

A typed representation for HTML and XML documents in Haskell

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

February 15, 2001

Abstract

We define a family of embedded domain specific languages for generating HTML and XML documents in Haskell. To this end, we have designed and implemented a combinator library which provides the means to create and modify HTML/XML elements. The resulting HTML/XML code is guaranteed to be well-formed. In addition, the library can guarantee that the generated documents are valid XML documents (for HTML only a weaker guarantee is possible). Haskell can then be used as a meta language to define parameterized documents, to map structured documents to HTML/XML, to define conditional content, to extract information from the documents, or to define entire web sites.

The combinators support a programming convention called *container-passing style*. Programs written in this style have a visual appearance similar to HTML/XML source code, without modifying the syntax of Haskell.

Key words: embedded domain specific language, HTML, XML, 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 [24, 12], 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 [27, 6, 7]. 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 by the tool at hand. In particular, most systems only provide fixed means to structure documents and in the very few systems [7] where you can build your own structures these features have a low level of abstraction. 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 [9]. 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 guarantees.

Another potential use is the generation of dynamic Web pages. Such pages are constructed on-the-fly from templates and computed values (eg., the results of data base queries). Our library provides higher-order templates for free and ensures the validity of the generated pages.

Initially [29], we have built a prototype library in Haskell to model HTML 4 [10] and we were able to reap all the benefits alluded to above. Our domain specific language is based on a combinator library. 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 validity of the generated documents to a large degree. This exploitation of the type system is a key contribution of the paper. Using a Haskell interpreter [14] 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. The library has also been used successfully to generate dynamic content through CGI scripts written in Haskell.

In the next stage, we have parameterized the combinator library with respect to an XML DTD (document type definition). The DTD restricts the way in which elements may be nested in a valid XML document. Similar restrictions are built into HTML, but in the case of XML they may vary for each particular group of documents. Hence, we have defined and implemented a translation that maps an XML DTD to a custom version of the library. This version can then be incorporated into programs that generate

valid XML with respect to this particular DTD. In a similar way, document-generating libraries may be created for different versions of HTML as well as XHTML [33].

A version of the library, created for HTML 4.01 from the official DTD [10] is available from the author's webpage at <http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASP/>.

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 the typed representation of HTML/XML elements in Haskell, how the elements are filled with contents, and how they are supplied with attributes. Section 4 shows how multi-parameter type classes are used to enforce a particular DTD when assembling elements. Section 5 defines the translation from DTDs to instances of our library. In Section 6, 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 7 and conclude.

In the paper, we assume some familiarity with Haskell, HTML, and XML. Strictly speaking, the library is *not* a valid Haskell98 program due to the use of multi-parameter type classes [23]. However, a number of Haskell implementations support this extension.¹

2 Examples

To illustrate the use of the library, we work through some examples. The first example is a Hello World document. The second example demonstrates construction and use of a parameterized document. The last example describes a prototype implementation of a simplified Texinfo.

2.1 Hello World

The HTML standard [10] 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 :: ELT DOCUMENT
doc = build_document (doc_head ## doc_body)
```

¹In particular, the code in this paper has been tested with the January 2001 release of the Haskell interpreter Hugs in `-98` mode. The storage space for instances has been increased to 10000 (define `NUM_INSTS` in `src/prelude.h`).

Here, `build_document` creates a new HTML element from the components `doc_head` and `doc_body`. For the moment, the operator `##` may be regarded as a concatenation operator that glues components together. In this case, each component stands for particular HTML element. In the head of the document, we provide its title.

```
doc_head :: (AddTo a HEAD) => ELT a -> ELT a
doc_head = head (title (text "Hello World"))
```

The component `doc_head` stands for a `<head>` element, which contains a single `<title>` element, which in turn contains the string `Hello World`. The predicate `AddTo a HEAD` indicates that a `<head>` element can only be put inside another element, if its type `a` is related to `HEAD` by the type class `AddTo`.

Not every combination of tag and contents is legal in HTML. For example, putting some text directly into a `<head>` element is illegal. If we try this with our library, we get a type error. Text is modeled using the `CDATA` type.

```
Example> head (text "Hello")
ERROR - Unresolved overloading
*** Type      : (AddTo HEAD CDATA, AddTo a HEAD) => ELT a -> ELT a
*** Expression : head (text "Hello")
```

At this moment, we will not dig deeper into the type, but only consider the the predicates (`AddTo HEAD CDATA`, `AddTo a HEAD`). The type class `AddTo` governs which element can contain which other element. Whenever there is an instance `AddTo s t` then elements with tag `t` can be put into an element with tag `s`. The type error tells us that there is no instance `AddTo HEAD CDATA`, which indicates that plain text cannot be put directly into a `<head>` element in HTML.

For the body, we supply a minimal content.

```
doc_body :: (AddTo a BODY) => ELT a -> ELT a
doc_body =
  body (h1 (text ttl)
        ## text ("This is the traditional \" ++ ttl ++ "\" page.")
        ## hr empty
        ## address (hlink (text "Peter Thiemann") "mailto:thiemann@acm.org"))
  where ttl = "Hello World!"
```

Again, `body` creates a `<body>` element which contains a heading, some text, a horizontal rule `<hr>`, and an address. The parameter `empty` to `hr` indicates that there are neither sub-elements nor attributes. The address element contains a hyperlink, which is constructed using an auxiliary function `hlink`.

