Typing XHTML Web Applications in SMLserver

Martin Elsman
mael@itu.dk

Ken Friis Larsen
kfl@itu.dk

IT University of Copenhagen*

October 10, 2003

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 programs. The type system guarantees that only conforming XHTML documents are sent to clients and that the target scriptlets of client-submitted forms use form data consistently and in a type-safe way. The type system is encoded in the type system of Standard ML using so-called phantom types, which are used both to guarantee conformity of XHTML documents and to guarantee form consistency.

1 Introduction

In this paper we address two problems with the construction of Web applications. Traditionally, frameworks for developing Web applications give little guarantees about the conformity of generated HTML or XHTML documents. Moreover, although forms and form variables play a central rôle in the development of Web applications, most often, no static mechanism guarantees that the particular use of form data is consistent with the construction of a corresponding form.

SMLserver [7] is an efficient multi-threaded Web server platform for the programming language Standard ML [17]. We present a static type system for SMLserver scriptlets that guarantees that generated XHTML documents conform to the XHTML 1.0 specification [27]. In principle, *conformity* expresses that a document is well-formed, valid according to a specific DTD, and that a number of element prohibitions are satisfied.

Although many browsers render non-conforming XHTML well, the rendering is often browser specific. Thus, it is preferable for a Web application to generate conforming XHTML only. It turns out that it is difficult for a programmer to manually obey the many restrictions enforced by the XHTML 1.0 specification.

*ITU TR-2003-34, ISBN: 87-7949-047-6

In particular, the different element prohibitions (i.e., requirements that prohibit certain elements in certain contexts), but also the restrictions caused by the distinction between inline, block, and flow elements, can cause many programming mistakes.

The conformity requirements are enforced by requiring the Web programmer to construct XHTML documents through the use of a combinator library for which the different requirements are encoded in the types of element combinators using phantom types [8, 14, 1, 9, 24, 25, 23]. Because phantom types introduce type information that has no connection to the actual implementation of the combinators, phantom types are completely static and thus do not introduce a runtime overhead.

The Web application type system that we present guarantees that forms in generated documents are consistent with actual form data submitted by a client. A scriptlet in SMLserver is represented as a functor. Whereas the argument to the scriptlet functor represents form data received from a client, the body of the scriptlet functor represents program code that is executed when the scriptlet is requested by a client. In this sense, scriptlet functors are instantiated by SMLserver upon client requests, which can then result in new documents being sent to clients. Form data submitted by a client either corresponds to form variable arguments submitted with an HTTP GET request (e.g., a user follows a link) or to form data following a document in an HTTP POST request (a user fills out a form).

Because of limitations of Standard ML modules, it is not possible to encode the recursive nature of scriptlets directly using Standard ML modules. Instead, an *abstract scriptlet interface*, containing typing information about accessible scriptlets and their form arguments, is generated prior to compilation, based on a preprocessing of scriptlet functor arguments. The 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 system that we present 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 the form data. Similarly, form variable arguments appearing in hyper-link anchors to scriptlets must be consistent with the scriptlet's expectations of form variable arguments. In both cases, it is possible to construct XHTML documents that conform to the XHTML 1.0-Strict DTD, but that are not accepted by our system.

An important aspect of the type safe embedding of XHTML and form construction in the type system of Standard ML is that it immediately becomes possible to encode type safe polymorphic higher-order XHTML templates using Standard ML's support for Hindley-Milner style polymorphism and higher-order functions. Moreover, instead of using our type system as the basis for designing a domain-specific language for writing Web applications, the embedding of the type system within the type system of Standard ML makes it possible for the

programmer to reuse many existing tools and libraries.

1.1 Contributions

The main contributions of this paper are two-fold. First, we present a novel approach to enforce conformity of generated XHTML 1.0 documents, based entirely on the use of a typed combinator library based on phantom types. Although others have suggested type systems for guaranteeing conformity of generated XHTML documents [25, 23, 24, 21], none of these approaches can be used for embedding XHTML documents in ML-like languages that do not provide support for type classes [13]. To the best of our knowledge, the encoding of linearity constraints using phantom types is novel.

The second main contribution of this work is the type based technique we have developed for enforcing consistency between a scriptlet's use of form variables and the construction of forms that target that scriptlet. Whereas this technique is also based on phantom types, to encode *type lists* (i.e., lists at the type level), we also contribute with a type-indexed function [29, 10, 15] for swapping arbitrary elements in type lists.

Both of the two main contributions are formally justified and the techniques have been implemented in SMLserver and used for building various Web applications, including a form extensive quiz for employees at the IT University of Copenhagen. We are also in the process of porting existing SMLserver Web applications to utilize the new type support. 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.

1.2 Outline

The remainder of the paper is organized as follows. In Section 2 and 3, we describe how it can be enforced, statically, using phantom types, that generated XHTML documents conform to the XHTML 1.0 specification [27].

In Section 4, we present a mechanism that guarantees that forms generated with SMLserver are consistent with the target scriptlet's expectation of form data.

In Section 5, we present a basic formalization of scriptlets, in the style of [11], which captures the basic Web application model used by SMLserver. Related work is described in Section 6. Finally, in Section 7, we describe future work and conclude.

2 Generating Conforming XHTML

In essence, for a document to *conform* to the XHTML 1.0 specification [27], the document must be well-formed, valid according to a particular DTD, and obey certain element prohibitions. For a document to be well-formed, it is a requirement that it satisfies the following conditions:

1. All start-tags must have a corresponding closing-tag.

2. All elements must be properly nested.

3. No attribute name may appear more than once in the same start-tag.

We adopt the well-known technique for satisfying the first two well-formedness conditions by allowing the programmer to construct elements only through a library of combinators [23]. We return to the linearity condition for attributes in Section 3.

2.1 Static Validation of Generated Documents

Given the XHTML 1.0-Strict DTD, it is not difficult, for instance in Haskell or Standard ML, to construct a series of mutually recursive datatype declarations for representing documents in such a way that only valid documents are representable. However, for an XHTML interface to be convenient for a programmer, it is important that the interface is as narrow as possible, in the sense that no (or at least few) explicit datatype coercions are needed to coerce an element of some entity to be treated as an element of another entity.

