Typing XHTML Web Applications in ML

Martin Elsman
mael@itu.dk

Ken Friis Larsen
ken@friislarsen.net

IT University of Copenhagen.
Glentevej 67, DK-2400 Copenhagen NV, Denmark

Abstract. In this paper, we present a type system for typing Web applications in SMLserver, an efficient multi-threaded Web server platform for Standard ML scriptlets. The type system guarantees that only conforming XHTML documents are sent to clients and that forms are used consistently and in a type-safe way. The type system is encoded in the type system of Standard ML using so-called phantom types.

1 Introduction

Traditionally, frameworks for developing Web applications give little guarantees about the conformity of generated HTML or XHTML documents, and, most often, no static mechanism guarantees that the particular use of form data is consistent with the construction of a corresponding form.

We present a static type system for SMLserver [5] that guarantees generated XHTML documents to conform to the XHTML 1.0 specification [21]. The conformity requirements are enforced by requiring the Web programmer to construct XHTML documents using a combinator library for which the different requirements are encoded in the types of element combinators using phantom types [1, 7, 8, 13, 18–20]. The type system also guarantees that forms in generated documents are consistent with actual form data submitted by clients.

A scriptlet in SMLserver is represented as a functor with the argument representing form data to be received from a client and the body representing program code to be executed when the scriptlet is requested by a client. In this sense, scriptlet functors are instantiated dynamically by SMLserver upon client requests, which may result in new documents being sent to clients. Because it is not possible to encode the recursive nature of scriptlets directly using Standard ML modules, an *abstract scriptlet interface*, containing typing information about accessible scriptlets and their form arguments, is generated prior to compilation, based on preprocessing of scriptlet functor arguments. The generated abstract scriptlet interface takes the form of an abstract Standard ML structure, which can be referred to by scriptlets and library code to construct XHTML forms and hyper-link anchors in a type safe way.

The type safe encoding is complete in the sense that it does not restrict what conforming XHTML documents it is possible to write. There are two exceptions to this completeness guarantee. First, a form must relate to its target scriptlet

in the sense that form data submitted by a form is consistent with the scriptlet's expectations of form data. Second, form variable arguments appearing in hyperlink anchors to scriptlets must be consistent with the scriptlet's expectations of form variable arguments.

1.1 Contributions

The paper contains two main contributions. First, in Sect. 2 and 3, we present a novel approach to enforce conformity of generated XHTML 1.0 documents [21], based on a typed combinator library that makes use of phantom types. Although others have suggested type systems for guaranteeing conformity of generated XHTML documents [17–20], no other approaches can be used for embedding XHTML documents in ML-like languages without support for type classes [12]. To the best of our knowledge, our encoding of linearity constraints using phantom types is novel.

Second, in Sect. 4, we present a type based technique for enforcing consistency between a scriptlet's use of form variables and the construction of forms that target that scriptlet. For this technique, we introduce the concept of *type lists* (i.e., lists at the type level), also encoded using phantom types. Moreover, we contribute with a type-indexed function [9, 14, 22] for swapping arbitrary elements in type lists.

The contributions are formally justified and the techniques are implemented in SMLserver and have been used for building various Web applications, including a form extensive quiz for employees at the IT University of Copenhagen. It is our experience that the approach scales well to large Web applications and that the type system catches many critical programming mistakes early in the application development. It is also our experience that type errors caused by erroneous use of XHTML combinators are understandable and pinpoint problems directly.

A formalization of scriptlets, in the style of [10], that captures the basic Web application model used by SMLserver is given in a companion technical report [6]. Related work is described in Sect. 5. Finally, in Sect. 6, we conclude.

2 Conformity of XHTML Documents

In essence, for a document to *conform* to the XHTML 1.0 specification [21], the document must be well-formed, valid according to a particular DTD, and obey certain element prohibitions.

Well-formedness: For a document to be well-formed, all start-tags must have a corresponding closing-tag, all elements must be properly nested, and no attribute name may appear more than once in the same start-tag.

Validity: A valid XHTML document must be derivable from the grammar described by the XHTML DTD.

Element prohibitions: The XHTML 1.0 specification describes a set of prohibitions that are not specified by the XHTML DTD [21, App. B], but which must be satisfied for an XHTML document to be conforming. For example, an element prohibition specifies that, to all depth of nesting, an anchor element `<a ...> ... </a>` must not be contained in other anchor elements.

2.1 Motivating Example

The program code in Fig. 1 shows an abbreviated interface to a typical library of combinators for generating XHTML (the signature `MICRO_XHTML`). Fig. 1 also shows some sample utility code for generating an XHTML table from a list of `string` lists (the function `toTable`).

```
signature MICRO_XHTML = sig
 type elt
 val $ : string -> elt
 val & : elt * elt -> elt
 val td    : elt -> elt
 val tr    : elt -> elt
 val table : elt -> elt
end
structure MicroXHtml = ...
```

```
open MicroXHtml
fun concat es =
    foldr & ($"") es
fun toCols xs =
    concat (map (td o $) xs)
fun toRows xs =
    concat (map (tr o toCols) xs)
fun toTable strings =
    table(toRows strings)
```

Fig. 1. An unsafe XHTML combinator library and application code.