```

hlink :: (AddTo a b, AddAttr b HREF, AttrValue HREF c) =>
        (ELT A -> ELT b) -> c -> ELT a -> ELT a
hlink body url =
    a (body ## attr HREF url)

```

In `hlink`, the URL of the link is supplied as an attribute of the `<a>` (anchor) tag. It is parameterized over the body, which can assume everything that can be put into `<a>`.

The function `attr` takes an attribute name and a value and builds an attribute from it. Just like before, not every combination of tag and attribute is valid HTML. This is governed by another class, `AddAttr`.

For example, the `SRC` attribute can specify the URL of a picture with the image tag ``, but it is not valid with the `<a>` tag. An attempt to add a `SRC` attribute to an `<a>` tag yields a type error:

```

Example> a (attr SRC "xxx")
ERROR - Unresolved overloading
*** Type      : (AddAttr A SRC, AddTo a A) => ELT a -> ELT a
*** Expression : a (attr SRC "xxx")

```

Finally, the `show_document` function yields a rendition of the document in HTML. As a simple service, it translates each character into its HTML encoding if the character has a special meaning. In the example, the character `"` is translated to `"`.

```

Example> putStr (show_document doc)
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<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 parameterized documents are just as simple. 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 ttl contents =
    build_document

```

```
(head (title (text ttl))
  ## body (h1 (text ttl) ## contents))
```

Now is the time to look a bit closer at the type for `html_doc`:

```
html_doc :: AddTo HTML a => String -> (ELT BODY -> ELT a) -> DOCUMENT
```

That is, the function takes a string `ttl` (the text for the title) and a function `contents` of type `(ELT BODY -> ELT a)` to yield a `DOCUMENT`. The `contents` parameter is a “BODY transformer” that maps a `<body>` element to some element of type `a` that can be added to an `<html>` element. The latter restriction is indicated by the `AddTo HTML a` predicate. The type `ELT t` is the generic type of HTML elements. The index type `t` is used to denote the tag of the element.

In this particular case, only `BODY` makes sense for `a`, so we could restrict the type somewhat further. The type given above is the most general type inferred by Haskell.

It might seem strange that the type of `contents` is a function. However, it turns out that this is the most natural choice of a type for a component (read: group of elements). For example, the type of the `body` function is

```
body :: AddTo a b => (ELT BODY -> ELT b) -> ELT a -> ELT a
```

That is, the first argument of `body` is a function that transforms an empty `BODY` element into some element with tag `b` (usually `BODY`, too). The second argument is the container into which the `BODY` element should be put. The predicate `AddTo a b` guarantees that the combination of the container `a` and content `b` is legal. The result has again the type of the container.

We call this functional style of parameterization *container-passing style*. It facilitates the handling of groups of elements simply by composing the element functions. Hence, the `##` operator is nothing but reversed function composition and `empty` is the identity function:

```
f ## g = \x -> g (f x)
empty x = x
```

Grouping elements using function composition is more flexible than using lists. Due to its associativity, function composition can be nested arbitrarily. A list representation would run into efficiency problems due to repeated concatenation. In addition, a list representation would lose all type information about the individual elements.

As a final example of a parameterized document, we write a function that adds a signature.

```
signature :: (AttrValue HREF a, AddTo b ADDRESS, AddTo b HR) =>
    String -> a -> ELT b -> ELT b
signature name url =
    hr id
    ## address (hlink (text name) url)
```

From the type signature, we can extract that `signature` can be used in every container that allows the addition of the `ADDRESS` and `HR` tags.

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

```
my_doc :: (AddTo HTML a, AddTo a ADDRESS, AddTo a HR) =>
    String -> (ELT BODY -> ELT a) -> DOCUMENT
my_doc ttl contents =
    html_doc ttl
    (contents
     ## signature "Peter Thiemann" "mailto:thiemann@acm.org")
```

The type of `my_doc` specifies that the first argument must be a string and that the second argument must transform a body into something that allows the addition of an `ADDRESS` and an `HR`.

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

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

2.3 Linked nodes

The idea of Texinfo [28] 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 (in our simplified version) 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 node into the corresponding HTML data structure. Following the

```

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

```

Figure 1: The type Node

```

node2html :: Node -> Maybe Node -> Maybe Node -> Maybe Node -> DOCUMENT
node2html (Node name contents subnodes _ _ sec_count) m_next m_previous m_up =
  html_doc title
  ( maybe_link "Next" m_next
  ## maybe_link "Previous" m_previous
  ## maybe_link "Up" m_up
  ## hr empty
  ## pars contents
  ## my_menu node_ref subnodes)
where
  title = show_sec_count sec_count ++ name
  level = length sec_count

```

Figure 2: Translation of Node to HTML

```

my_menu :: (AddTo LIST a, AddTo b LIST) =>
    (c -> ELT LI -> ELT a) -> [c] -> ELT b -> ELT b
my_menu make_ref [] =
    empty
my_menu make_ref subnodes =
    menu (foldr add_node empty subnodes)
    where
        add_node node items = li (make_ref node) ## items

```

Figure 3: Creating the menu of subnodes

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

The `pars` function takes a list of strings and transforms it into an HTML composite component, by making a paragraph out of each string in the list.²

```

pars :: AddTo a P => [String] -> ELT a -> ELT a
pars = foldr (##) empty . Prelude.map (p . text)

```

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 into an IO action. Executing this main function results in automatically assigning a filename to each node, translating it to HTML, and writing the resulting HTML source texts to the respective files.

3 Modeling HTML elements

In this section, we explain the underlying data structures. The main tools are type classes and phantom types (parameterized types where the type parameter does not appear on the right side of the definition).

²The `Prelude` function `map` must be qualified because `map` is also an HTML tag.

3.1 Types for elements

The basic data type for HTML elements defines just a standard untyped representation (cf. [18, 8, 31]).

```
data ELEMENT_ =
  ELEMENT_ { tag    :: String
            , attrs :: [ATTR_]
            , elems :: [ELEMENT_]
            }
| EMPTY_   { tag    :: String
            , attrs :: [ATTR_]
            }
| CDATA_   String
| DOCTYPE_ { doctype :: [String]
            , elems  :: [ELEMENT_]
            }
```

That is, an element consists of a tag of type `String`, a list of attributes of type `ATTR_` and a list of sub-elements that are nested inside of the current element (of type `ELEMENT_`). The `EMPTY_` constructor stands for empty elements without a closing tag and without sub-elements. The constructor `CDATA_` represents just a string of text. The final alternative for an element is `DOCTYPE_`, which represents the document header.

To enable a typed representation of documents, we introduce a wrapper data type for elements. It has a phantom type parameter `t`, which does not occur in the types on the right side of the definition.

