed to contain HTML tags. Instead of prohibiting the characters '<' and '>' in CDATA we translate them to the HTML character

entities denoted by `<` and `>`. The transliteration is carried out by means of a systematic mapping of every characters in the textual contents of a LAML document. The map is represented by the *HTML character transformation table*. Most entries in the table will be identity entries, but besides the characters mentioned above it is also useful to translate a variety of other characters (such as the three Danish national characters 'æ', 'ø', and 'å') to the corresponding HTML character entities. It is expected that LAML users customize the HTML character transformation table in the LAML init file (`.laml`).

We see that besides generating the underlying HTML fragments based on a flexible Scheme input syntax, the mirror functions are able to carry out substantial document checking 'on the fly'. Certain anomalies cannot occur at all, and others will be identified during the analysis process preceding the HTML synthesis phase. The actual amount of checking depends on a few boolean variables such as `check-html-attributes?` and `validate-html?`. Errors are reported through a procedure `check-error`, the default value of which just gives warnings on standard terminal output. Alternatively, the user can redefine `check-error` to be the Scheme procedure `error` in order to stop the generation process in case of validation problems. In section 2.3 we will discuss the creation of the mirror functions, including the implementation of the validation aspects of the mirror.

As it appears from the discussion in section 2.1 we use the run time types of Scheme objects to distinguish between attribute names, attribute values, content strings, list of content strings, and white space suppression. Anomalies are first discovered at run time. This is the usual and well-known consequence of 'dynamic typing' which makes it harder to find certain kind of errors in an early phase of the web document development phase. On the positive side, however, the Lisp and Scheme approach to handling of types creates an ideal ground for flexible passing of arbitrary parameters to a function. This has been of central importance to the creation of the HTML mirror functions in LAML, as described in section 2.1, and as such it has contributed to the development of the Scheme flavor of HTML, as provided by LAML. In addition, we are able to issue domain specific error messages because most error messages are controlled by the LAML software, as opposed to the type checker of the compiler.

The flexible handling of types is the underlying prerequisite which enables us to write expressions like

```
(p 'class "main"
   "This paper has the following paragraphs:"
   (map as-string (list 1 2 3 4)) _ ".")
```

where `as-string` converts its parameter to a string. The crucial observation is that interpretation of the actual parameters depends not only on their run time types, but also on the context in which they appear. To illustrate the latter point, the application

```
(p "The" "main" "part of this paper has the following paragraphs:"
    (map as-string (list 1 2 3 4)) _ "." 'class "main")
```

also passes the string `"main"` as the second actual parameter to `p`, but due to mirror rule number 1 (see section 2.1) the second parameter is part of the paragraph content, because the preceding parameter is not an attribute name.

The mirror of HTML in Scheme could alternatively be implemented by syntactic abstractions in terms of Scheme macros (Kelsey *et al.*, 1998). Using this solution, it would not be necessary to rely on the run time types of data objects to distinguish between content elements, attributes, and other elements. On the down side, a syntactic surface based on macros will not work well together with higher-order functions (cf. the discussion in section 3). Macros cannot play the role of functions when passed as input to, or output from higher-order functions. In addition, the macro concept of Scheme is not uniformly implemented in all major Scheme implementations, although it has been standardized in the most recent Scheme report (Kelsey *et al.*, 1998). As such, a mirror based on macros would make it harder to use LAML from many different Scheme systems.

As it has been illustrated by several examples above, we simulate a simple keyword parameter mechanism in the HTML mirror functions. The keyword is represented as a symbol, and the actual parameter of the keyword is the succeeding string. The keyword parameters are used to pass the HTML attributes names and values. The identification of the keywords is done at run time, and as such it adds an overhead to each call of an HTML mirror function which is linear in the length of the parameter list. The simulated keyword parameter mechanism can be seen as a simple variant of the Common Lisp's inherent keyword parameter mechanism (Steele, 1990).

As a practical aspect of LAML, document fragments are represented as strings which are passed as parameters to mirror functions and thereby aggregated to the overall document. As a concrete illustration, look at the Scheme expression

```
(p "The journal of" (em "functional programming")_".")
```

which will be rendered as the HTML fragment

```
<p>The journal of <em>functional programming</em>.</p>
```

If additional markup is introduced in the expression, such as

```
(p "The" (b "journal") "of" (em "functional programming")_".")
```

the string `"the journal of"` is to be split up in three single word strings of which the middle is nested in the `b` mirror function. This causes the following problems:

1. **The editing problem.** In the practical authoring situation it is error prone to handle the string quoting and the need of string splitting.
2. **The problem of lexical clutter.** The amount of "lexical clutter", primarily the string quotes, dominates the appearance of the expression. This affects the readability of the document.

The editing problem can be dealt with effectively by specialized editing commands, such as **embed**, which embeds a selected string in an application of a Scheme function. The **embed** editing command also handles the necessary string

```
(element "PRE" "-" "-"
  "(#PCDATA | TT | I | B | U | S | STRIKE | BIG | SMALL | EM | STRONG | DFN |
    CODE | SAMP | KBD | VAR | CITE | ABBR | ACRONYM | A | IMG | APPLET | OBJECT |
    FONT | BASEFONT | BR | SCRIPT | MAP | Q | SUB | SUP | SPAN | BDO | IFRAME |
    INPUT | SELECT | TEXTAREA | LABEL | BUTTON)*
     -(IMG|OBJECT|APPLET|BIG|SMALL|SUB|SUP|FONT|BASEFONT)" " preformatted text ")

(attribute "PRE" (("id" "ID" "#IMPLIED") ("class" "CDATA" "#IMPLIED")
  ("style" "CDATA" "#IMPLIED") ("title" "CDATA" "#IMPLIED")
  ("lang" "NAME" "#IMPLIED") ("dir" ("ltr" "rtl") "#IMPLIED")
  ("onclick" "CDATA" "#IMPLIED") ("ondblclick" "CDATA" "#IMPLIED")
  ("onmousedown" "CDATA" "#IMPLIED") ("onmouseup" "CDATA" "#IMPLIED")
  ("onmouseover" "CDATA" "#IMPLIED") ("onmousemove" "CDATA" "#IMPLIED")
  ("onmouseout" "CDATA" "#IMPLIED") ("onkeypress" "CDATA" "#IMPLIED")
  ("onkeydown" "CDATA" "#IMPLIED") ("onkeyup" "CDATA" "#IMPLIED")
  ("width" "NUMBER" "#IMPLIED")))
```

