Modeling HTML in Haskell

Peter Thiemann

Universität Freiburg, Germany
Email: thiemann@informatik.uni-freiburg.de

Abstract. We define HTML as an embedded domain specific language in Haskell. To this end, we have designed and implemented a combinator library which provides the means to create and modify HTML elements. Haskell's type classes are employed to enforce well-formed HTML to a large degree. Haskell can then be used as a meta language to map structured documents to HTML, to define conditional content, to extract information from the documents, or to define entire web sites. We introduce container-passing style as a programming convention for the library. We also pinpoint some shortcomings of Haskell's type system that make the task of this library's implementor tedious.

Key words: embedded domain specific language, HTML, functional programming, type classes

1 Introduction

Programming one web page in isolation is not hard. Programming an entire web site can be a nightmare. While simple HTML editors help with the first task [23, 9], full blown web authoring systems are required for the other task. Meanwhile there is a plethora of systems available for this task [22]. These systems come in two basic flavors. Either they have a WYSIWYG front end [20], or they are purely textual [25, 4, 5]. The pure WYSIWYG tools never let you program a single line in HTML, you don't even have to know it. A disadvantage is that the HTML code produced by these tools is usually inscrutable and cannot easily be modified without the tool. The textual tools require some knowledge of HTML and add concepts on top of that. Usually, they come with some sort of previewing feature in a separate window. Of course, there are also tools which live in the middle ground between the extremes [19].

But you are always limited in the means to structure documents and to build your own abstractions by the tool at hand. However, programming languages are very good at structuring problems and at building abstractions. This is especially true for a functional programming language with an expressive type system like Haskell [6]. In such a language not only can you structure your problem and build your own abstractions, but on top of that the type system can provide additional structure and guarantees.

We have used Haskell to model a recent version of HTML and we are able to reap all the benefits alluded to above. Our domain specific HTML modeling language is based on a combinator library in Haskell. The basic idea is to build a data structure that can be rendered to HTML text. Due to our use of type classes, the Haskell type checker guarantees the well-formedness of the generated HTML to a large degree. This exploitation of the type system is in our view the main contribution of the paper. Using a Haskell interpreter [12] it is possible to interactively create and manipulate web pages as well as entire web sites. Since we can use Haskell in all stages the creation of customized styles, site maps, or multi-language pages is just a matter of writing some Haskell code.

Overview In Section 2, we work through three simple examples to get a glimpse of the programmer's view of the library. Section 3 explains how Haskell can represent HTML elements, how they are filled with contents, and how they are supplied with attributes. Section 4 shows how type classes are used to enforce HTML's restrictions on how elements can be assembled. Next, in Section 5, we discuss some problems and features of the library, some of them originating from limitations in the Haskell type system. The section concludes with a brief assessment. Finally, we discuss related work in Section 6 and conclude.

In the paper, we assume some familiarity with Haskell and HTML.

2 Examples

To illustrate the working of the library, we work through some simple examples. The first example is a simple Hello World document. The second example shows a simple parameterized document. The last example reports a prototype implementation of a simplified version of Texinfo.

Although we are giving type signatures for clarity and documentation below, we should emphasize that no type signature is actually required. The Haskell interpreter, Hugs98, is able to infer all of them.

2.1 Hello World

The HTML definition demands that every document consists of a head and a body. Therefore, to start a new document, we must provide a head and a body.

```
doc :: HTML
doc = make_html `add` doc_head `add` doc_body
```

Here, `make_html` creates a new HTML element and the function `add` adds contents to this element. Each content item is itself a HTML element. In the head of the document, we provide the title of the document.

```
doc_head :: HEAD
doc_head =
  make_head
  `add` (make_title `add` "Hello World")
```

Again, `make_head` creates a `<head>` element which contains a single `<title>` element, which in turn contains the string `Hello World`.

Not every combination of tag and contents is legal HTML. For example, putting some text into a document head is illegal. If we try this with our library, we get a type error.

```
Example> make_head `add` "some string"
ERROR: Unresolved overloading
*** Type       : AddTo HEAD [Char] => HEAD
*** Expression : add make_head "some string"
```

The type class `AddTo` governs which element can contain which other element. Whenever there is an instance `AddTo s t` then elements of type `t` can be put into an element of type `s`. The type error tells us that there is no instance `AddTo HEAD [Char]`, so strings cannot be put into a `<head>` in HTML.