```
data ELT t =
  ELT { unELT :: ELEMENT_ }
```

In addition, there is one data type for each HTML tag. These types are the candidates for the parameter `t` of `ELT`. For example, the tags for `<html>`, `<head>`, and `<body>` are defined thus

```
data HTML = HTML deriving Show
data HEAD = HEAD deriving Show
data BODY = BODY deriving Show
```

Every tag type is a member of the type class `TAG` defined in Figure 4. The member function `make` of this class maps a value of type `t` to a “wrapped” element of type `ELT t`. This way, the type of the wrapped elements reflects its toplevel tag. The default implementation of `make`, `make_standard`, uses the `ELEMENT_` constructor. Elements declared as empty in the DTD override `make` using `make_empty`. Elements constructed with `make` have neither contents nor attributes, initially.

```

class Show t => TAG t where
  make :: t -> ELT t
  show_tag :: t -> String
--
  make = make_standard
  show_tag = map toLower . show

make_standard t = ELT (ELEMENT_ (show_tag t) [] [])
make_empty     t = ELT (EMPTY_   (show_tag t) [])

```

Figure 4: Type class TAG

The choice to have a typed representation for HTML elements has an unfortunate consequence. It is not possible to construct a list that contains elements with different tags. For example, a `<head>` element has type `ELT HEAD` while a `<body>` element has type `ELT BODY` and it is not possible to construct a list containing both. But the documents in our examples above contained both.

Fortunately, the validity of an element only depends on the tag and on the tags of the immediate sub-elements. Hence, once it is clear that an element is a valid sub-element, we can discard its wrapper and collect the subelements in a list of type `ELEMENT_`.

It is also simple to display a typed element. After unwrapping the top-level `ELT` constructor, it is a standard programming exercise to display the element in HTML syntax.

3.2 Types for attributes

Attributes are dealt with in the same way as elements. Each attribute name has its own datatype, which may also be used as a tag name. It does not matter that HTML has different name spaces for tag names and attribute names (and also for attribute values) because Haskell is able to discriminate the uses by their context.

Figure 5 defines the basic data types and classes for attributes. Analogously to the previous representation of elements, a value of type `ATTR_` is a pair of strings, one for the name of the attribute and one for its value. Again, this defines an untyped layer and lists of type `ATTR_` with different attributes are possible.

The type `ATTR a` implements the typed layer. Again, `a` is a phantom type parameter. It is intended to take on only types of class `ATTRIBUTE`.

```

data ATTR_ =
  ATTR_ { attr_name  :: String
         , attr_value :: String
         }

data ATTR a =
  ATTR { unATTR :: ATTR_ }

class Show a => ATTRIBUTE a where
  show_name :: a -> String
--
  show_name = map toLower . show

```

Figure 5: Definitions for attributes

The type class `ATTRIBUTE` has a single member function `show_name`, which is used to print the attribute name. In essence, the class `ATTRIBUTE` defines a certain subset of the elements of the class `Show`.

For example, the type for the attribute `HREF` is

```

data HREF = HREF deriving Show
instance ATTRIBUTE HREF

```

and to create an `HREF` attribute, all we have to do is `mkAttr HREF "mailto:thiemann@acm.org"`. For the moment, we assume a simple implementation for `mkAttr`:

```

-- mkAttr :: (ATTRIBUTE a, Show v) => a -> v -> ATTR a
mkAttr a v = ATTR (ATTR_ (show_name a) (show v))

```

Later on, we will give a more refined type for `mkAttr`.

4 Assembling HTML

The datatypes from the previous section just provide a uniform means of accessing, modifying, and printing values of type `ELEMENT_` and `ATTR_`. But they do not provide any checks that the constructed data represents valid HTML. The validity of a particular combination of tag and sub-elements as well as attributes is governed by a DTD (document type definition). The DTD gives rise to type classes that implement the validity checks. This section considers a subset of SGML-DTDs since widely used versions of HTML are defined in this way. Dealing with XML-DTDs is analogous.

4.1 DTDs

Basically, a document type definition contains two kinds of entries, element definitions and attribute definitions³. An element definition defines a tag and declares its sub-elements. An attribute definition defines the admissible attributes for an element, their types, and sometimes their default values.

A typical element definition has the form

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

where DL defines the name of the tag, the two dashes state that both the opening tag and the closing tag must be written (an 0 would indicate that they are optional), and the (DT | DD)+ is the content specification. The latter specifies the tags of the immediate sub-elements. In this case, the sub-elements may have tags DT or DD, and at least one of them must be present. The content specification is a restricted regular expression using the operators , for sequencing, | for alternative, * for repetition, + for one or more repetitions, and ? for one or zero occurrences. Also, EMPTY is a content specification, which is self-explanatory.⁴

A typical attribute definition has the form (excerpt)

```
<!ATTLIST FORM
  action      CDATA          #REQUIRED -- server-side form handler --
  method      (GET|POST)     GET          -- HTTP method used to submit the form--
  enctype     CDATA         "application/x-www-form-urlencoded"
>
```

where

- FORM is the name of the tag to which the attributes belong,
- action, method, and enctype are the names of attributes,
- CDATA and (GET|POST) specify their respective types, the first and third are string types and the second is an enumeration type with elements GET and POST, and
- #REQUIRED, GET, and application/x-www-form-urlencoded specify default values for the attributes (the #REQUIRED one has no default and must always be supplied).

³We ignore the abbreviation mechanism through so-called entities and conditional sections, since they can be eliminated by a pre-pass.

⁴SGML has an additional binary operator a & b, which means that both a and b must occur once, but in arbitrary order.

