A Typed Representation for HTML and XML Documents in Haskell

PETER THIEMANN
Universität Freiburg, Germany
(*e-mail:* thiemann@informatik.uni-freiburg.de)

Abstract

We define a family of embedded domain specific languages for generating HTML and XML documents. Each language is implemented as a combinator library in Haskell. The generated HTML/XML documents are guaranteed to be well-formed. In addition, each library can guarantee that the generated documents are valid XML documents to a certain extent (for HTML only a weaker guarantee is possible). On top of the libraries, Haskell serves as a meta language to define parameterized documents, to map structured documents to HTML/XML, to define conditional content, or to define entire web sites.

The combinator libraries support *element-transforming style*, a programming style that allows programs to have a visual appearance similar to HTML/XML documents, without modifying the syntax of Haskell.

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

1 Introduction

HTML and XML (HTML 4.01, 1999; XML1.0, 2000; XHTML 1.0, 2000) have emerged as standard formats for the dissemination of information. HTML is primarily targeted at delivering information via the Web. Besides some fixed means for structuring documents, it includes facilities to control their layout on the screen. In contrast, XML has been developed as an extensible format for tree structured documents and it does not include a layout semantics per se. In both cases, trees are represented using strings with markup notation to delimit the individual nodes (*elements* in XML terminology) of the tree. Each element may have a finite number of *child elements* and a finite number of *attributes*. For example,

```
<DL compact="compact">
  <DT> DTD </DT>
  <DD> Document Type Definition </DD>
</DL>
```

represents a tree with five elements. The root element has *name* `DL`, an *attribute* with name `compact` and (string) value `compact`, and two child elements. The first child has name `DT` and, as only child, the text `DTD`. The second child has name `DD` and also a textual child. Each non-textual element starts with an *opening tag*, *e.g.*, `<DL>`, and ends with a *closing tag*, *e.g.*, `</DL>`, where both tags carry the name of the element.

Early information sources in the Web were static, in the sense that a request for a document resulted in delivering the contents of a file via the network. This simple picture changed quickly and radically. Today's information sources are dynamic and highly configurable. As requests are often parameterized (by language preference, image quality, and so on), servers must compose their responses from templates, results of computations, and data base accesses.

Hence, there is an increasing demand for convenient ways of creating HTML and XML documents dynamically. Unfortunately, many applications do so in an inappropriate way by treating documents as strings. However, HTML and XML have structural restrictions that are easily violated using a string-based approach. First, documents must be *well-formed*, which means that opening and closing tags match. Moreover, documents must be *valid*, which means that they conform to a DTD (document type definition). A DTD governs the nesting of tree elements. For example, the following part of HTML's DTD governs the contents of `DL` elements:

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

It specifies that the children of a `DL` element must be a non-empty sequence of either `DT` or `DD` elements. The *content description* `(DT | DD)+` is essentially a regular expressions.

The DTD also governs the set of attributes that each element may assume. For example, here is the declaration of the `compact` attribute for the `DL` element (excerpted from the HTML DTD):

```
<!ATTLIST DL compact     (compact)      #IMPLIED>
```

It tells us that a `DL` element can take an attribute named `compact`. The value of this attribute must be the string `compact`, but the attribute needs not be present (expressed by `#IMPLIED`).

In a valid document, each element carries at most one occurrence of each declared attribute. An attribute declaration may require that certain attributes be present and it can impose type restrictions on the value of the attribute.

Early browsers and HTML processors where fairly lax about these restrictions and corrected many of the blunders automatically. Such an approach was feasible because HTML is defined by one fixed DTD. In addition, HTML's DTD is rather permissive because it was created after the fact. With XML, each organization can define its own DTDs for its applications and validating XML processors (for example, browsers) must report violations of the DTD and may reject invalid documents. Hence, it is important that generated documents conform to their stated DTD.

It is easy to construct libraries that support the generation of *well-formed* HTML and XML. It is more demanding to support the generation of *valid* HTML and XML. The latter is the design goal of the libraries that we present in this work:

> Whenever the generating program is type correct,
> the generated document should be valid.

The main contribution of this work is the demonstration that it is possible to create practically useable Haskell libraries that achieve this goal by exploiting a few recent extensions to its type system. For pragmatic reasons, we are offering a number of alternatives with the following weaker guarantees.

Well-formedness. In a well-formed document, opening and closing tags match and attributes have the form `name="value"`, where `value` is an arbitrary string.

Weak validity. A document is *weakly valid* if it is well-formed and valid with respect to a flattened version of its DTD. To obtain the flattened version of a DTD, each content description c is replaced by $(\mathtt{L}_1 \mid \ldots \mid \mathtt{L}_n)*$ where $\mathtt{L}_1, \ldots, \mathtt{L}_n$ are the distinct element names that occur in c.

Attributes are only admissible if they are specified in the DTD. The types of the attribute values are also checked, but it is not checked whether required attributes are present, nor whether attributes are given multiple times.

Elementary validity. A document is *elementary valid* if it is weakly valid and if the contents of each element satisfies the content description given for it.

Validity. A document is valid if it is elementary valid and all attribute occurrences conform to the DTD (XML1.0, 2000).

We will sometimes say *full validity* instead of just validity to distinguish it from the weaker notions.

Our approach builds on a generic representation for XML elements. The representation ensures well-formedness and it comes with a pretty-printer that renders documents in XML syntax. We wrap this representation in three differently typed combinator libraries that guarantee weak, elementary, and full validity, respectively. Each library models the information from the DTD in Haskell's type class system, relying on multi-parameter type classes (Peyton Jones *et al.*, 1997) and functional dependencies (Jones, 2000), in particular.

All typed libraries wrap the underlying generic representation of an XML element into a value of type `ELT t`, where `t` is a *tag type* which determines the name of the represented element. The only operation on values of type `ELT t` is the function

```
add :: AddTo s t => ELT s -> ELT t -> ELT s
add elem child = ...
```

which adds a child of type `ELT t` to an element of type `ELT s`. The predicate `AddTo s t` in the type of `add` refers to a two-parameter type class. It ensures that the new child is admissible according to the DTD. The type class `AddTo` and the tag types both depend on the DTD and they are generated from it:

- for each element name `t`, generate a tag type `t` (by a slight abuse of language, we sometimes use the element name for the tag type and vice versa);
- for each pair of tag types `s` and `t`, where the name `t` occurs in the content description for `s` elements, generate an instance declaration `instance AddTo s t` (this achieves weak validity).

For instance, according to the HTML DTD, the tag type for the `DL` element is `DL`, the type for `DL` elements is `ELT DL`, and the instance declarations concerning `DL` elements and its children are

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

They express that `add` may be used at types

```
add :: ELT DL -> ELT DD -> ELT DL
add :: ELT DL -> ELT DT -> ELT DL
```

(among others defined by further instance declarations). Hence, both `DD` and `DT` elements are allowed as child elements of a `DL` element.

