WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms

Peter Thiemann

Universität Freiburg, Georges-Köhler-Allee 079, D-79085 Freiburg, Germany
Phone: +49 761 203 8051, Fax: +49 761 203 8052
Email: `thiemann@informatik.uni-freiburg.de`

Abstract. The common gateway interface (CGI) is one of the prevalent methods to provide dynamic contents on the Web. Since it is cumbersome to use in its raw form, there are many libraries that make CGI programming easier.
WASH/CGI is a Haskell library for server-side Web scripting. Its implementation relies on CGI, but its use avoids most of CGI's drawbacks. It incorporates the concept of a session, provides a typed, compositional approach to constructing interaction elements (forms), and relies on callbacks to specify control flow. From a programmer's perspective, programming WASH/CGI is like programming a graphical user interface (GUI), where the layout is specified using HTML via a novel monadic interface.

Key words: Haskell, Monads, CGI Programming

1 Introduction

The common gateway interface (CGI) is one of the oldest methods for deploying dynamic Web pages based on server-side computations. As such, CGI has a number of advantages. Virtually every Web server supports CGI. CGI requires no special functionality from the browser, apart from the standard support for HTML forms. And, on the programming side, CGI communicates via standard input/output streams and environment variables so that CGI is not tied to a particular architecture or implementation language. Hence, CGI is the most portable approach to providing dynamic contents on the Web.

The basic idea of CGI is straightforward. Whenever the Web server receives a request for a CGI-enabled URL, it treats the local file determined by the URL as an executable program, a *CGI script*, and starts it in a new process. Such a script receives its input through the standard input stream and through environment variables and delivers the response to its standard output stream as defined by the CGI standard [1].

CGI programming in raw form is fairly error-prone. A common source of errors lies in the parameter passing scheme between forms and CGI scripts. A form is an HTML element that contains named input elements. Each input element implements one particular input widget. When a special submit button is pressed, the browser sends a list of pairs of input element names and their string values to the server. This message in its raw form is passed to a CGI script, which must extract the values by name. Unfortunately, there is no guarantee that the form uses the names expected by the script and vice versa.

In particular, since parameter passing between forms and CGI scripts is string-based, it is completely untyped. It is not possible to specify the expected type of an input field. Of course, it is possible to check the input value upon receipt but this requires extra programming.

Another painful limitation of CGI is due to the statelessness of the underlying protocol, HTTP. Every single request starts a new CGI process, which produces a response page and terminates. There is no concept of a *session*, *i.e.*, a sequence of interactions between browser and server, and more importantly of *session persistence*. Session persistence means that the value of a variable is available throughout a session. Some applications even require global persistence where the lifetime of a variable is not tied to any session. Usually, CGI programmers build support for sessions and persistence from scratch. They distribute the units of a session over a number of CGI scripts and link them manually. To provide a notion of session persistence, they put extra information in their response pages (hidden input fields or cookies) or they maintain a session identification using URL rewriting. Clearly, all those solutions are tedious and error-prone.

The present work provides a solution to all the issues mentioned above. Our WASH/CGI library makes CGI programming simple and intuitive. It is implemented in Haskell [2] and includes the following features:

– a callback style of programming user interaction;
– input fields which are first-class entities; they can be typed and grouped to compound input fields (compositionality);
– the specification of an input widget and the collection of the input data are tied together so that mismatches are impossible;
– support for sessions and persistence.

The library is available through the WASH web page[1]. The web page also contains some live examples with sources. Beyond the topics discussed in this paper, WASH/CGI provides first-class clickable images where each pixel can be bound to a different action, as well as a simple implementation of global persistence.

Familiarity with the Haskell language [2] as well as with the essential HTML elements is assumed throughout.

Overview We follow the structure of the library in a bottom-up manner to give the reader a feel for the awkwardness of raw CGI programming, which is discussed in Sec. 2. The worst edges are already smoothed by using functional wrappers for input and output. In Section 3, we consider a structured way of specifying HTML documents, so that we can describe more complex output. Section 4 explains the concept of a session in WASH/CGI and discusses its implementation. Section 5 explains the construction of HTML documents with

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

active elements, forms and input fields. The issue of persistent store is considered in Sec. 6. Finally, Section 7 explains an extended example of using the library, followed by a section on related work (Sec. 8), and a conclusion with some pointers for further work.

2 Raw CGI Programming

A CGI script receives its input through the standard input stream and through environment variables. The input can be divided into meta-information and the actual data. Meta-information includes the URL of the script, the protocol, the name and address of the requesting host, and so on. The data arrives (conceptually) as a list of name-value pairs. The actual encoding is not important for our purposes and many approaches to Web programming contain facilities to access these inputs conveniently. For example, Meijer's CGI library [3] provides functionality similar to that explained in this section.

