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

Peter Thiemann
Universität Freiburg
Germany

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 domain specific embedded language for server-side Web scripting. It is implemented and hosted in Haskell. Its implementation relies on CGI, but it avoids most of CGI's drawbacks by incorporating the concept of a session and by providing a compositional approach to constructing interaction elements (forms). From a programmer's perspective, programming WASH/CGI is like programming a graphical user interface (GUI). In contrast to a GUI, the layout is specified using HTML. WASH/CGI generates HTML via a new monadic interface. Special combinators are available that provide typed input fields and graphics, which is generated on the fly.

Keywords

Haskell, Monads, GUI 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 form elements in HTML. On the programming side, CGI communicates via standard input/output streams and environment variables. It 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 and starts it in a new process. This kind of program is called a *CGI script*. It receives its input through the standard input stream and through environment variables and delivers the response to its standard output stream. The CGI standard [4] fixes the format of this communication.

Unfortunately, there are a number of limitations. The most painful one stems from the fact that the underlying HTTP protocol, which is used for communication between browser and server, is *stateless*. Every single request starts a CGI script. Then the script produces a response page and terminates. Hence, there is no concept of a *session*, *i.e.*, a sequence of alternating requests and responses. Usually, CGI programmers must build such sessions from scratch. They distribute the stages of a session over a number of CGI scripts and connect them manually through links in the response pages. To provide a notion of session-wise state they must resort to putting hidden information in their responses (hidden input fields) or to using cookies, which is not reliable because browsers can refuse them. Clearly, it is error-prone to manually maintain links in this way and also to have the code for a single interaction forcibly spread over many programs.

Another 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 kind of input behavior (a widget in GUI terminology). 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. Inside of a CGI script, these argument values can be accessed by their name. Unfortunately, there is no guarantee that the form uses the names expected by the script and vice versa.

Last but not least, all parameter passing between forms and CGI scripts is completely untyped. Each script must provide its own decoding functions to convert strings into whatever type is really required. It is not even possible to specify the expected type of an input field.

The present work provides a cure for all the issues mentioned above: the DSL WASH/CGI. WASH/CGI makes CGI programming easy and intuitive. It is implemented as a library for Haskell [9] and provides the following features:

- one program can implement entire sessions;
- the specification of an input field and the collection of

the input from this widget are tied together so that mismatches are not possible; the external name of an input field does not matter;

- input fields are first-class entities; they may be typed and grouped to compound input fields (compositionality); each group may be bound to a callback action;
- first-class images as active input fields where each pixel of the image can result in a different action;
- no explicit URLs need to be constructed in the script, except references to external pages;
- the script is "relocatable"; it can be moved in the directory hierarchy or to another server without change[1].

The library is available through the WASH web page [20]. The web page also provides some live examples, complete with sources.

Familiarity with the Haskell language [9] as well as with the essential HTML elements is assumed throughout.

2. EXAMPLE PROGRAMS

This section demonstrates the use of WASH/CGI with some examples. At first, the reader may be surprised that the examples have a distinct GUI flavor. But this is exactly the right impression: CGI programming should feel just like GUI programming, where the layout is determined by HTML.

The library is based on the monad `CGI`, which handles all interaction with the browser.

2.1 Hello world

```
mainCGI :: CGI ()
mainCGI =
  htell (standardPage "Hello World" empty)
```

As customary, the first program just displays `Hello World` on the screen. The combinator `htell` takes an HTML page produced by `standardPage` and sends it to the browser. The combinator `standardPage` takes a title and the contents of the HTML page (here: `empty`) and produces the usual combination of `html`, `head`, `body`, and `title` tags.

```
standardPage ttl elems =
  html (head (title (text ttl)) ##
        body (h1 (text ttl) ## elems))
```

Here, `text` transforms a string into an HTML element and the operator `##` concatenates groups of HTML elements. Hence, the browser receives the following response:

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

[1] Of course, provided that it can execute at all on the other machine.

```
mainCGI :: CGI ()
mainCGI =
  counter 0

counter :: Int -> CGI ()
counter n =
  ask (standardPage "Counter" $
       makeForm $
       do text "Current counter value "
          text (show n)
          br empty
          submitField (counter (n + 1))
                      (fieldVALUE "Increment"))
```

Figure 1: The counter example

2.2 The counter example

The counter example (Fig. 1) uses the `makeForm` combinator to start a form. The contents of the form are specified using a monad. Every content element (in fact, every HTML element) is a value in the monad `WithHTML CGI`. The `text` combinator produces plain text output, the `br empty` inserts a `<br>` element, and the `submitField` creates a submit-button. The first parameter of `submitField` is the action to be taken, when the form is submitted. The second parameter specifies the attributes of the submit-button. The `empty` parameter of `br` can also be replaced by attributes for `<br>`.