XHTML distinguishes between elements that are `inline` entities and elements that are `block` entities. For instance, the `p` element requires its content to be an `inline` entity and results in a `block` entity. Similarly, the `em` element requires its content to be an `inline` entity and results in an `inline` entity. A third entity in XHTML is the `flow` entity, which allows for a kind of subtyping in the sense that both `inline` and `block` entities can be treated as `flow` entities. All elements that are `flow` entities may be used in `td` and `div` elements, for instance. A serious problem with the mutually recursive datatype declaration approach is that it requires all subtype coercions to be explicit (via tagging).

Another problem with the mutually recursive datatype declaration approach is that it does not allow for the same datatype constructor to be used for sequencing elements of different entities in a way that prohibits the construction of invalid documents. Further, the XHTML 1.0 specification specifies a set of element prohibitions [27, Appendix B], which are not directly expressed in the XHTML 1.0-Strict DTD. For example, an element prohibition specifies that, to all depth of nesting, an anchor element `a` must not be contained in other anchor elements. For a document to conform to the specification, all element prohibitions must be satisfied. Element prohibitions are not directly expressible with the mutually recursive datatype declaration approach, and neither is the well-formedness linearity condition on attributes.

The approach that we are going to take to guarantee statically that generated documents conform to the XHTML 1.0 specification is to allow the programmer to construct XHTML elements only through a library of typed combinators. The types of the combinators make use of phantom types to an extend that disallows combinators to be composed to construct non-conforming documents.

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

The DTD defines a context free grammar for Mini XHTML documents. Although it appears to be overly simplistic, the DTD captures an essential subset of XHTML. Not only does it capture the distinction between `inline`, `block`, and `flow` entities, but it also captures the notion of sequencing and a weakened form of element prohibitions expressible in a DTD. We postpone the discussion of attributes to Section 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:

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

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`, and `flow`) 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 $\boxed{\models d : \kappa}$

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

$$\frac{\models d : \texttt{inline}}{\models \texttt{em}(d) : \texttt{inpre}} \qquad \frac{\models d : \texttt{flow}}{\models \texttt{div}(d) : \texttt{block}} \qquad \frac{\models d : \texttt{inpre}}{\models \texttt{pre}(d) : \texttt{block}}$$

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

```
signature MINI_XHTML =
  sig
    type inl and blk and flw
    type inpre and preclosed
    type ('flow,'pre) elt
    val $   : string -> (inl,'p) elt
    val p   : (inl,'p)elt -> (blk,'p)elt
    val em  : (inl,'p)elt -> (inl,'p)elt
    val pre : (inl,inpre)elt -> (blk,'p)elt
    val big : (inl,'p)elt -> (inl,preclosed)elt
    val div : ('f,'p)elt -> (blk,'p)elt
    val &   : ('f,'p)elt * ('f,'p)elt -> ('f,'p)elt
    val &&  : ('f,'p)elt * ('g,'p)elt -> (flw,'p)elt
    val emp : unit -> ('f,'p)elt
  end
```

Figure 1: Mini XHTML combinator library.

$$\frac{}{\models \varepsilon : \kappa} \qquad\qquad \frac{\models d_1 : \kappa \qquad \models d_2 : \kappa}{\models d_1\ d_2 : \kappa}$$

A Standard ML signature for an abstract combinator library for Mini XHTML is given in Figure 1. The signature specifies a type constructor `('flow,'pre)elt` for element sequences. The type constructor takes two type parameters, corresponding to two separate requirements. The first parameter is used to specify whether the element is an `inline` entity, a `block` entity, or a `flow` entity. For instance, the `p` combinator requires its argument to be an `inline` entity and the result is a `block` entity. Using the infix sequence combinator `&`, it now becomes impossible to combine a `block` entity with an `inline` entity, as in `p($"hello") & $" world"`. Notice that because the `div` combinator is polymorphic in the `'flow` parameter, it can take either an `inline` entity or a `block` entity as argument, which amounts to implicit subtyping [9] for the subtyping hierarchy defined by the partial order `flow` $<$ `inline` and `flow` $<$ `block`.

It is apparent here that the approach has its limitations. Although the element `<div><p>Hello</p> world</div>` is a valid `block` element according to the DTD, the expression `p($"hello") & $" world"` cannot be given a type in our system, even if it appears as an argument to the `div` combinator. Instead, the element can be constructed with the expression `div(p($"hello") && $" world")`, which makes use of the `&&` 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 Figure 1 follows. The rules allow inference of sentences of the form $\vdash e : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt}$, where e ranges over expressions, where τ_{f} ranges over the nullary type constructors `inl`, `blk`, and `flw`, and where τ_{p} ranges over nullary type constructors `inpre` and `preclosed`.

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

$$\frac{}{\vdash \texttt{\$}\ c : (\texttt{inl}, \tau_{\mathrm{p}})\texttt{elt}} \qquad \frac{\vdash e : (\texttt{inl}, \tau_{\mathrm{p}})\texttt{elt}}{\vdash \texttt{p}\ e : (\texttt{blk}, \tau_{\mathrm{p}})\texttt{elt}} \qquad \frac{\vdash e : (\texttt{inl}, \tau_{\mathrm{p}})\texttt{elt}}{\vdash \texttt{em}\ e : (\texttt{inl}, \tau_{\mathrm{p}})\texttt{elt}}$$

$$\frac{\vdash e : (\texttt{inl}, \texttt{inpre})\texttt{elt}}{\vdash \texttt{pre}\ e : (\texttt{blk}, \tau_{\mathrm{p}})\texttt{elt}} \qquad \frac{\vdash e : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt}}{\vdash \texttt{div}\ e : (\texttt{blk}, \tau_{\mathrm{p}})\texttt{elt}} \qquad \frac{}{\vdash \texttt{emp()} : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt}}$$

$$\frac{\vdash e_1 : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt} \quad \vdash e_2 : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt}}{\vdash e_1 \texttt{\&} e_2 : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt}} \qquad \frac{\vdash e_1 : (\tau_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt} \quad \vdash e_2 : (\tau'_{\mathrm{f}}, \tau_{\mathrm{p}})\texttt{elt}}{\vdash e_1 \texttt{\&\&} e_2 : (\texttt{flw}, \tau_{\mathrm{p}})\texttt{elt}}$$

$$\frac{\vdash e : (\texttt{inl}, \tau_{\mathrm{p}})\texttt{elt}}{\vdash \texttt{big}\ e : (\texttt{inl}, \texttt{preclosed})\texttt{elt}}$$

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

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

An implementation of the `MINI_XHTML` signature in terms of a Standard ML structure appears in Figure 2.