Although the `MicroXHtml` library ensures that well-formed XHTML code is generated [18] (ignoring the linearity condition for attributes), the library does not ensure that the generated XHTML code is valid. In particular, if the empty list is given as argument to the function `toTable`, a `table` element is generated containing no `tr` elements, which is not valid XHTML code according to the XHTML DTD. Similarly, if one of the lists in the argument to the function `toRows` is the empty list, a `tr` element is constructed containing no `td` elements.

An alternative interface could allow the programmer to construct XHTML documents directly through a set of datatype constructors. It turns out, however, that, due to XHTML 1.0 element prohibitions and the linearity well-formedness condition on attributes, such an approach cannot be made type safe in ML. Moreover, the approach would burden the programmer with the need for an excessive amount of tagging and datatype coercions.

2.2 Mini XHTML

To demonstrate the phantom type approach for a small subset of XHTML 1.0, consider the language Mini XHTML defined by the following DTD:

```
<!ENTITY  %block  "p|table|pre">        <!ELEMENT p     (%inline)*>
<!ENTITY  %inline "%inpre|big">         <!ELEMENT em    (%inline)*>
<!ENTITY  %flow   "%block|%inline">     <!ELEMENT big   (%inline)*>
<!ENTITY  %inpre  "#PCDATA|em">         <!ELEMENT pre   (%inpre)*>
<!ENTITY  %td     "td">                 <!ELEMENT td    (%flow)*>
<!ENTITY  %tr     "tr">                 <!ELEMENT tr    (%td)+>
                                        <!ELEMENT table (%tr)+>
```

The DTD defines a context free grammar for Mini XHTML, which captures an essential subset of XHTML, namely the distinction between `inline`, `block`, `flow`, `td`, and `tr` entities, the notion of sequencing, and a weakened form of element prohibitions easily expressible in a DTD. We postpone the discussion of attributes to Sect. 3.

For constructing documents, we use the following grammar, where t ranges over a finite set of *tags* and c ranges over finite sequences of character data:

$$d ::= c \mid t(d) \mid d_1\ d_2 \mid \varepsilon$$

The construct $d_1\ d_2$ denotes a sequence of documents d_1 and d_2 and ε denotes the empty document.

To formally define whether a document d is valid according to the Mini XHTML DTD, we introduce the relation $\models d : \kappa$, where κ ranges over entity names (i.e., `inline`, `inpre`, `block`, `flow`, `tr`, and `td`) defined in the DTD. The relation $\models d : \kappa$ expresses that d is a valid document of entity κ. The relation is defined inductively by a straightforward translation of the DTD into inference rules, which allow inference of sentences of the form $\models d : \kappa$.

Valid documents $\boxed{\models d : \kappa}$

$$\frac{\models d : \texttt{inpre}}{\models d : \texttt{inline}} \qquad \frac{\models d : \texttt{inline}}{\models \texttt{em}(d) : \texttt{inpre}} \qquad \frac{\models d : \texttt{inline}}{\models \texttt{p}(d) : \texttt{block}} \qquad \frac{\models d : \texttt{inline}}{\models \texttt{big}(d) : \texttt{inline}}$$

$$\frac{\models d : \texttt{inpre}}{\models \texttt{pre}(d) : \texttt{block}} \qquad \frac{\models d : \texttt{flow}}{\models \texttt{td}(d) : \texttt{td}} \qquad \frac{\models d : \texttt{td}}{\models \texttt{tr}(d) : \texttt{tr}} \qquad \frac{\models d : \texttt{tr}}{\models \texttt{table}(d) : \texttt{block}}$$

$$\frac{}{\models c : \texttt{inpre}} \qquad \frac{\models d : \texttt{inline}}{\models d : \texttt{flow}} \qquad \frac{\models d : \texttt{block}}{\models d : \texttt{flow}}$$

$$\frac{\models d_1 : \kappa \quad \models d_2 : \kappa}{\models d_1\ d_2 : \kappa} \qquad \frac{\kappa \in \{\texttt{block}, \texttt{inline}, \texttt{inpre}, \texttt{flow}\}}{\models \varepsilon : \kappa}$$