For the body, we supply a minimal content.

```
doc_body :: BODY
doc_body =
  make_body
  `add` (make_heading 1 `add` title)
  `add` ("This is the traditional \"" ++ title ++ "\" page.")
  `add` make_hr
  `add` (make_address
         `add` (make_a
                `add` "Peter Thiemann"
                `attr` A_HREF "mailto:thiemann@acm.org"))
  where title = "Hello World!"
```

Here, `make_body` creates a `<body>` element which contains a heading, some text, a horizontal rule, and an address. The address element contains a hyperlink, where the URL of the link is supplied as an attribute of the `<A>` (anchor) tag. Just like `add` adds content items to an element, `attr` adds attributes to it. Also, just like before, not every combination of tag and attribute is legal HTML. This is governed by the class `AddAttr` with the single method `attr`.

For example, trying to add a `src` attribute to an `<a>` tag yields a type error:

```
Example> make_a `attr` A_SRC "xxx"
ERROR: Unresolved overloading
*** Type       : AddAttr A A_SRC => A
*** Expression : attr make_a (A_SRC "xxx")
```

Finally, the `show_html` function yields a rendition of the document in HTML.

```
Example> putStr (show_html doc)
<html><head><title>Hello World</title>
</head>
<body><h1>Hello World!</h1>
This is the traditional &#34;Hello World!&#34; page.<hr>
<address><a HREF="mailto:thiemann@acm.org">Peter Thiemann</a>
</address>
</body>
</html>
```

2.2 Parameterized documents

So far, we have shown plain HTML programming with some legality checking. But it is just as simple to program parameterized documents. Suppose, you want all your documents to look similar in structure to the `Hello World` document of the preceding section. Abstracting the title and the contents yields

```
html_doc :: (AddTo HTML a, AddTo HEADING b, AddTo TITLE b) =>
            Int -> b -> (BODY -> a) -> HTML
html_doc level title add_contents =
  make_html
  `add` (make_head
         `add` (make_title `add` title))
  `add` (make_body
         `add` (make_heading level `add` title)
         $$ add_contents)
```

where `$$` is reverse function application defined by

```
x $$ f = f x
```

The parameter `add_contents` is a "`BODY` transformer" that maps a `<BODY>` to some element of type `a` that can be added to an `<HTML>` element. Only `BODY` makes sense for `a`, so we could restrict the type somewhat further.

We call this functional style of parameterization *container-passing style*. It makes it possible to add multiple elements into a single element and it simplifies dealing with conditionals. Had we just said

```
html_body :: AddTo BODY a => a -> BODY
html_body contents =
  make_body `add` contents
```

then we could only add one item to the `<BODY>`. This is because the library does not support grouping of elements. If it did we would lose static typechecking of HTML code, which relies on every element having a special type[1].

As an aside, there are alternative entry points for the library which provide all elements in container-passing style. For example, instead of `make_body` we would use `add_body` of type `AddTo a b => (BODY -> b) -> a -> a`. This function takes a `BODY` transformer, which is intended to add contents or attributes to the `BODY`, and inserts it in an element of type `a`. As an example of a `BODY` transformer and a use of the library in container-passing style, we write a function that adds a signature.

```
add_signature :: (AddTo a ADDRESS, AddTo A b, AddTo a HR) =>
                 b -> String -> a -> a
add_signature name url =
  add_hr
  # add_address (add_a (add' name # attr' (A_HREF url)))
```

[1] This problem could be addressed if the Haskell type system supported overlapping instances.

```
data Node =
  Node  String          -- name of node
        [Node_Content]  -- contents of the node
        [Node]          -- subnodes
  -- administrative fields:
        String Int      -- name stub & number  of node
        [Int]           -- section counter

type Node_Content = String
```

Fig. 1. The type `Node`

Here `add' = flip add`, `attr' = flip attr`, and `(f # g) x = g (f x)` (reversed function composition). The `add_hr` function does not have a transformer argument because this element has neither contents nor attributes. Although some definitions look more appealing in container-passing style, we stick to the "direct" style for the rest of this paper.

Finally, we can pick up these blocks and produce a template for the `Hello World` example considered above.

```
my_doc :: (AddTo HEADING a, AddTo BODY b, AddTo TITLE a) =>
          a -> b -> HTML
my_doc title contents =
  html_doc 1 title
    (add' contents
     # add_signature "Peter Thiemann" "mailto:thiemann@acm.org")
```

The type of `my_doc` specifies that the first argument must be suitable for adding it to a heading as well as to a title and the second argument is put into a body.