2.3 Extended counter

```
counter :: Int -> CGI ()
counter n =
  ask $ standardPage "UpDownCounter" $ makeForm $
  do text "Current counter value "
     activeInputField counter (fieldVALUE (show n))
     submitField (counter (n + 1)) (fieldVALUE "++")
     submitField (counter (n - 1)) (fieldVALUE "--")
```

In this example, we consider a replacement for the `counter` function from the previous example. It displays the current value in an input field and it has two submit-buttons. The generated Web page has the following functionality: Clicking on the `++` and `--` button increments or decrements the counter's value. Alternatively, a new value may be entered by typing it into the `activeInputField` and hitting return to submit the form. The input field is actually typed. It accepts only inputs that parse as elements of type `Int`:

```
activeInputField :: Read a =>
        (a -> CGI ()) -> HTMLField (InputField a)
```

2.4 Calculator

A pocket calculator consists of a display and an array of buttons (Fig. 2). The corresponding code uses a HTML table to specify the layout (Fig. 3). Each button has an action, specified by `calcAction`, attached to it. The `textInputField` is a specialized input field for Strings. Including it into a form yields a value `dsp` of type `InputField String`. This value is a handle to extract the input from the field using the function `value :: InputField a -> Maybe a`.

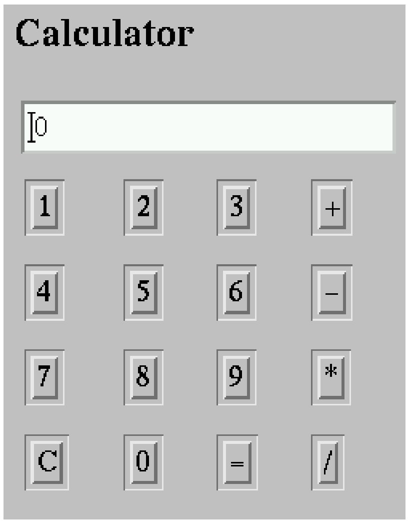

Figure 2: Screenshot of Calculator

```
mainCGI :: CGI ()
mainCGI =
  calc "0" id

calc :: String -> (Integer -> Integer) -> CGI ()
calc dstr f =
  ask $ standardPage "Calculator" $ makeForm $ table $
  do dsp <- tr (td (textInputField (fieldVALUE dstr)
                  ## attr "colspan" "4"))
     let btn c = td (submitField (calcAction dsp c f)
                                 (fieldVALUE [c]))
     tr (btn ’1’ ## btn ’2’ ## btn ’3’ ## btn ’+’)
     tr (btn ’4’ ## btn ’5’ ## btn ’6’ ## btn ’-’)
     tr (btn ’7’ ## btn ’8’ ## btn ’9’ ## btn ’*’)
     tr (btn ’C’ ## btn ’0’ ## btn ’=’ ## btn ’/’)

calcAction :: InputField String ->
              Char -> (Integer -> Integer) -> CGI ()
calcAction dsp c f
  | isDigit c = calc (dstr ++ [c]) f
  | c == ’C’  = mainCGI
  | c == ’=’  =
      calc (show (f (read dstr :: Integer))) id
  | otherwise =
      calc "0" (optable c (read dstr :: Integer))
  where Just dstr = value dsp
       optable ’+’ = (+)
       optable ’-’ = (-)
       optable ’*’ = (*)
       optable ’/’ = div
```

Figure 3: Calculator

The `table`, `tr`, and `td` functions construct HTML elements with the same tag. The argument of each function specifies the list of sub-elements and attributes of the element. The operator `##` serves to concatenate (lists of) HTML elements and attributes. In the example code,

```
td (textInputField (fieldVALUE dstr) ##
    attr "colspan" "4")
```

creates the element

```
<td colspan="4"> <input type="text" value="..."> </td>
```

2.5 Graphics

```
mainCGI =
  ask $ standardPage "UseGraphics" $ makeForm $
  activeImage testImage

canvasRed = newImage (100,100) red
ovalBlue = fillOval canvasRed (20,20) (70,50) blue
background = activate ovalBlue hitNothing
testImage = activateColor background blue hitOval

hitOval = htell (standardPage "Hit the Oval!" empty)
hitNothing = htell (standardPage "Missed." empty)

red  = (255,0,0)
blue = (0,0,255)
```

The connection to GUIs also extends to images where parts of the image may be bound to certain actions. The example program constructs a red square (`canvasRed`) and paints a blue oval in it (`ovalBlue`). Next it activates the image, so that an action is triggered when the image is clicked. The image `background` executes the action `hitNothing` everywhere. The image `testImage` has the action `hitOval` attached to each blue pixel in `background`, and the action `hitNothing` to any other pixel.

