Electronic Notes in Theoretical Computer Science 76 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume76.html` 19 pages

Programmable Type Systems for Domain Specific Languages

Peter Thiemann [1,2]

Institut für Informatik
Universität Freiburg
Georges-Köhler-Allee 079
D-79110 Freiburg, Germany

Abstract

A language with a programmable type system is vital for the construction of an embedded domain specific language (EDSL). Driven by the requirements posed by the implementation of an EDSL for server-side Web scripting, we examine two major of extensions to the type system of the host language, Haskell. We show that a component that ensures the generation of correct HTML documents can take good advantage of type-level functions, as implemented using functional logic overloading. We further show that a function that ensures the consistency of data submitted to a Web script with the data expected by the script is less awkward to use in the presence of lambda expressions in the type language. In both cases we assess the guarantees obtained by the use of the typing and explore alternative solutions.

1 Introduction

Domain specific languages (DSLs) are addressing the programming needs for particular domains. They are intended to address and solve problems in the domain in terms of the concepts of the domain itself. Thus, they can improve the productivity of domain experts, who need not be programming experts.

Since DSLs have a limited user community, the time spent for their development and implementation must be carefully weighted against the productivity gain of their users. For that reason, a popular way of implementing a DSL is to embed it in a general purpose programming language, the host language. In particular, functional programming languages have proven to be good host languages because of their orthogonal abstraction facilities and

[1] Email: `thiemann@informatik.uni-freiburg.de`
[2] This work was completed during a stay at Oregon Health & Science University supported by VolkswagenStiftung.

their advanced type systems. Another advantage of the embedding approach is that the DSL is easily extensible and customizable to particular problem instances.

However, it turns out that the type system is an asset and a drawback at the same time. While it is possible to express a number of domain concepts in the type language, the resulting typings are often far from natural — some typing requirements even force a particular programming style. Such "programming conventions" are hard to motivate for users that are not expert-programmers in the host language, in particular, if they lead to incomprehensible error messages from the type checker. At this point the domain expert needs a programming languages expert to decipher the compiler's utterances.

Hence, we are striving to identify better ways to integrate domain concepts into the type language, in the context of the functional programming language Haskell. First, we develop a motivating example drawn from a DSL for Web scripting[16,17,18]. Guided by this example, we show that a viable way of extending type languages is by installing a (first-order) functional logic programming language at the type level [4]. The idea is that a term rewriting system at the type level captures the domain-specific concepts and the execution machinery of functional logic programming (*e.g.*, narrowing or residuation) is the foundation of the type inference procedure. Ideally, the semantics of the term rewriting system at the type level coincides with the semantics of the programming language at the value level, thus allowing to lift the value level functions to the type level without changing their semantics. The underlying theory builds on the HM(X) type-inference framework [15].

The next step is the addition of lambda expressions at the type level [13]. This step is again driven by practical needs from the application domain where we would like to express problems in the simplest and most natural possible way. We exhibit a problem whose Haskell98-based solution leads to contorted programs that are hard to explain and understand. Next, we discuss another solution using lambda abstractions in the type language. It leads to fairly natural programs. We further explore an alternative API for the motivating example and show that it also requires facilities for generic programming. Finally, we assess the properties that we achieve through typing and refine them by using rank-2 types.

Throughout the paper, we assume familiarity with the functional programming language Haskell [8].

2 WASH

WASH (Web Application Services in Haskell) is a collection of domain specific languages for designing and implementing Web services. Each language is embedded in Haskell and is available in form of a Haskell library. The idea is that the languages may be mixed and matched according to the needs of the application under development.

Two important parts of WASH are WASH/HTML and WASH/CGI. The first part, WASH/HTML, [16] deals with the generation of HTML and XML documents whereas WASH/CGI [17] provides a convenient API for server-side Web scripting. The distinctive feature of WASH/HTML is the fact that document generators programmed in WASH/HTML are guaranteed to only generate well-formed and to a large extent also valid HTML pages, subject to the condition that the generator program itself is type correct. WASH/CGI provides a simple callback-based programming model for server-side Web scripting by hiding most of the tedious details of the communication between browser and server. It also employs the type system to guarantee consistency between forms submitted by a browser and the program that processes these forms.

3 Type Checking HTML Generators

In this section, we consider the problem of guaranteeing that a program generates correct XHTML just by type checking the program. After discussing the design of the released version of WASH/HTML, we consider a more easily adaptable solution which relies heavily on computation taking place in the type checker at compile time.