With this definition in place, the document from the first example is just

```
my_doc "Hello World" "This is the traditional \"Hello World!\" page."
```

2.3 Linked nodes

The idea of Texinfo [26] is to structure a text as a set of nodes which are interconnected by hyperlinks. Each node has a unique name, by which it can be referred to, and three links. The links point to the next, previous, and up nodes, that is, the next or previous one on the same hierarchical level of nodes, whereas the up link points to a node higher up in the hierarchy. Each of these nodes is rendered to HTML in essentially the same way. This pattern is captured in a function `node2html`. But first, we need to discuss the basic datatype.

The datatype for a node has six fields (see Figure 1). It contains the name of the node, the contents of the node, the subnodes, and some administrative fields which are filled in automatically (name stub for the generated files, a unique number for the generated file, and a section counter).

The author of such a structure only has to specify the contents of each node and to list the subnodes. The function `node2html` in Figure 2 translates one

```
node2html :: Node -> Maybe Node -> Maybe Node -> Maybe Node -> HTML
node2html (Node name contents subnodes _ _ sec_count)
          m_next m_previous m_up =
  html_doc level title
  ( makelink "Next" m_next
  # makelink "Previous" m_previous
  # makelink "Up" m_up
  # add_hr
  # makepars contents
  # makemenu node_ref subnodes)
  where
    title = show_sec_count sec_count ++ name
    level = length sec_count
```

Fig. 2. Translation of `Node` to `HTML`

```
makemenu make_ref [] =
  id
makemenu make_ref subnodes =
  add_list "menu" (foldr add_node id subnodes)
  where
    add_node node add_items = add_li (add' (make_ref node)) # add_items
```

Fig. 3. Creating the menu of subnodes

node into the corresponding HTML data structure. Following the standard title and heading defined above, there are three hyperlinks `Next`, `Previous`, and `Up` created by `makelink`. Then there is a horizontal rule, followed by the text structured in paragraphs (`makepars`) and finally a menu of the immediate subnodes (`makemenu`). Of these, `makemenu` (see Figure 3) is probably the most interesting. If the list of subnodes is non-empty, it creates a function that adds a `<MENU>` and then adds to it one list item for each subnode.

The complete implementation only requires some simple auxiliary functions and a main function `tree2html :: Node -> Maybe Node -> String -> IO ()` which takes a `Node` data structure, possibly a reference to an enclosing document, and a filename stub. Executing this main function results in automatically assigning a filename to each node, translating every node to HTML, and writing the resulting text to the respective files.

3 Modeling HTML elements

In this section, we explain the underlying data structures. The main tools are type classes and heterogeneous lists (lists with partially abstract elements).

Each basic HTML element (or tag) is modeled by its own datatype. For example, the datatype for the `<hr>` tag (*horizontal rule*) is

```
class ELEMENT a where
  contents :: a -> ELEMENT_List
  set_contents :: a -> ELEMENT_List -> a
  attributes :: a -> ATTRIBUTE_List
  set_attributes :: a -> ATTRIBUTE_List -> a
  tag :: a -> String
  show_html :: a -> String
-- default implementations
  contents a = ELEMENT_Nil
  set_contents a x = a
  attributes a = ATTRIBUTE_Nil
  set_attributes a x = a
  tag a = ""
  show_html = show_element_standard
```

Fig. 4. Type class ELEMENT

```
data HR = HR
```

In addition, every element type is a member of the type class `ELEMENT` defined in Figure 4. There are two methods to manipulate contents (if any) of the element. The method `contents` retrieves the contents and the method `set_contents` sets new contents. The `set_contents` method is purely functional, no destructive updating takes place.

Contents are values of type `ELEMENT_List`. This is a heterogeneous list of values of type class `ELEMENT`:

```
data ELEMENT_List =
  forall e . ELEMENT e => ELEMENT_Cons e ELEMENT_List | ELEMENT_Nil
```

The actual type of each element of the list is abstract, since the all-quantification of `e` in the type of the constructor amounts to the element having an existential type. It is only known that each element has a type belonging to `ELEMENT`, so the methods of this class are applicable [15, 16]. This is an extension to Haskell, which is available in many implementations (e.g., Hugs98 [12]).