3. USER-LEVEL CONCEPTS

This section presents an application programmer's view of the concepts and function of WASH/CGI.

3.1 HTML

Each HTML element is constructed by a function of the same name as shown with the function `table` below. Each of these "constructor functions" has a type like

```
type HTMLCons m a = WithHTML m a -> WithHTML m a
table :: Monad m => HTMLCons m a
```

There are also constructor functions for attributes that will be attached *to the enclosing element.* The generic attribute constructor is `attr`.

```
attr :: Monad m => String -> String -> WithHTML m ()
```

It constructs an attribute instance from an attribute name and an attribute value.

Although a value of type `WithHTML m a` stands for an ordered collection of HTML elements and attributes, it is impossible to examine elements and attributes once they are constructed. Passing such a value to a constructor function incorporates the elements as sub-elements of the new element and also attaches the attributes to it.

The values

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

serve as the empty collection and as the concatenation operation. Since `WithHTML m` is a monad (provided that `m` is), HTML elements may also be combined using the standard monad operations as well as the `do` notation.

In most cases, the parameter `m` will be the monad `CGI`. But there are exceptions, as we will see below in 3.4.

3.2 Input fields

There are special combinators to construct input fields. They add an input field to the current collection of HTML elements and return a handle for accessing the input value. The type of such an input field is

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

That is, it takes a collection of attributes (of type `WithHTML CGI ()`), attaches them to a new `<input>` field, and embeds the new field into another collection. The generic, typed textual input field is constructed by

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

The `Read a` predicate comes from the fact that all communication between browser and server is through strings. Hence, each value of type `a` must be converted from a string.

Once again, the value of type `InputField a` is merely a handle to access its input values through the two functions

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

The `value` function provides access to 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).

The remaining input elements are provided in the same manner (see Fig. 4). 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.

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

Figure 4: Input fields (excerpt)

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.

3.3 Forms

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

The constructor for forms takes a collection of attributes and returns a `<form>` element. At least one form is necessary in each page that contains input fields since input fields do not make sense outside of a form. It is not necessary to set the standard attributes of the form element. The `action` attribute, which contains the URL for processing the form's contents, the `enctype` attribute, which determines the encoding of the form's contents, and the `method` attribute are all determined automatically by WASH/CGI.

3.4 Sessions

Programming of interactions is based on just four combinators in the monad `CGI`.

```
run  :: Translation -> CGI () -> IO ()
ask  :: WithHTML CGI a -> CGI ()
tell :: CGIOutput a => a -> CGI ()
io   :: (Read a, Show a) => IO a -> CGI a
```

The combinator `run` introduces the `CGI` monad. The standard main program (for the examples above) is as follows:

```
main = run NoTranslation mainCGI
```

The `NoTranslation` argument is appropriate for all scripts that do not make use of graphics generation (see Sec. 3.5).

The combinator `ask` displays a page on the browser. Its argument of type `WithHTML CGI a` constructs a web page, which should contain a form. It returns a `CGI` action. This action never produces its value. To extract values from the form, a callback action must be tied to one of its input elements.

The combinator `tell` displays a page and terminates the interaction. The argument of `tell` can have any type of

```
-- inserting into HTML
activeImage   :: CGIImage -> WithHTML CGI ()
-- creating new images
newImage      :: (Int, Int) -> Pixel -> CGIImage
makeText      :: String -> Pixel -> CGIImage
gifImage      :: String -> CGIImage
-- drawing
drawOval, fillOval, drawRectangle, fillRectangle,
  drawLine    :: CGIImage -> (Int, Int) -> (Int, Int) ->
                 Pixel -> CGIImage
-- composing images
overlay       :: CGIImage -> CGIImage -> (Int, Int) ->
                 Pixel -> CGIImage
-- attaching actions
type ActionFun = Int -> Int -> Maybe (CGI ())
activateXY    :: CGIImage -> ActionFun -> CGIImage
activate      :: CGIImage -> CGI () -> CGIImage
activateColor :: CGIImage -> Pixel -> CGI () -> CGIImage
```

Figure 5: Interface for graphics

class `CGIOutput`. These types are `Status` (error message), `Location` (redirect response to a URL), and `WithHTML IO a` (a HTML page). Actually, `tell` is a member function of `CGIOutput`. Appendix A contains some instance definitions.