```

class (TAG t) => AddAttr t a where
  add_attr :: ELT t -> ATTR a -> ELT t
  add_attr (ELT e_) (ATTR att) =
    ELT (e_ { attrs = att : attrs e_ })

```

Figure 6: Admissible attributes for an element

4.2 Attributes and their values

First, we consider the relation between elements and their attributes. Not every attribute makes sense for a particular element. The type class `AddAttr` (see Fig. 6) defines a single member function `add_attr` that unwraps the type of the new attribute, joins it to the underlying untyped representation of the element, and wraps the element back into its typed representation. The instance declarations govern exactly which typed attribute is admissible for a particular typed element.

For example, the `FORM` element can take (among others) three attributes, an `ACTION`, a `METHOD`, and an `ENCTYPE`:

```

instance AddAttr FORM ACTION
instance AddAttr FORM METHOD
instance AddAttr FORM ENCTYPE

```

From the attribute definition in the previous section we know that the attributes are typed. While the `ACTION` attribute can take on an arbitrary string value⁵, the `METHOD` and `ENCTYPE` attributes take values only from a limited set of choices.

The attribute `METHOD` only takes on `GET` and `POST`, and for `ENCTYPE` only `application/x-www-form-urlencoded` and `multipart/form-data` make sense. The DTD for HTML states such restrictions by declaring enumeration types for these attributes.

Fortunately, it is possible to restrict the formation of attributes using yet another type class, which relates an attribute name to the type(s) of its possible values.

The corresponding type class `AttrValue` has no member functions and is just used to restrict the type of the `mkAttr` function defined above.

```

class (ATTRIBUTE a, Show v) => AttrValue a v

mkAttr :: AttrValue a v => a -> v -> ATTR a
-- mkAttr a v = ATTR (ATTR_ (show_name a) (show v))

```

⁵It should be a valid URL, but this is hard to check.

```

class (TAG s, TAG t) => AddTo s t where
  add :: ELT s -> ELT t -> ELT s
  add (ELT e_) (ELT e'_) =
    ELT (e_ { elems = e'_ : elems e_})

```

Figure 7: Type class relating an element and a sub-element

This typing in connection with the instance declarations for `AttrValue` ensures that only values of the correct type can be adopted for attributes.

Here are implementations for the `ACTION`, `METHOD`, and `ENCTYPE` attributes. An `ACTION` is simply a string (although a user of the library is free to define a type for URIs and create suitable instances of the classes `Show` and `AttrValue`). A `METHOD` attribute can take on values of types `GET` or `POST`, thus implementing an enumeration type. As an alternative implementation, `ENCTYPE` defines its own enumeration type `ENCTYPE_` and makes it an instance of `Show`.

```

instance AttrValue ACTION String

data GET = GET deriving Show
data POST = POST deriving Show
instance AttrValue METHOD GET
instance AttrValue METHOD POST

data ENCTYPE_ = URLENCODED | MULTIPART
instance Show ENCTYPE_ where
  show URLENCODED = "application/x-www-form-urlencoded"
  show MULTIPART = "multipart/form-data"
instance AttrValue ENCTYPE ENCTYPE_

```

The first alternative allows the greatest amount of sharing of types and names in the library, while the second alternative avoids a proliferation of types. Either implementation makes sense. The second alternative might be preferable if an attribute name is overloaded so that it takes different types of values depending on the element to which it belongs. The current implementation employ the first alternative plus some name mangling to avoid conflicts.

4.3 Elements

Valid nesting of elements is governed by the two-parameter type class `AddTo` shown in Figure 7. In `AddTo s t` the `s` is the type of the container and `t` is the type of the content. The default method unwraps the sub-element,

enters it in the `elems` field of the unwrapped container, and wraps it back into an `ELT s`. It is 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) elements. 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 6.

4.4 Character data

Up to now, we assumed that all elements can be constructed from a tag using the `make` function. Unfortunately, this is not true. The notable exception is character data. The data type for elements already provides a constructor `CDATA_` for it. It remains to define a function that turns a string into a component of the right type. The type `CDATA` serves as a pseudo tag type.

```
data CDATA = CDATA deriving Show
instance TAG CDATA
```

```
text :: (AddTo a CDATA) => String -> ELT a -> ELT a
text str = flip add (ELT (CDATA_ str) :: ELT CDATA)
```

The function `text` takes a string, turns it into a value of type `ELEMENT_`, and then wraps it into a value of type `ELT CDATA`.

4.5 Main document

The main document is constructed using the function

```
build_document :: (ELT HTML -> ELT HTML) -> ELT DOCUMENT
build_document contents =
  make DOCUMENT # html contents
```

It transforms an empty HTML element using `contents` and applies the result to the document constructed by `make DOCUMENT`. The latter just constructs a data structure, which contains the document type information at the beginning of an HTML document and adds the top-level HTML element. The `#` operator is just reversed function application: `(#) = flip ($) .`

```
data DOCUMENT = DOCUMENT deriving Show
instance AddTo DOCUMENT HTML
instance TAG DOCUMENT where
  make DOCUMENT =
    ELT (DOCTYPE_
        ["HTML"
         ,"PUBLIC"
         ,"\"-//W3C//DTD HTML 4.01//EN\""
         ,"\"http://www.w3.org/TR/html4/strict.dtd\""]
        [])
```

4.6 Interface functions

The implementation of the tag functions, which are exported to the users of the library, is straightforward. Here is the implementation for the `<head>` element (all others are identical).