Attributes are dealt with in the same way as elements. Each attribute has its own datatype. Since HTML has different name spaces for tag names and attribute names whereas Haskell uses datatypes for both, the library renames all attributes consistently by prepending `A_` to attribute types and constructors. For example, the type for the attribute `HREF` is

```
data A_HREF = A_HREF String
instance ATTRIBUTE A_HREF where
  attr_name (A_HREF url) = "HREF"
  attr_value (A_HREF url) = Just url
```

Analogously to elements, each attribute is an instance of the type class `ATTRIBUTE` which is defined in Figure 5. Each attribute has a name (`attr_name`) and it can

```
class ATTRIBUTE a where
  attr_name :: a -> String
  attr_value :: a -> Maybe String
  show_attr :: a -> String
-- default implementations
  attr_value a = Nothing
  show_attr = show_attr_standard
```

Fig. 5. A type class for attributes

be converted to a string (`show_attr`). In addition, some attributes carry a value (`attr_value`).

Returning to elements, there are two methods to access (`attributes`) and to set attributes (`set_attributes`) of an element. They deal with a type `ATTRIBUTE_List`, which is a heterogeneous list of instances of `ATTRIBUTE` in analogy to `ELEMENT_List`.

```
data ATTRIBUTE_List =
  forall a . ATTRIBUTE a => ATTRIBUTE_Cons a ATTRIBUTE_List | ATTRIBUTE_Nil
```

The remaining two methods deal with the identification of the element (`tag`) and with creating a string representation (`show_element`).

As an example, we show the full datatype definition for the `<A>` element (HTML *anchors*).

```
data A = A ATTRIBUTE_List ELEMENT_List
instance ELEMENT A where
  contents (A aaa ccc) = ccc
  set_contents (A aaa _) ccc = A aaa ccc
  attributes (A aaa ccc) = aaa
  set_attributes (A _ ccc) aaa = A aaa ccc
  tag (A aaa ccc) = "a"
```

Anchors have attributes and also contents, so the data constructor takes two parameters. The get and set methods are obvious and the `tag` method returns the string `"a"`. The library provides a shortcut `make_a` for constructing an anchor and an anchor-adding function in container-passing style:

```
make_a = A ATTRIBUTE_Nil ELEMENT_Nil
add_a f el  = el `add` f make_a
```

There are shortcuts defined for each element. The idea is now to use the `add` and `attr` methods to add contents and attributes to this empty anchor.

4 Assembling HTML

The datatypes from the previous section just provide a uniform means of accessing, modifying, and printing instances of `ELEMENT` and `ATTRIBUTE`. But they do not provide any checks that the constructed data represents valid HTML. To

```
class (ELEMENT s, ELEMENT t) => AddTo s t where
  add :: s -> t -> s
  add element subelement =
    set_contents element (ELEMENT_Cons subelement (contents element))

class (ELEMENT e, ATTRIBUTE a) => AddAttr e a where
  attr :: e -> a -> e
  attr e a =
    set_attributes e (ATTRIBUTE_Cons a (attributes e))
```

Fig. 6. Type classes for adding elements and attributes

implement the validity checks, the library provides two additional type classes `AddTo` and `AddAttr` shown in Figure 6. Both type classes have two parameters. In `AddTo s t` the `s` is the type of the container and `t` is the type of the content. In `AddAttr e a`, `e` is the type of the element and `a` is the type of the attribute. Both have default methods which rely on the provided methods for accessing and modifying contents and attributes. They are never overridden in the current version of the library.

To specify that an element is a legal content of another element, all we need to do is declare an instance of `AddTo`. For example,

```
instance AddTo DL DT
instance AddTo DL DD
```

states that the only allowed contents of a definition list (`<DL>`) are `<DT>` (term in definition list) and `<DD>` (definition of a term). It corresponds directly to the HTML DTD (document type definition) which defines the `<DL>` tag like this:

```
<!ELEMENT DL    - -  (DT | DD)+>
```

Actually, this phrase says a little more than our instance declarations because it insists that each `<DL>` contains *at least one* `<DT>` or `<DD>`. We'll return to that point later in Section 5.

Similarly, to specify that an attribute is legal for some element, we need to provide an instance of `AddAttr`. For example, the only possible attribute for a definition list is `COMPACT`, indicating that the list is to be rendered in a compact style. The declaration

```
instance AddAttr DL A_COMPACT
```

states this exactly.