Before we state a soundness property for the combinator library, we define

```
structure MiniXHtml :> MINI_XHTML =
  struct
    datatype e = Elt of string * e | Emp
               | Seq of e * e | S of string
    type ('flow,'pre)elt = e
    type inl = unit and blk = unit and flw = unit
     and inpre = unit and preclosed = unit
    fun $ s = S s
    fun op & p = Seq p
    fun op && p = Seq p
    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 op div e = Elt("div",e)
    fun emp() = Emp
  end
```

Figure 2: Mini XHTML Implementation.

a function *entity*, which relates instantiations of `elt` types to DTD entities:

$$\begin{aligned} \mathit{entity}(\texttt{(inl,preclosed)elt}) &= \texttt{inline} \\ \mathit{entity}(\texttt{(inl,inpre)elt}) &= \texttt{inpre} \\ \mathit{entity}(\texttt{(blk,preclosed)elt}) &= \texttt{block} \\ \mathit{entity}(\texttt{(blk,inpre)elt}) &= \texttt{block} \\ \mathit{entity}(\texttt{(flw,preclosed)elt}) &= \texttt{flow} \\ \mathit{entity}(\texttt{(flw,inpre)elt}) &= \texttt{flow} \end{aligned}$$

It is now possible to state a soundness lemma for the combinator library. The soundness lemma states that well-typed expressions results in documents that validate according to the Mini XHTML DTD. The lemma is easily proved by structural induction on the derivation $\vdash e : \tau$.

Lemma 1 (Soundness) *If* $\vdash e : \tau$ *then* $\models \mathit{doc}(e) : \mathit{entity}(\tau)$.

We now consider whether the combinator library is complete, in the sense that all valid documents can be constructed using the combinator library. It turns out that because of the element prohibition encoded in the combinator library and because element prohibitions are not—and cannot be—expressed by the DTD, there are documents that are valid according to the DTD, but which cannot be constructed using the combinator library. It also turns out, however, that it is possible to weaken the types for the combinators so that the element prohibitions are not enforced at arbitrary depth, but only to the extend that the prohibitions are encoded in the DTD.

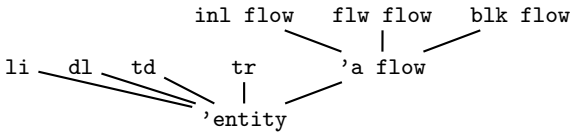

Figure 3: Partial order for the entity subtyping relation.

2.3 Entity Subtyping for Full XHTML

To support full XHTML, we extend the entity subtyping relation to be defined by the partial order in Figure 3. With this subtyping relation, it is possible to use the same sequencing combinator (i.e., `&`) to create sequences of `td` elements and sequences of `tr` elements, for instance. The benefits hereof are many. For instance, in the following Standard ML code, polymorphism makes it possible for the same function `iter` to be used for gluing `td` elements and `tr` elements:

```
fun iter f n = if n <= 1 then f 1
               else iter f (n-1) & f n

fun col r c = td ($(Int.toString ( r * c )))
fun row sz r = tr (iter (col r) sz)
fun tab sz = table (iter (row sz) sz)
```

The function, `tab` takes an integer N as argument and calls the function `iter`, via the `col` and `row` functions, to construct an $N \times N$ multiplication table. Yet, the type system guarantees that only valid XHTML code is constructed.

Notice also that with this subtyping relation, the flow-sequence combinator `&&` has the type:

```
('f1 flow,'p)elt * ('f2 flow,'p)elt -> (flw flow,'p)elt
```

It is still necessary to use this combinator to combine `inline` and `block` entities in a `div` element, for instance.

2.4 Composing Element Prohibitions

As mentioned earlier, the XHTML 1.0 specification specifies other element prohibitions than the element prohibition for the `big` element [27, Appendix B]. The specified element prohibitions, which apply to all depth of nesting, are:

1. An `a` element must not contain other `a` elements.
2. A `pre` element must not contain the elements `img`, `object`, `big`, `small`, `sub`, or `sup`.
3. A `button` element must not contain the elements `input`, `select`, `textarea`, `label`, `button`, `form`, `fieldset`, `iframe`, or `isindex`.

4. A `label` element must not contain other `label` elements.

5. A `form` element must not contain other `form` elements.

As we have seen in Section 2.2, it is possible to express element prohibitions using phantom types in an ML or Haskell combinator library. It turns out that it is possible to compose element prohibitions in an orthogonal way such that the encoding of one element prohibition has no influence on other element prohibitions or other requirements expressed by the phantom type approach. The way element prohibitions are composed is by having separate type parameters for each element prohibition. In this way, the `elt` type constructor for full XHTML 1.0 is parameterized over six type parameters, one for the encoding of the entity subtyping relation and five for the encoding of the five element prohibitions.

3 XHTML 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.

One important XML well-formedness requirement, which is therefore also an XHTML well-formedness requirement, is that no attribute name may appear more than once in a start tag or empty-element tag [26, Section 3.1]. In this section we shall see how this linearity constraint on attribute lists can be enforced statically using phantom types.

3.1 Attributes in Mini XHTML

The signature `MINI_XHTML_ATTR` in Figure 4 specifies operations for constructing linear lists of attributes, that is, lists of attributes for which an attribute with a given name appears at most once in a list. For simplicity, the 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 the function's right argument. The result type provides incoming and outgoing

```
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
```

Figure 4: Mini XHTML attribute library.

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 that the attribute list contributes with.

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 Figure 4 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 the τ_n, $n \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$ ranges over the types `na`, `align`, `width`, and `height`.