```
head :: AddTo a b => (ELT HEAD -> ELT b) -> ELT a -> ELT a
head f elt = elt 'add' f (make HEAD)
```

Hence, the first argument, `f`, of `head` is a transformer for the newly created `HEAD` element. The second argument, `elt`, is the element, in which the transformed `HEAD` element should be inserted in the end. The predicate `AddTo a b` originates from the use of the `add` function and indicate that the result, `b`, of transforming the new `HEAD` element is suitable for putting it into the enclosing `elt` of type `ELT a`.

5 Translation

In this section we collect the various bits and pieces and show the complete translation from a DTD to Haskell code. Figure 8 defines the translation distributed over a number of functions. All functions yield top-level Haskell definitions.

- **DT** translates an item from a DTD;
- **ET** translates an element definition by generating a data type for the tag, defining its interface function, and passing on to **CT**;

```

DT[[<!ELEMENT tag begin end content>]]
= ET tag content
DT[[<!ATTLIST tag body>]]
= AT tag [[body]]

ET tag content
= data U[[tag]] = U[[tag]] deriving Show
  L[[tag]] f elt = elt 'add' f (make U[[tag]])
  CT tag content

CT tag EMPTY
= instance TAG U[[tag]] where make = make_empty

CT tag content
= instance TAG U[[tag]]
  instance AddTo U[[tag]] U[[sub-tag]]    for each sub-tag ∈ content

AT tag []
= -- nothing
AT tag [[name type default rest]]
= data U[[name]] = U[[name]] deriving Show
  instance ATTRIBUTE U[[name]]
  instance AddAttr U[[tag]] U[[name]]
  VT name type
  AT tag [[rest]]

VT name CDATA
= instance AttrValue U[[name]] String
VT name ID
= instance AttrValue U[[name]] String
VT name NUMBER
= instance AttrValue U[[name]] Integer
VT name (val1 | . . . | valn)
= data U[[val1]] = U[[val1]] deriving Show
  instance AttrValue U[[name]] U[[val1]]
  ⋮
  data U[[valn]] = U[[valn]] deriving Show
  instance AttrValue U[[name]] U[[valn]]

```

Figure 8: Translation from DTD to Haskell

- **CT** generates the instance declaration for **TAG** (which depends on whether the content is **EMPTY**) and definitions for the content part of an element;
- **AT** translates an attribute list definition, by generating its data type, its instance declarations for **ATTRIBUTE** and **AddAttr**, and generating the value definitions using **VT**;
- **VT** generates the data types for attribute values (if necessary) and instance declarations for class **AttrValue**.
- \mathcal{U} and \mathcal{L} are name mangling functions that transform a name in a DTD to a valid Haskell identifier, starting with an uppercase character or with a lowercase one.

The definition of the translation glosses over the following problems, which are addressed in the implementation.

- A DTD might use the same name for an element tag, an attribute name, and an attribute value from an enumerated type. For example, HTML 4.01 uses the names **CITE**, **DIR**, **LINK**, and **TITLE** as element tags and also as attribute names. The translation must only define a single data type for those names.
- A name in a DTD may contain characters, which are not allowed in Haskell identifiers (for example, the attribute name **HTTP-EQUIV** in HTML 4.01). Hence there must be a mapping to valid Haskell identifiers and a particular instance of **Show** must be defined for these attribute names:

```
data HTTP_EQUIV = HTTP_EQUIV
instance Show HTTP_EQUIV where
    show HTTP_EQUIV = "HTTP-EQUIV"
instance ATTRIBUTE HTTP_EQUIV
```

Translating the HTML 4.01 DTD [10] in this way yields 5220 lines (roughly 148k) of Haskell code. Most of these lines (4850) are instance declarations. There are 281 lines of data declarations and the remaining lines define the interface functions.

6 Extensions

In this section, we consider extensions that make the library more powerful by increasing the extent of the validity checks. These checks could be

conducted in two ways, static and dynamic. In the dynamic approach, the `ELT` data type would have an additional field containing some value that encodes the current state of the element. Each `add` and `attr` operation would change this state in some way, and there would be a test for a final state, whenever an element becomes closed. The unfortunate thing about the dynamic approach is that correctly typed code might give rise to errors while generating the HTML output. This is clearly undesirable, but still it would reveal many errors during testing.

Hence, we will concentrate on the static approach where the additional information is encoded in the types, too. It is interesting to see, to what extent the Haskell type checker can perform validity checks at compile time.

6.1 Regular expressions

One particular feature of HTML (or rather of DTDs) that the library as explained so far does not model is the use of regular expressions to specify the contents of an element. For example, the declarations

```
<!ELEMENT HTML O O (HEAD, BODY)>
<!ELEMENT DL - - (DT | DD)+>
```

specify that the contents of `<html>` should be exactly one `<head>` followed by exactly one `<body>` and that the contents of `<dl>` should be a sequence of `<dt>` and `<dd>` elements with at least one member. The present library only approximates this by allowing an arbitrary number of `<head>`s and `<body>`s (respectively, `<dt>`s and `<dd>`s) in arbitrary sequence. While this approximation seems good enough for many practical purposes, it is still interesting to consider the construction of a library that guarantees valid documents.

6.1.1 Elements

To illustrate the basic ideas, we will first consider a dynamic approach and then concentrate on the static approach. In the dynamic approach, the `ELT` data type would have an additional field containing a regular expression that specifies in which sequence elements can legally be added. Initially, this expression is the content specification from the DTD. Whenever a new element is added to the contents, the `add` function computes the next state by taking the derivative of the regular expression [4] (this approach uses regular expressions as the set of states of a finite automaton). In addition, the code checks that the new content element is complete by demanding that its state is a final state. The following code fragment illustrates the idea:

```

add (ELT state e_) (ELT state' e'_)
  | finalState state' =
    let nextState = derivative state (tag e'_) in
    if sinkState nextState then
      error "Illegal content sequence"
    else
      ELT nextState (e_ { elems = e'_ : elems e_})
  | otherwise = error "Incomplete element"

```

The static approach encodes the states of this finite automaton into data types and it employs type classes to model the transition function. It requires type classes with functional dependencies [15], yet another extension of Haskell's type system. More specifically, the following two type classes replace the class `AddTo`:

```

class NextState s t s' | s t -> s' where
  nextState :: s -> t -> s'

class FinalState initial final | final -> initial

add :: NextState s t s' => ELT s -> ELT t -> ELT s'
add (ELT e_) (ELT e'_) =
  ELT (e_ { elems = e'_ : elems e_})

```

The class `NextState` encodes a transition function. Hence, the functional dependency `s t -> s'` specifies that the types `s` and `t` determine the next-state-type `s'`. The data type for each tag is split into as many types as required for the finite automaton encoding the contents of the corresponding element. In our prototype implementation, the member function `nextState` remains unused, but it could be used to provide debugging information.

The class `FinalState` relates the initial state of a tag automation to any of its final states. In fact, `FinalState` is a function that maps each final state to its initial state, due to the functional dependency `final -> initial`. This class is required to test whether an element is complete before it is added to the contents.

The function `add` takes a container element of type `ELT s` and a new sub-element of type `ELT t` to produce an updated container element of type `ELT s'`, provided that `NextState s t s'`. Its untyped implementation remains the same as before.

In the case of the `HTML` element, the DTD prescribes

```
<!ELEMENT HTML O O (HEAD, BODY)>
```

This means there must be a `HEAD` followed by a `BODY`. The corresponding finite automaton has three states. The initial one (`HTML`), the state af-

ter adding HEAD (HTML_1), and the final state after adding HEAD and BODY (HTML_2).

```
data HTML = HTML deriving Show
data HTML_1 = HTML_1 deriving Show
data HTML_2 = HTML_2 deriving Show

instance FinalState HTML HTML_2

instance FinalState HEAD head => NextState HTML head HTML_1 where
  nextState HTML head = HTML_1

instance FinalState BODY body => NextState HTML_1 body HTML_2 where
  nextState HTML_1 body = HTML_2
```

The two instance declarations for `NextState` say that if `head` is a final state for tag `HEAD` then a value of type `ELT head` changes `ELT HTML` to `ELT HTML_1`. Further, if `body` is a final state for tag `BODY` then a value of type `ELT body` changes `ELT HTML_1` to `ELT HTML_2`.

Initially, we expected that there would be a proliferation of states. However, it turns out that the number of states is small. For the majority of tags, the number of states is one because their content description has the form $(elt1 | \dots | eltn)^*$. Even a complicated element like `TABLE` has just seven different states.

6.1.2 Attributes

Using the same approach, multiple occurrences of attributes could be rejected and the presence of required attributes could be guaranteed. In practice, the `ELT` data type would receive a second phantom variable to keep track of the attribute automaton. Implicitly, this constructs the product of the element automaton and the attribute automaton for each tag.

Unfortunately, this approach is not practical because the automata have a huge number of states. Most elements take 16 or more attributes in arbitrary order, and in a valid element each attribute may not occur more than once. An automaton that checks this restriction has at least 2^{16} states.

However, an alternative approach is possible, which is inspired by work on record types [25, 32]. The idea is to encode the state using one type `ATTRS` with as many parameters as there are different attributes. Each of these type parameters ranges over two one-element types:

```
data PRESENT = PRESENT
data ABSENT  = ABSENT
```

```
data ATTRS action method enctype =
  ATTRS action method enctype
```

(For illustration purposes, the type `ATTRS` only considers some attributes of `FORM`. HTML 4.01 defines 132 attributes in total, hence `ATTRS` has 132 parameters.) The element type `ELT` receives an additional (phantom) type parameter. The `make` function creates an empty `FORM` element with type

```
ELT FORM (ATTRS ABSENT ABSENT ABSENT)
```

meaning that no attribute is present, yet.

To keep track of the presence or absence of particular attributes, the functions `add` and `add_attr` receive suitable types (their implementations remain the same as before):

```
add_attr :: (AddAttr t a, AttrValid v a v') =>
  ELT t v -> ATTR a -> ELT t v'
add :: (AddTo s t, AttrFinal t v) =>
  ELT s v' -> ELT t v -> ELT s v'
```

The types mention two new type classes `AttrValid` and `AttrFinal`. The predicate `AttrValid v a v'` implements the transition function: If `v` (= `ATTRS ...`) is the current attribute state and `a` is the next attribute, then `v'` is the next attribute state. Clearly, `AttrValid` is a function because `v'` depends on `v` and `a`. This is specified using a functional dependency. The predicate `AttrFinal t v` determines if the attribute state `v` is a final state for the element with tag `t`.

Here are the class definitions and some instances.

```
class ATTRIBUTE a => AttrValid v a v' | v a -> v'

class TAG t => AttrFinal t v

instance AttrValid
  (ATTRS ABSENT method enctype) ACTION (ATTRS PRESENT method enctype)
instance AttrValid
  (ATTRS action ABSENT enctype) METHOD (ATTRS action PRESENT enctype)
instance AttrValid
  (ATTRS action method ABSENT ) ENCTYPE (ATTRS action method PRESENT)

instance AttrFinal FORM (ATTRS PRESENT method enctype)
instance AttrFinal CDATA (ATTRS ABSENT ABSENT ABSENT)
instance AttrFinal BODY (ATTRS ABSENT ABSENT ABSENT)
```

The instance of `AttrFinal` for FORM states that an action attribute must be present, the other attributes can be arbitrary. `CDATA` and `BODY` elements do not take any of these attributes. Hence, their final attribute states contain `ABSENT` only.

For example:

```
Main> putStr $ show_document $ build_document (body (form id))
ERROR - Unresolved overloading
*** Type      : AttrFinal FORM (ATTRS ABSENT ABSENT ABSENT) => IO ()
*** Expression : putStr $ show_document $ build_document (body (form id))
```

The term is rejected because there is no suitable instance of `AttrFinal`. If we provide the required attribute, then the result is displayed.