Building on this framework, the library defines, for each element name `t`, an *element constructor*, *i.e.*, a function that creates an empty `t` element. There is also a higher-order version of the element constructor for `t` that takes as a parameter an *element transformer* function of type `ELT t -> ELT t`, which adds children to the `t` element, and that returns another element transformer of type `ELT s -> ELT s`, which adds the constructed `t` element to an `s` element. The resulting higher-order element constructor has a type of the form

```
AddTo s t => (ELT t -> ELT t) -> (ELT s -> ELT s)
```

For instance, the higher-order constructors for `DL`, `DD`, and `DT` elements are

```
dl :: AddTo s DL => (ELT DL -> ELT DL) -> ELT s -> ELT s
dd :: AddTo s DD => (ELT DD -> ELT DD) -> ELT s -> ELT s
dt :: AddTo s DT => (ELT DT -> ELT DT) -> ELT s -> ELT s
```

The higher-order constructors enable *element-transforming style*, a programming style which treats single elements, attributes, and groups of elements and attributes in a uniform way. Element-transforming style is the preferred way of using the library because it avoids some typing problems (see Sec. 2.2.1) and also leads to a natural appearance of document generators. For example, the expression

```
dl (attr COMPACT "compact" ##
    dt (text "DTD") ## dd (text "Document Type Definition"))
```

generates the example document at the beginning of this section (the infix function `##` composes element transformers). The function `attr` inserts the attribute `COMPACT`. It will be discussed in Sec. 2.2.4.

We develop the typed encodings with example HTML documents and concentrate on weak validity, since our experience indicates that weak validity gives sufficient guarantees in practice. Later on, we generalize to XML by specifying a translation from an XML DTD to a specialized Haskell module. We also consider strengthening the guarantees of the library for elementary and full validity. By choosing other translations, the generated library enforces weak, elementary, or full validity. It is not possible to achieve full validity for HTML 4.01 because it is defined by an SGML DTD which relies on features not present in XML. Still, we obtain a very good approximation. However, full validity can be achieved for XHTML. See Sec. 6.3 for a detailed discussion.

While the encodings for weak validity and elementary validity are practically useful for generating parameterized documents, the encodings for full validity are too restrictive for that purpose. Still, they demonstrate that a Haskell compiler may be used as a validation tool for XML documents.

Our libraries are being used to generate Web pages offline and to construct CGI programs (WASH, 2001). The current version of the library, generated for HTML 4.01 from the official DTD (HTML 4.01, 1999), is available from the author's Web page[1].

[1] `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH/`

1.1 Related Work

In previous work (Thiemann, 2000) we made a first attempt at the library presented here. The previous implementation is less flexible and unnecessary complicated.

MAWL (Atkinson *et al.*, 1997) is one of the first languages for generating HTML documents. It relies on first-order templates, where holes can be filled with data items, but not with other document templates. There is a repetition construct, which can fill a hole repeatedly with the elements of a list.

MAWL has been refined in a number of ways by Sandholm and Schwartzbach (2000), who define a language and a type system for dynamically composable documents with higher-order templates from scratch. They define an inference engine based on standard flow analysis techniques and prove its soundness. Their type system provides form-specific type information and it ensures that composition does not destroy the document structure. Our libraries also provide for dynamically composable documents. While we do not provide form-specific information, we guarantee validity of the generated documents in various degrees. Furthermore, our library is integrated into Haskell.

Subsequent work by Brabrand and others (2001) addresses the issue of validity using a type system.

Wallace and Runciman (1999) describe Haskell libraries for parsing, unparsing, and processing XML. They have two different approaches for processing XML. The generic approach uses one fixed data type to represent documents and it comes with a powerful set of combinators for processing documents. While this library is suitable for document generation, it only guarantees well-formedness of the output. In contrast, our library provides different degrees of validity with respect to a given DTD.

In their second approach, Wallace and Runciman transform an XML DTD into a number of specialized Haskell data types and provide functions for parsing from and unparsing to XML syntax. The actual functions to process elements of these data types must be written from scratch; the document-processing combinators are not applicable. This approach guarantees that the resulting XML document is valid by giving up a lot of flexibility in processing. In contrast, our approach uses an underlying generic representation and employs the type system for further guarantees.

Some libraries for CGI programming (Hanus, 2000; Hughes, 2000; Meijer, 2000) rely on generic representations, similar to the one chosen by Wallace and Runciman, to generate their output. These libraries make no guarantees beyond well-formedness.

XDuce (Hosoya & Pierce, 2000; Hosoya *et al.*, 2000) is a typed first-order language for processing XML documents. Its type system is based on regular expressions, which describe the nesting of elements, and subtyping. The element part of a DTD can be translated to XDuce types without changing its semantics.

A companion paper (Hosoya & Pierce, 2001) defines a pattern-matching facility which allows for regular expressions in patterns. The typing discipline (regular expression types) provides precise typings for pattern variables and guides the checks for redundant patterns and exhaustiveness of pattern matching.

Type-indexed rows (Shields & Meijer, 2001) form the basis of the language XMλ, a higher-order functional language for typed processing of XML documents. Type-indexed rows (TIR) generalize record typing by indexing a type not with a set of field names, but rather with a set of types. TIRs are well suited for XML document processing because they can express untagged unions (by using a TIR to generate a sum type) as well as unordered sequences (by using a TIR to generate a "record" type). Like XML, TIRs require one-unambiguous content descriptions (Brüggemann-Klein & Wood, 1998).

Both approaches, XDuce and XMλ, are applicable to document generation as a special case of document transformation. In comparison to either approach, our libraries are specialized to a particular DTD, they do not consider the typed inspection of documents, and their types fit into Haskell's type system, so that it is not necessary to learn a new type system.

There are quite a few approaches to XML query languages that are loosely related to our work. We just pick two illustrative examples. YAT (Cluet *et al.*, 1998) is a system for building mediators. A mediator performs transparent data conversion between different formats. YAT consists of a number of converters from external formats into an internal, tree-based format. A particular feature of YAT is its transformation language YATL that works on this internal format. It is a pattern-based language with the distinctive feature that programs may be instantiated according to a particular subject pattern. Programs may also be composed. YATL has a type system whose primary purpose is to check that composition is safe. Generated trees conform to the patterns that generate them, and these output patterns can encode similar information than a DTD.

XML Query Algebra (Fernandez *et al.*, 2001) is a typed XML transformation language. It works on a generic representation of XML and enforces further guarantees through the type system. The type system is closely related to XDuce and is largely compatible with XML Schema (World-Wide Web Consortium, 2000a; World-Wide Web Consortium, 2000b).

In the logic programming world, there are several toolkits for generating HTML pages. The PiLLoW toolkit (Cabeza & Hermenegildo, 1997) allows for easy creation of documents including CGI functionality. It is widely used to connect logic programs to the WWW. LogicWeb (Loke & Davison, 1996) offers an even tighter integration which includes client-side scripting. None of these offers advanced typing features.

1.2 Overview

In Section 2, we first introduce the main concepts and then work through three simple examples to explain the programmer's view of the library. Section 3 gives a brief introduction to document type definitions (DTDs). Then, Section 4 deals with the implementation of the library for the special case of weak validity. Starting from the underlying, untyped representation, we move on to define the typed wrapper on top of the untyped layer. Finally, we discuss the type classes that determine how elements and attributes may be put together. Section 5 defines the translation from a DTD to an instance of the library for weak validity. In Section 6, we discuss the progression from weak validity to full validity (for XML) and explain the problem with modeling full validity for HTML in Haskell's type system. Section 7 concludes.

In the paper, we assume some familiarity with Haskell, HTML, and XML. Strictly speaking, the libraries are *not* valid Haskell98 programs due to the use of multi-parameter type classes (Peyton Jones *et al.*, 1997) and functional dependencies (Jones, 2000) (only for elementary and full validity). However, a number of Haskell implementations support this extension.[2]

2 Examples

After a brief overview of the functionality of the HTML library, we work through some examples. The first example is a Hello World document. For pedagogical reasons, the example does not use the higher-order element constructors mentioned in the introduction. In the next section, when we move on to parameterized documents, we point out the deficiencies of using plain element constructors and introduce element-transforming style. The second example describes a prototype implementation of a simple hypertext system.