Our raw library `RawCGI` implements parameter passing through the function

```
start :: (CGIInfo -> CGIParameters -> IO a) -> IO a
```

where `CGIInfo` is a record containing the meta-information and `CGIParameters` is a list of (name-value) pairs of strings. Now, a CGI programmer just has to write a function of type `CGIInfo -> CGIParameters -> IO a` that processes the input and creates an output document and returns an `IO` action. The intention is that this action writes a CGI response to the standard output, but it can also do other things, *e.g.*, access a data base or communicate with other hosts.

The output of a CGI script also has to adhere to a special format, which is parsed by the web server. Since that format depends on the type of the output data, it is natural to implement this output format generically using a type class.

```
class CGIOutput a where
  cgiPut :: a -> IO ()
```

This declaration introduces an overloaded function `cgiPut` which can be instantiated differently for each type `a`. Currently, the library includes instances for HTML, strings, files, return status (to indicate errors), and relocation requests.

Here is a very simple CGI script written with this library.

```
main =
  start cgi
cgi info parms =
  case assocParm "test" parms of
    Nothing -> cgiPut "Parameter ‘test’ not provided"
    Just x  -> cgiPut ("Value of test = "++x)
```

It checks whether the parameter `test` was supplied on invocation of the script and generates an according message. The value of `assocParm "test" parms` is `Nothing` if there is no binding for the name `test`, and `Just x` if the value of `test` is the string `x`.

```
type Element   -- abstract
type Attribute -- abstract

element_  :: String -> [Attribute] -> [Element] -> Element
doctype_  :: [String] -> [Element] -> Element

attr_     :: String -> String -> Attribute

add_      :: Element -> Element   -> Element
add_attr_ :: Element -> Attribute -> Element
```

Fig. 1. Signature of `HTMLBase` (excerpt)

3 Generation of HTML

The next ingredient for CGI programming is a disciplined means to generate HTML. This task is split into a low-level ADT `HTMLBase` (Fig. 1), which provides the rudimentary functionality for constructing an internal representation of HTML pages and printing it, and a high-level module `HTMLMonad`.

In the low-level ADT, the function `element_` constructs a HTML element from a tag, a list of attributes, and a list of children. The function `doctype_` constructs the top-level element of a document. Given an attribute name and a value, `attr_` constructs an attribute. The expression `add_ e e'` adopts the element `e'` as a child of `e`, and `add_attr_` attaches an attribute to an element.

The interface in Fig. 1 is not intended for direct use but rather as a stepping stone to provide representation independence for the high-level interface. A key point of the high-level interface is its parameterization. Later on, it will be necessary to thread certain information through the construction of an HTML document. Since the particulars of this threading may vary, we parameterize the construction by a monad that manages the threading. Later, we will instantiate the parameter to the `IO` monad and to the `CGI` monad, a new monad which is discussed below.

Given that the parameter is a monad, we formulate HTML generation as a monad transformer `WithHTML` [4]. Hence, if `m` is a monad, then `WithHTML m` is a monad, too. In particular, it composes `m` with a state transformer that transforms HTML elements:

```
data WithHTML m a = WithHTML (Element -> m (a, Element))
```

For each HTML tag, the library provides a function of the same name, that constructs the correponding element. This function maps one `WithHTML` action into another and has a generic definition.

```
makeELEM :: Monad m => String -> WithHTML m a -> WithHTML m a
makeELEM tag (WithHTML g) =
  WithHTML (\elem ->
    g (element_ tag" [] []) >>= \(a, tableElem) ->
    return (a, add_ elem tableElem))
```

The argument action `WithHTML g` is applied to the newly created, empty `table` element. The intention is that this action adds children and attributes to the new element. Afterwards, it includes the new element in the list of children of the current element, `elem`, which is threaded through by the state transformer. The resulting value, `a`, is the value returned by `g` after constructing the children.

From this it is easy to define special instances like the `table` function.

```
table :: Monad m => WithHTML m a -> WithHTML m a
table = makeELEM "table"
```

Attributes are also attached to the current element. The corresponding function `attr` is straightforward.

```
attr :: Monad m => String -> String -> WithHTML m ()
attr a v =
  WithHTML (\ elem -> return ((), add_attr_ elem (attr_ a v)))
```

Due to the underlying language Haskell, parameterized documents and (higher-order) document templates can be implemented by functions. As an example for the elegant coding style supported by the library, we show the definition of a standard document template, which is parameterized over the title of the page, `ttl`, and over the actual contents, `elems`.

```
standardPage :: String -> WithHTML m a -> WithHTML m a
standardPage ttl elems =
  html (head (title (text ttl))
     ## body (h1    (text ttl)  ## elems))
```

Only two combinators must be explained. The function `text` constructs a textual node from a string. The operator `##` composes state transformations. Intuitively, `##` concatenes (lists of) nodes.