Most Web pages are generated on the fly by scripts running on web servers. It is a widely recognized problem that these pages often violate the W3C standard for XHTML [19]. The standard requires that documents are well-formed and valid with respect to the XHTML document type definition (DTD). In a well-formed document, opening and closing tags are properly nested so that that the document is really a flat rendition of a labeled and attributed tree. The main kind of node in such a tree is called an *element* and the label of an element provides its name, which is reflected in the opening and closing tag. Validity states that the restrictions imposed by the DTD are met. The DTD basically associates with each element name a regular expression over element names. A well-formed document is valid if, for each element named N, the sequence of names of the subelements of N is accepted by the regular expression associated to N by the DTD.

For example, the DTD for XHTML states that an element named `dl` may only have a non-empty sequence of elements named either `dt` or `dd` as subelements.

```
<!ELEMENT dl (dt|dd)+>
```

3.1 Guaranteeing Well-formedness

The main reason why many generated Web pages are not even well-formed is the use of an inadequate programming interface for their generation. Quite often, pages are generated by sequences of print statements. Hence, a programmer easily looses track of the stack of open tags. In consequence, he might close elements that are either not open at all or obscured by other elements that were opened later on, but which are not yet closed.

Our approach to avoid these problems is to build an internal tree representation of the document first. Only after the entire document tree has been constructed, a single traversal of the document serializes it into XHTML syntax. Hence, the generator library must provide for operations that build this internal tree.

WASH/HTML provides for each element a combinator (a Haskell function) that has the same name as the element. The combinator does not simply build the new element. Rather, it *transforms* an existing (parent) element by attaching the new element at the end of its sequence of subelements. Likewise, it does not just accept a sequence of subelements, but rather it takes as its argument a *transformer function* that may add subelements and attributes.

As an example, we show a preliminary typing for an element combinator.

```
dt :: (Element -> Element) -> (Element -> Element)
```

The argument function transforms the `dt` element by attaching subelements and attributes whereas the resulting function is the transformer that adds the `dt` element to a parent.

This approach is attractive for a number of reasons. First, all kinds of nodes, elements and attributes, can be treated uniformly. Second, we obtain a notion of "sequence of elements and attributes" for free by composing the transformation functions.

3.2 Guaranteeing Validity

To guarantee validity, we propose to enhance the typing for element combinators. Since we need to keep track of element names for checking validity, the first step is to make the element names available at the type level. To this end, the library provides, for each element name, a data type of the same name (subject to name mangling to work around syntactical restrictions imposed by Haskell). For example, the type for the `dt` tag is `DT` and its definition is as follows.

```
data DT = DT
```

This data type has only one observable value, namely `DT`.[3] Hence, we will often call these types singleton types, as such types are called in type theory.

The idea is now to parameterize the `Element` type by a phantom type that ranges over the singleton types. That yields another potential typing for the `dt` combinator

```
dt :: (Element DT     -> Element DT)
   -> (Element parent -> Element parent)
```

[3] The identifier `DT` is once used as the name of a type on the left side and once as the name of a data constant on the right side.

With this typing in place, two things remain to be done.

First, the type `parent` should no be arbitrary. Since the DTD allows `dt` elements *only* as subelements of `dl` elements, we might want to replace `parent` by `DL`. However, this approach does not work in general because there may be a number of different admissible parent elements. Fortunately, this kind of restriction can be modeled using a standard Haskell type class.

For each element name N, we introduce a type class that encompasses exactly the set of those element types that admit child elements named N.[4] For example, the `dt` element gives rise to the class

```
class AdmitChildDT parent where
  dt :: (Element DT -> Element DT)
     -> (Element parent -> Element parent)

instance AdmitChildDT DL
```

In this example, there is only one instance (*i.e.*, only one element name admits a `dt` element as its child) but in general there may be up to 50 element names admissible for parents.

It makes perfect sense to stop at this point and use this typing for an API with only a partial validity guarantee. This typing already rules out the majority of errors because many regular expressions in the XHTML DTD have the form $(N_1|\dots|N_k)^*$ which is exactly what the `AdmitChild` type classes achieve. Moreover, further strengthening the typing constraints makes the library significantly harder to use.

Second, the sequence of subelements must not be arbitrary, either. Shifting the focus of our attention back to the `dl` element, where the subelements must form a *non-empty* sequence of `dd` and `dt` elements, the typing must prevent the attempt to add an empty `dl` element into its parent.