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

The combinator `io` injects an `IO` action into the `CGI` monad.

3.5 Graphics

Many web pages contain graphics that are prefabricated. The WASH/CGI library contains facilities to create simple, click-sensitive graphics on the fly. As expected from a functional language, an image is a first-class value. Images can be created from scratch (`newImage`), from text strings (`makeText`), and from existing GIF images (`gifImage`). They can be manipulated in the usual ways by drawing ovals, rectangles, and lines and by overlaying one image on top of another (see Figure 5 for a summary of the interface). In addition, `CGI` actions may be attached to parts of an image using `activate`, `activateXY`, and `activateColor`. Finally, an image can be inserted into a web page using `activeImage`. In principle, it is possible to write interactive graphics programs, although the interaction is a bit slow.

4. IMPLEMENTATION

4.1 Raw CGI functionality

The module `RawCGI` implements the input part of the low level CGI functionality. It is usually invisible to a WASH/CGI programmer. Its interface is the function `start`:

```
start :: (CGIInfo -> [CGIParameter] -> IO a) -> IO a
```

It reads and decodes the information passed to the CGI script in environment variables and through the standard input. This information is passed to the first argument, which is supposed to generate the output of the CGI script.

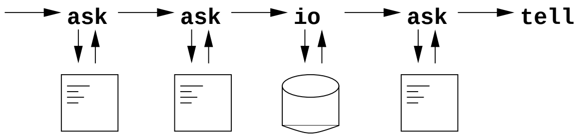

Figure 6: A CGI session

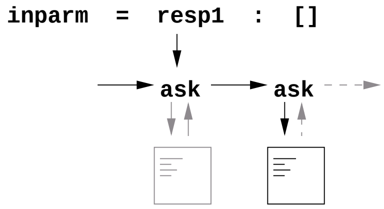

Figure 7: Partial CGI session

The `CGIInfo` argument contains the computed URL of the script, the content type of the input to the script, and the raw input stream.

On submission of a form, the browser sends an encoded version of a list of pairs of the form *fieldname* = *value* to the Web server. In this list, *fieldname* is the name of an input field and *value* is the value entered by the user. The argument list of type `[CGIParameter]` is the decoded version of this list. It can be accessed like a finite map using the following lookup functions for parameter lists:

```
assocParm  :: String -> [CGIParameter] -> Maybe String
```

4.2 Session management

Figure 6 shows a CGI session as it looks to the WASH/CGI programmer and also to the user. A session is a dynamically evolving sequence of `ask` and `io` actions (in the CGI monad). Each of these actions queries the external world, either by displaying a form on a Web browser or by performing and IO action, and receives a response. At each point of the session, the previous responses determine the next CGI action.

Unfortunately, whenever a response is required from the Web browser, we know that the CGI script must terminate. Thus, when the response comes back, we need a way to get back to the action that is waiting for it. To do so, it is necessary to keep track of all the responses that have been received up to that action. These responses are put into a hidden input field on the form, so that they can be reconstructed on the next invocation of the CGI script. Hidden fields are not displayed but they are always sent along when the form is submitted. Thus the hidden field plays the role of a continuation.

For illustration, Figure 7 shows a prefix of the CGI session after the first `ask` has been completed and the second `ask` has delivered its form to the browser. The list `inparm` is the list of responses to previous actions. Its `resp1` corresponds to the response to the first `ask`. If an `ask` (or `io`) has a corresponding value in the `inparm` list, then this value is taken as the response, without sending anything to the browser.

At the second ask, the list inparm is empty, hence the CGI monad must generate the form and get the response from the browser.

On the next invocation of the CGI script, the value of inparm is `resp1 : resp2 : []` so that both ask actions take their responses from the list. Then the next action takes over and computes its response.