```
Main> putStr $ show_document $
      build_document (body (form (attr ACTION "mailto:thiemann")))
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">
<html><body><form action="mailto:peter"></form>
</body>
</html>
```

Finally, if we provide the same attribute twice, a type error occurs, too.

```
Main> putStr $ show_document $ build_document
      (body (form (attr ACTION "mailto:peter"
                  ## attr ACTION "mailto:peter"))))
ERROR - Unresolved overloading
*** Type      : (AttrValid (ATTRS PRESENT ABSENT ABSENT) ACTION a,
                  AttrFinal FORM a) => IO ()
*** Expression : putStr $ show_document $ build_document
      (body (form (attr ACTION "mailto:peter"
                  ## attr ACTION "mailto:peter"))))
```

We have presently chosen not to include this check into the library because it gives raise to another 1595 instance declarations, which bumps the library's size from 148k to 3.2M.

6.2 Exceptions and inclusions

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

```

<!ELEMENT A - - (%inline;)* -(A)      -- anchor -->
<!ELEMENT FORM - - (%flow;)* -(FORM)  -- interactive form -->
<!ELEMENT BUTTON - -
      (%flow;)* -(A|%formctrl;|FORM|ISINDEX|FIELDSET|IFRAME)
      -- push button -->

```

indicates that an `<a>` element may not appear nested inside an `<a>` element. Likewise, `<form>` elements may not be nested, and neither `<a>`, `<form>`, `<isindex>`, ... may appear inside of `<button>` elements.⁶

Interestingly, it seems possible to encode exceptions using a multi-parameter type class, whereas the encoding of inclusions seems to require an extension of the type class model. The key idea to encode negative information comes again from type systems for records, which also need to express the absence of a particular label [25, 32]. We demonstrate the approach using the example above.

In the absence of special row types, we define a data type `ELEMS` with as many type parameters as there are different element tags. In our example, this amounts to

```

data ELEMS a form button isindex =
  ELEMS a form button isindex

```

The two types `PRESENT` and `ABSENT` are again used to signal the presence or absence of particular tags using the `ELEMS` type.

The type `ELT` receives two additional parameters, an `above` parameter and a `below` parameter. The `below` parameter reflects the use of tags *below* the element, whereas the `above` parameter reflects the use of tags *above* and *among the siblings* of the element. Both will be instantiated with a particular instance of the `ELEMS` type. The type class `EXCEPTION` governs the propagation of information between `below` and `above` through the type of the `add` function.

```

add :: (AddTo s t, EXCEPTION s above below) =>
      ELT s above below -> ELT t below oo -> ELT s above below

class EXCEPTION tag above below | tag -> above below

instance EXCEPTION
  A (ELEMS PRESENT form button isindex) (ELEMS ABSENT form button isindex)
instance EXCEPTION
  FORM (ELEMS a PRESENT button isindex) (ELEMS a ABSENT button isindex)

```

⁶The entity references `%inline;`, `%flow;`, and `%formctrl;` macro-expand into content descriptions, that is, regular expressions.