2.1 Hello World

In this section, we construct a generator for a static document:

```
<html><head><title>Hello World!</title>
</head>
<body><h1>Hello World!</h1>
</body>
</html>
```

The basic pattern for constructing a document is to create an empty element and then add child elements to it. The function

```
make :: TAG t => t -> ELT t
```

maps a tag type `t` to an empty element with the corresponding name. The type of an element `ELT t` indicates its name `t`. The predicate "`TAG t =>`" restricts the possible instances of `t` to elements of the type class `TAG`. A type class is a set of types that permit a particular set of operations, the *member functions*. In this case, the type class `TAG` characterizes the set of admissible tag types and `make` is its member function.

For each element name *`Tag`* of HTML, there is a data type *`Tag`* with a single element *`Tag`* and each of these types is an instance of `TAG`. Hence, the expression `HTML` has type `HTML` and the predicate `TAG HTML` is satisfied so that the expression

[2] In particular, the code in this paper has been tested with `ghc` version 5.00 and the February 2001 release of the Haskell interpreter Hugs in `-98` mode. The storage space for instances in Hugs has been increased to 10000 (define `NUM_INSTS` in `src/prelude.h`).

```
make HTML
```

has type

```
ELT HTML
```

and stands for the document

```
<html></html>
```

The function `text' :: String -> ELT CDATA` is the constructor for textual elements. The type `CDATA` is the tag type for these elements. Hence,

```
hwtext :: ELT CDATA
hwtext = text' "Hello World!"
```

constructs the textual element used in the example.

The next task is the addition of a child element to an element. The function

```
add :: AddTo s t => ELT s -> ELT t -> ELT s
```

serves this purpose. The parent element has type `ELT s` whereas the child element has type `ELT t`. The two-parameter type class `AddTo` implements a binary relation between the name `t` of a child element and the name `s` of its parent. An *instance declaration* states that a particular pair of types belongs to `AddTo`. For example, the declarations

```
instance AddTo TITLE CDATA
instance AddTo H1    CDATA
```

indicate that

```
make TITLE `add` text' "Hello World!"
make H1    `add` text' "Hello World!"
```

are both acceptable expressions of type `ELT TITLE` and `ELT H1`, respectively[3].

Consulting the DTD assures us that that following combinations must also be acceptable.

```
instance AddTo HEAD TITLE
instance AddTo HTML HEAD
instance AddTo BODY H1
instance AddTo HTML BODY
```

Hence, the complete code for our example is

```
make HTML
 `add` (make HEAD `add` (make TITLE `add` hwtext))
 `add` (make BODY `add` (make H1    `add` hwtext))
```

This expression has type `ELT HTML` and it stands for the document shown at the beginning of this section.

If there is no instance declaration for a particular combination of element name and child element name, then the type checker prevents us from adding the child element. For example, it is illegal to put a header `H1` element into a title element:

```
> make TITLE `add` make H1
ERROR - Unresolved overloading
*** Type       : AddTo TITLE H1 => ELT TITLE
*** Expression : add (make TITLE) (make H1)
```

[3] In Haskell, an identifier in grave accents (like `` `add` ``) is an infix operator.

2.2 Parameterized Documents

Although direct use of `make` and `add` as in the preceding subsection is sufficient to demonstrate the basic features of the library, their use is cumbersome and has severe limitations. In this subsection, we identify the limitations and propose a solution that works well for many parameterized documents.

2.2.1 Limitations of Simple Element Construction

Suppose we want to generate the body of a document with a parameterized function that adds a fixed header and footer:

```
genBody' contents =
  make BODY `add` (make H1)
            `add` contents
            `add` (make ADDRESS)
```

This function type-checks, but the inferred type is unfortunate news:

```
genBody' :: AddTo BODY t => ELT t -> ELT BODY
```

Why is it unfortunate? Because the parameter `contents` is restricted to *exactly one element*. It cannot be empty and it cannot stand for more than one element. The straightforward idea of "somehow" passing a standard list of elements fails because standard lists are homogeneous. In a homogeneous list, each element has the same type. Such a restriction rules out a list containing both, a textual element of type `ELT CDATA` and a definition list of type `ELT DL`. Hence, standard lists are not suitable for grouping elements.

A similar problem occurs when we consider conditional content.

```
mytext' italic =
  if italic then make I `add` text' "mytext"
            else text' "mytext"
```

In this case, the type checker rejects the definition. The `True`-branch of the conditional has type `ELT I` whereas the `False`-branch has type `ELT CDATA`. Hence, a type clash results.

We solve the above problems by proposing *not* to use the constructors directly, but rather wrap the them into higher-order functions. The resulting *element-transforming style* of programming yields a satisfactory and natural programming model for document generators.

2.2.2 Element-Transforming Style

The idea of element-transforming style is to never return elements directly but rather deal with them indirectly using transformer functions. Hence, an element constructor takes as a parameter a transformer that modifies the constructed element and returns as its result a transformer that adds the constructed element to an enclosing element. For example, the combinator for `TITLE` is

```
title :: AddTo s TITLE => (ELT TITLE -> ELT TITLE) -> (ELT s -> ELT s)
```

that is, it maps a transformer for `TITLE` elements to a transformer for `s` elements, provided that the `s` element admits the addition of `TITLE`.

The combinator for textual elements is simpler, since text cannot be transformed:

```
text :: AddTo s CDATA => String -> (ELT s -> ELT s)
```

Let us now first review our "Hello World" example in this style, and then check that element-transforming style addresses the two problems mentioned above. The revised expression

```
html (head (title (text "Hello World!"))
   ## body (h1    (text "Hello World!")))
```

type checks with type `AddTo s HTML => ELT s -> ELT s`. The code is visually more appealing than the previous attempt and arguably more concise than the HTML source generated from it.

It remains to explain the combinator `##`. The argument of an element constructor is a transformer of the element. An addition of a child element is an elementary transformation. If we want to add more children, then we need to compose transformations using `##`. Since transformations are just functions, composition is just (forward, diagrammatic) composition of functions:

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

Whenever we do not want to transform an element, we plug in the empty transformation, the identity function:

```
empty x = x
```

It turns out that element-transforming style gives us a powerful means to deal with sequences of elements (and attributes, as we will see). Intuitively, each constructor returns a singleton sequence, `empty` returns an empty sequence, and `##` concatenates sequences. Due to the implementation by function composition, concatenation is an associative operation that runs in constant time.

The ability to talk about sequences of elements solves our problem with the parameterized document:

```
genBody contents =
  body (h1 empty ## contents ## address empty)
```

The function `genBody` has type `(AddTo s BODY) => (ELT BODY -> ELT BODY) -> ELT s -> ELT s`. Hence, contents can assume an arbitrary sequence of elements, provided that each participant of the sequence is a transformer for `BODY`. All three expressions below are legal and have type `(AddTo s BODY) => ELT s -> ELT s`.

```
genBody empty
genBody (text "Heureka!")
genBody (h2 empty ## h2 empty)
```

Element-transforming style also solves the problem with conditional content: The function `mytext` defined by

```
mytext italic =
  if italic then i (text "mytext")
            else text "mytext"
```

has type `(AddTo s I, AddTo s CDATA) => Bool -> ELT s -> ELT s`, which expresses that the enclosing element must be ready to accept an `I` element as well as a `CDATA` element.