4.2.1 The types

```
data CGI a
  = CGI { unCGI :: CGIState -> IO (a, CGIState) }
data CGIState
  = CGIState { inparm   :: [PARAMETER]
             , outparm  :: [PARAMETER]
             , cgiInfo  :: CGIInfo
             , pageInfo :: PageInfo
             }
data PARAMETER
  = PAR_RESULT String
  | PAR_VALUES [CGIParameter]
  deriving (Show, Read)
```

The CGI monad is a state monad composed with the IO monad. The state of type CGIState contains two lists with elements of type PARAMETER, the cgiInfo record passed by RawCGI.start, and some data about the currently generated page in pageInfo. While cgiInfo remains constant, pageInfo evolves continually as requests and responses are processed.

The datatype PARAMETER is the type of responses from CGI actions. The result of an io action corresponds to a PAR_RESULT str and an ask action corresponds to a PAR_VALUES str response. In each case the results are encoded in a string using methods of Read and Show.

The list inparm contains the responses to previous actions in the order that they occurred. It indicates the path from the start of the interaction to the current action. Each ask or io action takes its response from the first element of inparm and passes its tail to the remaining actions.

The list outparm initially contains the same values as inparm, but in reverse order. This field plays the role of the history log. Whenever a new response is computed, it is put *in front* of this list. This outparm list is actually put in the hidden input field of every form.

The pageInfo field contains all information to generate input elements and forms and for extracting results from them. It is reset by every ask and io action.

```
data PageInfo =
     PageInfo { count      :: Int
              , bindings   :: Maybe CGIParameters
              , enctype    :: String
              , nextaction :: Element -> CGI ()
              }
```

The CGI monad generates unique names for input elements and forms from the count field. The field bindings contains the parameters for the current form if they can be computed from the inparm field. The enctype field contains the content type of the most recently submitted form (in this session). Finally, the nextaction field determines the CGI action to be executed after the output page has been computed. This action is either simply to display the page or it is the action specified in a submit field.

4.2.2 The actions

Now that we have gathered the basic intuitions and the basic types involved, we can look at the code of io:

```
io :: (Read a, Show a) => IO a -> CGI a
io ioAction =
  CGI (\cgistate ->
    case inparm cgistate of
      -- parameter of correct form present:
      --   extract result
      PAR_RESULT aStr : rest ->
        return (read aStr, nextCGIState cgistate)
      -- no parameter present:
      --   perform action and prepend result
      [] ->
        do a <- ioAction
           return (a, cgistate
                  { outparm = PAR_RESULT (show a) :
                              outparm cgistate })
      -- wrong parameter: generate an error
      _ -> reportError "Out of sync" empty cgistate)
```

The function nextCGIState has two duties. It drops the first element of the inparm field, which belongs to the current action, and it re-initializes the pageInfo field.[2]

```
nextCGIState cgistate = cgistate'
  where cgistate' = cgistate
         { inparm = drop 1 (inparm cgistate)
         , pageInfo = initialPageInfo cgistate'
         }
```

The initialPageInfo sets the count to 0 and it extracts the current bindings from the first element of the inparm field of the CGI state. The enctype is also initialized from the state. The nextaction field is either tell (if there are no parameters for this form, then the form must be displayed on the browser) or it is tellError (which generates an error unless an input field overrides the action).

```
initialPageInfo cgistate =
  let bnds = do PAR_VALUES parms <-
                  listToMaybe (inparm cgistate)
                return parms
  in  PageInfo
  { count = 0
  , bindings = bnds
  , enctype = contentType cgistate
  , nextaction = case bnds of
```

[2]There is no real recursion going on, but the recursive binding makes the code much simpler.

```
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
```

Figure 8: Signature of `HTMLBase` (excerpt)

```
        Nothing -> tell
        Just _  -> tellError "Unspecified action"
  }
```

This leaves not much to do for the combinator `ask`. It first builds the output document from its argument of type `WithHTML CGI a` using `build_document`, which may change the `nextaction` field, then it extracts the `nextaction` field from the `pageInfo` and applies it to the document just constructed. The resulting `CGI` action is applied to the next CGI state. To reiterate, if there are no parameters for this document (or rather for its form), then it displays the document. Otherwise, some action is performed that is collected while constructing the document.

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

We defer the discussion of `run`'s implementation to Section 4.7 because it is easier to look inside the monad `WithHTML CGI` first, to see how input fields really work. But before we can do that, we have to consider the generation of HTML.

4.3 Generating HTML

The low-level module `HTMLBase` provides some simple functionality for generting raw HTML (Fig. 8). The constructors `element_` and `doctype_` construct ordinary elements and the top-level element of a document, `attr_` constructs an attribute from the name and the value string, `add_ e e'` appends `e'` to the list of sub-elements of `e`, and `add_attr_` attaches an attribute to an element.