Using a monad for HTML generation has further advantages. For example, standard monadic operations and in particular the `do` notation can be used to construct HTML. The `##` operator behaves very much like the `>>` operator, but its typing is different. While `##` returns the result of its first parameter action, the `>>` operator returns the result of the second parameter:

```
(>>) :: Monad m => m a -> m b -> m b
(##) :: Monad m => m a -> m b -> m a
```

4 Sessions

A session is a sequence of interactions between the server and a browser that logically belong together. In raw CGI programming, sessions must be implemented by the programmer. For example, Figure 2 shows a typical session which guides a user through a couple of forms (`ask`), performs some I/O (`io`), and finally yields some document as an answer (`tell`). Usually, each of the blocks A, B, C, and D must be implemented as a separate program. In contrast, our library provides combinators `ask`, `tell`, and `io` in such a way that the entire interaction can be described in one program.

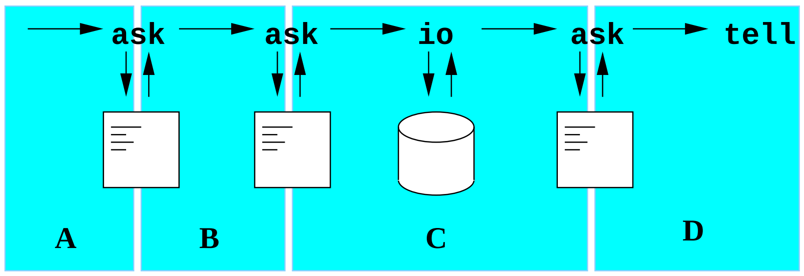

Fig. 2. A CGI session

Since we cannot change the CGI protocol, our program also has to terminate each time a user interaction is required: at each `ask`. However, `ask` saves the current state and arranges matters so that receipt of the user's response reloads the state and makes the program continue exactly where it left off.

CGI programmers have a number of alternatives for saving the state, and these are exactly the alternatives for implementing `ask`. The simplest approach is to include a so-called hidden input field in the form sent to the user and store the state in that field. Hidden input fields are not displayed by the browser, but are always transmitted on submittal of the form. Clearly, this approach is only feasible when the state is small. Once the state gets bigger, it should be stored in a local file on the server and only a pointer to it is put into a hidden field, a cookie, or in an extension of the script's URL. The present version of our library puts the whole state in a hidden field.

The next question is which part of the state we save: for efficiency reasons, it is not feasible to save the entire program state. In our approach, we save a log of the responses from the user and from I/O operations. These responses are the only factors that affect the flow of control in the program. Since we are in a pure language, all other values can be recomputed with equal outcomes. This approach proved sufficiently efficient for the applications that we considered.

Our implementation encapsulates this machinery in the monad `CGI`.

```
data CGI a = CGI { unCGI :: CGIState -> IO (Maybe a, CGIState) }
```

This definition declares the selector `unCGI` along with the data constructor `CGI`. It defines, again, a state monad composed with the `IO` monad. The `CGIState` contains two copies of the log mentioned. The first copy is used to provide responses to questions that have been answered in previous parts of the session. The second copy retains the whole log so that it can be sent to the browser. Both copies are used like queues: each input operation tries to get its result from the front of the first queue. Only if the queue is empty it performs the actual input. The second queue is only used to register new results, which are put at the end of the queue. Whenever the CGI program (re-) starts, both queues are initialized to the same restored state.

The `CGI` monad has the following actions.

```
tell :: CGIOutput a => a -> CGI ()
tell a = CGI (\cgistate -> cgiPut a >> exitWith ExitSuccess)
```

The action `tell` takes any value whose type is an instance of `CGIOutput`, prints it as a CGI response, and terminates the program.

```
io :: (Read a, Show a) => IO a -> CGI a
```

The combinator `io` implements the embedding of `IO` into `CGI`. If `ioa :: IO a` is an `IO` action returning values of type `a`, then `io ioa` is the corresponding `CGI` action. Its implementation is slightly complicated because `IO` actions can return different results each time they are run. Hence, an `IO` action must be executed at most once in a session and its value must be saved for later invocations of the script.

First, `io ioa` must check if the result of action `ioa` is already present in the first queue. In that case, it simply returns the logged answer and removes the front element. If the first queue is empty, `io ioa` executes the action `ioa`, logs the result (enqueues it in the second queue), and returns it. The result type of the operation must be an instance of the classes `Show` and `Read`, so that each result can be stored as a string (`Show`) and reconstructed from a string (`Read`)[2].