Fig. 2: *An element and attribute descriptor for the HTML* `p` *element.*

splitting. The editor command **embed** and other similar commands are available in Emacs, and they are discussed in more details in section 5. We see no good solution to the second problem within the context of LAML.

As already discussed, the HTML mirror functions return instances of abstract syntax trees. Eventually, these trees must be transformed to HTML or XML text (rendering). Earlier versions of LAML had the reputation of causing heavy garbage collection due to concatenation of lots of strings in the rendering process. In the most recent version of LAML we linearize the abstract syntax trees, either directly to an output stream (which is the best approach if the final target is a file) or into fixed segments of strings which finally are concatenated. It is worth noticing that this kind of rendering calls for imperative processing of the abstract syntax trees.

2.3 The creation and organization of the mirror

It would be a major challenge to manually create an accurate mirror of a given version of HTML or XHTML. In the LAML system, the mirror of HTML is created automatically from the document type definition (DTD), apart from some aspects of the full validation analysis which is explained below.

The accuracy of the HTML mirrors in Scheme depends on full syntactical knowledge of the HTML language, as it is represented in the DTD of a particular HTML version. As part of the preparation for LAML, we have written an ad hoc DTD parser which produces lists of *element and attribute descriptors*, represented as lists. Figure 2 shows and example such descriptors. As one of the main assets of the descriptors, the much used character entities (textual macros) in the HTML DTDs are unfolded, such that the full information about the elements are available in single, convenient representation to be used by the mirror generation tools.

Given the unfolded list representation of the DTD, it is relatively straightforward to automatically synthesize all the Scheme `define` forms of the mirrors. We also

```
(define (html4:em contents  . attributes)
  (let ((attributes-of-elements attribute-descriptor)
        (req-n 0))
    (if check-html-attributes?
       (check-attributes! attributes attributes-of-elements req-n "em"))
    (if validate-html?
       (validate-contents!
         contents
         (zero-or-more "#pcdata" "tt" "i" "b" ...)
         "em"))
    (if (not contents)
        (display-warning ...))
    (internal-ast-node "em" contents attributes)))
```

Fig. 3: *An outline of the basic mirror functions of the* `em` *element.*

generate a substantial amount of useful documentation extracted from the DTD; This information is processed by the SchemeDoc tool (see section 5) and presented as manual pages. The validation of the attributes is also easy to deal with on the ground of the attribute descriptors.

The validation of the HTML document composition is the most difficult part of the analysis. In general it is known to be difficult to automatically produce validators from the grammatical model of a DTD. We have chosen an approach where the easy and most frequently occurring cases are handled automatically. The remaining cases are left to special purpose checkers, which we write specifically for a particular mirror.

The validation of the HTML composition is based on the content models of the element descriptors (corresponding to 'right hand sides of productions' in context free grammars). In the HTML 4.01 transitional DTD, the majority of the content models (58 out of the 78 non-single elements) are on one of the forms:

```
"(X | ... | Y)*"
"(X | ... | Y)+"
```

From the element descriptor shown above it can be seen that the content model of the `pre` element is more complicated. As part of the DTD parsing, we transform the simple content model strings to the lists

```
(zero-or-more "X" ... "Y")
(one-or-more "X" ... "Y")
```

respectively. Based on these clauses it is easy to automatically synthesize checking predicates for these simple elements. The validation of the remaining HTML elements, such as `pre`, is done manually by writing predicates for each of these. We had to write 20 such predicates for HTML4.01.

The DTD of HTML4.01 is a context sensitive grammar which uses both general *inclusions* and *exclusions* (Bradley, 1997). Thus, it may be specified that a certain element is generally allowed or prohibited in a given HTML fragment. The LAML

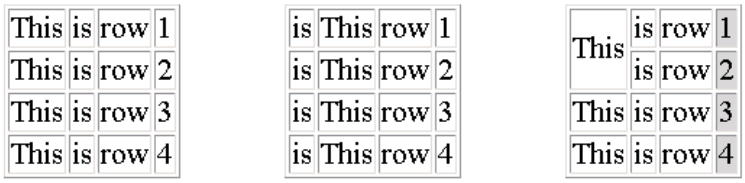

Fig. 4: *Three tables produced in section 3.*

validator handles exclusions, but not inclusions. As a consequence, the rarely used `ins` and `del` elements, which a generally allowed in `body` elements, are not properly dealt with by the LAML HTML validator. As a practical consequence, there will be issued 'false warnings' when `ins` or `del` elements are encountered within a `body` element.

The HTML mirror functions are organized in two library files: the *surface mirror* and the *basic mirror*. Most users will only be interested in the surface level (which is the one described in section 2.1), but for efficiency reasons some applications (such as CGI programs) can profit from the underlying basic mirror. The surface mirror identifies attributes and contents elements and passes these to the underlying basic mirror. Figure 3 shows an outline of basic mirror function.