A signature for a combinator library for Mini XHTML is given in Fig. 2 together with a concrete implementation. The signature specifies a type constructor `('ent,'pre)elt` for element sequences, which takes two *phantom type* parameters. The first phantom type parameter is used for specifying the entity

```
signature MINI_XHTML = sig
 type ('blk,'inl)flw and tr and td
 type blk and inl and NOT and inpre
 type ('ent,'pre)elt and preclosed
 val $ : string->(('b,inl)flw,'p)elt
 val p : ((NOT,inl)flw,'p)elt
      -> ((blk,'i)flw,'p)elt
 val em : ((NOT,inl)flw,'p)elt
      -> (('b,inl)flw,'p)elt
 val pre : ((NOT,inl)flw,inpre)elt
      -> ((blk,'i)flw,'p)elt
 val big : ((NOT,inl)flw,'p)elt
      -> (('b,inl)flw,preclosed)elt
 val table : (tr,'p)elt
      -> ((blk,'i)flw,'p)elt
 val tr : (td,'p)elt -> (tr,'p)elt
 val td : ((blk,inl)flw,'p)elt
      -> (td,'p)elt
 val & : ('e,'p)elt * ('e,'p)elt
      -> ('e,'p)elt
 val emp : unit -> (('b,'i)flw,'p)elt
end
```

```
structure MiniXHtml :> MINI_XHTML =
struct infix &
 datatype e = Elt of string*e
   | Emp | Seq of e*e | S of string
 type ('ent,'pre)elt = e
 type blk = unit and inl = unit
  and NOT = unit
 type ('b,'i)flw = unit
  and tr = unit and td = unit
 type inpre = unit
 type preclosed = unit
 fun $ s = S s
 fun em e = Elt("em",e)
 fun p e = Elt("p",e)
 fun big e = Elt("big",e)
 fun pre e = Elt("pre",e)
 fun td e = Elt("td",e)
 fun tr e = Elt("tr",e)
 fun table e = Elt("table",e)
 fun emp() = Emp
 fun e & e' = Seq(e,e')
end
```

Fig. 2. Mini XHTML combinator library.

of the element in terms of *entity types*, which are types formed with the `NOT`, `blk`, `inl`, `flw`, `tr`, and `td` type constructors (all implemented as type unit). For instance, the `p` combinator requires its argument to be an `inline` entity, expressed with the entity type `(NOT,inl)flw`, which classifies sequences of `flow` entities that do not contain `block` entities. The result type of the `p` combinator expresses that the result is a `block` entity, which may be regarded either as a pure `block` entity of type `(blk,NOT)flw` or as a `flow` entity with type `(blk,inl)flw`.

Using the infix sequence combinator `&`, it is impossible to combine a `block` entity with an `inline` entity and use the result as an argument to the `p` combinator, for example. The result of combining `block` entities and `inline` entities can be used only in contexts requiring a `flow` entity (e.g., as argument to the `td` combinator).

The `'pre` type parameter of the `elt` type constructor is used for implementing the element prohibition of XHTML 1.0 that, to all depth of nesting, prohibits `big` elements from appearing inside `pre` elements. This element prohibition implies the satisfaction of the weaker DTD requirement that prohibits a `big` element to appear immediately within a `pre` element.

Specialized elaboration rules for constructing documents in Standard ML with the combinators presented in Fig. 2 follow. The rules allow inference of sentences of the form $\vdash e : (\tau_{\mathrm{e}}, \tau_{\mathrm{p}})\texttt{elt}$, where e ranges over expressions, τ_{e} over entity types, and τ_{p} over nullary type constructors `inpre` and `preclosed`. We

shall also use τ_b to range over the type constructors `blk` and `NOT`, and τ_i to range over the type constructors `inl` and `NOT`.

Expressions $\boxed{\vdash e : (\tau_\mathrm{e}, \tau_\mathrm{p})\texttt{elt}}$

$$\frac{}{\vdash \texttt{\$}\ c : ((\tau_\mathrm{b}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}} \qquad \frac{\vdash e : ((\texttt{NOT}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}{\vdash \texttt{p}\ e : ((\texttt{blk}, \tau_\mathrm{i})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}$$

$$\frac{\vdash e : ((\texttt{NOT}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}{\vdash \texttt{em}\ e : ((\tau_\mathrm{b}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}} \qquad \frac{\vdash e : ((\texttt{NOT}, \texttt{inl})\texttt{flw}, \texttt{inpre})\texttt{elt}}{\vdash \texttt{pre}\ e : ((\texttt{blk}, \tau_\mathrm{i})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}$$

$$\frac{\tau_\mathrm{p} = \texttt{preclosed} \quad \vdash e : ((\texttt{NOT}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}{\vdash \texttt{big}\ e : ((\tau_\mathrm{b}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}} \qquad \frac{\vdash e_1 : (\tau_\mathrm{e}, \tau_\mathrm{p})\texttt{elt} \quad \vdash e_2 : (\tau_\mathrm{e}, \tau_\mathrm{p})\texttt{elt}}{\vdash e_1\ \texttt{\&}\ e_2 : (\tau_\mathrm{e}, \tau_\mathrm{p})\texttt{elt}}$$

$$\frac{\vdash e : (\texttt{tr}, \tau_\mathrm{p})\texttt{elt}}{\vdash \texttt{table}\ e : ((\texttt{blk}, \tau_\mathrm{i})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}} \qquad \frac{\vdash e : (\texttt{td}, \tau_\mathrm{p})\texttt{elt}}{\vdash \texttt{tr}\ e : (\texttt{tr}, \tau_\mathrm{p})\texttt{elt}}$$

$$\frac{\vdash e : ((\texttt{blk}, \texttt{inl})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}{\vdash \texttt{td}\ e : (\texttt{td}, \tau_\mathrm{p})\texttt{elt}} \qquad \frac{}{\vdash \texttt{emp()} : ((\tau_\mathrm{b}, \tau_\mathrm{i})\texttt{flw}, \tau_\mathrm{p})\texttt{elt}}$$