In the library, as it is implemented [16], we take the following approach. For each XHTML element name, we construct a finite automaton by compiling its associated regular expression from the DTD using standard techniques [1]. The `Element` type receives another phantom type parameter that ranges over the states of such a finite automaton. The type of the `dt` combinator is thus refined to

```
class AdmitChildDT parent where
  dt :: (FinalStateDT state, DeltaDT parent pstate pstate')
     => (Element DT     STATE0 -> Element DT     state)
     -> (Element parent pstate -> Element parent pstate')
```

In that type, `FinalStateDT` is a Haskell type class that characterizes the set of final states of the automaton associated with `DT`. However, `DeltaDT parent pstate pstate'`, which implements the transition function of the automaton,

[4] This simplifie an earlier version of WASH/HTML that required a two-parameter type class for this purpose[16].

is an application of a *three parameter type class*, a widely implemented Haskell extension, where `parent` is the name of the parent element, `pstate` is the state of the parent element's automaton before reading `DT`, and `pstate'` is the state of that automaton after reading `DT`. The type `STATE0` stands for the initial state of the automaton associated with the element `dt`. To avoid problems with ambiguity, it is advantageous to make use of another Haskell extension, functional dependencies [11]. Briefly put, declaring the functional dependency `parent pstate -> pstate'` for `DeltaDT` allows the type checker to treat `DeltaDT` like a function that maps `parent` and `pstate` into `pstate'`.

While this approach is viable, it exhibits a number of drawbacks. In particular, once a DTD is compiled to a library of `AdmitChild` classes and transition functions, it is very hard to extend the library on the fly. Hence, the solution is not very flexible. Another point is the use of a relation like `DeltaDT` in a place where a functional formulation would be much easier to understand.

3.3 Validity with Functional Logic Overloading

Hence, we propose to encode the transition function using functional logic overloading [4]. The main idea of this framework is to allow the use of functions in types and have the type inference engine deal with the evaluation of these functions at compile time. Since the variables present in types during type inference are logical variables, this places the type inference engine in the setting of functional logic programming [6]. This, in turn, enables the use of well-known and well-established implementation techniques for functional-logic programming languages [7] in the implementation of the type checker.

Using this approach, we proceed as follows. Instead of precompiling the regular expressions to finite automata and hardcoding them in the type structure, we take up the idea of derivatives of regular expressions [2] to avoid the explicit construction of a finite automaton.

Brzozowski [2] shows that starting from a regular expression r and an input symbol a it is possible to compute another regular expression $d(r, a)$ (the a-derivative of r) such that $L(d(r, a)) = a \setminus L(r) = \{w \mid aw \in L(r)\}$. Iterating this construction yields a finite set of regular expressions that is closed under taking derivatives. This set can be regarded as the set of states of a finite automaton with d as its transition function. The final states are those regular expressions that recognize the empty word.

It is a straightforward exercise to code the function d and a function e for checking if a regular expression recognizes the empty word in Haskell. However, it is a quite involved task to lift this implementation into type-level functions, in particular with just multi-parameter type classes and functional dependencies at hand [14]. For example, a simple two-line definition of an `equals` function for input symbols has a surprisingly awkward transcription on the type level. With functional logic overloading, however, the code need not be rewritten but the two-line definition can be reused on the type level.

The execution engine guarantees that the semantics of the code at the type level is the same as the semantics on the value level.

As an example, we look at the respective version of the typing of `dl` which involves the regular expression `(dt|dd)+`. The latter is encoded as `(PLUS (UNION (ATOM DT) (ATOM DD)))` where `PLUS` is a type constructor corresponding to the `+` operator, `UNION` corresponds to the `|` operator, and `ATOM` maps an element of the alphabet to a regular expression.

```
class AdmitChildDL parent where
  dl :: (AcceptsEmptyWord state)
     => (Element DL (PLUS (UNION (ATOM DT) (ATOM DD)))
                -> Element DL state)
     -> (Element parent pstate
                -> Element parent (Delta pstate DL))
```

Due to the shift from a relational point of view to a functional one, the spurious `pstate'` variable disappears and the functionality of `Delta` is immediately obvious. Furthermore, `Delta` does not depend on `parent` anymore because the regular expression `pstate` contains all the necessary information. [5]

4 Type Checking Web Scripts

The developer of a Web application is faced with a number of problems. Due to the stateless nature of the underlying HTTP protocol, the application often has to be split into many scripts, where each script corresponds to one or more states in the interaction with the user. Hence, the developer must ensure that there are no dangling interactions, which means that whenever the user submits an answer (a so-called form) to the Web server, then there is a script installed on the server which is ready to process this answer. This property is quite fragile and it can be affected by accidental renaming or deletion of a script or by improper installation. In addition, the submitted form must at least contain the information that the script wishes to process, *i.e.*, if a script asks for a particular field in the form, then this field should be present.