2.2.3 The Toplevel Element

There is no direct access to the constructed elements anymore. Hence, the library provides a combinator

```
build_document :: (ELT HTML -> ELT HTML) -> ELT DOCUMENT
```

to construct a toplevel element. The expression `build_document` *tr* creates an empty `HTML` element, applies the `HTML` transformer *tr* to it, and returns a data structure that represents an entire HTML document, including header information. For example, pretty-printing the value of

```
build_document (head (title (text "Hello World!"))
            ## body (h1    (text "Hello World!")))
```

yields

```
<!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>
</body>
</html>
```

2.2.4 Attributes

Attributes names are treated in the same way as element names. For each attribute name, there is a one-element type *Attr* with element *Attr*. Each of these types is an instance of a type class `ATTRIBUTE`. For example, the `href` attribute gives rise to the type `HREF` with element `HREF`. Similar to the relation between tag types and elements, the type `HREF` only provides the *name* of the attribute. The actual attribute instance (a name-value pair) is represented by a value of type `ATTR HREF`.

Similar as with elements, there is a direct function to add an attribute to an element.

```
add_attr :: AddAttr t a => ELT t -> ATTR a -> ELT t
```

The type class `AddAttr t a` used in its type determines whether a `t` element admits an attribute with name `a`. For example, the declaration

```
instance AddAttr A HREF
```

determines that an `HREF` attribute is admissible for an `A` element.

Clearly, we want to group attributes in the same way as we have worked it out for elements above. Hence, we proceed immediately to the element-transforming style definition of the attribute constructors.

```
attr :: (AttrValue a v, AddAttr t a) => a -> v -> ELT t -> ELT t
```

The first parameter (of type `a`) is the attribute name. The second parameter (of type `v`) is the attribute value. The result is an element transformer for elements with name `t`, provided that

- `AddAttr t a`: the attribute name is admissible for a `t` element and

- `AttrValue a v`: the type of the attribute value is admissible for this attribute name.[4]

For example, the expression `attr HREF "mailto:thiemann@acm.org"` evaluates to an element transformer of type `AddAttr t HREF => ELT t -> ELT t`.

Again, element-transforming style makes it easy to define parameterized attributes.

```
hlink :: (AddTo t A, AttrValue HREF v)
      => (ELT A -> ELT A) -> v -> ELT t -> ELT t
hlink body url =
  a (body ## attr HREF url)
```

The definition of `hlink` shows that the attributes for an element can appear anywhere in the transformer for this element. Supplying them through transformers immediately enables grouping of attributes and it frees us from supplying extra arguments to the constructors or having special attribute-sensitive constructors, which is the approach commonly taken in HTML libraries (Meijer, 2000; Hanus, 2001).

2.3 A Larger Example: Simple Hypertext

In this subsection, we consider the translation of a simple hypertext system to HTML. The system structures 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.

The datatype for a node has six fields.

```
data Node =
  Node  String          -- name of node
        [NodeContent]   -- contents of the node
        [Node]          -- list of children
  -- administrative fields (filled in automatically):
        String          -- name stub for generated files
        Int             -- unique number  of node
        [Int]           -- section counter

type NodeContent = String
```

The author of such a structure only has to specify the contents of each node and to list the children. The function `node2html` below translates one node into the corresponding HTML data structure.

```
node2html :: Node -> Maybe Node -> Maybe Node -> Maybe Node -> ELT DOCUMENT
node2html (Node name contents children _ _ count) m_next m_previous m_up =
  html_doc title
```

[4] This pattern should be clear by now, so we defer the explanation of `AttrValue` to Sec. 4.5.

```
  (  maybe_link "Next" m_next
  ## maybe_link "Previous" m_previous
  ## maybe_link "Up" m_up
  ## hr empty
  ## pars contents
  ## my_menu node_ref children)
  where
    title = show_sec_count count ++ name
```

The function `html_doc` takes a title `String` and an element transformer for a `<body>` element to create a simple standard document structure. In the body, there are three hyperlinks labeled `Next`, `Previous`, and `Up` created by `maybe_link`. The `maybe_link` function takes a label of type `String` and an optional node. If there is a node present, then `maybe_link` creates a labeled link. Otherwise, the label appears as plain text. Next there is a horizontal rule, followed by the text structured in paragraphs (`pars contents`) and finally a menu of the children (`my_menu node_ref children`), where `node_ref` creates a link to a node. Of these, the functions `my_menu` and `pars` are probably the most interesting ones.

```
my_menu :: (AddTo MENU a, AddTo b MENU) =>
        (item -> ELT LI -> ELT a) -> [item] -> ELT b -> ELT b
my_menu make_ref [] =
  empty
my_menu make_ref children =
  menu (foldr add_node empty children)
  where
    add_node node items = li (make_ref node) ## items
```

The function `my_menu` takes as arguments a function `make_ref` that constructs a link from an `item` and a list of menu `items`. If the list of items is empty, then no element is constructed. Otherwise, `my_menu` creates a `<menu>` that contains one `<li>` element with a link for each `item`.

The `pars` function takes a list of strings and transforms it into a sequence of paragraphs.

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

Each input string is first transformed by `text` into a textual element, which is wrapped into a paragraph by `p`. The function `Prelude.map` is the usual map function for lists.[5]

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, an optional 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. The code is available through the WASH-web page (WASH, 2001).

[5] The qualification by the module name `Prelude.` is necessary because the library also defines a function `map`, which implements the HTML element `map`. The other occurrences of dots `.` are infix operators that denote (backwards) function composition.

```
<!ATTLIST OL
  type       CDATA        #IMPLIED  -- numbering style --
  compact    (compact)    #IMPLIED  -- reduced interitem spacing --
  start      NUMBER       #IMPLIED  -- starting sequence number --
  >
```

Fig. 1. Definition of attributes for `OL` (excerpt)

3 Document Type Definitions

The validity of a particular combination of an element name and a child element name as well as of an element name and an attribute is governed by a DTD (document type definition). 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.

Basically, a DTD contains two kinds of entries, element definitions and attribute definitions[6]. An element definition defines an element name and declares its child elements using a content description. 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 element, the two dashes state that both the opening tag and the closing tag must be written (an `O` indicates that they are optional), and the `(DT | DD)+` is the content description. The latter specifies the names of the child elements. In this case, the child elements may have names `DT` or `DD`, and at least one of them must be present. The content description 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.[7] Also, `EMPTY` is a content description, which is self-explanatory. The restriction on regular expressions is that they must be one-unambiguous (Brüggemann-Klein & Wood, 1998).

Figure 1 shows a typical attribute definition from the HTML4.01 DTD. It declares admissible attributes and enforces a simple type discipline on attribute values. In the definition

- `OL` is the name of the element to which the attributes belong,
- `type`, `compact`, and `start` are the names of attributes,
- `CDATA`, `(compact)`, and `NUMBER` specify their respective types: a string, an enumeration type with one element `compact`, and a number, and
- `#IMPLIED` specifies that the attribute is optional and has no default value. Alternatively, attributes can be `#REQUIRED` or this column can provide a default value.

[6] We ignore the abbreviation and structuring mechanisms of entities and conditional sections, since they can be eliminated by a pre-pass.
[7] SGML has an additional operator `a & b`, which denotes an arbitrary interleaving of `a` and `b`.

The text between the pairs of dashes `--` is a comment.

4 Implementation

In this section, we explain the implementation of the library for weak validity. We start out with the underlying representation for well-formed HTML documents and build a typed layer on top of it. The main tools for building the typed layer are type classes and *phantom types* (parameterized types where the type parameter does not appear on the right side of the definition).