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

$$\frac{}{\vdash \texttt{align left} : (\texttt{na}, \texttt{align}, \tau_\mathrm{b}, \tau_\mathrm{b}, \tau_\mathrm{c}, \tau_\mathrm{c})\texttt{attr}}$$

$$\frac{}{\vdash \texttt{align right} : (\texttt{na}, \texttt{align}, \tau_\mathrm{b}, \tau_\mathrm{b}, \tau_\mathrm{c}, \tau_\mathrm{c})\texttt{attr}}$$

$$\frac{}{\vdash \texttt{width}\ n : (\tau_\mathrm{a}, \tau_\mathrm{a}, \texttt{na}, \texttt{width}, \tau_\mathrm{c}, \tau_\mathrm{c})\texttt{attr}}$$

$$\frac{}{\vdash \texttt{height}\ n : (\tau_\mathrm{a}, \tau_\mathrm{a}, \tau_\mathrm{b}, \tau_\mathrm{b}, \texttt{na}, \texttt{height})\texttt{attr}}$$

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

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

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

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.

Lemma 2 (Linearity of attribute lists)
If $\vdash e : (\tau_\mathrm{a}^0, \tau_\mathrm{a}^1, \tau_\mathrm{b}^0, \tau_\mathrm{b}^1, \tau_\mathrm{c}^0, \tau_\mathrm{c}^1)\texttt{attr}$ *then*

1. *the number of* `align` *attributes in* e *is* $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}^0$
2. *the number of* `width` *attributes in* e *is* $\tau_\mathrm{b}^1 \div \tau_\mathrm{b}^0$
3. *the number of* `height` *attributes in* e *is* $\tau_\mathrm{c}^1 \div \tau_\mathrm{c}^0$

Proof The interesting case is the case for appending two attribute lists. We have $e = e_1\ \texttt{\%}\ e_2$. From (1), it follows that there exist types τ_a, τ_b, and τ_c, so that we can apply induction twice to get that the number of `align` attributes in e_1 is $\tau_\mathrm{a} \div \tau_\mathrm{a}^0$ and that the number of `align` attributes in e_2 is $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}$. We now proceed by case analysis on $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}$. There are two cases:

case: $\tau_\mathrm{a}^1 \div \tau_\mathrm{a} = 0$. In this case we have $\tau_\mathrm{a} = \tau_\mathrm{a}^1$. It follows that $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}^0 = \tau_\mathrm{a} \div \tau_\mathrm{a}^0$. Thus the number of `align` attributes in e is $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}^0$, as required.

case: $\tau_\mathrm{a}^1 \div \tau_\mathrm{a} = 1$. In this case we have $\tau_\mathrm{a} = \texttt{na}$. From the definition of $\div$, for $\tau_\mathrm{a} \div \tau_\mathrm{a}^0$ to be defined, we have $\tau_\mathrm{a}^0 = \texttt{na}$, which means that the number of `align` attributes in e_1 is 0. Thus, the number of `align` attributes in e is $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}$, which equals $\tau_\mathrm{a}^1 \div \tau_\mathrm{a}^0$, as required.

Similar arguments are used for the `width` and `height` attributes. □

3.2 Narrowing the Attribute Interface

The simplest way to provide support for more attribute names in Mini XHTML is to add two new type parameters to the type constructor `attr` for each new attribute name. Following this strategy, however, the number of type variable parameters for the `attr` type constructor amounts to twice the total number of attributes in XHTML. Fortunately, 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, it makes sense to use particular type variable parameters for different attributes in different contexts.

To do so, we refine the strategy so that each attribute makes use of a triple of type variable parameters for each attribute. The first type variable parameter now denotes the particular attribute that the triple is used for, whereas the two other type variable parameters are used to encode the linearity information as before.

We first construct an *attribute interference graph* with the nodes in the graph being all attribute names supported by element entries in the DTD. Further, an edge is added to the graph between two attribute names if the attribute names are supported by the same element. To construct a coloring of the attribute names in the graph, where a *color* denote a type variable triple in the `attr` type constructor, we keep removing the nodes with the fewest neighbors from the graph and pushing them onto a stack. When the graph is empty, we pop nodes off the stack and reinsert them in the graph. Whenever an attribute name is reinserted in the graph, the attribute name is given a color different from its existing neighbors in the graph.

As an example, consider the following simplified subset of the XHTML 1.0-Strict DTD:

```
<!ELEMENT img EMPTY>
<!ELEMENT table (tr)+>
<!ELEMENT tr    (td)+>
<!ELEMENT td    (%flow)*>
<!ATTLIST img   id      ID           #IMPLIED
                height  %Number;     #IMPLIED
                width   %Number;     #IMPLIED>
<!ATTLIST table id      ID           #IMPLIED
                width   %Number;     #IMPLIED>
<!ATTLIST tr    id      ID           #IMPLIED>
<!ATTLIST td    id      ID           #IMPLIED
                colspan %Number;     #IMPLIED
                rowspan %Number;     #IMPLIED>
```

From the attribute specifications in this DTD subset, we generate the attribute interference graph shown in Figure 5. Following the graph coloring algorithm, we can obtain the coloring annotated on the graph. The resulting attribute interface for the specified DTD looks as follows:

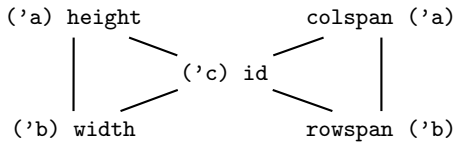

Figure 5: Attribute interference graph.

```
signature ATTR =
  sig
    type ('a,'a1,'a2,'b,'b1,'b2,'c,'c1,'c2)attr
    type na and id and width and height
     and colspan and rowspan
    val id : string ->
     ('a0,'a,'a,'b0,'b,'b,id,na,id)attr
    val width : int ->
     ('a0,'a,'a,width,na,width,'c0,'c,'c)attr
    val height : int ->
     (height,na,height,'b0,'b,'b,'c0,'c,'c)attr
    val colspan : int ->
     (colspan,na,colspan,'b0,'b,'b,'c0,'c,'c)attr
    val rowspan : int ->
     ('a0,'a,'a,rowspan,na,rowspan,'c0,'c,'c)attr
    val % : ('a,'a1,'a2,'b,'b1,'b2,'c,'c1,'c2)attr
          * ('a,'a2,'a3,'b,'b2,'b3,'c,'c2,'c3)attr
         -> ('a,'a1,'a3,'b,'b1,'b3,'c,'c1,'c3)attr
  end
```

We have used this approach to generate an attribute combinator library for a large part of the XHTML 1.0-Strict DTD. At present, the combinator library provides support for 18 different attributes, using only 21 type variable parameters in the `attr` type constructor.