The WASH/CGI library ensures these two requirements. First, the entire interaction can be programmed as a single program and it is also installed as such. The WASH/CGI library automatically keeps track of the state of the interaction without requiring attention from the programmer. Instead, the programmer just attaches callback actions to the submit-buttons of a form to specify the flow of control inside the application. Thus, the problem of dangling interactions is completely avoided. Second, the fields of a form are never referred to by name in a WASH/CGI script. Instead, whenever a WASH/CGI script creates an input field it obtains a *typed handle* for the field. Initially, the handle is *invalid*, *i.e.*, it is not possible to extract its value. Then the

[5] To be fair, this dependency can be removed in the relational case, too, by incorporating the name of the parent into the state.

handle is passed through the submission mechanism to the callback actions. The submission mechanism typechecks the handle, rejects improper or otherwise malformed form submissions, and has the user repeat the submission if an error was detected. After checking the handle, the submission mechanism passes a *validated handle* to the callback action. The callback action can now extract the handle's value in a fully type safe way. Of course, many handles can be processed at once in this manner.

In the following, we take a deeper look at the mechanism to validate submissions. First, we discuss the standard Haskell98 solution as it is implemented in WASH/CGI. Second, we show an alternative solution that improves upon the Haskell98 solution by avoiding some of its inconvenience. Third, we consider further possible typings for the submission functions and discuss their merits. Finally, we assess the entire validation mechanism.

4.1 Submission Validation Using Haskell98

The creation of a new widget (*e.g.*, a text input field, a button, or a selection box) always returns a handle to the value of that widget. Initially, the handle is invalid and its value cannot be accessed. For example, the function that creates a text input field has the typing

```
textInputField :: HTMLField (InputField String INVALID)
```

where `InputField String INVALID` is the type of the widget's handle. The phantom type `INVALID` indicates that the handle is not yet validated and the function that extracts the value from a handle cannot be applied to the handle: [6]

```
value :: InputField a VALID -> a
```

`INVALID` (as well as `VALID`) is another instance of a singleton type.

The validation of an input handle takes place during the submission of a form. This is reflected in the type of the submission action:

```
submit :: InputHandle h
       => h INVALID -> (h VALID -> CGI ()) -> HTMLField ()
```

The function `submit` is overloaded over the type of a widget. It takes an invalid widget of type `h INVALID` as its first argument and a callback action of type `h VALID -> CGI ()` as its second argument to produce an `HTMLField ()`. The callback action is parameterized over a validated widget of type `h VALID`. This change in the type of the widget requires that the overloading cannot simply be performed over the type of the widget, but rather over the type constructor that abstracts the widget type over its `VALID`/`INVALID` type argument.

[6] The typing of `value` is simplified. In the implementation, `value` is overloaded to work with different types of handles, not just with `InputField`.

Unfortunately, this arrangement has a catch. Suppose that a form contains two input widgets and that the callback action must process both of them. In this situation, the two handles for the widgets might both have type

```
InputField String INVALID
```

While it is easy to pass either one of the handles to the callback, it is not straightforward to pass both. Simply pairing them yields a value of type

```
(InputField String INVALID, InputField String INVALID)
```

but this type is not of the form `h INVALID`. Of course, the same problem reappears with arbitrary tuples of handles, as well as with lists of handles. In either case, the resulting type has not the form required by the type of `submit`.

It turns out that a few newly introduced data types save the day. For each data constructor that is to be used with handles, a *lifted version* must be defined. For example

```
data F2 a b x = F2 (a x) (b x)
```

defines the datatype `F2` of lifted pairs, whereas

```
data FL a x = FL [a x]
```

defines the datatype `FL` of lifted lists.

Applying the lifted pairing constructor to the two input handles yields a value of type

```
F2 (InputField String) (InputField String) INVALID
```

which indeed has the form `h INVALID` (where `h` is `F2 (InputField String) (InputField String)`) and is thus a suitable first argument to `submit`. To make `F2` and `FL` acceptable input handles they need to become members of class `InputHandle`:

```
instance (InputHandle h)
        => InputHandle (FL h) where ...
instance (InputHandle h1, InputHandle h2)
        => InputHandle (F2 h1 h2) where ...
```

Last, but not least, in case we want to submit no data (the notorious continue button), that “nothing” needs to be lifted, too:

```
data F0 x = F0
```

Clearly, `F0 INVALID` has the required form `h INVALID` and can be made an instance of`InputHandle`:

```
instance InputHandle F0 where ...
```

4.2 Alternative Approach Using Higher-Order Unification

The problem with the Haskell98 solution outlined in the last subsection is precisely the use of spurious data constructors like `F2` and `FL`. It is quite hard to explain to non-expert Haskell programmers that they cannot simply build an arbitrary data structure containing handles. In fact, `F2` and `FL` are really artifacts caused by the weakness of the type system and it would be more natural to just use the standard pair and list type constructors.