The implementation of the `MINI_XHTML` signature is defined in terms of documents by the function *doc*:

$$\begin{aligned} doc(\texttt{\$}\ c) &= c & doc(\texttt{table}\ e) &= \texttt{table}(doc(e)) \\ doc(\texttt{p}\ e) &= \texttt{p}(doc(e)) & doc(\texttt{tr}\ e) &= \texttt{tr}(doc(e)) \\ doc(\texttt{em}\ e) &= \texttt{em}(doc(e)) & doc(\texttt{td}\ e) &= \texttt{td}(doc(e)) \\ doc(\texttt{pre}\ e) &= \texttt{pre}(doc(e)) & doc(e_1\ \texttt{\&}\ e_2) &= doc(e_1)\ doc(e_2) \\ doc(\texttt{big}\ e) &= \texttt{big}(doc(e)) & doc(\texttt{emp()}) &= \varepsilon \end{aligned}$$

Before we state a soundness property for the combinator library, we define a binary relation $\tau \sim \kappa$, relating element types τ and DTD entities κ. As before, τ_p ranges over the type constructors $\{\texttt{inpre}, \texttt{preclosed}\}$.

$$\begin{aligned} \texttt{((blk,NOT)flw,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{block} & \texttt{((NOT,NOT)flw,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{inpre} \\ \texttt{((NOT,NOT)flw,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{inline} & \texttt{((blk,inl)flw,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{flow} \\ \texttt{((NOT,NOT)flw,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{block} & \texttt{(td,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{td} \\ \texttt{((NOT,inl)flw,inpre)elt} &\sim \texttt{inpre} & \texttt{(tr,}\tau_\mathrm{p}\texttt{)elt} &\sim \texttt{tr} \\ \multicolumn{4}{c}{\texttt{((NOT,inl)flw,preclosed)elt} \sim \texttt{inline}} \end{aligned}$$

The soundness lemma states that well-typed expressions are valid according to the Mini XHTML DTD. The lemma is easily demonstrated by structural induction on the derivation $\vdash e : \tau$.

Lemma 1 (Soundness). *If* $\vdash e : \tau$ *and* $\tau \sim \kappa$ *then* $\models doc(e) : \kappa$.

The soundness lemma is supported by the property that if $\vdash e : \tau$ then there exists an entity κ such that $\tau \sim \kappa$.

The library is not complete. It turns out that because of the element prohibitions encoded in the combinator library and because element prohibitions are not enforced by the DTD, there are documents that are valid according to the DTD, but cannot be constructed using the combinator library. It is possible, to weaken the types for the combinators so that the element prohibitions are enforced only to the extend that the prohibitions are encoded in the DTD.

The orthogonal five element prohibitions of XHTML 1.0 [21, Appendix B] can be composed using separate type parameters.

2.3 Motivating Example Continued

If the utility code from Fig. 1 (the function `toTable` and friends) is used with the `MiniXHtml` library from Fig. 2, two type errors occur. The type errors are caused by the use of the `concat` function for composing lists of `td` and `tr` elements. The problem with the `concat` function from Fig. 1 is that it may return the empty element, which cannot be used for the `tr` and `table` elements.

To resolve the type error (and avoid that invalid XHTML is sent to a browser), the `concat` function can be replaced with the following function `concat1`, which fails in case its argument is the empty list:

```
fun concat1 []      = raise List.Empty
  | concat1 [x]     = x
  | concat1 (x::xs) = x & concat1 xs
```

The remainder of the utility code in Fig. 1 can be left unchanged (except that all calls to `concat` must be replaced with calls to `concat1`).

3 Linearity of Attributes

An *attribute* is a pair of an attribute name and an attribute value. In general, we refer to an attribute by referring to its name. Each kind of element in an XHTML document supports a set of attributes, specified by the XHTML DTD. All elements do not support the same set of attributes, although some attributes are supported by more than one element. For instance, all elements support the `id` attribute, but only some elements (e.g., the `img` and `table` elements) support the `width` attribute. In this section we show how the linearity well-formedness constraint on attribute lists can be enforced statically using phantom types.