4.1 Data Representation

A generic representation for XML elements must define two abstract datatypes of attribute instances and elements.

```
data ATTR_    -- abstract

attr_      :: String -> String -> ATTR_
attr_name  :: ATTR_ -> String
attr_value :: ATTR_ -> String
```

A value of type `ATTR_` is an attribute instance. It is created from two strings, the attribute name and its value, by the constructor `attr_`. The selector functions `attr_name` and `attr_value` extract the respective components, again.

```
data ELEMENT_ -- abstract

element_   :: String -> [ATTR_] -> [ELEMENT_] -> ELEMENT_
empty_     :: String -> [ATTR_] -> ELEMENT_
cdata_     :: String -> ELEMENT_
doctype_   :: [String] -> [ELEMENT_] -> ELEMENT_

add_       :: ELEMENT_ -> ELEMENT_ -> ELEMENT_
add_attr_  :: ELEMENT_ -> ATTR_ -> ELEMENT_

putElement :: ELEMENT_ -> IO ()
```

The datatype `ELEMENT_` models elements. It has four constructors. The first one, `element_` creates an element from an element name (a string), a list of attribute instances, and a list of child elements. The `empty_` constructor is intended to model HTML elements like `<hr>` whose content description is empty and whose closing tags may be omitted. Elements constructed with `empty_` are only printed differently to elements constructed with `element_`.[8] The `cdata_` constructor creates a textual element from a string, and the `doctype_` constructor creates the toplevel element of a document.

The expression `add_` *`el child`* adds the element *`child`* to the list of elements of *`el`*. The expression `add_attr_` *`el at`* adds the attribute instance *`at`* to the list

[8] This constructor will probably be phased out in the transition to XML due to XML's shorthand notation for empty elements `<hr/>`.

of attributes of element *el*. Finally, the expression putElement *el* returns an IO action that prints the element *el* in HTML syntax.

For example,

```
element_ "DL" []
  [element_ "DD" [] [cdata_ "Document Type Definition"],
   element_ "DT" [] [cdata_ "DTD"]])
```

prints as

```
<DL><DT>DTD</DT>
<DD>Document Type Definition</DD>
</DL>
```

The low level representation keeps the list of child elements in reverse order so that the add_ operation runs in constant time.

4.2 Types for Attributes

The typed layer for attributes consists of a phantom type for attribute instances, a number of singleton types that stand for attribute names, a type class that collects these singleton types, and a type class that enforces a simple type discipline on the values of an attribute.

The phantom type is realized by

```
data ATTR a = ATTR { unATTR :: ATTR_ }
```

which defines the constructor ATTR and the selector unATTR. The intention is that the type variable, a, is only ever instantiated by types that stand for attribute names. Later on, we use this typing to relate admissible attributes to elements using the type class AddAttr. The attribute-name types are collected in the type class ATTRIBUTE:

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

The first line says that every member type of ATTRIBUTE must belong to the predefined class Show[9]. Every type t that belongs to Show has a show function of type t -> String. The default implementation of ATTRIBUTE's member function show_name is to convert the attribute name to a string and then convert this string to all lower case.

For example, the declarations for the attribute TYPE of <ol> (cf. Fig. 1) are as follows.

```
data TYPE = TYPE deriving Show
instance ATTRIBUTE TYPE
```

[9] See section 6.3.3 of (Haskell98, 1998)

The first line constructs a one-element type `TYPE` and instructs the compiler to automatically make it into ("derive") an instance of `Show`.[10] The second line makes `TYPE` an instance of the `ATTRIBUTE` class. Since there is no overriding definition for `show_name`, its default definition (from the class declaration) is used for `TYPE`.

As explained in Sec. 3, a DTD enforces a simple type discipline on the attribute values. Consequently, we provide a type class `AttrValue` that relates an attribute-name type to the types of its potential values. The class `AttrValue` has no member functions and is just used to restrict the type of a function `mkAttr` that takes an attribute name and a value and constructs an attribute of the right type. The type of the attribute name, `a`, must be a member of `ATTRIBUTE` and the type of the value, `v`, must be a member of `Show`, so that it can be converted to a string.

```
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))
```

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

For example, the attributes specific to `<ol>` (cf. Fig. 1) require the following instance declarations:

```
instance AttrValue TYPE String
instance AttrValue COMPACT COMPACT_compact
instance AttrValue START Integer
```

The type `COMPACT_compact` has just a single element, which shows as `compact`. With the above definitions in place, we can write code like this:

```
mkAttr TYPE "i"                 :: ATTR TYPE
mkAttr COMPACT COMPACT_compact  :: ATTR COMPACT
mkAttr START (42::Integer)      :: ATTR START
```

4.3 Types for Elements

The typed representation of elements introduces another phantom type.

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

In addition, there is one data type (tag type) for each HTML element name. These types are the candidates for the parameter `t` of `ELT`. For example, the tag types for `<dl>`, `<dd>`, and `<dt>` are defined thus

```
data DL = DL deriving Show
data DD = DD deriving Show
data DT = DT deriving Show
```

Every tag type is a member of the type class `TAG`.

```
class Show t => TAG t where
  make :: t -> ELT t
  show_tag :: t -> String
--
```

[10] The `deriving` mechanism is only available for a few predefined classes.

```
  make = make_standard
  show_tag = map toLower . show

make_standard t = ELT (element_ (show_tag t) [] [])
make_empty    t = ELT (empty_   (show_tag t) [])
```

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 a wrapped element reflects its name. The default implementation of `make`, `make_standard`, uses the `element_` constructor. Elements declared as empty in the DTD override `make` with `make_empty` in their instance declaration. Elements constructed with `make` have neither children nor attributes, initially.

4.4 Adding Elements

This section considers the addition of a weakly valid child element to a weakly valid element. Such an addition preserves weak validity if the name of the child element is mentioned in the content description of the parent element. The type checker can guarantee preservation because the name of each element is exactly the tag type in the type of an element, `ELT t`.

4.4.1 Relating Elements to Children

The library models the relation between the name of an element and the name of a child element by the two-parameter type class `AddTo`.

```
class (TAG s, TAG t) => AddTo s t

add :: AddTo s t => ELT s -> ELT t -> ELT s
add (ELT e_) (ELT e'_) =
  ELT (add_ e_ e'_)
```

In `AddTo s t` the `s` is the name of the parent element and `t` is the name of the child element. The function `add` unwraps both elements, adds the "raw" child element `e'_` into the raw element `e_`, and wraps the result back into an element of type `ELT s`. The type class `AddTo` merely restricts the polymorphic type of `add`.

Each instance of `AddTo` specifies that a certain parent element accepts a certain child element. For example,

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

state 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` element 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.

4.4.2 Element Transformers

In Sec. 2.2.1, we have seen that direct programming with `make` and `add` is awkward and limiting. Hence, we introduced higher-order element constructor functions that yield element transformers. The implementation of these higher-order element constructors is straightforward. Here is the implementation for the `dl` element (all others are analogous).

```
dl :: AddTo s t => (ELT DL -> ELT t) -> ELT s -> ELT s
dl f elt = elt `add` f (make DL)
```

The first argument, `f`, of `dl` is a transformer for the newly created `<dl>` element. The second argument, `elt`, is the element, in which the transformed `<dl>` element will be inserted. The predicate `AddTo s t` originates from the use of the `add` function and indicates that the result, `t`, of transforming the new `<dl>` element is suitable for putting it into the enclosing `elt` of type `ELT s`.

The type of `dl` is a little bit more general than necessary because the transformer function may change the type of the newly generated element from `ELT DL` to `ELT t`. In most cases, the type parameter `t` will be equal to `DL`. Later, in Sec. 6.1, we exploit the extra generality.