5 Discussion

In this section, we consider some of the unpleasant facts of life pertaining to the library that we constructed. Some are just inconveniences for the implementor of the library, some can be fixed using a more elaborate type structure, and some may only be fixed using dynamic tests at run time.

5.1 HTML entities

Besides the elements considered above, HTML defines *parameter entities* which can be used to name groups of elements. For example,

```
<!ENTITY % font " TT | B | I ">
<!ENTITY % phrase "EM | STRONG | CODE | SAMP | KBD | VAR | CITE ">
```

defines that %font stands for one of the elements <TT>, <B>, or <I>, and similarly for %phrase. It would be nice to model this relation directly using subclasses FONT and PHRASE of ELEMENT, but unfortunately we cannot.

To see why, consider the definition of another entity %text which relies on %font and %phrase:

```
<!ENTITY % text "#PCDATA | A | IMG | BR | %phrase | %font">
```

Hence, %text can be a character string, one of the <A>, <IMG>, or
 tags, or it can be a %phrase or a %font. However, the corresponding definitions

```
class ELEMENT a => FONT a
instance FONT TT
instance FONT B
instance FONT I

class ELEMENT a => PHRASE a
instance PHRASE EM
...

class ELEMENT a => TEXT a
instance TEXT A
instance TEXT IMG
instance TEXT BR
instance PHRASE a => TEXT a
instance FONT a => TEXT a
```

are rejected due to overlapping instances in the last two lines (by Hugs98). This declaration is already syntactically illegal in Haskell98 which restricts instance declarations to `C (T u1 ... uk)` where `C` is the type class and `T` is a type constructor.

In the present case, this is overly restrictive because the elements of PHRASE and FONT are disjoint. And even if they were not, since the instance declaration does not provide any definition, there can be no semantic ambiguities regardless of which instance path is used.

Another viable approach would be to define not type classes, but datatypes for FONT, PHRASE, TEXT, and so on. Here is an example definition.

```
data FONT = forall a . ELEMENT a => FONT a
instance ELEMENT FONT where
  contents (FONT a) = contents a
  set_contents (FONT a) x = FONT (set_contents a x)
  attributes (FONT a) = attributes a
```

```
  set_attributes (FONT a) x = FONT (set_attributes a x)
  tag (FONT a) = tag a
  show_element (FONT a) = show_element a
```

This approach can be made to work, but it places a heavy burden on the user of the library. The user should not be concerned whether some use of `make_tt args` should really be `TEXT (FONT (make_tt args))`. So we would like to have the type checker insert these type coercions for us. (A related technique has been employed to locate a type in an extensible union type [13].) The declarations

```
class (ELEMENT source, ELEMENT target) => COERCE source target where
  coerce :: source -> target

class (ELEMENT source, ELEMENT target) => COERCE1 source target where
  coerce1 :: source -> target

instance ELEMENT x => COERCE x x where coerce x = x

instance (COERCE1 a b, COERCE b c) => COERCE a c where
  coerce a = coerce (coerce1 a)
instance COERCE1 TT   FONT where coerce1 = FONT
instance COERCE1 FONT TEXT where coerce1 = TEXT
-- and so on
```

specify the required relation `COERCE` between element types. Unfortunately, the declaration

```
instance (COERCE1 a b, COERCE b c) => COERCE a c where
  coerce a = coerce (coerce1 a)
```

is rejected because `b` does not occur in the type of `coerce :: a -> c` (this could cause a semantics ambiguity, in general). In the present case, the relation `COERCE1` is a partial function, hence it cannot give rise to semantic problems. (This kind of declaration was possible in Gofer [11], which was used to program the extensible type mentioned above [13].)

The current implementation of the library is flat. All uses of `%text`, `%font`, and so on in the DTD are tediously expanded by hand. A tool to perform this task automatically is currently in the works. Besides, the use of parameter entities is not restricted to grouping element names, but rather it is a general macro mechanism, which is best supported by such a tool.

5.2 Regular expressions

Another feature of HTML that the library does not model is the use of regular expressions to specify the contents of an element. For example,

```
<!ELEMENT HTML O O  "HEAD, BODY">
```

specifies that the contents of `<HTML>` should be exactly one `<HEAD>` followed by exactly one `<BODY>`. The present library only approximates this by allowing an arbitrary number of heads and bodies in arbitrary sequence.