The monad transformer `WithHTML` adds to any monad `m` a component that generates HTML. To this end, it composes `m` with a state monad that propagates a *currently constructed* HTML element. The `return`, `>>=`, and `lift` operations are standard, and thus omitted.

```
data WithHTML m a = WithHTML
     { unWithHTML :: Element -> m (a, Element) }
```

Each element-producing combinator adds to the current element, as demonstrated with the `table` combinator.

```
table :: Monad m => WithHTML m a -> WithHTML m a
table ma =
  WithHTML (\elem ->
    unWithHTML ma (element_ "table" [] []) >>=
    \(a, tableElem) ->
      return (a, add_ elem tableElem))
```

Here, `element_` constructs a `<table>` element with empty attribute list and empty sub-element list. This element is first transformed by `unWithHTML ma`, which attaches attributes and adds sub-elements to it. Finally, the complete `<table>` element is added to the current element `elem`.

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)))
```

The top-level function `build_document` constructs the outermost `<!DOCTYPE ...>` declaration using the low-level function `doctype_`, passes it to the monad that fills it with content, and returns the full element in the end.

```
build_document :: Monad m => WithHTML m a -> m Element
build_document ma =
  unWithHTML ma
    (doctype_
      ["HTML"
      ,"PUBLIC"
      ,"\"-//W3C//DTD HTML 4.01//EN\""
      ,"\"http://www.w3.org/TR/html4/strict.dtd\""]
      [])
  >>= \ (a, elem) ->
  return elem
```

The implementations of `empty` and the gluing operator `##` are unsurprising. `empty` corresponds to `return` and gluing is the monad's bind operator, modified to return the result of the first action.

```
empty :: Monad m => WithHTML m ()
empty = return ()

(##)  :: Monad m => WithHTML m a ->
                    WithHTML m b -> WithHTML m a
x ## y = x >>= \a -> y >> return a
```

The remaining functions are routine. More interesting are the functions that construct input fields. They live in the monad `WithHTML CGI`, which is discussed next.

4.4 The monad `WithHTML CGI`

A typed input field takes a collection of attributes, attaches them to an `<input>` element, and yields the input field.

```
type HTMLField a = WithHTML CGI () -> WithHTML CGI a
inputField :: Read a => HTMLField (InputField a)
```

The combinator has three obligations.

1. Generate an `<input>` element and insert it into the implicit HTML document.
2. Generate the input handle of type `InputField a`.
3. Obtain the value of the input field.

These tasks must happen at different stages. On the first execution of a particular `inputField` action in a session, the HTML document must be generated and the handle `InputField a` must be constructed, but there are no parameter values to process. In all subsequent executions, there are parameter values present. The `inputField` action extracts its value from the `bindings` field in the `pageInfo` field of the CGI state. Hence, the following code implements an input field:

```
data InputField a =
     InputField { string :: Maybe String
                , value  :: Maybe a
                }

inputField :: Read a => HTMLField (InputField a)
inputField fieldAttrs =
  do fieldName <- lift nextName
     pageInfo  <- lift getPageInfo
     let maybestring =
           bindings pageInfo >>= assocParm fieldName
         maybevalue  = maybestring >>= (g . reads)
         g ((a,""):_) = Just a
         g _ = Nothing
     input (attr "type" "text" ##
            attr "name" fieldName ##
            fieldAttrs)
     return $ InputField
       { string = maybestring
       , value  = maybevalue
       }
```

Here, `nextName :: CGI String` is an action that computes a new field name from the `count` field in the `pageInfo` component of the `CGIState`. The `getPageInfo :: CGI PageInfo` action fetches the current value of the `pageInfo` field. Both must be `lift`ed into `WithHTML CGI` because they are `CGI` actions. The combinator `input` adds an `<input>` element to the current element of `WithHTML CGI`. The variable `maybestring` is set to `Just` *astring* if there are parameter bindings for the current page and the parameter corresponding to this field is present. Finally, the `value` field is computed by `read`ing from the `maybestring` if it is present.

All other input widgets are simple variations of this scheme. Some (for example, images and selection boxes) are a little more complicated, but the basic idea is the same. This leaves not much to do for the `makeForm` combinator.