4.5 Relating Elements with Attributes

Not every attribute makes sense for a particular element. Analogously to the class `AddTo` for elements, the type class `AddAttr` is used to restrict the polymorphic type of `add_attr`.

```
class (TAG t, ATTRIBUTE a) => AddAttr t a

add_attr :: AddAttr t a => ELT t -> ATTR a -> ELT t
add_attr (ELT e_) (ATTR att) =
  ELT (add_attr_ e_ att)
```

The function `add_attr` unwraps the element and the attribute instance, joins the new attribute using `add_attr_`, 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, according to Fig. 1, the `OL` element can take three attributes, `TYPE`, `COMPACT`, and `START`. Our library encodes this restriction with the following three instance declarations.

```
instance AddAttr OL TYPE
instance AddAttr OL COMPACT
instance AddAttr OL START
```

Finally, for a smooth integration with element processing, we provide the attribute functions in the form of element transformers. It is a straightforward combination of `add_attr` and `mkAttr`.

```
attr :: (AttrValue a v, AddAttr t a) => a -> v -> ELT t -> ELT t
attr a v into = add_attr into (mkAttr a v)
```

4.6 Character Data

Up to now, we assumed that all elements can be constructed from the element name using the `make` function. The only 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 elta = add elta (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` using an explicit type annotation.

4.7 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: `a # f = f a`.

The following definition introduces the type `DOCUMENT` as a tag type and defines its constructor function by overloading `make`.

```
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\""]
          [])
```

5 From HTML to XML

Since the DTD of HTML is fixed, we can perform the construction outlined in the previous section once and for all by hand. In practice, it is more convenient to automatize the construction. Hence, we have designed and implemented a translation that converts a DTD to Haskell code. The generated Haskell library provides all the datatype and instance declarations discussed in the previous section.

```
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⟦child-tag⟧      for each child-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⟧
```

Fig. 2. Translation from DTD to Haskell

Figure 2 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 element name, defining its interface function, and passing on to **CT**;
- **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
  - its 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 name, 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 names 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 that 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 (HTML 4.01, 1999) 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 Beyond Weak Validity

The typed encoding presented so far guarantees weak validity of the generated HTML/XML documents. While our experience shows that weak validity works well in practice, it is still interesting to see if Haskell's type classes can deliver stronger guarantees. This section shows that it is indeed possible.

6.1 Elementary Validity

Consider again the content description for the `dl` element:

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

This definition requires that

1. the children of `<dl>` are either `<dd>` or `<dt>` elements *and*
2. that at least one child is present.

However, the encoding introduced in Sec. 2–4 only enforced the first requirement, thus tacitly changing the content description to `(DT | DD)*`.

How can we instruct Haskell's type inference engine to enforce a content description more accurately? To see this, we first consider how a validating XML processor performs this task. Such a processor builds a finite automaton from each content description. Whenever it enters the list of children of a particular element, the processor retrieves the appropriate automaton and checks that the sequence of element names of the children is accepted by the automaton. The contents of the children are checked recursively.

While a typical processor performs this task dynamically at run-time, we intend to perform it statically at compile-time. To this end, we have to model the dynamic processing engine at compile-time.

6.1.1 Basic Approach

The natural place for maintaining additional compile-time information is in the phantom type for elements. The type of an element is now `ELT (s, qs)`, where `s` is a tag type and `qs` is a type that tracks the state of the automaton. This idea leads to the following typing for the `add` function:

```
add :: (AddTo s t, FinalState' t qt, NextState' s qs t qs')
    => ELT (s, qs) -> ELT (t, qt) -> ELT (s, qs')
```

- The predicate `AddTo s t` relates element names to the names of child elements, as before.
- The predicate `FinalState' t qt` expresses that the parameter `qt` of the child element must be a final state for the automaton of `t`. Otherwise, an incomplete element (for example, an element `<dl></dl>` without contents) can sneak into another element.
- The predicate `NextState' s qs t qs'` defines the state transition from state `qs` to state `qs'` on input `t` of the automaton that implements the content description for `s` elements. As usual, `t` stands for the name of the child element.

As an example, we consider the automaton for `<dl>`. It has two states, `State0` and `State1`, where `State0` is the initial state. Beyond the instance declarations for `AddTo`, the following declarations are required to implement the finite automaton.

```
data State0 = State0
data State1 = State1

instance FinalState' DL State1

instance NextState' DL State0 DD State1
instance NextState' DL State0 DT State1
instance NextState' DL State1 DD State1
instance NextState' DL State1 DT State1
```

The types `State0`, `State1`, and so on, are singleton types like the types for element names and attribute names before. They can be shared among all automata.

The instance for `FinalState'` indicates that a `<dl>` element can only be added to another element if its state is `State1`. In this particular case, it means that there must be at least one child.

The instances for `NextState'` implement the transition function of the minimal deterministic finite automaton for `(DD | DT)+`. The construction of the finite automaton from a regular expression is standard (Hopcroft & Ullman, 1979).

The creation of new elements must correctly initialize the state part of the `ELT` type. The automaton of each element must be set to its initial state:

```
make' :: TAG t => t -> ELT (t, State0)
```

6.1.2 Refined Approach

Unfortunately, the basic approach outlines in 6.1.1 defers type errors to a fairly late stage. Suppose that we have two elements `dd :: ELT (DD, State0)` and `dl :: ELT (DL, State0)`. Adding `dd` to `dl` gives rise to the typing

```
add dl dd ::
    (AddTo DL DD, FinalState' DD State0, NextState' DL State0 DD qs')
         => ELT (DL, qs')
```

Since `DD` is an admissible child for `DL`, there is an instance `AddTo DL DD`. Further, the automaton for `DD` has just one state `State0`, which is also the final state. Hence, all but the `NextState'` predicate can be reduced by the type checker.

```
add dl dd :: (NextState' DL State0 DD qs')
          => ELT (DL, qs')
```

To add the element `add dl dd` into another element, the type checker has to show that `FinalState' DL qs'` but all it has is `NextState' DL State0 DD qs'`, which cannot be reduced because `qs'` is not instantiated. At the toplevel, the type checker reports these remaining constraints as unresolved overloading.

Fortunately, it is possible to save the situation by providing additional information. The key to the solution is the fact that `NextState'` is not just a relation on types but also a function on types. Jones's extension of Haskell's type system by functional dependencies (Jones, 2000) enables us to express this property as follows:

```
class NextState' s qs t qs' | s qs t -> qs'
```

The *functional dependency* `| s qs t -> qs'` reads "where `s`, `qs`, and `t` determine `qs'` uniquely". The type checker takes advantage of this information during simplification of predicates. From the predicate `NextState' DL State0 DD qs'` it derives that `qs'` must be `State1` and determines the typing of the example expression as

```
add dl dd :: ELT (DL, State1)
```

With this typing, we can easily insert the element `add dl dd` into another element. Since the state is determined, the predicate `FinalState' DL State1` can always be reduced.

6.1.3 Implementation

In the actual implementation, which is also generated automatically by translation from the DTD, we perform a product construction. We merge the element name and the state information into one type, so that each element name gives rise to as many types as the finite automaton derived from its content description has states. This way, we can collapse all three type classes, `AddTo`, `NextState'`, and `FinalState'` into one class:

```
class NextState s t s' | s t -> s'
```

If we regard the "new" tag types as pairs `(s, qs)` of "old" tag types and states, then the connection is as follows: There is an instance `NextState (s, qs) (t, qt) (s, qs')` if and only if `NextState' s qs t qs'` and `AddTo s t` and `FinalState' t qt`.

For example, here are the instances for the `<dl>` element:

```
instance NextState DL   DD DL_1
instance NextState DL   DT DL_1
instance NextState DL_1 DD DL_1
instance NextState DL_1 DT DL_1
```

where `DL` stands for `(DL, State0)`, `DL_1` stands for `(DL, State1)`, `DD` for `(DD, State0)`, and `DT` for `(DT, State0)`.