3.1 Attributes in Mini XHTML

The signature `MINI_XHTML_ATTR` in Fig. 3 specifies operations for constructing linear lists of attributes, that is, lists of attributes for which an attribute with

```
signature MINI_XHTML_ATTR = sig
  type ('a0,'a,'b0,'b,'c0,'c) attr
  type na and align and width and height
  val left  : align
  val right : align
  val align : align -> (na,align,'b,'b,'c,'c) attr
  val width  : int -> ('a,'a,na,width,'c,'c) attr
  val height : int -> ('a,'a,'b,'b,na,height) attr
  val % : ('a0,'a,'b0,'b,'c0,'c)attr * ('a,'a1,'b,'b1,'c,'c1)attr
       -> ('a0,'a1,'b0,'b1,'c0,'c1)attr
end
```

Fig. 3. Mini XHTML attribute library.

a given name appears at most once in a list. For simplicity, the Mini XHTML attribute interface provides support for only three different attribute names (i.e., `align`, `width`, and `height`). Singleton attribute lists are constructed using the functions `align`, `width`, and `height`. Moreover, the function `%` is used for appending two attribute lists. The interface specifies a nullary type constructor `na` (read: no attribute), which is used to denote the absence of an attribute. The type constructor `attr` is parameterized over six type variables, which are used to track linearity information for the three possible attribute names. Two type variables are used for each possible attribute name. The first type variable represents "incoming" linearity information for the attribute list, whereas the second type variable represents "outgoing" linearity information. The type of `%` connects outgoing linearity information of its left argument with incoming linearity information of its right argument. The result type provides incoming and outgoing linearity information for the attribute list resulting from appending the two argument attribute lists. In this respect, for each attribute name, the two corresponding type variables in the attribute type for an attribute list expression represent the decrease in linearity imposed by the attribute list.

As an example, consider the expression

```
width 50 % height 100 % width 100
```

This expression does not type because the `width` combinator requires the incoming linearity to be `na`, which for the second use of the combinator contradicts the outgoing linearity information from the first use of the `width` combinator. Notice also that the type of a well-typed attribute list expression is independent of the order attributes appear in the expression.

Specialized elaboration rules for constructing attribute lists in Standard ML with the combinators presented in Fig. 3 are given below. The rules allow inference of sentences of the form $\vdash e : (\tau_\mathrm{a}, \tau_\mathrm{a}, \tau_\mathrm{b}, \tau_\mathrm{b}, \tau_\mathrm{c}, \tau_\mathrm{c})\texttt{attr}$, where e ranges over Standard ML expressions, and where τ_n, $n \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$ ranges over the types `na`, `align`, `width`, and `height`.

Attribute Typing Rules $\boxed{\vdash e : \tau}$

$$\frac{\tau = (\texttt{na}, \texttt{align}, \tau_\text{b}, \tau_\text{b}, \tau_\text{c}, \tau_\text{c})\texttt{attr}}{\vdash \texttt{align left} : \tau} \qquad \frac{\tau = (\texttt{na}, \texttt{align}, \tau_\text{b}, \tau_\text{b}, \tau_\text{c}, \tau_\text{c})\texttt{attr}}{\vdash \texttt{align right} : \tau}$$

$$\frac{\tau = (\tau_\text{a}, \tau_\text{a}, \texttt{na}, \texttt{width}, \tau_\text{c}, \tau_\text{c})\texttt{attr}}{\vdash \texttt{width}\ n : \tau} \qquad \frac{\tau = (\tau_\text{a}, \tau_\text{a}, \tau_\text{b}, \tau_\text{b}, \texttt{na}, \texttt{height})\texttt{attr}}{\vdash \texttt{height}\ n : \tau}$$

$$\frac{\vdash e_1 : (\tau_\text{a}^0, \tau_\text{a}, \tau_\text{b}^0, \tau_\text{b}, \tau_\text{c}^0, \tau_\text{c})\texttt{attr} \qquad \vdash e_2 : (\tau_\text{a}, \tau_\text{a}^1, \tau_\text{b}, \tau_\text{b}^1, \tau_\text{c}, \tau_\text{c}^1)\texttt{attr}}{\vdash e_1\ \texttt{\%}\ e_2 : (\tau_\text{a}^0, \tau_\text{a}^1, \tau_\text{b}^0, \tau_\text{b}^1, \tau_\text{c}^0, \tau_\text{c}^1)\texttt{attr}}$$

To state a soundness lemma for the attribute typing rules, we first define a partial binary function $\div$ according to the following equations:

$$\begin{aligned} \texttt{align} \div \texttt{na} &= 1 & \qquad \texttt{height} \div \texttt{na} &= 1 \\ \texttt{width} \div \texttt{na} &= 1 & \qquad \tau \div \tau &= 0 \end{aligned}$$

The following lemma expresses that there is a correlation between the number of attributes with a particular name in an attribute list expression and the type of the expression; a proof appears in the companion technical report [6].