```

instance EXCEPTION
  BUTTON (ELEMS a form PRESENT isindex) (ELEMS ABSENT ABSENT ABSENT ABSENT)
instance EXCEPTION
  ISINDEX (ELEMS a form button PRESENT) (ELEMS ABSENT ABSENT ABSENT ABSENT)

```

The instance declaration for **A** says that the elements below cannot contain an **A**. If there were an **A**, then the type of the corresponding variable would be instantiated to **PRESENT**, thus colliding with the type **ABSENT** required by the **EXCEPTION** class. The instance declaration for **FORM** is quite similar. The remaining elements, **FORM**, **BUTTON**, and **ISINDEX** are “inherited”.

The instance declaration for **BUTTON** says that there must be neither an **A**, nor a **FORM**, nor another **BUTTON**, nor an **ISINDEX**.

For inclusions, the type of **add** is too restrictive. It would be necessary to express the following information:

- **s** and **t** are related by **AddTo** *or* **t** is allowed by an enclosing inclusion declaration
- and **t** is not disallowed by an enclosing exception declaration.

While disallowance and allowance can be formalized using the **EXCEPTION** class above and another type class (using two additional type variables), there remains the problem of expressing the disjunction in the type class system. Presently implemented type checkers can only deal with conjunctions of class predicates.

Progressing from HTML to XML [3] would also solve the problem because XML does not support exceptions, anymore. In fact, in XHTML [33] the side conditions on **<a>** and **<form>** are only mentioned informally because they are not expressible using an XML DTD.

The current library implements neither inclusions nor exceptions. First, they are not necessary due to the imminent transition to XML and XHTML. And second, they would render the library useless due to the enormous increase in size caused by the instance declarations for a type with 89 parameters (the number of tags supported by HTML 4.01), as demonstrated with the attributes.

7 Related work

TkGofer [30] is a toolkit for building graphical user interfaces using Gofer [13] and Tcl/Tk [21]. It employs multi-parameter constructor classes to relate configuration parameters to widgets. This use of type classes statically

avoids errors that would otherwise only be detected dynamically at execution time. This idea inspired our approach of handling attributes using the class `AddAttr`. Still, 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 programming (a standard interface for generating dynamic documents) by Hanus [8], Hughes [11], and Meijer [18]. Meijer’s work provides a thin layer of abstraction on top of the raw access to CGI parameters and convenience functions for generating different types of output. On top of this basic functionality, Hughes has a more sophisticated method for dealing with interactions, based on a generalization of monads. While the previous works deal with Haskell, Hanus’s library is implemented in the functional logic language Curry. It introduces powerful abstractions that allow the specification of event handlers for interactions taken on an HTML document. None of these works makes an attempt to provide more than untyped representations of HTML documents. Hence, we believe that the present work is complementary to these works and something could be gained from their combination.

Type classes are also used as a structuring tool in the various functional hardware modeling frameworks [17, 2]. 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 simple logic program (in fact, a finite automaton) to rule out invalid programs.

Wallace and Runciman [31] propose two alternative ways of using Haskell to represent XML documents. Their type-based encoding maps XML DTDs directly to specialized Haskell data types, thus sacrificing some flexibility. In addition, Haskell data types can only express a subset of content models of elements, so their encoding suffers from similar problems as our simple model. While a more elaborate type structure can address these shortcomings (cf. Sec. 6), Haskell’s data types cannot be extended in this way.

In addition, Wallace and Runciman present a generic encoding, which represents documents in a similar way as our untyped encoding, and they have developed a combinator library for processing XML data in this encoding. This encoding does not impose any validity constraints on generated or processed documents. However, it might be possible to combine this representation with our typed layer. The result would be a highly flexible XML processing library, which admits generic operations like searching and restructuring but which also admits a guaranteed, typed way of dealing with documents.

In the logic programming world, there are several toolkits for generating HTML pages. The PiLLoW toolkit [5] allows for easy creation of documents including CGI functionality. It is widely used to connect logic programs to the WWW. LogicWeb [16] offers an even tighter integration which includes client-side scripting. None of these offers advanced typing features.

The DynDoc facility by Sandholm and Schwartzbach [26] has similar goals as the present paper. They define a language and a type system for dynamically composable documents from scratch and prove its soundness using standard flow analysis techniques. In contrast, our work fully relies on the type system of Haskell (with some standard extensions) and its parameterization facilities, which can exploit the full power of functional programming. Beyond that, DynDoc has facilities that provide additional safety for HTML forms. These facilities extend the work on MAWL [1]. MAWL only admits first-order document templates, where holes can be filled with data items, but not with other document templates. In addition, there is a repetition construct, which can fill a hole repeatedly with the elements of a list.

Last not least, in our previous work [29] we made a first attempt at the library presented here. In retrospect, the implementation considered there was less flexible and unnecessary complicated. In particular, the previous implementation could not easily be extended with generic operations like the ones from Wallace and Runciman's work. The present work addresses all these shortcomings.

8 Conclusion

We have designed and implemented a family of convenient embedded domain specific languages for meta programming of web pages and web sites in Haskell. Haskell's type classes and in particular multi-parameter classes with functional dependencies were instrumental in the construction. We have introduced container-passing style as a means to concisely construct abstractions and fragments of web pages. The resulting programming style is very natural and yields visually appealing programs.

We found the library surprisingly easy and intuitive to use. 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 simple version (without strict DTD checking) sufficient because it captures many common errors (using an element in the wrong place). Initial experiments with the more elaborate static scheme outlined in Section 6.1 yield quite natural and precise typings.

On the negative side, type errors are a fairly 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 might also appreciate a syntax which is closer to HTML. This is subject to further investigation.

Acknowledgements

Thanks to Sebastian Schulz for testing the library and helping with its construction.

References

- [1] David Atkinson, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. USENIX.
- [2] 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.
- [3] Tim Bray, Jean Paoli, and C. Michael Sperberg-MacQueen. Extensible markup language (XML) 1.0 (W3C Recommendation). <http://www.w3.org/TR/REC-xml>, feb 1998.
- [4] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
- [5] 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.
- [6] Allaire Corp. HomeSite. <http://www.allaire.com/>.
- [7] Ralf S. Engelschall. Website meta language (wml). <http://www.engelschall.com/sw/wml/>.

- [8] Michael Hanus. Server side Web scripting in Curry. In *Workshop on (Constraint) Logic Programming and Software Engineering (LPSE2000)*, London, July 2000.
- [9] Haskell 98, a non-strict, purely functional language. <http://www.haskell.org/definition>, December 1998.
- [10] Html 4.01 specification. <http://www.w3.org/TR/html4/>, December 1999.
- [11] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
- [12] SoftQuad Software Inc. HoTMetaL. <http://www.softquad.com/>.
- [13] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.
- [14] Mark P. Jones. Hugs Online — embracing functional programming. <http://www.haskell.org/hugs/>, June 1999.
- [15] Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Proc. 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244, Berlin, Germany, March 2000. Springer-Verlag.
- [16] Seng Wai Loke and Andrew Davison. Logic programming with the World-Wide Web. In *Proceedings of the 7th ACM Conference on Hypertext (Hypertext '96)*, pages 235–245, Washington DC, USA, March 1996.
- [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*, 10(1):1–18, January 2000.
- [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] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: An exploration of the design space. In John Launchbury, editor, *Proc. of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997. Yale University Research Report YALEU/DCS/RR-1075.
- [24] John Punin and Mukkai Krishnamoorthy. ASHE (a simple HTML editor) - xhtml. <http://www.cs.rpi.edu/~puninj/ASHE/>, November 1996.
- [25] 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.
- [26] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic web documents. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 290–301, Boston, MA, USA, January 2000. ACM Press.
- [27] Internet Software Technologies. HTMLed Pro32. <http://www.ist.ca>.
- [28] Texinfo. <http://texinfo.org>.
- [29] Peter Thiemann. Modeling HTML in Haskell. In *Practical Aspects of Declarative Languages, Proceedings of the Second International Workshop, PADL'00*, volume 1753 of *Lecture Notes in Computer Science*, pages 263–277, Boston, Massachusetts, USA, January 2000.
- [30] Ton Vullings, 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.
- [31] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In Peter Lee, editor, *Proc. International Conference on Functional Programming 1999*, pages 148–259, Paris, France, September 1999. ACM Press, New York.
- [32] 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*.

- [33] XHTML 1.0: The extensible hypertext markup language. <http://www.w3.org/TR/xhtml1>, January 2000.