For that reason, we have sought for ways to get rid off these artifacts. Getting back to the type of a pair of input handles

```
(InputField String INVALID, InputField String INVALID)
```

we see that this type can be unified with `h INVALID` by considering it as a higher-order unification problem. That is, instead of limiting ourselves to substituting constructor terms during unification, we also allow certain lambda terms. In the example, the substitution that maps `h` to `\x -> (InputField String x, InputField String x)` would do the job.

Since it is well-known that higher-order unification is an undecidable problem and moreover one with a high degree of non-determinism [3], the introduction of full-blown higher-order unification into a type inferencer is not advisable. In particular, an analysis shows [13] that examples can be constructed where Huet's higher-order unification procedure yields two different solutions, but where only one solution makes sense in such an example. Unfortunately, the standard procedure cannot be extended to pick the acceptable solution automatically because this information cannot be derived from the unification problem without further information. For that reason, we have restricted higher-order unification to *guided higher-order unification* [13], where the set of lambda terms is restricted and where the substitution selected by the unification procedure is guided by a set of substitutions supplied separately.

In particular, substitutions generated by Huet's higher-order unification procedure are all *general substitutions* of the form

$$f \mapsto \lambda \overline{x}_m . c \; (h_1 \; \overline{x}_m) \; \ldots \; (h_n \; \overline{x}_m)$$

where f is the free variable substituted for, $h_1, \ldots, h_n$ are newly introduced free variables, and c is a constant or a bound variable (*i.e.*, one of the $\overline{x}_m$). These substitutions are generated for equations of kind *flex-rigid*, where the head symbol on one side is a free variable, f, and the head symbol on the other side is a constant (*e.g.*, c).

In guided higher-order unification, we are not using general substitutions, but rather rely on substitutions supplied by the programmer. Since we are only interested in applying them in the presence of overloading, we tie the specification of the substitution to the instance declaration for a particular type class.

For example, in the case of the `InputHandle` class considered in the context of the `submit` function above, we supply the following instance declaration

```
instance (InputHandle h1, InputHandle h2)
      => InputHandle (\x -> (h1 x, h2 x))
```

The effect of this declaration is the following: When attempting to unify

```
(InputHandle h) =>
h INVALID =?= (InputField String INVALID,
               InputField String INVALID)
```

and we know that `InputHandle h` must hold (the typical situation while type checking an application of `submit` to a pair of handles), then the type checker applies the substitution `\x -> (h1 x, h2 x)` to obtain

```
(InputHandle h1, InputHandle h2) =>
(h1 INVALID, h2 INVALID) =?= (InputField String INVALID,
                              InputField String INVALID)
```

which is then easily reduced (by standard decomposition steps as in first-order unification) to the solution

```
h1 =?= InputField String
h2 =?= InputField String
```

which satisfies all constraints.

4.3 Alternative API for Submission Validation

Let's step back and consider if the API for validated form submission cannot be changed in ways that simplify the problems for the implementor and the user of the library. For once, it is not clear if the callback actions have to see the widget handlers. Instead, the `submit` function might extract the values from the handlers after validation and just pass them directly to the callback. (We simplify somewhat by only looking at one type of handle, `Hdl a`.)

```
submitHandle :: Hdl a -> (a -> CGI ()) -> HTMLField ()
```

While this typing works well in the case where there is only one handle, the problems with pairs and lists of handles aggravate. Let's review the types for `submit` for these two cases and the case where there is no handle at all (a "continue" button):

```
submitVoid :: ()             -> (()     -> CGI ()) -> HTMLField ()
submitPair :: (Hdl a, Hdl b) -> ((a, b) -> CGI ()) -> HTMLField ()
submitList :: [Hdl a]        -> ([a]    -> CGI ()) -> HTMLField ()
```

Comparing the three types, it looks like the type of the argument to the callback action should be a function of the type of the handle. This function `SH` just strips away the handles. The resulting type is

```
submitGeneric :: h -> (SH h -> CGI ()) -> HTMLField ()
```

where `SH` must at least satisfy the equations

```
SH (Hdl a) = a
```

```
SH ()          = ()
SH (h1, h2)    = (SH h1, SH h2)
SH [h1]        = [SH h1]
```

More concisely, we might define

$$
\begin{aligned}
\texttt{SH}\ (H\ t) &= t \\
\texttt{SH}\ (C) &= C \\
\texttt{SH}\ (f\ t) &= (\texttt{SH}\ f)\ (\texttt{SH}\ t)
\end{aligned}
$$

where H stands for `Hdl`, C for a constant (of arbitrary kind), and f and t are constructor variables. Furthermore, we restrict C to range only over polynomial data constructors (that contain no function types). Not only does this restriction simplify the development, but it is also required by the application. The point is that the validation function should perform its duty at the place where it is invoked. If the data structure contained functions that processed or returned handles, then the validation function could at best coerce these functions into functions that perform the required checks later. However, this potential delay is against the intent of the validation.