Lemma 2 (Attribute linearity). *If* $\vdash e : (\tau_\text{a}^0, \tau_\text{a}^1, \tau_\text{b}^0, \tau_\text{b}^1, \tau_\text{c}^0, \tau_\text{c}^1)\texttt{attr}$ *then (1) the number of* `align` *attributes in* e *is* $\tau_\text{a}^1 \div \tau_\text{a}^0$*, (2) the number of* `width` *attributes in* e *is* $\tau_\text{b}^1 \div \tau_\text{b}^0$*, and (3) the number of* `height` *attributes in* e *is* $\tau_\text{c}^1 \div \tau_\text{c}^0$*.*

It turns out that no element supports more than a dozen attributes. As a consequence, to decrease the number of type variable parameters for the `attr` type constructor, we refine the strategy such that each attribute makes use of a triple of type variable parameters for each attribute, where the first type variable parameter denotes the particular attribute that the triple is used for and the two other type variable parameters are used to encode the linearity information as before. Given a DTD, it is possible to construct an *attribute interference graph*, which can be colored using a simple graph coloring algorithm and used to construct an attribute interface with the desired properties; see the companion technical report for details [6]. We have used this approach to generate an attribute combinator library for a large part of the XHTML 1.0-Strict DTD. As a result, 18 different attributes are supported using 21 type variable parameters in the `attr` type constructor.

3.2 Adding Attributes to Elements

To add attributes to elements in a type safe way, for each element name, we introduce a new attribute-accepting combinator, which takes as its first argument an attribute list. The attribute argument-type of the combinator specifies which attributes are supported by the element.

4 Form Consistency

Form consistency guarantees that form data submitted by clients, either as form variable arguments in a GET request or as posted data in a POST request, is consistent with the scriptlet's expectations of form data.

The programmer writes a Web application as a set of ordinary Standard ML library modules and a set of scriptlets. An example scriptlet looks as follows:

```
functor bmi (F : sig val h : int Form.var
                     val w : int Form.var
                 end) : SCRIPTLET =
  struct
    infix &
    val h = Form.getOrFail Page.page "Height" F.h
    val w = Form.getOrFail Page.page "Weight" F.w
    val bmi = Int.div(w * 10000, h * h)
    val txt = if bmi > 25 then "too high!"
              else if bmi < 20 then "too low!"
              else "normal"
    val response =
      Page.page "Body Mass Index"
      (p ($ ("Your BMI is " ^ txt)))
  end
```

The signature `SCRIPTLET` specifies a value `response` with type `Http.response`, which represents server responses.

SMLserver cannot guarantee that a user indeed enters an integer. In the `bmi` example, both the `h` and `w` form arguments are specified with the type `int Form.var`, which suggests that the user is supposed to provide integers for these form arguments. Using the function `Form.getOrFail`, the `bmi` scriptlet converts the form arguments into integers and responds with an error page in case one of the form arguments is not present in the form or is not an integer. The structure `Form` provided by SMLserver contains a library of combinators for checking form variables of different types.

If both form arguments `h` and `w` are integers, the `bmi` scriptlet computes the body mass index and constructs a message depending on the index. Finally, an XHTML page containing the message is bound to the variable `response`, using the user provided function `Page.page`, which takes a string title and a `block` element (a page body) as arguments and constructs a conforming XHTML document.

4.1 Static Tracking of Form Variables

The following simplified `SIMPLE_XHTML` signature specifies operations that propagate information about form input elements in a type safe way:

```
signature SIMPLE_XHTML = sig
  type ('a,'b)elt and nil and ('n,'t)name
  val inputtext : ('n,'t)name -> ('n->'a,'a)elt
  val inputsubmit : string -> ('n->'a,'a)elt
  val $ : string -> ('a,'a)elt
  val & : ('a,'b)elt * ('b,'c)elt -> ('a,'c)elt
end
```

As before, the type `('a,'b)elt` denotes an XHTML element, but the type variables `'a` and `'b` are here used to propagate information about form variable names at the type level, where form variable names are represented as abstract nullary type constructors. For readability, we reuse the function type constructor `->` as a list constructor for variable names at the type level. For representing the empty list of names, the nullary type constructor `nil` is used. We use the term *type list* to refer to lists at the type level constructed with `->` and `nil`.

Consider the value `inputtext` with type `('n,'t)name -> ('n->'a,'a)elt`. In this type, the type variable `'n` represents a form variable name and `'t` represents the ML type (e.g., `int`) of that variable. In the resulting element type, the name `'n` is added to the list `'a` of form variables used later in the form.

Whereas `inputtext` provides one way of constructing a leaf node in an `elt` tree, the operator `$` provides a way of embedding string data within XHTML documents. The type for `$` suggests that elements constructed with this operator do not contribute with new form variable names. The binary operator `&` constructs a new element on the basis of two child elements. The type of `&` defines the contributions of form variable names used in the constructed element as the contributions of form variable names in the two child elements.

To continue the Body Mass Index example, consider the scriptlet functor `bmiform`, which creates a form to be filled out by a user:

```
functor bmiform () : SCRIPTLET =
  struct open Scriptlets infix &
    val response = Page.page "Body Mass Index Form" (bmi.form
       (p(  $"Enter your height (in cm)" & inputtext bmi.h & br()
          & $"Enter your weight (in kg)" & inputtext bmi.w
          & inputsubmit "Compute Index")))
  end
```

The `bmiform` scriptlet references the generated abstract scriptlet interface to construct a `form` element containing input elements for the height and weight of the user. The use of the functions `inputtext` and `inputsubmit` construct `input` elements of type `text` and `submit`, respectively.