To understand the `ask` operation that sends a form to the browser and waits for the user's response, we first need to examine `cgistate` more closely. Beyond the queues explained above, there is a `pageInfo` field that contains information pertinent to the currently created HTML page. From this information and a document to display, the function in the `nextaction` field determines the `CGI` action that is executed by `ask`. The type of this function is `Element -> CGI ()`. There is also a function, `nextCGIState`, that removes the front element from the first queue and resets a few internal data structures (*e.g.*, `pageInfo`) that are required to implement forms and input fields.

```
ask :: WithHTML CGI a -> CGI ()
ask ma =
  do doc <- build_document ma
     CGI (\cgistate ->
       unCGI (nextaction (pageInfo cgistate) doc) (nextCGIState cgistate))
```

First, the element `ma` is wrapped into a document, `doc`, using `build_document`. Then, `ask` grabs the current `cgistate` and performs the `CGI` action `nextaction (pageInfo cgistate) doc` on the advanced CGI state `nextCGIState cgistate`.

The computation of `nextaction` is necessary because the action depends on a number of things which cannot be determined in advance.

– If there is no response in the log, then the constructed page must be sent to the user and the program must terminate.
– If there is a logged response, then the response is analyzed during the construction of the request page (which is not sent anymore). This analysis also registers the required action in the `pageInfo` field.

[2] More efficient means for storing and restoring are possible, but currently not implemented.

Now we see why the construction of HTML is parameterized over a monad, in this case over `CGI`. During the construction, the input parameters are analyzed and saved in the `pageInfo` field. The next logical step is to examine the construction of forms.

5 Forms

The constructors for active HTML components, like input fields and forms, have a more specific type than constructors for passive HTML elements.

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

The constructor for forms takes a collection of attributes and child elements and returns a `<form>` element. Each input field must have exactly one enclosing form element.

```
makeForm :: HTMLField ()
```

It is not necessary to set the standard attributes of the form element. The `action` attribute, which contains the URL for processing the content of the form, the `enctype` attribute, which determines the encoding of the content of the form, and the `method` attribute are all determined automatically by WASH/CGI.

Input elements specify the input widgets that appear in a form. The function

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

generates a simple textual input field whereas

```
inputField :: Read a => HTMLField (InputField a)
```

generates a text input field, which is restricted to values of type `a`. The resulting value of type `InputField a` is the *handle* for the input field. The handle contains the actual input value, which is accessible through the functions

```
value  :: InputField a -> Maybe a
string :: InputField a -> Maybe String
```

The `value` function extracts the parsed value (if there was a parsable input), whereas the `string` function is meant for error analysis and provides access to the raw input (if the input element was filled in at all).

Further input widgets are specified in the same manner (see Fig. 3). A `fileInputField` returns the contents of the chosen file as a string. A `resetField` just clears all input fields, it has no I/O functionality. Radio buttons and selection boxes have a slightly more complicated interface. They are omitted for brevity.

It remains to discuss the `submitField`. It takes a `CGI` action and generates a button in the HTML page. Clicking such a button executes its action. The action is similar to a continuation. Since a form may contain more than one submit button, multiple continuations are possible. In particular, a large form may be composed from small interaction groups that consist of input fields and one or more submit buttons.

For example, consider programming a simple login screen:

```
passwordInputField :: HTMLField (InputField String)
checkboxInputField :: HTMLField (InputField Bool)
fileInputField     :: HTMLField (InputField String)
resetField         :: HTMLField (InputField ())
submitField        :: CGI () -> HTMLField ()
```

Fig. 3. Input fields (excerpt)

```
<html><head><title>LOGIN</title></head>
<body><h1>LOGIN</h1>
<form enctype="application/x-www-form-urlencoded" name="f3" method="POST"
      action="http://localhost:80">
<table><tr><td>Enter your name </td>
<td><input size="10" name="f0" type="text"></td>
</tr>
<tr><td>Enter your password </td>
<td><input size="10" name="f1" type="text"></td>
</tr>
<input value="LOGIN" name="s2" type="submit">
</table>
<input value="%5B%5D" name="=CGI=parm=" type="hidden">
</form></body></html>
```

Fig. 4. Generated HTML form

```
login = ask $ standardPage "LOGIN" $ makeForm $ table $
  do nameF <- tr (td (text "Enter your name ") >>
                  td (textInputField (attr "size" "10")))
     passF <- tr (td (text "Enter your password ") >>
                  td (textInputField (attr "size" "10")))
     submitField (check nameF passF) (attr "value" "LOGIN")

check nameF passF =
  htell $ standardPage "LOGIN" $
    (text "You said " ## fromJust (value nameF) ##
     text " and " ## fromJust (value passF))
```

This program is self-explanatory with a small amount of knowledge on HTML. Clicking the submit field triggers the callback action `check name pass` and the variables `name` and `pass` are bound to the handles for the two `textInputFields`. This is due to our convention that HTML constructors always return the value constructed by their children.