3.3 Adding Attributes to Elements

We have still to show how particular attributes may be added to elements in a type safe way. The approach we take here is, for each element name, to introduce a new attribute-accepting combinator, which takes as its first argument an attribute list. The idea is then that the type of the argument specifies which attributes are supported by the element. For instance, for the DTD subset given above, the combinator library provides a function `tda` with the following type:[1]

```
(colspan,na,'a,rowspan,na,'b,id,na,'c)attr
 -> 'f flow elt -> td elt
```

[1]For brevity, type variable parameters for encoding element prohibitions in the `elt` type constructor are not shown.

Given an attribute list and a `flow` element, the function returns a `td` element, which may then be composed with other `td` elements before it is passed to the `tr` element combinator. Notice that the `tda` combinator is polymorphic in the outgoing linearity information for each supported attribute. This polymorphism means that none of the attributes are required attributes.

Another approach to adding attributes to elements is to follow the element transforming style introduced by Thiemann [24], which has the property that no additional attribute-accepting combinator needs to be introduced for each element. A drawback of this approach, however, is that an additional type variable parameter is needed for the `elt` type constructor.

4 Form Consistency

In this section, we demonstrate how form consistency is checked in SMLserver. 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*, which are functors with arguments that specify the expected form variables.

Prior to the compilation proper, SMLserver performs a pre-scan of all scriptlet functor arguments and generates an abstract scriptlet API (in terms of a structure with an abstract interface) that, in a type safe way, exposes functions for constructing hyper-link anchors and `form` elements for all scriptlets. Both library modules and scriptlets may depend on the abstract scriptlet API, which makes it possible to write library functions and scriptlets that, in a type safe way, result in documents containing any number of different forms and links.

Code for instantiating scriptlet functors is generated automatically by SMLserver and executed upon client requests. For increased efficiency, SMLserver caches the result of library code execution [7].

An example scriptlet looks as follows:

```
functor bmi (F : sig val h : int Form.var
                     val w : int Form.var
                 end) : SCRIPTLET =
  struct open Scriptlets infix &
    val h = Form.get Page.page "Height" h
    val w = Form.get Page.page "Weight" 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 responses with different status codes, including HTTP responses with status code 200 followed by a valid XHTML document, or HTTP redirect responses (status code 302).

The argument of a scriptlet functor specifies form variables, which represent either form arguments in a GET request or posted data in a POST request. Although SMLserver guarantees that forms and links appearing in XHTML documents match the respective target scriptlet's expectations of form arguments, there is no way that SMLserver can guarantee that a user indeed enters an integer, for instance.[2] In the `bmi` example, both the `h` and `w` form arguments are specified with type `int Form.var`, which suggests that the user is supposed to provide integers for these form arguments. Using the function `Form.get`, the `bmi` scriptlet converts the form arguments into integers and responds with an error page in case one of the form arguments is not an integer.

In case both the 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`.

4.1 Static Tracking of Form Variables

Before we present a scriptlet that responds with a form for the `bmi` scriptlet, we first present a simplified `XHTML` signature, which provides operations that propagate information about form input elements in a type safe way:

```
signature XHTML =
  sig
    type ('a,'b)elt
    type nil
    type ('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
```

Similarly 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`.

[2] Neither can SMLserver guarantee that a user does not obfuscate a GET request by altering form arguments in a browser's location bar. In this case, SMLserver responds with a standard error page.

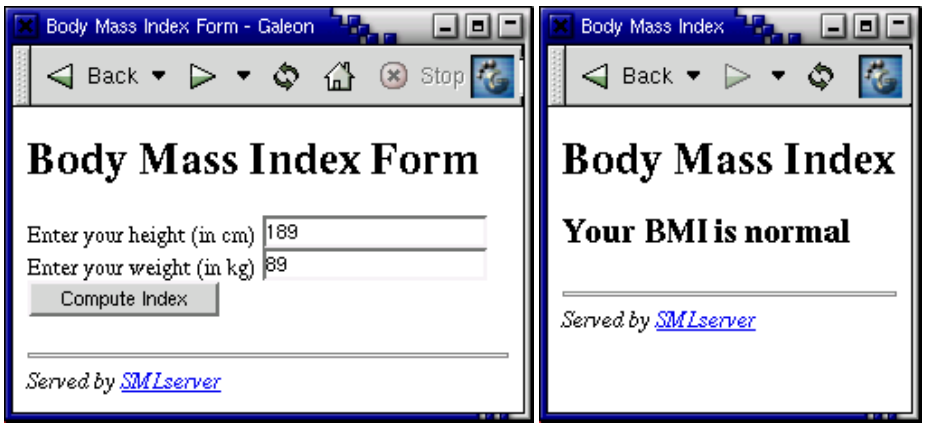

Figure 6: Pages served by the BMI Web service.

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 does 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
```

Figure 6 shows the result of requesting the `bmiform` scriptlet. The target of the form in this scriptlet is the `bmi` scriptlet.

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. Whereas the applications of the `inputtext` function are used to construct `input` elements with type `text`, the `inputsubmit` function is used to construct an `input` element with type `submit` and the text "Compute Index".

4.2 Abstract Scriptlet Interfaces

As mentioned earlier, inter-scriptlet form typing depends critically on the generation of an abstract scriptlet interface based on a pre-scan of all scriptlet functor arguments. The abstract scriptlet interface specifies a structure for each scriptlet. Each structure provides type safe functions for creating XHTML hyper reference anchors (i.e., links) and form elements for targeting the specific scriptlet. The interface also specifies a type safe function for sending an HTTP 302 redirect response to a client.