3 Programming with the HTML mirror functions

In this section we will illustrate applications of the HTML surface mirror together with higher-order Scheme functions.

The HTML `table` element is an important element, not only for tabular presentations, but also for more complicated typographical arrangements (despite recommendations to avoid such usage in recent HTML specifications). Basically, a table is composed of a number of `tr` table row element instances inside which each cell is nested in a `td` element instance.

In a programmatic context, it is attractive to represent a table as an appropriate data structure instead of authoring a table with plain nesting of `tr` and `td` element instances. Using a Lisp language, it is natural to represent a table as a list of rows, like

```
(list
 (row "This" "is" "row" "1")
 (row "This" "is" "row" "2")
 (row "This" "is" "row" "3")
 (row "This" "is" "row" "4"))
```

where `row` is an alias of the `list` function. We will in the following assume that the variable `sample-table` is bound to this structure. The table can be rendered as HTML4.01 by the following expression:

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

The result is shown as the leftmost table in Figure 4. The higher-order function `compose` combines a number of one-parameter functions to a single, aggregated function. The function `(compose tr (map td))`, which is applied on each row in the table, embeds the elements in the necessary `tr` and `td` element instances. The simplicity of the table rendering in Scheme and LAML depends critically on Rule 5 of mirror, which allows us to pass lists of contents elements to the HTML mirror functions (see section 2.1). The table expressions returns a HTML fragment which is rendered as

```
<table border="1">
 <tbody>
   <tr><td>This</td> <td>is</td> <td>row</td> <td>1</td> </tr>
   <tr><td>This</td> <td>is</td> <td>row</td> <td>2</td> </tr>
   <tr><td>This</td> <td>is</td> <td>row</td> <td>3</td> </tr>
   <tr><td>This</td> <td>is</td> <td>row</td> <td>4</td> </tr>
 </tbody>
</table>
```

Notice here that the Scheme `map` function, which requires two or more parameters (a function an a number of lists) is generalized to accept only a single parameter (the function). With this, an expression like `(map td)` is a `td` mapper. To obtain this generalization we redefine `map` as

```
(define map (curry-generalized map))
```

where `curry-generalized` is a higher-order function (part of the *general LAML library*) which performs ad hoc currying of a function, which normally requires at least two parameters:

```
(define (curry-generalized f)
 (lambda rest
  (cond ((= (length rest) 1)
            (lambda lst (apply f (cons (car rest) lst))))
        ((>= (length rest) 2) (apply f rest)))))
```

In order to illustrate the flexibility of handling tables as lists of rows, we will assume that we decide to switch the first and second column of the table. Instead of re-arranging the table as such, we write the function `switch` which does the job:

```
(define (switch row-lst)
 (cons (second row-lst)
  (cons (first row-list)
   (cddr row-list))))

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

The result is shown in the middle table of Figure 4.

The versatility of the Scheme HTML mirror allows us to exploit the HTML `td` attributes of the table cells directly. Here is an example where we colorize the background of the numeric cells, and where we join two cells in the upper leftmost corner of the table:

```
(define (grey-numeric x)
 (if (and (string? x) (numeric-string? x))
     (list x 'bgcolor (rgb-color 200 200 200))
     x))

(table
 'border "1"
 (tbody
  (map (compose tr (map (compose td grey-numeric)))
       (list
        (row (cell "This" 'rowspan "2") "is" "row" "1")
        (row                            "is" "row" "2")
        (row "This"                     "is" "row" "3")
        (row "This"                     "is" "row"  "4")))))
```

The `cell` function is again just an alias of `list`. The resulting table is shown as the rightmost table of Figure 4.

It is often useful to define a variant of an HTML mirror function which binds certain attributes to fixed values. This can be done by use of the higher-order function `modify-element`. Let us, as an example, assume that we wish to bind the `target` attribute of the `a` element to the fixed value `"main"` and the `title` attribute to a fixed explanation:

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

The higher-order function `modify-element` can be defined as

```
(define (modify-element element . attributes-and-contents)
 (lambda parameters
  (apply element (append parameters attributes-and-contents))))
```

From this we see that we can also bind parts of the content elements if this should turn out to be useful. With this function

```
(a-main 'href "http://www.dcs.glasgow.ac.uk/jfp" "JFP")
```

will be rendered as