Executing the `login` function generates the HTML page in Fig. 4 (see also Fig. 5, left part). In this page, the widgets are automatically named (`f0`, `f1`, `f3`, and `s2`). The numbers are assigned during construction of the page in a depth-first traversal. The `pageInfo` field administers this information.

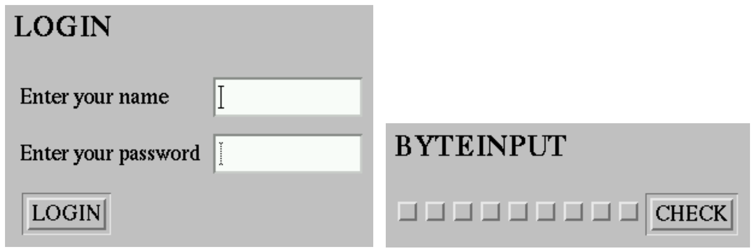

Fig. 5. Generated Web forms

This functionality is largely implemented in the constructors for the input fields like `textInputField`. They have the following responsibilities, beyond constructing the HTML element and creating the field's handle:

- Assign a unique name `f`*i* to the input field.
- Check the log for an input pair with name `f`*i*.
- If such a pair is present, the corresponding value is put into the handle. In the case of a typed field, the value is parsed from the input string.
- If no suitable pair is present (either because there is no logged response, yet, or because the browser did not send it), then `string` and `value` of the handle are set to `Nothing`.

Fields of type "submit" are counted separately because there may be several such fields for each form and each submit field may be bound to a different action. The constructor `submitField` for a submit field works as follows:

- Assign a unique name `s`*i* to the submit field.
- Check the log for an input with name `s`*i*.
- If such an input is present, it registers the callback action in the `action` component of the `pageInfo`.
- Otherwise, nothing happens.

The constructor `makeForm` of the form element is only responsible to set up the attributes and to include the hidden field that contains the log (`=CGI=parm=` in Fig. 4) in the document. The `value` attribute contains the log information (in this case, the empty list `[]`), which is URL encoded.

It is easy to define custom input elements from the given primitives. For example, a byte-input widget might be programmed from checkboxes as shown in Fig. 6. The code generates a sequence of eight checkboxes, extracts their boolean values, multiplies them by the place value, and adds them all together. The first checkbox determines the most significant bit. Like any other HTML element constructor, `byteInput` accepts a parameter `attr` which is distributed to all the checkboxes. Fig. 5 (right part) displays the result.

```
byteInput = byteInput' 256 InputField{string=Just "", value=Just 0}
byteInput' i acc attr =
  if i > 0 then return acc else
  do bi   <- checkboxInputField attr
     acc' <- byteInput' (i `div` 2) acc attr
     return acc'{value= value acc' >>= \v ->
                        value bi >>= \b ->
                        return (v + i * b2i b)}
  where b2i False = 0
        b2i True = 1
```

Fig. 6. A Byte-Input Widget

6 Persistency

In the context of Web Programming, a persistent value has a lifetime which is independent of any particular session. Typically, a persistent value would be stored in a database. To access such a value, a WASH/CGI-script might perform a database query as an `IO` action. Unfortunately, this simplistic approach has a number of drawbacks.

- The (potentially big) result of the query would appear in the log.
- It is virtually impossible to guarantee that a value read earlier in a session is still consistent with the value stored in the database at a later point in a session. Hence, it is impossible to implement an update operation.

For these reasons, we designed an interface for accessing persistent data which overcomes these problems.

The basic idea of the library is to manage all accesses to persistent data via time-stamped handles. The type of a handle is

```
data T a     -- abstract
```

where the type parameter `a` indicates the type of the persistent value. To obtain a handle, it must be initialized:

```
init    :: (Read a, Show a, Types a) => String -> a -> CGI (Maybe (T a))
```

The operation `init` takes the name of a persistent value and an initial value (of type `a`) and returns a `CGI` action, which may return a handle for the value. If the persistent value does not exist, yet, then the system creates it, fills it with the initial value and returns a handle to it. Otherwise, the handle contains the most recently stored value. The contexts `Read a` and Show a indicate that a textfile is used to store the actual data. The context `Types a` indicates that it must be possible to generate a type stamp from type `a`. This type stamp is used to guarantee type consistency across module boundaries. When trying to `init` an existing persistent value at the wrong type then the `CGI` action returns `Nothing`.