Likewise, other regular operators are only approximated. For example, the contents of `<DL>` should be `(DT | DD)+` (see above), but this is not checked either.

This problem could be addressed in two ways, one static and one dynamic. In the dynamic approach, the `ELEMENT` class would have additional methods to model a deterministic finite automaton [7] and each addition of a content element would step the automaton. The `add` method of the `AddTo` class would have to check dynamically that the automaton of the added element is in a final state, otherwise it would raise an exception.

The static approach would encode the states of this finite automaton into the datatypes. In the case of the `HTML` element, this would mean

```
data HTML_0 = HTML_0
data HTML_1 = HTML_1 HEAD
data HTML_2 = HTML_2 HEAD BODY
instance HTML HTML_2            -- final state
instance HEAD h => AddTo' HTML_0 h HTML_1 where
  add HTML_0 h = HTML_1 h
instance BODY b => AddTo' HTML_1 b HTML_2 where
  add (HTML_1 h) b = HTML_2 h b
```

and there would be a type class for each element capturing the types corresponding to final states of the automaton (cf. `HTML_2`). Despite the imminent flood of types and classes, we expect that the usability of the library is not affected by this gain of precision. Hence, we are currently pursuing this approach by automatically generating all type and class definitions using a parser for DTDs.

Using the same approach, multiple occurrences of attributes could be rejected and the presence of required attributes could be guaranteed.

5.3 Exceptions

An exception declaration in an element declaration of a DTD indicates that, within the scope of the declared element, certain elements are admissible regardless of the content indicator of the nested elements. Dually, there are negative exceptions, which abolish the use of some elements, regardless of the content indicator of the nested elements. For example,

```
<!ELEMENT SELECT - - (OPTION+) -(INPUT|SELECT|TEXTAREA)>
```

indicates that a `<SELECT>` element is composed of a non-empty sequence of `<OPTION>` elements *and these elements must not contain the elements `<INPUT>`, `<SELECT>`, or `<TEXTAREA>`.*

While type classes can encode "permissions" which allow values to be used in certain ways, they cannot encode "denials", which would be required in the present case. However, this kind of negative information is required in type systems for records, which need to express the absence of a particular label [24, 30]. Similar information can also be expressed using qualified types [10].

Progressing from HTML to XML [2] would also solve the problem because XML does not support exceptions, anymore.

5.4 Assessment

We found the library surprisingly easy and intuitive to use. Especially, programming HTML in container-passing style results in a fairly natural syntax. The possibility to abstract commonly used patterns pays off enormously, its benefits are already visible in the examples shown in Section 2. We also found the type checking capabilities of the current version sufficient because it captures common errors (using an element in the wrong place). We need some more experience to see if the more elaborate scheme outlined in Section 5.2 is worth the effort.

On the negative side, type errors are a bit hard on users who are not deeply into Haskell. It would be nice if type errors could be filtered and translated so that they are more informative to casual users of the library. These users would also appreciate a syntax which is closer to HTML.

Another idea that would widen the scope of the library is to put the `show_element` method into a monad, i.e., give it the type `Monad m => a -> m String`. In this way, the show methods could print directly to a file and we would gain some more flexibility in modeling configurable web sites (e.g., parameterized over a language).

6 Related work

TkGofer [28, 27] is a toolkit for building graphical user interfaces using Gofer [11] and Tcl/Tk [21]. It employs multi-parameter constructor classes to govern parameters passed to instances of widgets. This use of type classes statically avoids errors that would otherwise only be detected dynamically at execution time. Our approach is simpler because there are fewer type classes to learn and there is no parameterization over constructors. In addition, we can type check the way in which HTML elements (widgets) are put together.

There are libraries for CGI (a standard for generating dynamic documents) programming by Hughes [8] and Meijer [18], both are monad-based (or a generalization thereof) and concentrate on modeling the interaction between client and server and on encoding issues. They provide only little support for checking the validity of generated HTML. We believe that the present work is complementary to these works.

Type classes are also used as a structuring tool in the various functional hardware modeling frameworks [17, 1]. However, they employ type classes in a more conventional way, namely to structure the operations that may be applied to certain values. In contrast, we use the class and instance declarations as a logic program (just a finite automaton) to rule out invalid programs (see our complaints in Section 5.1).