```
makeForm :: WithHTML CGI a -> WithHTML CGI ()
makeForm contents =
  form $
    do contents
       name <- lift nextName
       info <- lift getInfo
       parm <- lift getParm
       url  <- lift getUrl
       input (attr "type" "hidden"
           ## attr "name" "=CGI=parm="
           ## attr "value" (urlEncode $ show parm))
       attr "action" url
       attr "name" name
       attr "enctype" (enctype info)
```

It extracts all information from the `CGI` monad, creates an additional hidden field that contains the value of the `outparm` field, and sets the `action` and `enctype` attributes accordingly. A hidden input field does not appear on the screen when the form is displayed, but its value is transmitted along with the other values whenever the form is submitted. The contents of the `value` attribute are URL-encoded because some characters are illegal in an attribute value. This encoding is glaringly inefficient, but hidden fields are not intended to transmit big chunks of data, anyway.

4.5 Graphics

Graphics is a little more complicated to integrate. Let's first refresh our memory about input fields with graphics: An element

```
<input type="image" src="image-url" name="id">
```

displays the image specified by the `src` attribute. When the image is clicked, the browser submits the form and passes the coordinates of the click in the form of two parameters *id*`.x` and *id*`.y`. It is straightforward to transform these into a pair of `Int`s and create a handle of type `InputField (Int, Int)` from them.

But this is not enough to produce graphics on the fly. In our approach, we start from a declarative description of the image. To simplify the description, we first concentrate on inactive images. The WASH/CGI library needs to accomplish two tasks for such an image.

1. Generate an image from the declarative description.
2. Create a URL for the image.

To generate an image, the library transforms all image sources into a pixmap format, and then processes these pixmaps internally as indicated by the declarative description. In the end a PPM (portable pixmap) file is written in a temporary directory and this file is transformed to a GIF image using `ppmtogif`[3]. Subsection 4.6 gives more information on this.

At this point, we have a GIF image stored in a temporary directory. Since we claim relocatability for our CGI scripts, the library cannot make any assumptions about which part of the file system is accessible to the web server. Hence, the

[3]From the `PBMPLUS` suite of image conversion programs[14].

generated URL of the image points again to the CGI script. It uses the query string part of the URL (the part after the ?) to indicate accesses to files stored in the temporary directory.

The main function of WASH/CGI, `run`, takes a parameter of type `Translation` to map query strings to files:

```
data Translation =
     NoTranslation
   | forall a . CGIOutput a =>
                Translation ([String] -> a)
```

The value `NoTranslation` indicates that this feature of WASH/CGI is turned off. The value `Translation` *trans* supplies a function *trans* that maps a query string to something that may be sent as the body of a HTTP response.[4]

As an example, here is the translation function for the graphics example from Section 2.5.

```
main = run (Translation translator) mainCGI

translator (name:_) =
  FileRef
   { fileRefName = storeDirectory ++ name
   , fileRefContentType = guessContentType name
   }

guessContentType name
  | ".gif" `isSuffixOf` name = "image/gif"
  | otherwise = "application/octet-stream"
```

Here, `FileRef` is the obvious record type and it is printed using an instance of `CGIOutput` (see appendix A).

4.6 Image generation

```
type CGIImage  = IO CGIImage'
type ActionFun = Int -> Int -> Maybe (CGI ())

data CGIImage' =
     CGIImage'{ pixmap    :: Pixmap
              , actionFun :: ActionFun
              }
```

Surprisingly, an image is really an IO action! When performed, this action computes two things, a `pixmap`, which corresponds to an image in PPM format (the type `Pixmap` and its operations live in a separate module `PPM`), and an `actionFun`, which maps two coordinates to a CGI action.

[4]The datatype declaration uses an existentially quantified type for the field. This is an extension with respect to Haskell98. It is supported by many systems.
This existential quantification seems unavoidable. The obvious approach is to use `Maybe ([String] -> a)` as the type of a translation. But the resulting type for `run :: CGIOutput a => Maybe ([String] -> a) -> CGI () -> IO ()` generates an ambiguous type for `run Nothing :: CGIOutput a => CGI () -> IO ()`. Hence, it is not appropriate.