The `get` operation retrieves the value from a handle:

```
get     :: Read a => T a -> CGI a
```

The implementation guarantees that `get h`, for a fixed handle `h`, always returns the same value.

The `set` operation stores a value in the persistent store.

```
set     :: (Read a, Show a) => T a -> a -> CGI (Maybe (T a))
```

Interestingly, the action `set h v` may fail (*i.e.*, return `Nothing`). Failure occurs if the handle `h` is not *current*. A handle is current if the value accessible via the handle has not been set otherwise since the handle was created. In other words, a `get` operation on a non-current handle does not return the last value `set` through the handle.

Hence, the last operation

```
current :: Read a => T a -> CGI (T a)
```

takes a handle and returns its most recent version.

As said above, the implementation stores the persistent value in a textfile. Since many scripts accessing the persistent value may run concurrently on the server, we need some kind of concurrency control. Hence, the module `Persistent` locks each file before it starts using it and unlocks it once it is done with the file. Locks are implemented in a portable way by creating a lock directory for each persistent value.

7 Extended Example

As an extended example, we consider a time tabling service[3]. It implements a collaborative interface to construct time tables and publish them on the Web. Time tables can be entered from scratch, saved on the server, and later reviewed. Changing a time table requires a password, whereas viewing is possible for anybody who knows the URL. The whole application is implemented in just 136 lines of Haskell.

Clearly, this task requires global persistence, which is implemented in a module `Persistent`. As an example, here is the code implementing the anonymous view function.

```
cgigen owner =
  do Some hdl <- Persistent.init ('T':'T':owner) Nothing
     alltt <- get hdl
     case alltt of
       Nothing ->
         htell (standardPage "Time Table Service"
                  (text "No time table available for " ## text owner))
       Just (passwd, headers, tt) ->
         timetable owner passwd headers tt True
```

[3] http://nakalele.informatik.uni-freiburg.de:80/cgi/WASH/TimeTable.cgi

The code accesses the timetable for `owner` by first creating a *persistence handle*, `hdl`, using the function `Persistent.init` applied to the name of the persistent value and its initial value. If the persistent value already exists, the handle points to its current value. Otherwise, `init` creates a new persistent value initialized to `Nothing`, in this case. We access the persistent value through function `get`. It returns the current contents of the handle.

The function `timetable`, which is invoked if the time table is available, performs the main formatting work. It is heavily parameterized and we only show an excerpt from this function.

```
timetable owner passwd headers tt final =
  ask $ standardPage (Prelude.head hdrs) $ makeForm $
  do xys <- table $
       do attr "border" "3"
          thead $ tr $ mapM_ (\d -> th (text d ## attr "width" "150")) hdrs
          mapM (\hour -> tr (td (text (show hour) ## attr "align" "right") >>
                         mapM (\d -> ttentry final tt d hour)
                              [1 .. 5]))
                [8 .. 19]
     unless final
       (submitField (updateAction owner passwd hdrs (concat xys) Nothing True)
                    (fieldVALUE "SUBMIT"))
```

The code either generates a form with 60 input fields or the final output formatted as a table, depending on the value of `final`. The second `do` specifies the contents of the table. First, its `border` attribute is set to `3`. Next, it creates the header line of the table `thead`. It creates the header entries by applying the monadic map operator `mapM_` to the list of headers. The second `mapM` is responsible for the contents of the table. It uses the monadic map operator `mapM` for constructing a rectangular arrangement of 5×12 input fields, where each single field is generated by `ttentry`. The submit button (last three lines) only appears if we are generating an input form (`final == False`).

It is interesting to consider how the return value `xys` from the `table` combinator is computed. The innermost `mapM` generates a list of handles to input elements. This list is also returned as the value of the `\hour` lambda expression. The outer `mapM` wraps each of the items into a list.

8 Related work

Meijer's CGI library [3] implements a low-level facility for accessing the input to a CGI script and for creating its output. It is nicely engineered and its functionality is at about the level of our own `RawCGI` library. Meijer's library offers additional features like cookies and its own HTML representation, which we felt should be separated from the functionality of `RawCGI`.

Hughes [5] has devised the powerful concept of arrows, a generalization of monads. His motivating application is the design of a CGI library that implements sessions. Indeed, the functionality of his library was the major source of inspiration for our work. Our work indicates that monads are sufficient to implement sessions (Hughes also realized that [6]). Furthermore, it extends the functionality offered by the arrow CGI-library with a novel representation of HTML and typed compositional forms. Also, the callback-style of programming advocated here is not encouraged by the arrow library.