```
<a href = "http://www.dcs.glasgow.ac.uk/jfp"
   target = "main" title = "Goes to the main window">JFP</a>
```

In conclusion we find that the combination of HTML mirror functions in LAML and higher-order functions (general as well as more specialized ones) provide elegant and powerful solutions to Scheme programmers, who do web development with LAML.

4 Raising the level of abstraction

In the previous section we have illustrated that it is possible to write HTML documents by using the HTML mirror functions together with a number of higher-order functions. As pointed out already in section 2, the real potential of the LAML approach is to develop layers of abstraction on top of the HTML mirror functions. We will now discuss this key aspect of LAML.

4.1 Simple ad hoc abstractions

Let us use the example from Figure 1 as a starting point. In section 2 we observed that it is unlikely that we will write documents with pure HTML markup in Scheme syntax. We now introduce a number of simple abstractions leading to the document source shown in Figure 5. The applied abstractions can be summarized as follows:

1. The function `html-document` implements the standard document preamble (including application of the `html`, `head`, `title`, and `body` elements) together with an application of the `h` form with the same content as the document title. Use of this function ensures a proper wrapping of the main contents into a plain and simple HTML envelope.
2. The function `a-href` which turns the anchor `a` element into a function (with positional parameter correspondence) of exactly two parameters. Use of this function ensures proper use of an URL `href` attribute in the `a` element, which is not a required HTML attribute.
3. The function `kn`, which is the author's signature function that returns his name, email address, home page, etc. This function allows the typical document trailer to be defined once and for all. The LAML init file, `.laml`, is the natural location of this function.

Each of these functions adds a bit of convenience for the web author. Seen together the use of such functions ease the task of the practical web author. Some of the functions are simple "one shot" functions to be used in a single document only. Many of the functions, however, are generally useful and can therefore be organized in libraries which are loaded initially. We have accumulated a substantial collection of such functions, and organized them in the so-called *LAML convenience library*. We realize that many of the convenience functions are primarily valuable for the author who conceived them, and we therefore recommend the definition of personal LAML convenience collections.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-html4.01-transitional-validating")

(define (html-document ttl . real-body)
 (html (head (title ttl)) (body (h1 ttl) real-body)))

(define (laml-paper-url suffix)
  (string-append "http://www.cs.auc.dk/~normark/laml/papers/" suffix))

(define (a-href url anchor-text)
  (a 'href url anchor-text))

(write-html '(pp)
 (html-document
   "WEB Programming in Scheme - the LAML approach"

    (p "The paper"
       (a-href (laml-paper-url "jfp.pdf")
        (em "WEB Programming in Scheme - the LAML approach"))
       "authored by Kurt Nrmark is written for people who are
        interested in functional programming. The"
        (a 'href "abstract.html" "abstract") "is available as a separate page.")

    (p "The paper contains the following sections:")

    (ol
      (li "Introduction")
      (li "Markup language mirroring")
      (li "Programming with the HTML mirror functions")
      (li "Raising the level of abstraction")
      (li "LAML overview")
      (li "Reflections and similar work")
      (li "Conclusions"))

    (p "There exists other papers about LAML, such as:")

    (ul
     (li (a-href (laml-paper-url "www2002/p296-normark.html" )
                 "Programmatic WWW authoring using Scheme and LAML")))

    (kn)))
```

Fig. 5: *A sample LAML document with convenient abstractions.*

4.2 Domain specific Lisp languages

Development of complex web pages and sites call for use of more powerful abstraction ideas than definition of a few convenience functions that help out at designated locations in a single HTML document. In these situations it is attractive to design a new language with means of expressions that fit well with the concepts of the domain in question. In this section we will discuss and give examples of domain specific web language designed on the ground of s-expressions and parenthesized prefix notation ala Lisp and Scheme.

One of the most substantial LAML-based languages, called LENO, has been developed for the domain of web-based teaching material. LENO is described and

```
(note-page 'laml-basics
  (title "LAML basics"
         "On this page we describe LAML briefly")

  (point
   "LAML brings HTML and XML to the Scheme programming language"
   "LAML is software package that supports authoring of HTML
    and XML documents in Scheme Syntax.")

  (concept-list
   (concept "LAML"
            "LAML is a Lisp Abstracted Markup Language"
            "LAML consists of HTML mirrors, document styles, and tools"))

  (items
   (item "LAML Characteristics"
         "We here mention a number of important LAML characteristics"
         (items
          (item "Supports a number of different HTML mirrors"
                ""
                (items
                 (item "LAML 4.0 loose.
                        XHTML1.0 strict, transitional, and frameset")))

          (item "Supports a number of document styles"
                ""
                (items
                 (item "Domain specific WEB languages")
                 (item "LENO, Scheme manual pages, questionnaire, ..."))))))

  (index-words "LAML" "mirror" "LENO"))
```

Fig. 6: *A LENO note page in the original syntax*

discussed in separate papers (Nørmark, 2001b; Nørmark, 2000c). Figure 6 shows an example of a LENO source document fragment. The fragment represents a single note page with with `title`, `point`, `concept-list` and `index-words` subclauses. A note page gives rise to a number of different underlying HTML pages that represent the note page at different levels of abstractions.

There are several different ways to implement a domain specific web language in Scheme:

1. **Via functional abstraction in Scheme.** Each new language construct is defined by a function. As an implication, each constituent of an expression is evaluated uniformly and eagerly.
2. **Via syntactical abstraction in Scheme.** Each new language construct is defined by a Scheme macro. With this approach, the surface syntax can be defined more freely, and without uniform evaluation of all constituents of a language construct.
3. **Via interpretation of a new list-based language.** The new language is implemented by an interpretor written in Scheme. Using this approach, the language designer has full freedom to design the language as wanted.

Using the first and second approach the interpretation is done by the Scheme processor, and as such it is trivial to mix Scheme fragments with fragments of the domain specific web language. This provides for a flavor of web authoring which we call *programmatic authoring* (Nørmark, 2001a). Using these approaches, programmed solutions can be used anywhere in a web document, and it can be used at any time during the development process.

LENO and other similar LAML-based web languages have all been implemented via functional abstraction in Scheme. The surface syntax of a concept `c` is typically established by a function of the form:

```
(define (c . p) (make-element 'c p))
```

where `make-element` creates a tagged list structure with the given constituents, as bound to the formal rest parameter `p`. Functions applied at the outer level, such as `note-page` in LENO, take responsibility of interpreting the nested and tagged structures.

The original version of LENO, as illustrated in Figure 6, uses surface functions with positional parameter correspondence, and the surface functions accept only a fixed number of parameters. Recently, we have reengineered LENO to support a more flexible syntax similar to the surface syntax of the HTML mirror functions (see section 2.1). This syntactical surface is called *XML-in-LAML*, and as the name suggests the syntax is compatible with the generic XML syntax (Consortium, 1998). Figure 7 shows the example from Figure 6 in XML-in-LAML syntax.