It remains to determine the transformation $\texttt{sh}\langle t :: *\rangle$ that `SH` induces on values of type t. For simplicity, we first construct a function of type `h -> SH h` and then consider the actual validation function. Fortunately, the recursive definition of `SH` on the type level almost follows the pattern for polytypic programming defined by Hinze [9]. Hence, we just need to define a function

```
unHdl :: Hdl a -> a
```

then set

$$\texttt{sh}\langle H\ t :: *\rangle = \texttt{unHdl}$$

and for the remaining cases fall back to the definition of a generic identity function. That is

$$
\begin{aligned}
\texttt{sh}\langle () :: *\rangle &= \lambda x.x \\
\texttt{sh}\langle \texttt{Int} :: *\rangle &= \lambda x.x \\
&\quad \text{and so on for the remaining base types} \\
\texttt{sh}\langle (t_1, t_2) :: *\rangle &= \lambda (x_1, x_2).(\texttt{sh}\langle t_1 :: *\rangle x_1, \texttt{sh}\langle t_2 :: *\rangle x_2) \\
\texttt{sh}\langle t_1 + t_2 :: *\rangle &= \lambda x.\texttt{case}\ x\ \texttt{of}\ \texttt{Inl}\ x_1 \to \texttt{Inl}\ (\texttt{sh}\langle t_1 :: *\rangle x_1) \\
&\qquad\qquad\qquad\quad \texttt{Inl}\ x_2 \to \texttt{Inl}\ (\texttt{sh}\langle t_2 :: *\rangle x_2)
\end{aligned}
$$

In reality, instead of `unHdl` there is a function `validate` that has typing

```
validate :: Hdl a -> Either [String] a
```

That is, it either returns the value itself or an error message that takes the form of a list of strings. Clearly, `Either [String]` is a monad and so we abstract from the particulars of the error handling mechanism by assuming that it can be expressed using a monad. Hence, it remains to define a function `processHdl` that propagates a function of type

```
process :: Monad m => Hdl a -> m a
```

through a value that might contain `Hdl`s. Here is the generic definition of `ph` (process handles):

$$
\begin{array}{lll}
\mathtt{ph} & :: & \forall h, m.\mathtt{Monad}\ m \Rightarrow h \rightarrow m\ (\mathtt{SH}\ h) \\
\mathtt{ph}\langle H\ t :: *\rangle & = & \mathtt{process} \\
\mathtt{ph}\langle () :: *\rangle & = & \mathtt{return} \\
\mathtt{ph}\langle \mathtt{Int} :: *\rangle & = & \mathtt{return} \\
 & & \text{and so on for the remaining base types} \\
\mathtt{ph}\langle (t_1, t_2) :: *\rangle & = & \lambda(x_1, x_2).\,\mathtt{do}\ y_1 \leftarrow \mathtt{ph}\langle t_1 :: *\rangle x_1 \\
 & & \qquad\qquad\quad\ y_2 \leftarrow \mathtt{ph}\langle t_2 :: *\rangle x_2 \\
 & & \qquad\qquad\quad\ \mathtt{return}\ (y_1, y_2) \\
\mathtt{ph}\langle t_1 + t_2 :: *\rangle & = & \lambda x.\mathtt{case}\ x\ \mathtt{of}\ \mathtt{Inl}\ x_1 \rightarrow\ \mathtt{do}\ y_1 \leftarrow \mathtt{ph}\langle t_1 :: *\rangle x_1 \\
 & & \qquad\qquad\qquad\qquad\qquad\quad\ \mathtt{return}\ (\mathtt{Inl}\ y_1) \\
 & & \qquad\qquad\qquad\ \ \mathtt{Inl}\ x_2 \rightarrow\ \mathtt{do}\ y_2 \leftarrow \mathtt{ph}\langle t_2 :: *\rangle x_2 \\
 & & \qquad\qquad\qquad\qquad\qquad\quad\ \mathtt{return}\ (\mathtt{Inr}\ y_2)
\end{array}
$$

The beauty of Hinze's approach is that such a definition extends automatically to all kinds without further programming. Unfortunately, as the theory is presented [9], it does not allow the use of overloading in the base case, which is required to make our definition work.

Alternatively, a version limited to first-order kinds can be readily implemented using functional logic overloading [4]. However, this task is considerably more tedious because we must specialize the cases for sum and product type manually for each data type that will be wrapped around handles.