Initially, we expected a huge number of states in the automata derived from the content descriptions. However, it turns out that the number of states is small. For the majority of element names, the number of states is one because their content description has the form `(`*elt1*`|...|`*eltn*`)*`. Even the automaton for a complicated element like `TABLE` has just seven different states.

6.2 Full Validity

From elementary validity, there is only a small step to full validity where each attribute occurs at most once and required attributes are guaranteed to be present. The first requirement is expressed by the regular language

$$L = \{w \in \Sigma^* \mid \text{each symbol of } a \in \Sigma \text{ occurs at most once in } w\}$$

where Σ is the set of attribute names. If attribute a is required, then we must consider the language $L \cap R_a$ where $R_a = \Sigma^* a \Sigma^*$. Clearly, R_a is regular and so is $L \cap R_a$.

Hence, the task is again to recognize a regular language so that the automaton approach demonstrated in the previous subsection is applicable, in principle. The phantom variable of `ELT` can again keep track of the state of the attribute automaton. As before, this constructs implicitly the product of the element automaton and the attribute automaton for each element name.

Unfortunately, this approach is not practical because the automata recognizing L 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. A deterministic automaton that checks this restriction has at least 2^{16} states.

6.2.1 Vector of States

However, an alternative approach is possible, which is inspired by work on record types (Rémy, 1992; Wand, 1989). The idea is to encode the state using one type

```
<!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"
  >
```

Fig. 3. Attributes of FORM (excerpt)

ATTRS with as many parameters as there are different attributes. Each of these type parameters determines the presence or absence of a particular attribute.[11] Hence, they range 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, defined in Fig. 3. HTML 4.01 defines 132 attributes in total, hence ATTRS has 132 parameters in reality.) The element type ELT receives an additional (phantom) type parameter. The make function creates an empty FORM element with all parameters of ATTRS set to ABSENT, meaning that no attribute is present, yet.

```
make' :: TAG t => t -> ELT (t, ATTRS ABSENT ABSENT ABSENT)
```

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 name of an attribute to be added, 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 name t.

Here are the class definitions and some illustrative 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
```

[11] In fact, this is yet another instance of the product construction: the type ATTRS models the combined state space of the two-state automata for the attributes.

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

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:alex")))
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html><body><form action="mailto:alex"></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:alex"
              ## attr ACTION "mailto:alex")))
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:alex"
              ## attr ACTION "mailto:alex")))
```

6.2.2 More Accurate Type Errors

It should be noted that these type errors are often deferred to a point where they are difficult to comprehend. For example, the predicate `AttrValid (ATTRS PRESENT ABSENT ABSENT) ACTION a` indicates an attempt to add the `ACTION` attribute to an element that already has an `ACTION` attribute. Fortunately, it is possible to force these errors to occur earlier by using functional dependencies, again.

While type classes are only good for encoding positive information, due to the open-world assumption underlying their design, functional dependencies can supply some negative information. In this case, the idea is to add another parameter to `AttrValid'` (and leaving `AttrFinal` as before):

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

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

The new parameter `r` is determined by `v` and `a`, as indicated by the functional dependency, and records the state of the attribute *before* adding the new attribute instance. The trick is now to change the type of the `add_attr` function.

```
add_attr :: (AddAttr t a, AttrValid' v a v' ABSENT) =>
            ELT (t, v) -> ATTR a -> ELT (t, v')
```

This type requires that, whenever we add an attribute named `a` to the element, then it must have been `ABSENT` before. Now, the type error occurs as soon as the types for `v` and `a` are known, that is, at the application of the `add_attr` function.

A similar improvement to error reporting is possible by changing the `AttrFinal` class:

```
class TAG t => AttrFinal t v | t -> v
```

This is again based on the observation that `AttrFinal` is really a function.

6.2.3 Conditional Content Revisited

The more precise typing needed for full validity has some unpleasant consequences. As an example, let us consider the topic of conditional content. Suppose we want to write a function that conditionally adds the `COMPACT` attribute to an ordered list:

```
maybeCompact flag =
  if flag then attr COMPACT COMPACT_compact
          else empty
```

Given the typings for full validity, we find that

```
attr COMPACT COMPACT_compact ::
         (AddAttr t COMPACT, AttrValid v COMPACT v') =>
         ELT (t, v) -> ELT (t, v')

empty :: ELT (t, v) -> ELT (t, v)
```

Hence the typing for `maybeCompact`:

```
maybeCompact :: (AddAttr t COMPACT, AttrValid v COMPACT v)
             => Bool -> (ELT (t, v) -> ELT (t, v))
```

Clearly, the predicate `AttrValid v COMPACT v` is not satisfiable because there is no instance of `AttrValid` where the state, `v`, before adding the attribute is identical to the state, `v`, after adding the attribute. The source of the problem is the typing of `empty`, which forces us to unify the two states. While it is possible to define `empty'` with type `ELT (t, v) -> ELT (t, v')`, it is not appropriate to do so because one conditional that uses `empty'` in both branches completely defeats the purpose of

```
<!ELEMENT BODY O O (%flow;)* +(INS|DEL) -- document body -->
<!ELEMENT HEAD O O (%head.content;) +(SCRIPT|STYLE|META|LINK|OBJECT)
     -- document head -->
<!ELEMENT A - - (%inline;)* -(A)       -- anchor -->
<!ELEMENT FORM - - (%flow;)* -(FORM)   -- interactive form -->
<!ELEMENT BUTTON - -
     (%flow;)* -(A|%formctrl;|FORM|ISINDEX|FIELDSET|IFRAME)
     -- push button -->
```

Fig. 4. Element declarations with inclusions and exceptions

having the state variable `v` at all. Our conclusion is that restrictions on attribute occurrences should better be checked dynamically using run-time tests.

We have presently chosen not to implement full validity in the library because of the above drawbacks. In addition, the last group of checks dealing with attributes gives rise to another 1595 instance declarations (without the last two tricks from Sec. 6.2.2), which bumps the size of the library's source code from 148k to 3.2M.

6.3 Exceptions and Inclusions

Exceptions and inclusions pose problems that are specific to HTML and other markup languages that are instances of SGML. Both concepts, exceptions and inclusions, have been removed from XML.

An *inclusion* in an element declaration of a DTD indicates that, within the declared element, certain elements are admissible regardless of the content description of their immediate parent element. Dually, there are *exceptions*, which abolish the use of some elements, regardless of the content description of their immediate parent element.

For example, consider the element declarations in Fig. 4 extracted from the HTML4.01 DTD.[12] The first two element declarations, for `<body>` and `<head>`, specify inclusions indicated by `+`: anywhere in the descendants of a `<body>` element (not just the children!), it is legal to use `<ins>` and `<del>` elements, regardless of the current content description. Likewise, anywhere deep in a `<head>` element, one of the `SCRIPT`, `STYLE`, `META`, `LINK`, or `OBJECT` elements may be used.

The remaining three element declarations contain exceptions indicated by `-`. The first 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.

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 for exceptions comes again from type systems for records, which express the absence of a particular field

[12] The entity references like `%flow;` and `%inline;` can be safely ignored in our discussion.

name (Rémy, 1992; Wand, 1989). 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 names. 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 elements 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 element names *below* the element, whereas the `above` parameter reflects the use of element names *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)
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 remaining elements, `FORM`, `BUTTON`, and `ISINDEX` are “inherited”. The instance declaration for `FORM` is similar.

The instance declaration for `BUTTON` says that there must not be an element with name `A`, `FORM`, `BUTTON`, or `ISINDEX` nested within a `BUTTON` element. The instance declaration for `ISINDEX` is similar.