The XML-in-LAML syntax is more flexible with respect to support of additional attributes than the original surface syntax of LENO. On the downside, the format is more verbose than the original LENO format, and as such it calls for definition of extensive editor templates to be competitive with the more purified Scheme syntax. Also, it is quite elaborate to introduce lists of subclauses in an XML-in-LAML document. As an example from LENO, we denote a list of index words as

```
(index-words
  (index-word "first")
  (index-word "second"))
```

We need to define the element `index-word` to accommodate individual index words. If we accept Lisp lists as a separate means of expression, we could just write

```
(index-words (list "first" "second"))
```

This, however, would harm the one-to-one correspondence between XML and XML-in-LAML.

If desired, it is straightforward to translate a LENO XML document to the internal document representation of LENO, using an XML parser. It is not part of our plans to utilize this possibility. We find it more attractive to author web documents in the context of Scheme (programmatic authoring) than in the static poverty of XML.

```
(note-page 'id "laml-basics"
  (title
    (main-text "LAML basics ")
    (annotation "On this page we describe LAML briefly"))

  (point
    (main-text "LAML brings HTML and XML to the Scheme programming language")
    (annotation "LAML is software package that supports authoring of HTML
                 and XML documents in Scheme Syntax."))

  (concept-list
    (concept
      'concept-name "LAML"
      (main-text "LAML is a Lisp Abstracted Markup Language")
      (annotation "LAML consists of HTML mirrors, document styles, and tools")))

  (items
     (item
      (main-text "LAML Characteristics")
      (annotation "We here mention a number of important LAML characteristics")
      (items
        (item
          (main-text "Supports a number of different HTML mirrors")
          (items
            (item (main-text "LAML 4.0 loose.
                              XHTML1.0 strict, transitional, and frameset")))
        )
       (item
          (main-text "Supports a number of document styles")
          (items
            (item (main-text "Domain specific WEB languages"))
            (item (main-text "LENO, Scheme manual pages, questionnaire, ..."))))))))

  (index-words (index-word "LAML" )  (index-word "mirror") (index-word  "LENO")))
```

Fig. 7: *The LENO note page from Figure 6 using the XML-in-LAML syntax.*

Using the XML-in-LAML syntax we can think of our documents as XML documents in slightly different surface syntax than that of an SGML language. Most differently, we have to accept that all string contents are passed as quoted strings. But as for LAML documents using the HTML mirror functions (like in Figure 1) we do not want to stay at this level. We wish to mix programmatic means (not least higher-order functions) with the XML-in-LAML functions, exactly as illustrated in section 3, and we also wish to be able to introduce ad hoc abstractions, like in section 4.1.

It is worth a consideration what should be the value of the `note-page` expression in Figure 7 or one of its subexpressions. LENO takes a very pragmatic stand on this issue, because the `note-page` expression causes creation of a number of underlying HTML pages, or contributions to HTML pages. As such `note-page` is not a function, but a procedure. Thus, in LENO, the implementation of the XML-in-LAML abstractions directly realizes a non-trivial transformation from the domain-specific language to a set of HTML files. Alternatively, we could chose an intermediate lan-

guage as target of the XML-in-LAML functions, much like the abstract syntax trees generated by the validating HTML mirror functions (see section 2.1). Such a representation would serve as a canonical representation of the source document, where evaluation of ad hoc abstractions and outer higher-order functions have already taken place.

5 LAML Overview

In this section we will in relative brief terms describe the LAML system, with special emphasis on the aspects that do not pertain to the HTML mirrors.

It has been a goal to support the LAML system on a number of major platforms, operating systems, and not least Scheme Systems. LAML assumes compliance with the fourth revised Scheme Report, R4RS, which is supported by almost all implementations of Scheme. LAML comes with its own *general library* of functions, instead of relying on one of the non-standard set of libraries, such as SLIB (Jaffer, 2002) or PLT's libraries (Flatt, 2000). In order to use LAML it is necessary somehow to implement between 6 and 9 non-standard functions and procedures (most important `current-time`, `sort-list`, `file-exists?`, `directory-exists?`, `delete-file`, and `copy-file`). Most Scheme systems already support these functions, perhaps using other function names or parameters. LAML is configured to a given platform, operating system, and Scheme system by running a Scheme installation program, which takes a *configuration description* (in terms of an association list) as input.

5.1 LAML document styles

A LAML document style represents a domain specific Lisp language, as discussed in section 4.2. We have already described the LENO document style (see section 4.2). The *manual document style* is a little language for interface description of Scheme libraries. The manual document style is used together with the SchemeDoc tool (see section 5.2) and as such it plays a central role for the documentation of the LAML libraries. The *questionnaire document style* is another little language for formulation of questionnaires on the web. The questionnaire document style is accompanied by a set of CGI programs for registration and presentation of the questionnaire answers at different levels of abstractions. The *course home page document style* supports a high level description of a series of lectures in a university course (Nørmark, 2000c). The processing of the course home page document produces a course calendar, a number of course overviews, and lecture specific pages.

5.2 LAML tools

The LAML system supports a number of web related and Scheme related tools. The *Scheme Elucidator* is a LAML based tool for Elucidative Programming (Nørmark, 2000b; Nørmark, 2000a) such as documentation of internal aspects of a Scheme program. *SchemeDoc* is a tool that extracts interface comments from a library of

Scheme procedures. As the name indicates, SchemeDoc is similar to the JavaDoc tool (Friendly, 1995). SchemeDoc works in concert with the manual document style. The *calendar tool* generates a web calendar. The calendar tool is based on a LAML time library. The LAML *Bibtex tool* is able to parse simple bibtex files (Lamport, 1986) to association lists, and to render these as HTML fragments. LAML also comes with *XML and HTML parsing and pretty printing tools* and a *Scheme pretty printing tool*. In addition, there are a number of internal LAML tools for DTD parsing and mirror generation.