In the case for the `bmiform` and `bmi` scriptlets, the generated abstract scriptlet interface `Scriptlets` includes the following structure specifications:[3]

```
structure bmiform : sig
  val form : (nil,nil) elt -> (nil,nil) elt
  val link : ('x,'y) elt -> ('x,'y) elt
  val redirect : Http.response
end
structure bmi : sig
  type h
  type 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
  val redirect : {h:int, w:int} -> Http.response
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`. Similarly as for the `bmi.link` function, the type for the `bmi.redirect` function requires integers for the form variables `h` and `w` to be passed to the function in a record. Contrary, 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

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

`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` and other functions for constructing form input elements. Notice that 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 requires that form input elements for the form variables `h` and `w` 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 that the order in which abstract type constructors appear within type lists does matter. SMLserver induces the order in which abstract type constructors for form variable names appear within the argument to `form` functions from the order in which the corresponding form variables are specified in the scriptlet functor argument.

4.3 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 in particular necessary if two different forms entailing different orderings of form variable names use the same target scriptlet. What one could hope for is a system where type lists are interpreted as sets. The Standard ML type system, however, does not—or so it seems—allow us to provide a type construction for sets in the case that the maximum number of elements in the sets is not known in advance.

A first solution could be to introduce a function `swap` with type

```
('n->'m->'a,'b)elt -> ('m->'n->'a,'b)elt
```

which allows the programmer to explicitly swap the first and second element in a type list appearing in an element type. Again, the implementation of the `swap` function is simply the identity. Although this mechanism can be used to reorder type lists in many ways by applying the `swap` function to element values in a program, the mechanism is not complete in the sense that any reordering can be obtained, given an arbitrary type list. By applying the `swap` function to different points in the construction of an element of type `(a->b->c->nil,nil)elt`, it is possible to convert this element into an element where the components of the type list have been reordered arbitrarily. However, it turns out that it is not possible to convert an element of type `(a->b->c->d->nil,nil)elt` to an element of type `(c->a->d->b->nil,nil)elt` by inserting applications of the `swap` function at different points in the construction of the element.

A general solution to the problem of reordering type lists would be to provide the programmer with a type safe mechanism for swapping, within an element type, the head component of a type list with any other component of the type list. While it remains to be shown how to retrieve this functionality, it would provide 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.

4.4 Type-indexed Type List Reordering

We now present a general type safe function `swapn`, which allows the programmer to swap, within an element type, the head component of a type list with any other component of the type list. The function `swapn`, which is implemented as a type-indexed function [29, 10, 15], 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.*

4.5 Other Input Controls

Whereas the radio button input-control in XHTML gives a user the choice between several options, of which at most one can be selected at any particular time, check boxes allow the user to select multiple options. SMLserver provides the programmer with a series of type safe combinators for constructing radio buttons and check boxes, as well as other input controls such as selection boxes and various text controls.

SMLserver also provides support for passing values in `input` elements of type `hidden` in a type safe way. This is done by providing combinators for

constructing *form values* (of type (τ)`Form.var`) from values (of type τ) that support marshalling.

For constructing check boxes, three combinators are provided:

```
val chkbox  : ('n,'t list)name -> 't Form.var
           -> ('n chk->'x,'x)elt
val chkbox' : ('n,'t list)name -> 't Form.var
           -> ('n chk->'x,'n chk->'x)elt
val chkdrop : ('n chk->'x,'y)elt -> ('n->'x,'y)elt
```

The `chkbox` combinator may be used to construct a first check box in a group of check boxes. The combinator takes as arguments a form variable name and a form value associated with this particular check box. In the case a user selects the check box, this value is passed to the target script along with values for other checked boxes. The second combinator `chkbox'` assumes that at least one radio button (with the same form variable name) appears somewhere in the remainder of the form. In this way, it is possible to write code for constructing forms with a dynamic number of check boxes in a type safe way. The combinator `chkdrop` has no operational meaning, but eliminates the information that the form variable name is associated with a group of check boxes in the form.

For a target scriptlet to read the list of values associated with checked boxes in a form, the scriptlet may use the `Form.get` function described in Section 4.

5 A Formalization of Scriptlets

In this section we formalize a small scriptlet language. The formalization justifies the implementation of scriptlets in SMLserver, where, as we have seen, scriptlets are implemented as functors and where the mutually recursive typing of scriptlets is implemented by the generation of an abstract scriptlet interface prior to type inference and compilation proper.

The scriptlet language that we introduce does not allow the programmer to maintain state on the Web server. Instead, state in scriptlet programs must be modeled using form variables.

5.1 A Scriptlet Language

We assume denumerably infinite sets of *program variables*, ranged over by x and f, and *scriptlet identifiers*, ranged over by s. We also assume a denumerably infinite set of integers, ranged over by i. The grammars for *documents* (d), *values* (v), *expressions* (e), *scriptlet bindings* (S), and *programs* (p) are defined as follows:

$$
\begin{array}{rcl}
d & ::= & \texttt{link}\ s\ v \mid d_1\ \texttt{\&}\ d_2 \mid \varepsilon \\
v & ::= & d \mid i \mid \texttt{fix}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e \mid \texttt{(}v_1\texttt{,}\ v_2\texttt{)} \\
e & ::= & v \mid x \mid e_1\ e_2 \mid e_1\ \texttt{\&}\ e_2 \mid \texttt{(}e_1\texttt{,}\ e_2\texttt{)} \\
 & \mid & \texttt{link}\ s\ e \mid \texttt{\#1}\ e \mid \texttt{\#2}\ e
\end{array}
$$

$$
\begin{array}{rcl}
S & \texttt{::=} & s\texttt{(}x\texttt{) = }e\texttt{ ; }S \mid \bullet \\
p & \texttt{::=} & \texttt{scripts }S\texttt{ in }e
\end{array}
$$

A document d may either represent the empty document ε, a link to a scriptlet s with form argument v, or a document representing the composition of two sub-documents. A value represents either a document, an integer i, a (possibly recursive) function closure, or a value pair. Expressions include support for extracting components of pairs, constructing documents, applying functions, and representing values.

We sometimes interpret a scriptlet binding S as a set of bindings.

5.2 Scriptlet Evaluation

Scriptlet evaluation represents evaluation on the Web server. The grammars for *evaluation contexts* (E) and *instructions* (ι) are defined as follows:

$$
\begin{array}{rcl}
E & \texttt{::=} & [\cdot] \mid \texttt{link }s\ E \mid E\ \texttt{\&}\ e \mid v\ \texttt{\&}\ E \mid E\ e \\
 & \mid & v\ E \mid \texttt{(}E\texttt{,}e\texttt{)} \mid \texttt{(}v\texttt{,}E\texttt{)} \mid \texttt{\#1 }E \mid \texttt{\#2 }E \\
\iota & \texttt{::=} & \texttt{\#1 (}v_1\texttt{,}v_2\texttt{)} \mid \texttt{\#2 (}v_1\texttt{,}v_2\texttt{)} \mid \texttt{(fix }f\texttt{(}x\texttt{) = }e\texttt{)}\ v
\end{array}
$$

Evaluation rules for expressions and programs allow inference of sentences of the forms $e \rightsquigarrow e'$ and $p \rightsquigarrow p'$, respectively.

Expressions $\boxed{e \rightsquigarrow e'}$

$$\texttt{(fix }f\texttt{(}x\texttt{) = }e\texttt{)}\ v \rightsquigarrow e[v/x][\texttt{(fix }f\texttt{(}x\texttt{) = }e\texttt{)}/f]$$

$$\texttt{\#1 (}v_1\texttt{,}v_2\texttt{)} \rightsquigarrow v_1 \qquad\qquad \texttt{\#2 (}v_1\texttt{,}v_2\texttt{)} \rightsquigarrow v_2$$

$$\frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']}$$

Programs $\boxed{p \rightsquigarrow p'}$

$$\frac{e \rightsquigarrow e'}{\texttt{scripts }S\texttt{ in }e \rightsquigarrow \texttt{scripts }S\texttt{ in }e'}$$

5.3 Link Reduction

Link reduction represents evaluation on the Web client. Document contexts are defined according to the following grammar:

$$D \quad \texttt{::=} \quad [\cdot] \mid D\ \texttt{\&}\ d \mid d\ \texttt{\&}\ D$$

Programs that represent a document may be reduced according to the following rule for *link reduction*, which models that a user, non-deterministically, selects a link in a document. Link reduction allows inference of sentences of the form $p \xrightarrow{\text{LINK}} p'$.

Link reduction $\boxed{p \xrightarrow{\text{LINK}} p'}$

$$\frac{(s(x) \texttt{ = } e) \in S}{\texttt{scripts } S \texttt{ in } D[\texttt{link } s\ v] \xrightarrow{\text{LINK}} \texttt{scripts } S \texttt{ in } e[v/x]}$$

5.4 A Type System for Scriptlets

We now present a type system for the scripting language. The type system guarantees that well-typed scriptlet programs do not get stuck, except if the program reduces to the empty document ε, which provides no continuation links.

The grammars for *types* (τ), *type environments* (Γ), and *scriptlet environments* (Σ) are as follows:

$$\begin{array}{lcl} \tau & \texttt{::=} & \texttt{int} \mid \texttt{doc} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\ \Gamma & \texttt{::=} & x : \tau\ ;\ \Gamma \mid \bullet \\ \Sigma & \texttt{::=} & s : \tau\ ;\ \Gamma \mid \bullet \end{array}$$

In any type environment Γ, we assume that a variable is associated with a type at most once. We sometimes interpret type environments as sets and we write $\Gamma(x)$ to denote the type τ such that $(x : \tau) \in \Gamma$. Similarly for scriptlet environments.

The typing rules allow inference of sentences of the forms $\Sigma, \Gamma \vdash e : \tau$ and $\vdash S : \Sigma$ and $\vdash p : \texttt{doc}$.

Expressions $\boxed{\Sigma, \Gamma \vdash e : \tau}$

$$\frac{}{\Sigma, \Gamma \vdash i : \texttt{int}} \qquad \frac{}{\Sigma, \Gamma \vdash \varepsilon : \texttt{doc}} \qquad \frac{\Sigma(s) = \tau \rightarrow \texttt{doc} \quad \Sigma, \Gamma \vdash e : \tau}{\Sigma, \Gamma \vdash \texttt{link } s\ e : \texttt{doc}}$$

$$\frac{\Sigma, \Gamma \vdash e_1 : \texttt{doc} \quad \Sigma, \Gamma \vdash e_2 : \texttt{doc}}{\Sigma, \Gamma \vdash e_1 \texttt{ \& } e_2 : \texttt{doc}} \qquad \frac{\Sigma, \Gamma \vdash e_1 : \tau_1 \quad \Sigma, \Gamma \vdash e_2 : \tau_2}{\Sigma, \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Sigma, \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Sigma, \Gamma \vdash e_2 : \tau}{\Sigma, \Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{\Sigma, \Gamma \vdash e : \tau_1 \times \tau_2}{\Sigma, \Gamma \vdash \texttt{\#1 } e : \tau_1} \qquad \frac{\Sigma, \Gamma \vdash e : \tau_1 \times \tau_2}{\Sigma, \Gamma \vdash \texttt{\#2 } e : \tau_2}$$

$$\frac{\Sigma, \Gamma \ ; \ f : \tau \to \tau' \ ; \ x : \tau \vdash e : \tau'}{\Sigma, \Gamma \vdash \texttt{fix}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e : \tau \to \tau'}$$

Scripts $\boxed{\vdash S : \Sigma}$

$$\frac{\begin{array}{c}\Sigma = \{s_1 : \tau_1 \to \texttt{doc}, \cdots, s_n : \tau_n \to \texttt{doc}\} \\ \Sigma, x_i : \tau_i \vdash e_i : \texttt{doc} \quad i = 1..n\end{array}}{\vdash s_1\texttt{(}x_1\texttt{)}\ \texttt{=}\ e_1 \ ; \ \cdots \ ; \ s_n\texttt{(}x_n\texttt{)}\ \texttt{=}\ e_n : \Sigma}$$

Programs $\boxed{\vdash p : \texttt{doc}}$

$$\frac{\vdash S : \Sigma \quad \Sigma \vdash e : \texttt{doc}}{\vdash \texttt{scripts}\ S\ \texttt{in}\ e : \texttt{doc}}$$

The proof of type safety is based on well-known techniques for proving type safety for statically typed languages [18, 28]. We first state a property saying that a well-typed expression is either a value or can be separated into an evaluation context and an instruction:

Lemma 4 (Unique Decomposition) *If $\Sigma \vdash e : \tau$, then either (1) e is a value, or (2) there exists a unique E, e', and τ' such that $e = E[e']$ and $\Sigma \vdash e' : \tau'$ and e' is an instruction.*

Proof By induction on the structure of e. □

The following type preservation and progress lemmas hold:

Lemma 5 (Type Preservation) *If $\Sigma \vdash e : \tau$ and $e \rightsquigarrow e'$ then $\Sigma \vdash e' : \tau$. Moreover, if $\vdash p : \texttt{doc}$ and $p \rightsquigarrow p'$ then $\vdash p' : \texttt{doc}$. Finally, if $\vdash p : \texttt{doc}$ and $p \xrightarrow{\text{LINK}} p'$ then $\vdash p' : \texttt{doc}$.*

Lemma 6 (Progress) *If $\Sigma \vdash e : \tau$ then either (1) e is a value (possibly a document), or (2) $e \rightsquigarrow e'$ for some e'.*

The following lemma suggests that documents served by scripts are indeed documents and that reduction is stuck only if a served document contains no links:

Lemma 7 (Web Progress) *If $\vdash p : \texttt{doc}$ then either (1) $p \rightsquigarrow p'$ for some p' (evaluation), or (2) p is a document containing no links, or (3) $p \xrightarrow{\text{LINK}} p'$ for some p' (user follows a link).*

6 Related Work

The approach that we present here is not the first approach to guarantee well-formedness and validity of XHTML documents and consistent use of forms. First, the Haskell WASH/CGI library [24, 25, 23] provides a type safe interface for constructing Web services in Haskell. Although Thiemann's element transforming style is very much related to the element type encoding that we use here, the WASH/CGI 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, 5] (previously the `<bigwig>` project [21, 2, 3]) 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. In contrast to SMLserver and WASH/CGI, 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 of the above mentioned systems, 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 [7]. 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 claim resources and how it behaves in the presence of memory leaks and system failures.

Another branch of work related to this paper uses phantom types to restrict the composition of values and operators in domain specific languages embedded in Haskell and Standard ML [14, 9, 24, 25, 23]. This branch of work also includes the work by Thiemann on WASH/CGI. Also related to this work is the work on using phantom types to provide type safe interaction with foreign languages from within Haskell and Standard ML programs [8, 1]. As far as we know, we are the first to express all validity requirements of XHTML using phantom types. Moreover, we are aware of no other work that uses phantom types to express linear requirements, as we do to restrict an attribute from being added to an element more than once, for instance.

Phantom types have also been used in Haskell and Standard ML to encode dependent types in the form of type indexed functions [29, 10, 15]. 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 [16], which provided low-level functionality for accessing CGI parameters and sending responses to clients. The `mod_haskell`

project [6] took the approach of embedding the Hugs Haskell interpreter as a module for the Apache Web server [22]. Peter Sestoft's ML Server Pages implementation for Moscow ML [20] and SMLserver [7] provide good support for programming Web applications in Standard ML, although these approaches give no guarantees about the well-formedness and validity of generated documents.

Queinnec [19] suggests using continuations to implement the interaction between clients and Web servers. Graunke et al. [12] demonstrate how Web programs can be written in a traditional direct style and transformed into CGI scripts using CPS conversion and lambda lifting. In contrast to Queinnec, their approach uses the client for storing state information (i.e, continuation environments) between requests. It would be interesting to investigate if this approach can be made to work for statically typed languages, such as Standard ML or Haskell.

7 Conclusion and Future Work

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.

The technical contributions are justified theoretically and we have devised a theoretical concise model for understanding the essence of scriptlets, which form a basis for the implementation of scriptlets in SMLserver. Although the theoretical frameworks presented in this paper might seem heavy handed, the formalisms have given us a precise vocabulary for discussing future developments and desired properties.

There are several directions for future work. First, a natural question to ask is to which extend the type safe embedding of XHTML in Standard ML can be generalized to work for other DTDs than the XHTML 1.0-Strict DTD. Another line of future work is to use some of the techniques that we have described here for embedding database queries—in the form of SQL code—in ML-like languages, extending the work by Leijen and Meijer [14].

References

[1] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively.". In *Workshop on Multi-language Infrastructure and Interoperability (BABEL'01)*, September 2001.

[2] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001.

[3] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X, October 2002.

[5] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 2003. To appear.

[6] Eelco Dolstra and Armijn Hemel. *mod_haskell*, January 2000. `http://losser.st-lab.cs.uu.nl/mod_haskell`.

[7] Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.

[8] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *ACM International Conference on Functional programming*. ACM Press, 1999.

[9] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *International Conference on Theoretical Computer Science (TCS'2002)*, August 2002.

[10] Daniel Fridlender and Mia Indrika. Functional pearl: Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, July 2000.

[11] Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In *European Symposium On Programming (ESOP'03)*, April 2003.

[12] Paul Graunke, Shriram Krishnamurthi, Robert Bruce Findler, and Matthias Felleisen. Automatically restructuring programs for the Web. In *17th IEEE International Conference on Automated Software Engineering (ASE'01)*, September 2001.

[13] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[14] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Conference on Domain-specific languages*. ACM Press, 2000.

[15] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, July 2002.

[16] Erik Meijer. Server side Web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, January 2000.

[17] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[18] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1995.

[19] Christian Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In *Fifth International Conference on Functional Programming (ICFP'00)*, September 2000.

[20] Sergei Romanenko, Claudio Russo, and Peter Sestoft. *Moscow ML Owner's Manual*, June 2000. For version 2.00. 35 pages.

[21] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic Web documents. In *ACM Symposium on Principles of Programming Languages (POPL'2000)*, January 2000.

[22] Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, April 1999. ISBN 1-56592-567-X.

[23] Peter Thiemann. Programmable type systems for domain specific languages. In *Workshop on Functional and Logic Programming (WFLP'02)*, June 2002.

[24] Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002.

[25] Peter Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Conference on Practical Aspects of Declarative Languages (PADL'02)*, January 2002.

[26] W3C. Extensible markup language (XML) 1.0 (second edition), October 2000. W3C Recommendation.

[27] W3C. XHTML$^{\text{TM}}$ 1.0: The extensible hypertext markup language (second edition), January 2000. W3C Recommendation. Revised August 2002.

[28] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[29] Zhe Yang. Encoding types in ML-like languages. In *ACM International Conference on Functional Programming (ICFP'98)*, September 1998.