Wallace and Runciman [29] propose two alternative ways of using Haskell represent XML documents. They have one generic encoding, which represents documents in a similar way than we propose. But their generic encoding is a general tree structure which does not impose any validity constraints on generated or processed documents. In particular, the use of the type system for guaranteeing validity to a large degree is novel to our work. In addition, Wallace and

Runciman do have a type-based encoding, which guarantees valid documents by modeling them directly with Haskell datatypes. However, this approach is less flexible when it comes to parameterized documents. Finally, they have developed a combinator library for processing the generic XML representation. This library could be ported to our representation, thus reaping the benefits of their combinators while enjoying strongly typed XML at the same time.

In the logic programming world, there are several toolkits for generating HTML pages. The PiLLoW toolkit [3] allows for easy creation of documents including CGI functionality. It is widely used to connect logic programs to the WWW. LogicWeb [14] offers an even tighter integration which includes client-side scripting.

7 Conclusion

We have designed and implemented a convenient embedded domain specific language for meta programming of web pages and web sites in Haskell. Haskell's type classes and facilities to program with partially abstract types were instrumental in the construction. We have introduced container-passing style as a means to concisely construct abstractions and fragments of web pages.

References

1. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In Paul Hudak, editor, *Proc. International Conference on Functional Programming 1998*, pages 174–184, Baltimore, USA, September 1998. ACM Press, New York.
2. Tim Bray, Jean Paoli, and C.M. Sperberg-MacQueen. Extensible markup language (xml) 1.0 (w3c recommendation). http://www.w3.org/TR/REC-xml, feb 1998.
3. Daniel Cabeza and Manuel Hermenegildo. Www programming using computational logic systems (and the PiLLoW/CIAO library). http://www.clip.dia.fi.upm.es/Software/pillow/pillow_www6/pillow_www6.html, March 1997.
4. Allaire Corp. HomeSite. http://www.allaire.com/.
5. Ralf S. Engelschall. Website meta language (wml). http://www.engelschall.com/sw/wml/.
6. Haskell98, a non-strict, purely functional language. http://www.haskell.org, December 1998.
7. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.
8. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 1999. To appear.
9. SoftQuad Software Inc. HoTMetaL. http://www.softquad.com/.
10. Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brückner, editor, *Proc. 4th European Symposium on Programming '92*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306, Rennes, France, February 1992. Springer-Verlag.

11. Mark P. Jones. *Gofer, Functional Programming Environment*, June 1994. Version 2.30.
12. Mark P. Jones. Hugs Online — embracing functional programming. http://www.haskell.org/hugs/, June 1999.
13. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, CA, January 1995. ACM Press.
14. Seng Wai Loke and Andrew Davison. Logic programming with the World-Wide Web. pages 235–245, Washington DC, USA, mar 1996.
15. Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.
16. Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proc. ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, San Francisco, CA, June 1992.
17. John Matthews, John Launchbury, and Byron Cook. Microprocessor specification in Hawk. In *IEEE International Conference on Computer Languages, ICCL 1998*, Chicago, USA, May 1998. IEEE Computer Society Press.
18. Erik Meijer. Server-side web scripting with Haskell. *Journal of Functional Programming*, 1999. To appear.
19. Microsoft. Microsoft Frontpage98. http://www.microsoft.com/frontpage.
20. NetObjects Fusion. http://www.netobjects.com.
21. J. K. OusterHout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.
22. PC Magazine. Web authoring tools. http://www.zdnet.com/pcmag/features/htmlauthor/_open.htm, January 20 1998.
23. John Punin and Mukkai Krishnamoorthy. ASHE (a simple HTML editor) - xhtml. http://www.cs.rpi.edu/ puninj/ASHE/, November 1996.
24. Didier Rémy. Typing record concatenation for free. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, pages 166–176, Albuquerque, New Mexico, January 1992. ACM Press.
25. Internet Software Technologies. HTMLed Pro32. http://www.ist.ca.
26. Texinfo. http://texinfo.org.
27. Ton Vullinghs, Wolfram Schulte, and Thilo Schwinn. The design of a functional GUI library using constructor classes. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 398–408, Novosibirsk, Russia, June 1996. Springer-Verlag.
28. Ton Vullinghs, Daniel Tuijnman, and Wolfram Schulte. Lightweight GUIs for functional programming. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, pages 341–356, Utrecht, The Netherlands, September 1995. Springer-Verlag.
29. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? pages 148–259.
30. Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proc. of the 4th Annual Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989. IEEE Computer Society Press. To appear in *Information and Computation.*