5.3 Environmental support

Execution of a LAML program is plain and normal Scheme execution with a tiny bit of environmental information defined. The environment information amounts to the current directory and the name of the current source file. In addition, the LAML Scheme interpretator must know the location of the LAML installation (`laml-dir` which is an absolute path to the LAML directory) such that a central file `laml.scm` file can be loaded.

LAML can be activated from the operating system's command prompt, from an interactive Scheme prompt, and from Emacs. For Emacs users, the latter possibility is the most attractive; Via the "laml" file extension and an Emacs mode, LAML files can be processed asynchronously and synchronously by different Emacs commands, such as **laml-process-current-buffer**. This command is bound to a single keystroke (defaulted to `C-o`) and it is also available in a menu of LAML related commands. Within Emacs it is also possible to start an interactive LAML session with the command `run-laml-interactively`. This defines the environmental information, and it loads the HTML mirror functions and other useful libraries.

5.4 CGI Programming in LAML

We have done a large amount of CGI programming in Scheme using the LAML CGI libraries together with the HTML mirror libraries. As a historical remark, LAML was initiated with the purpose of supporting CGI programming in Scheme. Since then we have realized that LAML is useful for production of static web pages as well.

The most substantial CGI application is a distance education environment (IDA-FUS) which has been in daily use at the Computer Science Department of Aalborg University since the fall of 1999. The LAML CGI libraries provide for decoding of URL encoded and multipart encoded data, as submitted via the so-called POST method in the CGI. The decoded data are represented as association lists in Scheme. (As a side effect, uploaded files are copied to a destination in the server's file system). Similarly, association lists can be linearized and URL encoded via a function in the CGI libraries.

The CGI support of LAML is fairly simple and straightforward. It has not been a goal to hide the details of the CGI protocol for the LAML programmer. In that respect, the LAML CGI support runs counter to Meijer's CGI framework in Haskell

(Meijer, 2000), which elegantly protects the a Haskell programmer from most idiosyncrasies of the Common Gateway Interface.

6 Reflections and Similar Work

We will first discuss similar work in the area of Haskell, ML, Erlang, and Curry. Following that we will discuss similar work related to Scheme. We conclude with some reflections on static versus dynamic typing.

6.1 Web Programming in Haskell and related languages

Wallace and Runciman (1999) discuss two different representations of XML documents in Haskell. The first is based on a generic tree representation of XML documents. The second 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. The authors contribute with a collection of higher-order functions (combinators) that are intended to ease the processing of XML documents from Haskell. The LAML approach is located in between the two approaches introduced by Wallace and Runciman. The mirror functions in LAML are primarily oriented towards HTML, and they are derived from the DTD. XML-in-LAML documents are currently not derived from a DTD. In addition, LAML is not geared towards general transformation of XML documents, but rather towards specific transformation of XML-in-LAML documents to HTML.

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). As already mentioned in section 5.4, the Haskell CGI framework hides the low level CGI details from the Haskell programmer. The paper also presents a modelling of HTML (similar to the generic tree representation of Wallace and Runciman) together with a rudimentary layer of HTML surface syntax (called 'HTML combinators'). In comparison, the LAML CGI support is at a lower and more basic level, but the HTML modelling in LAML is more advanced and complete than Meijer's.

In a second paper, Meijer and Shields (2000) define a new language called XMλ which is indented for generation of dynamic XML documents. As described in the paper, XMλ is not yet completely defined. 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 in which program fragments are escaped. Program fragments are expressed in a language similar to Haskell, but as an important extension, verbatim XML fragments are part of the language. In comparison, LAML is based on programmatic notation, and textual contents are passed as quoted strings. We have in this paper described the virtues of this approach, and we have argued against a mixing of HTML/XML fragments and (escaped) program fragments.

In a third paper, Meijer and van Velzen (2001) describe HSP (Haskell Server

Pages). HSP is similar to ASP, PHP, JSP, and others. As such, HSP documents are HTML/XML documents with escaped Haskell expressions. In contrast to ASP, it is possible in HSP to mix HTML/XML and Haskell fragments in ways that preserve the principle of abstraction (HSP documents are compositional). Technically, XML fragments are added as atomic expressions and patterns to Haskell. As such, HSP builds on the same ideas as XMλ.

Thiemann (2000) describes another modelling of HTML in Haskell. Each HTML element and each HTML attribute correspond to its own datatype in Haskell. The modelling is based on type classes in Haskell and an overloaded `add` function which aggregates HTML fragments into each other. As such, the syntactical composition of HTML documents does not resemble the syntax of HTML. Thiemann's approach always gives well-formed HTML documents, but it is not powerful enough to guaranty full document validity on a static type checking basis. The derivation of the numerous datatypes for elements and attributes calls for an automatic derivation of these from the DTD of HTML. As reported in the paper, this work has not yet been completed.

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. As the main contribution in this work, HTML `form` expressions and the handling of `form` input are described together. This is a contrast to the fragmentation of conventional CGI web programs. Hanus also shows how to make use of aspects from the logical programming paradigm. High level Curry web programs are transformed automatically to programs that use the CGI.

With respect to SML, we are aware of Neumann's fxp parser for XML (Neumann, 1999). In addition, Sestoft and colleagues have implemented ML server pages (Sestoft, 2002), which is similar to ASP, JSP, PHP, and HSP. In the functional programming language Erlang, XMerL provides an Erlang modelling of XML (Wiger, 2000).