SMLserver also provides a series of type safe combinators for constructing radio buttons, check boxes, selection boxes, and input controls of type `hidden`; see the companion technical report for details [6].

4.2 Abstract Scriptlet Interfaces

In the case for the `bmiform` and `bmi` scriptlets, the generated abstract scriptlet interface `Scriptlets` includes the following structure specifications:[1]

```
structure bmiform : sig
  val form : (nil,nil)elt -> (nil,nil)elt
  val link : ('x,'y)elt -> ('x,'y)elt
end
structure bmi : sig
  type h and w
  val h : (h,int) XHtml.name
  val w : (w,int) XHtml.name
  val form : (h->w->nil,nil)elt -> (nil,nil)elt
  val link : {h:int, w:int} -> ('x,'y)elt -> ('x,'y)elt
end
```

The abstract scriptlet interface `bmi` specifies a function `link` for constructing an XHTML hyper-link anchor to the `bmi` scriptlet. The function takes as argument a record with integer components for the form variables `h` and `w`. Because the `bmiform` scriptlet takes no form arguments (i.e., the functor argument is empty), creating a link to this scriptlet using the function `bmiform.link` takes no explicit form arguments.

The abstract scriptlet interface `bmi` specifies two abstract types `h` and `w`, which represent the form variables `h` and `w`, respectively. The variables `h` and `w` specified by the `bmi` abstract scriptlet interface are used as witnesses for the respective form variables when forms are constructed using the function `XHtml.inputtext` or other functions for constructing form input elements. The Standard ML type associated with the form variables `h` and `w`, here `int`, is embedded in the type for the two form variable names. This type embedding makes it possible to pass hidden form variable to forms in a type safe and generic way.

Central to the abstract scriptlet interface `bmi` is the function `bmi.form`, which makes it possible to construct a `form` element with the `bmi` scriptlet as the target action. The type list `h->w->nil` in the type of the `bmi.form` function specifies that form input elements for the form variables `h` and `w` must appear within the constructed `form` element. Notice that the types `h` and `w` within the type list `h->w->nil` are abstract type constructors and that the type lists in type parameters to the `elt` type can be constructed only through uses of the function `XHtml.inputtext` and other functions for constructing form input elements.

Notice also the importance of the order in which abstract type constructors appear within type lists.[2] For generating the abstract scriptlet interface, SMLserver induces the order of abstract type constructors from the order form variables are specified in the scriptlet functor argument.

---

[1] The abstract scriptlet interface has been simplified to include only `elt` type parameters that are used to track form variables.

[2] The Standard ML type system, does not—or so it seems—allow us to provide a type construction for sets if the maximum number of elements in the sets is not fixed.

4.3 Type-indexed Type List Reordering

In some cases it is desirable to reorder the components of a type list appearing in a type parameter to the `elt` type. Such a reordering is necessary if two forms entailing different orderings of form variable names use the same target scriptlet.

To allow for arbitrary reorderings, we now present a function `swapn`, which allows, within an element type, the head component of a type list to be swapped with any other component of the type list. The function provides the programmer with a type safe mechanism for converting an element of type `(`l`,nil)elt` to an element of type `(`l'`,nil)elt` where l and l' are type lists representing different permutations of the same set of elements. The function `swapn`, which is implemented as a type-indexed function [9, 14, 22], takes as its first argument a value with a type that represents the index for the component of the type list to be swapped with the head component. The specifications for the `swapn` function and the functions for constructing type list indexes are the following:

```
type ('old,'new)idx
val One   : unit -> ('a->'b->'x,'b->'a->'x)idx
val Succ  : ('a->'x,'b->'y)idx -> ('a->'c->'x,'b->'c->'y)idx
val swapn : ('x,'xx)idx -> ('x,'y)elt -> ('xx,'y)elt
```

As an example, the application `swapn(Succ(One()))` has type

```
('a->'b->'c->'x,'y)elt -> ('c->'b->'a->'x,'y)elt
```

which makes it possible to swap the head component (i.e., the component with index zero) in the enclosed type list with the second component of the type list. Safety of this functionality relies on the following lemma, which is easily proven by induction on the structure of `(`τ, τ'`)idx`:

Lemma 3 (Type indexed swapping). *For any value of type* `(`τ, τ'`)idx`*, constructed using* `One` *and* `Succ`*, the type lists* τ *and* τ' *are identical when interpreted as sets.*

5 Related Work

The Haskell WASH/CGI library [18–20] provides a type safe interface for constructing Web services in Haskell. The library uses a combination of type classes and phantom types to encode the state machine defined by the XHTML DTD and to enforce constructed documents to satisfy this DTD. Because Standard ML has no support for type classes, another approach was called for in SMLserver.