4.4 Restricting the Scope of Widget Handles

What is the return for our investment into typing the submission primitive? Of course, we gain a lot of programming convenience: input widgets do not have to be given explicit names, problems with inconsistencies due to widget's names disappear, the communication between the browser and the script is typed, and all errors due to invalid entries into the input fields are caught by the system without programmer intervention.

However, one pressing concern remains. It is caused by a mismatch between the lexical scope of a widget handle and the handle's actual lifetime. A handle's actual lifetime starts with the creation of the handle in the context of

```
  ask (
    ...
    nameF  <- textInputField (fieldSIZE 40)
    ...
    submit F0 (getMoreInput nameF) empty)

getMoreInput nameF F0 =
  ask (
    ...
    submit nameF action empty)
```

Fig. 1. Skeleton of erroneous WASH/CGI script

a particular Web page and it ends before any callback action associated with the Web page starts to construct the response page. It corresponds to one interaction cycle: delivering a form to the browser and processing the answer up to constructing the next form. These checkpoints can be easily pinpointed in the program by looking for the combinator `ask`, which starts the construction of a new form. Thus, the actual lifetime also corresponds to a part of the program text.

To understand the importance of the actual lifetime, we need a bit of information about the implementation of widgets. For any input field in a form, its widget constructor is executed multiple times in at least two different modes. Initially, the constructor function builds the internal representation of the widget, an HTML element. When the form's input data is submitted, the same constructor function executes again, but this time it picks up the input corresponding to the widget and stores it into the handle. At the same time, it builds its part of an error page, which is only used in case the submission cannot be validated.

Next, a `submit` function is activated and it tries to validate the handles passed to it. If this process is successful, it passes control to its callback action. However, if one of the handles cannot be validated, it sends the error page containing the last form back to the browser so that the user can reenter the data. For convenience, in this form the widgets that did not validate are marked visually and the other widgets are initialized to the values previously entered.

Now, the problem is as follows. Suppose an action keeps an invalid handle in a free variable and passes it to the `submit` function in a subsequent interaction. If the entry validates now, then there is no problem and the program continues as intended. Otherwise, `submit` redisplays the last form. However, this form is not the one with the erroneous entry and so the error cannot be corrected (except through using the back button and some guesswork).

Figure 1 gives an example of this behavior. The first Web page (first five lines) has an input field for text, which is associated with the handle `nameF`. The page contains a submit button that does not validate anything (indicated by `F0`). However, the callback action `getMoreInput nameF` contains a reference to the (invalid) handle `nameF`. The action `getMoreInput` produces a new Web page and submits the `nameF` field through a button on this page. In the extreme case, the second page has no input fields, but just presents some text (for example, terms and conditions) and an "accept" button. A user that types an invalid name on the first page will only get an error message after clicking the "accept" button on the second page (and will then see the terms and conditions, again). Certainly, a puzzling situation.

The good news is that we can tackle this programming error with the type system, as well. Our solution relies on the use of a rank-2 type. It is inspired by Launchbury and Peyton Jones's solution to the problem of local state [12]. The trick is to index the underlying `CGI` monad with a phantom type variable. This phantom type variable also appears (as an extra type parameter) in the type of the input handles. The creation of an input widget has the following revised typing

```
textInputField :: HTMLField x (InputField String x INVALID)
```

where

```
type HTMLField x a = WithHTML (CGI x) () -> WithHTML (CGI x) a
```

and consequently, the submission function's type is revised to only accept input handles that match the type index of the currently executing thread of the `CGI` monad:

```
submit :: InputHandle h
       => h x INVALID
       -> (h x VALID -> forall y . CGI y ())
       -> HTMLField x ()
```

Actually, the `x` parameter in `h x VALID` does not matter since the `value` function is polymorphic in this parameter:

```
value :: InputField a x VALID -> a
```

Also `F0`, `F2`, and `FL` need be revised to take yet another type parameter that they just pass on to their arguments:

```
data F0 x y = F0
data F2 a b x y = F2 (a x y) (b x y)
data FL a x y = FL [a x y]
```

This approach effectively rules out the scope mismatch error. For example, the function call `submit F0 (getMoreInput nameF) empty` in the program in Fig. 1 does not typecheck. Of course, the offending part is the use of `nameF` in the callback argument to `submit`. Here is what happens starting with the assumption that `nameF :: InputField String x INVALID`:

- `submit nameF accept empty` has type `WithHTML (CGI x) ()`
- `F0` is used at type `F0 x INVALID`

- `getMoreInput nameF` has type `forall y z . FO y z -> CGI x ()` (note that `x` *cannot* be universally quantified because `nameF` occurs in the expression so that the assumption on `nameF` and hence the type variable `x` must occur in the environment)
- but `getMoreInput nameF` must assume type `forall y . FO y VALID -> CGI y ()`
- Since the expected type is *not* a generic instance of the first one, type inference fails at this point. The reported error is that the type variable `x` cannot be generalized.