Hanus's library [7] for server-side scripting in the functional-logic language Curry comes close to the functionality that we offer. In particular, its design inspired our investigation of a callback-style programming model. While his library uses logical variables to identify input fields in HTML forms, we are able to make do with a purely functional approach. Our approach only relies on the concept of a monad, which is fundamental for a real-world functional programmer.

Further approaches to Web programming using functional languages are using the Scheme language [8, 9]. The main idea is to use a continuation to take a snapshot of the state of the script after sending the form to the browser. This continuation is then stored on the server and the form contains a key for later retrieval of the continuation. Conceptually, this is similar to the log that we are using. Technically, there are two important differences. First, we have to reconstruct the current state from the log when the next user request arrives. Using a continuation, this reconstruction is immediate. Second, our approach relies only on CGI whereas the continuation approach implies that the script runs inside the Web server and hence that the Web server is implemented in Scheme, too.

A recent revision of the continuation-based approach overcomes the tight coupling of scripts to the server. Graunke et al [10] present an approach to transform arbitray interactive programs into CGI scripts. The transformation consists of three steps, a transformation to continuation-passing style, lambda lifting, and defunctionalization. The resulting program consists of a number of dispatch functions (arising from defunctionalization) and code pieces implementing interaction fragments. Compared to our approach, they reconstruct the state from a marshalled closure whereas we replay the interaction. Furthermore, they implement mutable state using cookies stored on the client side. It is not clear whether such functionality is needed for Haskell scripts since the language discourages the use of mutable state.

Bigwig [11] is a system for writing Web applications. It provides a number of domain specific customizable languages for composing dynamic documents, specifying interactions, accessing databases, etc. It compiles these languages into a combination of standard Web technologies, like HTML, CGI, applets, JavaScript. Like our library, it implements a session facility, which is more restrictive in that sessions may neither be backtracked nor forked. Each Bigwig session has a notion of a current state, which cannot be subverted. However, the implementation of sessions is different and relies on a special runtime system that improves the efficiency of CGI scripts [12]. In addition, Bigwig provides a sophisticated facility for generating documents and typed document templates. Moreover, there is a type system for forms. WASH/CGI provides typed document templates in the weak sense of Bigwig by keeping strings and values of type `Element` apart. A special type system for forms is not required since (typed) field values are directly passed to (typed) callback-actions. Hence, all necessary type checking is done by the Haskell compiler.

MAWL [13, 14] is a domain specific language for specifying form-based interactions. It was the first language to offer a typing of forms against the code that received its input from the form. It provides a subset of Bigwig's functionality, for example, the facilities for document templates are much more limited.

Guide [15] is a rule-based language for CGI programming. It supports a simple notion of document templates, similar in style to that offered by MAWL. It provides only rudimentary control structure: it sequentially processes a list of predicates and takes the action (that is, it displays a HTML page) associated to the first succeeding predicate. It supports the concept of a session, a limited form of concurrency control, as well as session-wide and global variables. However, it neither supports typed input, nor composition of input fields, nor facilities to ensure that the variables used in a script match the variables defined in a form.

In comparison to a GUI library [16–19] a CGI library does not have to deal with concurrency. All interaction is limited to exchanging messages between Web browser and Web server, so that nested interactions are not possible. This greatly simplifies the implementation. However, HTML is an expressive language to specify layout and Web-based user interfaces are ubiquitous, so there is a market for the kind of library that we are proposing.

In previous work[20, 21], we have investigated the use of Haskell for generating valid HTML and XML documents, where the type system enforces adherence to the DTD. That work is completementary to the present one. While the user interface is identical, the present work constructs HTML elements in a different way and it only guarantees well-formed output. It would be possible to join both works, but we have not done this, yet.

Java Server Pages [22] and Servlets are recent approaches to the convenient specification of dynamic contents. They also provide a notion of session Persistence and encapsulate its implementation. Unfortunately, they do not guarantee type safety and they do not provide the advanced support for generating HTML and forms as we do.

9 Conclusions and future work

The WASH/CGI library brings new power to CGI programmers. It offers a simple and declarative way to implement complicated interactive Web-based user interfaces. In particular, it treats the display of the Web browser like a graphical user interface with restricted facilities. This approach results in a natural use of HTML for the layout. Further, we can attach callback-actions to active input elements to specify the flow of control.

The WASH/CGI approach is not only suitable for CGI programming, but also for other kinds of server-side Web scripting. For example, it would be interesting to investigate a combination with Haskell server pages [23], with Bigwig's runtime system [12], or with proprietary APIs (NSAPI, ISAPI).