For inclusions, the type of `add` is too restrictive. It is 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 (Bray *et al.*, 1998) also solves the problem because XML does not support exceptions, anymore. In fact, in XHTML (XHTML 1.0, 2000) 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 element names used by HTML 4.01), as demonstrated with the attributes.

7 Conclusion

We have designed a family of embedded domain specific languages for meta programming of web pages and web sites. Each of these languages is implemented as a combinator library in Haskell. Haskell's type classes and in particular multi-parameter classes with functional dependencies were instrumental in the construction. We have introduced element-transforming 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 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 for weak validity sufficient because it captures many common errors (using an element or attribute in the wrong place). Initial experiments with the more elaborate static scheme for elementary validity outlined in Section 6.1 yield quite natural and precise typings, too.

On the negative side, type errors are 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/XML. This is subject to further investigation.

Acknowledgements

Thanks to Simon Helsen, Matthias Neubauer, and Sebastian Schulz for comments on this work and for using early versions of the library. Thanks are also due to the reviewers. Their extensive comments helped to improve the presentation considerably.

References

Atkinson, David, Ball, Thomas, Benedikt, Michael, Bruns, Glenn, Cox, Kenneth, Mataga, Peter, & Rehor, Kenneth. (1997). Experience with a domain specific language for form-based services. *Conference on domain-specific languages*. Santa Barbara, CA: USENIX.

Brabrand, Claus, Møller, Anders, & Schwartzbach, Michael I. (2001). Static validation of dynamically generated HTML. *Pages 38–45 of:* Field, John, & Snelting, Gregor (eds), *Workshop on program analysis for software tools and engineering, PASTE'01*. SIGPLAN Notices. Snowbird, Utah, USA: ACM.

Bray, Tim, Paoli, Jean, & Sperberg-MacQueen, C. Michael. 1998 (feb). *Extensible markup language (XML) 1.0 (W3C Recommendation)*. `http://www.w3.org/TR/REC-xml`.

Brüggemann-Klein, Anne, & Wood, Derick. (1998). One-unambiguous regular languages. *Information and computation*, **140**(2), 229–253. and 142(2): 182–206.

Cabeza, Daniel, & Hermenegildo, Manuel. 1997 (Mar.). *WWW programming using computational logic systems (and the PiLLoW/CIAO library)*. `http://www.clip.dia.fi.upm.es/Software/pillow/pillow_www6/pillow_www6.h%tml`.

Cluet, Sophie, Delobel, Claude, Siméon, Jérôme, & Smaga, Katarzyna. (1998). Your mediators need data conversion! *Pages 177–188 of:* Haas, Laura, & Tiwary, Ashutosh (eds), *Proceedings of the 1998 ACM SIGMOD international conference on management of data: June 1–4, 1998, Seattle, Washington, USA*. SIGMOD Record (ACM Special Interest Group on Management of Data), vol. 27(2). New York, NY 10036, USA: ACM Press.

Fernandez, Mary, Siméon, Jérôme, & Wadler, Philip. 2001 (Jan.). A semi-monad for semi-structured data. *ICDT*.

Hanus, Michael. 2000 (July). Server side Web scripting in Curry. *Workshop on (constraint) logic programming and software engineering (LPSE2000)*.

Hanus, Michael. (2001). High-level server side Web scripting in Curry. *Practical aspects of declarative languages, proceedings of the third international workshop, PADL'01*. Lecture Notes in Computer Science. Las Vegas, NV, USA: Springer-Verlag.

Haskell98. 1998 (Dec.). *Haskell 98, a non-strict, purely functional language*. `http://www.haskell.org/definition`.

Hopcroft, John E., & Ullman, Jeffrey D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley.

Hosoya, Haruo, & Pierce, Benjamin C. (2000). XDuce: A typed XML processing language. *Pages 226–244 of:* Suciu, Dan, & Vossen, Gottfried (eds), *The world wide web and databases. third international workshop WebDB2000*. Lecture Notes in Computer Science, no. 1997. Dallas, TX, USA: Springer-Verlag.

Hosoya, Haruo, & Pierce, Benjamin C. (2001). Regular expression pattern matching for XML. *In:* (Nielson, 2001).

Hosoya, Haruo, Vouillon, Jérôme, & Pierce, Benjamin C. (2000). Regular expression types for XML. *Pages 11–22 of:* Wadler, Philip (ed), *Proc. international conference on functional programming 2000*. Montreal, Canada: ACM Press, New York.

HTML 4.01. 1999 (Dec.). *HTML 4.01 specification*. `http://www.w3.org/TR/html4/`.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(May), 67–111.

Jones, Mark P. (2000). Type classes with functional dependencies. *Pages 230–244 of:* Smolka, Gert (ed), *Proc. 9th european symposium on programming*. Lecture Notes in Computer Science, no. 1782. Berlin, Germany: Springer-Verlag.

Loke, Seng Wai, & Davison, Andrew. 1996 (Mar.). Logic programming with the World-Wide Web. *Pages 235–245 of: Proceedings of the 7th ACM conference on hypertext (hypertext '96)*.

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

Nielson, Hanne Riis (ed). (2001). *Proc. 28th annual ACM symposium on principles of programming languages*. London, England: ACM Press.

Peyton Jones, Simon, Jones, Mark, & Meijer, Erik. 1997 (June). Type classes: An exploration of the design space. Launchbury, John (ed), *Proc. of the Haskell workshop*. Yale University Research Report YALEU/DCS/RR-1075.

Rémy, Didier. (1992). Typing record concatenation for free. *Pages 166–176 of: Proc. 19th annual ACM symposium on principles of programming languages*. Albuquerque, New Mexico: ACM Press.

Sandholm, Anders, & Schwartzbach, Michael I. (2000). A type system for dynamic web documents. *Pages 290–301 of:* Reps, Tom (ed), *Proc. 27th annual ACM symposium on principles of programming languages*. Boston, MA, USA: ACM Press.

Shields, Mark, & Meijer, Erik. (2001). Type-indexed rows. *In:* (Nielson, 2001).

Thiemann, Peter. 2000 (Jan.). Modeling HTML in Haskell. *Pages 263–277 of: Practical aspects of declarative languages, proceedings of the second international workshop, PADL'00*. Lecture Notes in Computer Science, no. 1753.

Wallace, Malcolm, & Runciman, Colin. (1999). Haskell and XML: Generic combinators or type-based translation? *Pages 148–259 of:* Lee, Peter (ed), *Proc. international conference on functional programming 1999*. Paris, France: ACM Press, New York.

Wand, Mitchell. (1989). Type inference for record concatenation and multiple inheritance. *Pages 92–97 of: Proc. of the 4th annual symposium on logic in computer science*. Pacific Grove, CA: IEEE Computer Society Press. To appear in *Information and Computation*.

WASH. 2001 (Mar.). *Web authoring system in Haskell (WASH)*. `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH`.

World-Wide Web Consortium. 2000a (Apr.). *XML schema part 1: Structures, working draft*. `http://www.w3.org/TR/xmlschema-1`.

World-Wide Web Consortium. 2000b (Apr.). *XML schema part 2: Datatypes, working draft*. `http://www.w3.org/TR/xmlschema-2`.

XHTML 1.0. 2000 (Jan.). *XHTML 1.0: The extensible hypertext markup language*. `http://www.w3.org/TR/xhtml1`.

XML1.0. 2000 (Oct.). *Extensible markup language (XML) 1.0 (second edition)*. `http://www.w3.org/TR/2000/REC-xml-20001006`.