The JWIG project [4] (previously the `<bigwig>` project [2, 3, 17]) provides another model for writing Web applications for which generated XHTML documents are guaranteed to be well-formed and valid and for which submitted form data is guaranteed to be consistent with the reading of the form data. JWIG is based on a suite of program analyses that at compile time verifies that no runtime errors can occur while building documents or receiving form input.

For constructing XHTML documents, JWIG provides a template system with a tailor-made plugging operation, which in SMLserver and WASH/CGI amounts to function composition.

Both WASH/CGI and JWIG, allow state to be maintained on the Web server in so-called sessions. SMLserver does not support sessions explicitly, but does provide support for type safe caching of certain kinds of values [5]. The possibility of maintaining state on the Web server (other than in a database or in a cache) introduces a series of problems, which are related to how the Web server claims resources and how it behaves in the presence of memory leaks and system failures.

Other branches of work related to this paper include the work on using phantom types to restrict the composition of values and operators in domain specific languages embedded in Haskell and ML [8, 13, 18–20] and the work on using phantom types to provide type safe interaction with foreign languages from within Haskell and ML [1, 7]. We are aware of no other work that uses phantom types to express linear requirements.

Phantom types have also been used in Haskell and ML to encode dependent types in the form of type indexed functions [9, 14, 22]. In the present work we also make use of a type indexed function to allow form fields to appear in a form in an order that is different than the order the corresponding form variables are declared in scriptlet functor arguments.

Finally, there is a large body of related work on using functional languages for Web programming. Preceding Thiemann's work, Meijer introduced a library for writing CGI scripts in Haskell [15], which provided low-level functionality for accessing CGI parameters and sending responses to clients. Peter Sestoft's ML Server Pages implementation and SMLserver [5] provide good support for programming Web applications in ML, although these approaches give no guarantees about the well-formedness and validity of generated documents.

Queinnec [16] suggests using continuations to implement the interaction between clients and Web servers. Graunke et al. [11] demonstrate how Web programs can be written in a traditional direct style and transformed into CGI scripts using CPS conversion and lambda lifting. It would be interesting to investigate if this approach can be made to work for statically typed languages.

6 Conclusion

Based on our experience with the construction and maintenance of community sites and enterprise Web applications—in SMLserver and other frameworks—we have contributed with two technical solutions to improve reliability and the quality of such applications. Our first contribution is a novel approach to enforce conformity of generated XHTML 1.0 documents, based entirely on the use of a typed combinator library in Standard ML. Our second technical contribution is a technique for enforcing consistency between a scriptlet's use of form variables and the construction of forms that target that scriptlet.

References

1. M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *Workshop on Multi-language Infrastructure and Interoperability*, September 2001.
2. C. Brabrand, A. Møller, and M. Schwartzbach. Static validation of dynamically generated HTML. In *Workshop on Program Analysis for Software Tools and Engineering*, June 2001.
3. C. Brabrand, A. Møller, and M. Schwartzbach. The `<bigwig>` project. *Transactions on Internet Technology*, 2(2):79–114, 2002.
4. A. Christensen, A. Møller, and M. Schwartzbach. Extending Java for high-level Web service construction. *Transactions on Programming Languages and Systems*, 25(6), November 2003.
5. M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Int. Symposium on Practical Aspects of Declarative Languages*, January 2003.
6. M. Elsman and K. F. Larsen. Typing XHTML web applications in SMLserver. Technical Report ITU-TR-2003-34, IT University of Copenhagen, Denmark, 2003.
7. S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling hell from heaven and heaven from hell. In *Int. Conference on Functional programming*, 1999.
8. M. Fluet and R. Pucella. Phantom types and subtyping. In *Int. Conference on Theoretical Computer Science*, August 2002.
9. D. Fridlender and M. Indrika. Functional pearl: Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, July 2000.
10. P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *European Symposium On Programming*, April 2003.
11. P. Graunke, S. Krishnamurthi, R. Findler, and M. Felleisen. Automatically restructuring programs for the Web. In *Int. Conference on Automated Software Engineering*, September 2001.
12. C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.
13. D. Leijen and E. Meijer. Domain specific embedded compilers. In *Conference on Domain-specific languages*, 2000.
14. C. McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, July 2002.
15. E. Meijer. Server side Web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, January 2000.
16. C. Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In *Int. Conference on Functional Programming*, September 2000.
17. A. Sandholm and M. Schwartzbach. A type system for dynamic Web documents. In *Symposium on Principles of Programming Languages*, January 2000.
18. P. Thiemann. Programmable type systems for domain specific languages. In *Workshop on Functional and Logic Programming*, June 2002.
19. P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.
20. P. Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Conference on Practical Aspects of Declarative Languages*, January 2002.
21. W3C. XHTML™ 1.0: The extensible hypertext markup language, January 2000. Second Edition. Revised August 2002. `http://www.w3.org/TR/xhtml1`.
22. Z. Yang. Encoding types in ML-like languages. In *Int. Conference on Functional Programming*, September 1998.