Finally, it turns out that there are situations where this typing is too restrictive. For example, consider a Web page with several different input fields, where a set of buttons (or a selection box) determines which other input fields are considered. That is, the widget handles that must be validated depend on the value of one or more input fields. Or they might even be selected randomly.

The obvious extension of the current programming model requires us to first validate and submit the selector fields, and then somehow validate the set of fields required by the particular choice. This could be done by a submission function with the following typing

```
submitx :: InputHandle h
        => h x INVALID
        -> (h x VALID
           -> (forall h' . InputHandle h'
                        => h' x INVALID
                        -> Either [String] (h' x VALID))
           -> (forall y  . Either [String] (CGI y ())))
        -> HTMLField x ()
```

The idea is that the parameter function takes a validated input handle and uses its value to select further input handles for validation. The second parameter is the validation function. Its type is polymorphic in the kind and type of the input handle to process, but at the same type specialized to handles created in the thread indexed with `x`. Since this validation might fail, the function does not directly return the action but wraps it into an error monad `Either [String]`, again.

5 Related Work

Thiemann [16] uses multi-parameter type classes with functional dependencies to generate correct HTML. We have simplified the basic approach that employed a two-parameter type class `AddTo` to relate an element to its parents to a pure Haskell98 approach relying on a number of specialized one-parameter type classes (`AdmitChild...`). We have re-expressed the multi-parameter classes that implemented the transition function using functional logic overloading [4].

WASH/CGI [17] provided motivation for the exploration of the `submit` function. The present paper extends that work by introducing a number of typings for `submit` that guarantee that there is no mismatch between the data posted by the browser and the data that the server-side script expects.

On the technical side, Sulzmann and others [5] have introduced a programmable type system that employs constraint handling rules. That system can also express most of the features supported by functional logic overloading, but it relies on constraint logic programming as its foundation.

Work on generic programming (for example, Hinze's approach to generic functional programming [9]) also relies heavily on powerful means to transform types. We have demonstrated that a fairly small extension to Hinze's framework is also amenable to express the type of a (suitably altered) `submit` function. In fact, since functions in his framework can be smoothly integrated with Haskell98 [10], we do not consider the required extension as a fundamental problem.

6 Conclusion

The development of the WASH family of DSLs has lead to a number of interesting typing problems. We have developed solutions for these typing problems, sometimes by encoding them into Haskell98, but also by relying on extensions of the language, like functional logic overloading, anonymous type functions, rank-2 polymorphism, and Hinze's framework for generic programming. The presented solutions guarantee high-level properties for WASH scripts: the correctness of generated HTML documents and the consistency between data submitted by the browser and the data expected by the Web browser.

Acknowledgement
Thanks to Matthias Neubauer for his careful reading of a draft of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[3] G. Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 16, pages 1009–1062. North-Holland, 2001.

[4] M. Gasbichler, M. Neubauer, M. Sperber, and P. Thiemann. Functional logic overloading. In J. Mitchell, editor, *Proc. 29th Annual ACM Symposium on Principles of Programming Languages*, pages 233–244, Portland, OR, USA, Jan. 2002. ACM Press.

[5] K. Glynn, P. Stuckey, and M. Sulzmann. Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming*, July 2000.

[6] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20):583–628, 1994.

[7] M. Hanus. A unified computation model for functional and logic programming. In N. D. Jones, editor, *Proc. 24th Annual ACM Symposium on Principles of Programming Languages*, pages 80–93, Paris, France, Jan. 1997. ACM Press.

[8] Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, Dec. 1998.

[9] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, June 2002.

[10] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *Electronic Notes in Theoretical Computer Science*, Montreal, Canada, Sept. 2001. Elsevier Science.

[11] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 230–244, Berlin, Germany, Mar. 2000. Springer-Verlag.

[12] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[13] M. Neubauer and P. Thiemann. Type classes with more higher-order polymorphism. In S. Peyton-Jones, editor, *Proc. International Conference on Functional Programming 2002*, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

[14] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In R. Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, 2001. to appear.

[15] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

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

[17] P. Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages, Proceedings*

of the Fourth International Workshop, PADL'02, number 2257 in Lecture Notes in Computer Science, pages 192–208, Portland, OR, USA, Jan. 2002. Springer-Verlag.

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

[19] XHTML 1.0: The extensible hypertext markup language. `http://www.w3.org/TR/xhtml1`, Jan. 2000.