Ongoing work addresses the integration with WASH/HTML, a generic typed representation of HTML, which representation ensures that only valid documents are generated [20]. Further, we investigate the integration with a Web server written in Haskell. Using Concurrent Haskell [24], there will be one thread in the server for each session. This approach reduces the management of session state via logs to thread management, as in the continuation approach discussed above. It also greatly simplifies issues like persistency and concurrency control. Last but not least, we are including support for style sheets and JavaScript.

References

1. : Cgi: Common gateway interface. (http://www.w3.org/CGI/)
2. : Haskell 98, a non-strict, purely functional language. http://www.haskell.org/definition (1998)
3. Meijer, E.: Server-side web scripting with Haskell. Journal of Functional Programming **10** (2000) 1–18
4. Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming. Volume 925 of Lecture Notes in Computer Science. Springer-Verlag (1995) 97–136
5. Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37** (2000) 67–111
6. Hughes, J. Private communication (2000)
7. Hanus, M.: High-level server side Web scripting in Curry. In: Practical Aspects of Declarative Languages, Proceedings of the Third International Workshop, PADL'01. Lecture Notes in Computer Science, Las Vegas, NV, USA, Springer-Verlag (2001)
8. Graunke, P., Krishnamurthi, S., Hoeven, S.V.D., Felleisen, M.: Programming the Web with high-level programming languages. In Sands, D., ed.: Proc. 10th European Symposium on Programming. Lecture Notes in Computer Science, Genova, Italy, Springer-Verlag (2001) 122–136
9. Queinnec, C.: The influence of browsers on evaluators or, continuations to program Web servers. [25] 23–33
10. Graunke, P., Findler, R.B., Krishnamurthi, S., Felleisen, M.: Automatically restructuring programs for the Web. In: Proceedings of ASE-2001: The 16th IEEE International Conference on Automated Software Engineering, San Diego, USA, IEEE CS Press (2001)
11. Sandholm, A., Schwartzbach, M.I.: A type system for dynamic web documents. In Reps, T., ed.: Proc. 27th Annual ACM Symposium on Principles of Programming Languages, Boston, MA, USA, ACM Press (2000) 290–301
12. Brabrand, C., Møller, A., Sandholm, A., Schwartzbach, M.I.: A runtime system for interactive web services. Journal of Computer Networks (1999)
13. Ladd, D.A., Ramming, J.C.: Programming the Web: An application-oriented language for hypermedia service programming, World Wide Web Consortium (1995) 567–586
14. Atkinson, D., Ball, T., Benedikt, M., Bruns, G., Cox, K., Mataga, P., Rehor, K.: Experience with a domain specific language for form-based services. In: Conference on Domain-Specific Languages, Santa Barbara, CA, USENIX (1997)
15. Levy, M.R.: Web programming in Guide. Software—Practice & Experience **28** (1998) 1581–1603

16. Carlsson, M., Hallgren, T.: FUDGETS: A graphical interface in a lazy functional language. In Arvind, ed.: Proc. Functional Programming Languages and Computer Architecture 1993, Copenhagen, Denmark, ACM Press, New York (1993) 321–330
17. Vullinghs, T., Tuijnman, D., Schulte, W.: Lightweight GUIs for functional programming. In Swierstra, D., Hermenegildo, M., eds.: International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95). Number 982 in Lecture Notes in Computer Science, Utrecht, The Netherlands, Springer-Verlag (1995) 341–356
18. Finne, S., Peyton Jones, S.: Composing Haggis. In: Proc. 5th Eurographics Workshop on Programming Paradigms in Graphics, Maastricht, NL (1995)
19. Sage, M.: FranTk — a declarative GUI language for Haskell. [25] 106–117
20. Thiemann, P.: Modeling HTML in Haskell. In: Practical Aspects of Declarative Languages, Proceedings of the Second International Workshop, PADL'00. Number 1753 in Lecture Notes in Computer Science, Boston, Massachusetts, USA (2000) 263–277
21. Thiemann, P.: A typed representation for HTML and XML in Haskell. Journal of Functional Programming (2001) To appear.
22. Peligrí-Llopart, E., Cable, L.: Java Server Pages Specification. http://java.sun.com/products/jsp/index.html (1999)
23. Meijer, E., van Velzen, D.: Haskell Server Pages, functional programming and the battle for the middle tier. In: Draft proceedings of the 2000 ACM SIGPLAN Haskell Workshop, Montreal, Canada (2000) 23–33
24. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: Conference Record of POPL '96: The 23$^{\text{rd}}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA (1996) 295–308
25. Wadler, P., ed.: International Conference on Functional Programming. In Wadler, P., ed.: Proc. International Conference on Functional Programming 2000, Montreal, Canada, ACM Press, New York (2000)