6.2 Web Programming in Scheme

It is interesting to notice that Scheme via DSSSL has played a relatively early role in the processing of SGML document. DSSSL (which means Document Style Semantics and Specification Language) is an ISO standard for specifying document transformation and formatting.

BRL is a language designed for server-side WWW-based applications (Lewis, 2000). BRL allows the WWW author to activate Scheme at designated places in an HTML document. The places are identified with square brackets. As such, a BRL program mixes fragments of HTML with fragments of Scheme. The Scheme program fragments within the square brackets are executed on the WWW server, using a slightly non-standard Scheme semantics. BRL is particularly strong with respect to access of a relational database on the server side.

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 mirrors a language similar to Scheme

in the markup framework. This means that it is possible to make programmatic contributions to a Latte document by writing Scheme definitions in a Latex syntax.

As an alternative to XSLT (Adler, 2000) Krishnamurthi et al. (2000) have proposed a similar XML transformation framework called XT3D. The XT3D work is based on the idea of 'transformation by example', which in turn is rooted in the work on a macro facility for Scheme done by Kohlbecker (1986). Like XSL and XSLT, the languages involved in XT3D are all XML languages with Scheme used at an internal level.

Scheme has been used in other web programming contexts as well. Queinnec (2000; 2001) uses the concept of continuations to support sessions on the web server. Instead of composing a server program of many state-less CGI programs, Queinnec recommends the use of a single Scheme program which in a coroutine-like fashion can be resumed when input is received from the client. Resumption points are handled by means of continuations. The major challenge in this work is to provide for persistence of the continuations on the server. Graunke et al. (2001a) work on a similar problem in the context of the PLT Scheme system. In their work, however, the goal is automatically to transform an interactive program to a set of CGI programs. Like in Queinnec's work the continuation concept plays an important role, but the programmer is not required to use it explicitly. As an interesting variation, Graunke et al. rely on persistent continuations kept at the client side.

Besides CGI related work, Scheme is also used for more dedicated WWW servers. The PLT group has demonstrated that excellent performance can be obtained by a WWW server written exclusively in Scheme (Graunke *et al.*, 2001b). As the downside of this approach, the services provided by a more conventional web server are not available. As a consequence of this observation we work on Scheme-based LAML module called SLAML (Hansen *et al.*, 2002) for the Apache web server.

6.3 Static versus dynamic typing

Scheme's use of dynamic typing stands as a contrast to static typing, as used in most other functional programming languages.

Although we fully acknowledge the ideals of *earliest possible identification of errors* we would like to point out that

- even with static type checking there will most likely be other kinds of errors that cannot be identified before run time (test time). Erroneous use of data relative to the static types of functions is a relatively trivial problem compared to these more severe errors.
- functions with static type checking provide for less flexibility than functions in Scheme. It would be difficult to mimic the conventions of the HTML Scheme mirror functions in an existing functional programming language, and both keeping the advantages of static type safety and notational elegance.

Taking an overall look the web-oriented literature that relates the functional programming paradigm, we find that the problem of type checking is overemphasized. Doing web work in a functional context is oriented towards the creation of high

quality web material; It is not entirely a game related to finding errors as early as possible in the development process.

Due to the use of 'dynamic typing' the LAML author must be aware that some type errors are identified at a late point in time, and that there is a risk that type errors will remain in the software. From the experience of the author (who have used LAML extensively and exclusively for all his web work the last four years) this has never been a major problem—at least not compared to the other problems which inevitably are experienced when dealing with development of non-trivial software.

7 Conclusions

With the use of LAML, the Scheme programming community can take advantage of the ideas of functional programming, both for everyday web production needs and for more specialized and demanding web production tasks. The latter calls for definition of domain-specific Scheme-based languages that parallels the definition of XML languages. Use of LAML brings functional programming power to the fingertips of the web developer at every location of the document, and at every time in the development process. Using the Emacs LAML support, the practical processing of a LAML document is conveniently streamlined. Using XML as an alternative, the computational power needs to be brought in via use of external tools. Such external tools typically apply new, limited and specially developed programming frameworks, such as XSL (Adler, 2000), as an alternative to well-proven, general purpose languages.

As a characteristic property, LAML makes markup aspects available through the means of the programming language. The use of the functional paradigm is in good accord with the descriptive nature and ideals of most markup languages. A LAML user is a programmatic author because the document source is an ordinary Scheme program. The LAML server side programmer is also writing Scheme programs, serving in another context than a 'static LAML document'. Most other systems make use of a mixed approach. Typically, the outer context is an XML or HTML document, in which pieces of programs are surrounded by particular tags.

In server side programs, we frequently encounter an outer context of program constructs in which HTML fragments are located as strings. Imperative server programs bring this to an extreme in which prefix or suffix parts of well-formed HTML clauses are printed in `print` commands. In this paper we have argued against the mixing of programming notation and markup notation.

In the other extreme there exist programmatic frameworks that are moved into the XML markup language. XSL (Adler, 2000) is a well-known example of functional nature. The XT3D work (Krishnamurthi *et al.*, 2000) mentioned in section 6.2 and the Lisp inspired SEXPR proposal (Nicol, 2000) are other examples. Such languages run counter to our aesthetic desires of a good programming notation, in part because of the verbosity implied. Moreover we are convinced that SGML was never envisioned as a syntactical framework for programming languages.

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

References

Adler, Sharon. 2000 (November). *Extensible stylesheet language (XSL) version 1.0.* Tech. rept. W3C.

Bos, Bert, Lie, Håkon Wium, Lilley, Chris, & Jacobs, Ian. 1998 (May). *Cascading style sheets, level 2 CSS2 specification.* Tech. rept. W3C.

Bradley, Neil. (1997). *The concise SGML companion.* Addison-wesley.

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

Coombs, James H., Renear, Allen H., & DeRose, Steven J. (1987). Markup systems and the future of scholarly text processing. *Communications of the ACM*, **30**(11), 933–947.

Eugene E. Kohlbecker, Jr. 1986 (August). *Syntactic extensions in the programming language lisp.* Ph.D. thesis, Indiana University. Tecnical Report no. 199.

Flatt, Matthew. 2000 (August). *PLT mzscheme: Language manual.* http://www.cs.rice.-edu/CS/PLT/packages/pdf/mzscheme.pdf.

Friendly, Lisa. (1995). The design of distributed hyperlinked programming documentation. Frass, Sylvain, Garzotto, Franca, Isakowitz, Toms, Nanard, Jocelyne, & Nanard, Marc (eds), *Proceedings of the international workshop on hypermedia design (iwhd'95), montpellier, france.*

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

Graunke, Paul, Findler, Robert Bruce, Krishnamurthi, Shriram, & Felleisen, Matthias. (2001a). *Automatically restructuring programs for the web.* Available from http://www.ccs.neu.edu/scheme/pubs/.

Graunke, Paul, Krishnamurthi, Shriram, der Hoeven, Steve Van, & Felleisen, Matthias. (2001b). Programming the web with high-level programming languages. *Pages 122–136 of:* Sands, D. (ed), *10th european symposium on programming, ESOP 2001.* Lecture Notes in Computer Science, no. 2028. Springer Verlag.

Hansen, Mikael, Iversen, Paw, & Juncker, Jimmy. 2002 (January). *SLAML - server side LAML.* Preliminary master thesis report. Available from `http://www.cs.auc.dk/-~normark/laml/`.

Hanus, Michael. (2001). High-level server side web scripting in Curry. *Pages 76–92 of:* Ramakrishnan, I.V. (ed), *Practical aspects of declarative languages, lncs 1990.* Lecture Notes in Computer Science. Third International Symposium, PADL 2001, Las Vegas, Nevada: Springer Verlag.

Jaffer, Aubrey. (2002). *SLIB - the portable scheme library version 2d3.* http://www-swiss.-ai.mit.edu/~jaffer/slib.pdf.

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

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

Lamport, Leslie. (1986). *Latex user's guide and reference manual.* Addison-Wesley Publishing Company.

Lewis, Bruce R. 2000 (October). *BRL—a database-oriented language to embed in HTML and other markup.* Located on 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/.

Meijer, Erik, & van Velzen, Danny. (2001). Haskell server pages - functional programming and the battle for the middle tier. *Electronic notes in theoretical computer science 41, no. 1.* Elsevier Science B.V. Available via http://www.elsevier.nl/locate/entcs/volume41.html.

Neumann, A. (1999). fxp - Processing Structured Documents in SML. Trinder, P., & Michaelson, G. (eds), *1st Scottish Functional Programming Workshop, Draft Proceedings.* Herriot-Watt University, Edinburgh, Scotland.

Nicol, Thomas. 2000 (November). *XEXPR - a scripting language for XML.* W3C note located at http://www.w3.org/TR/.

Nørmark, Kurt. (1999a). *The Elucidative Programming Home Page.* `http://www.cs.auc.dk/~normark/elucidative-programming/`.

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

Nørmark, Kurt. (2000a). Elucidative programming. *Nordic journal of computing*, **7**(2), 87–105.

Nørmark, Kurt. 2000b (May). An elucidative programming environment for Scheme. *Pages 109–126 of: Proceedings of NWPER'2000 - nordic workshop on programming environment research.* Available via (Nørmark, 1999a).

Nørmark, Kurt. (2000c). A suite of WWW-based tools for advanced course management. *Pages 65–68 of: Proceedings of the 5ht annual sigcse/sigcue conference on innovation and technology in computer science education.* ACM Press. Also available from `http://www.cs.auc.dk/~normark/laml/`.

Nørmark, Kurt. 2001a (November). *Programmatic WWW authoring using Scheme and LAML.* To be presented at the web engineering track of WWW2002. Also available via (Nørmark, 1999b).

Nørmark, Kurt. 2001b (November). *Web based lecture notes - the LENO approach.* Submitted for publication. Also available via (Nørmark, 1999b).

Nørmark, Kurt. (2002). *A collection of LAML examples.* WEB material available at `http://www.cs.auc.dk/~normark/scheme/examples/jfp/index.html`.

Queinnec, Christian. (2000). The influence of browsers on evaluators or, continuations to program web servers. *Pages 23–33 of: Proceedings of the fifth acm sigplan international conference on functional programming.* ACM Press.

Queinnec, Christian. 2001 (May). *Inverting back the inversion of control or, continuations versus page-centric programming.* Tech. rept. Technical Report 7, LIP6. Université Paris 6.

Sestoft, Peter. (2002). *ML server pages (version 1.1).* http://ellemose.dina.kvl.dk/~sestoft/msp/.

Steele, Guy L. (1990). *Common lisp, the language, 2nd edition.* Digital Press.

Tennent, R.D. (1981). *Principles of programming languages.* Prentice Hall.

Thiemann, Peter. (2000). Modeling HTML in haskell. *Pages 263 – 277 of:* Pontelli, E., & Costa, V. Santos (eds), *Practical aspects of declarative languages, lncc 1753.* Lecture Notes in Computer Science. Second International Workshop, PADL 2000, Boston, MA, USA: Springer Verlag.

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

Wiger, Ulf. 2000 (October). *XMerL - interfacing XML and Erlang.* Sixth International Erlang/OTP User Conference. http://www.erlang.se/euc/00/.