```
gifImage gifPath =
  do key <- inventStdKey
     let path = storeDirectory ++ key
         ppmPath = path ++ ".ppm"
     system (pbmPath ++ "giftoppm " ++ gifPath ++
             " > " ++ ppmPath)
     pixmapStr <- readFile ppmPath
     return $ CGIImage'
        { pixmap = read pixmapStr
        , actionFun = const (const Nothing)
        }
```

Figure 9: Implementation of `gifImage`

```
flushImage :: CGIImage -> CGI (URL, ActionFun)
flushImage img =
  do cgiImage <- unsafe_io img
     pageInfo <- getInfo
     baseUrl <- getUrl
     case bindings pageInfo of
       Nothing ->
         unsafe_io $ do
           key <- inventStdKey
           let path = storeDirectory ++ key
               ppmPath = path ++ ".ppm"
               gifPath = path ++ ".gif"
           writeFile ppmPath (show $ pixmap cgiImage)
           system (ppmtogif ++ ppmPath ++
                   " > " ++ gifPath)
           return (baseUrl ++ '?' : key ++ ".gif"
                   ,actionFun cgiImage)
       Just _ ->
         return ("", actionFun cgiImage)
```

Figure 10: Implementation of `flushImage`

Why is an image an IO action? For certain functionalities, like `makeText` (create a pixmap from a string) and `gifImage` (create a pixmap from a GIF image), WASH/CGI relies on the `pbmplus` library [14], which supplies external programs with the desired functionality. These programs must be run externally (using `System.system`) and their results must be read in for further processing. Both actions require the IO monad.

As an example, Fig. 9 shows the code for `gifImage`. The IO action `inventStdKey` randomly generates a unique name for use in the `storeDirectory`. The `actionFun` is initialized to "no action", it can be overridden later on using the `activate`, `activateXY`, and `activateColor` combinators. The latter work by overriding the `actionFun` field of a CGI image, as this example shows.

```
activateXY :: CGIImage -> ActionFun -> CGIImage
activateXY img fun =
  do cgiImage <- img
     return $ cgiImage { actionFun = fun }
```

Finally, Fig. 10 shows the code of `flushImage`, which converts a `CGIImage` into a URL and an action function. First, `flushImage` generates the image by executing it in the `IO` monad using `unsafe_io` of type `IO a -> CGI a`. This access of the `IO` monad from the `CGI` monad is different to the "memoized" one implemented by the `io` combinator. It is executed for each invocation of the script that must display the image or that displayed it earlier in its session. In fact, `cgiImage` has type `CGIImage'`: it is a pair of a pixmap and an `ActionFun`. Clearly, it makes no sense to save this result because the pixmap is way too big and the `ActionFun` cannot easily be made an instance of `Read` or `Show`.

Unfortunately, this is not avoidable because the `ActionFun` part may depend on the generated pixmap, so that the `cgiImage` must be recomputed on every execution. The remaining actions, writing the PPM file and converting it to a GIF file, need only be done once and for all, at the first time the image is required. The URL of the image is composed of the URL of the script and a query string, which is the file name of the generated graphics file.

The remaining functions are just straightforward variations and present no further challenges.

4.7 The `run` combinator

The implementation of `run` has two parts. The first part initializes the session parameters and the second part deals with images.

```
run :: Translation -> CGI () -> IO ()
run translator (CGI cgi) =
  start $ \ info decoded ->  do
  let maybeparm = liftM (read . urlDecode) $
                  assocParm "=CGI=parm=" decoded
      oldparm = case maybeparm of Just parm -> parm
                                  Nothing -> []
      newparm | null decoded = []
              | otherwise =
                (PAR_VALUES $ dropHidden decoded)
                : oldparm
      cgistate = CGIState
        { inparm = reverse newparm
        , outparm = newparm
        , cgiInfo = info
        , pageInfo = initialPageInfo cgistate
        }
```

Recall that the `outparm` field contains the log of the results of `io` actions and the inputs to `ask` actions in reverse order. The first part of the code reads the old `outparm` value and forms its value for the current execution by prepending the newly acquired input parameters for the current form. As already explained before, the value of `inparm` is the reverse of `outparm`. The `cgiInfo` and `pageInfo` fields are simply set to their initial values.

The next part deals with the `Translation` argument and starts the monad.

```
  args <- getArgs
  case translator of
    NoTranslation ->
      cgi cgistate >> return ()
    Translation trans ->
      if null args
      then cgi cgistate >> return ()
      else cgiPut (trans args)
  exitWith ExitSuccess
```

A CGI script can recognize the presence of a query string by looking for its argument list (using `System.getArgs`). If there is neither a translation nor a query string, then the code runs the `cgi` action on the initial state `cgistate`. Otherwise, it outputs the result of translating the query string using the `cgiPut` method of the class `CGIOutput`.

5. DISCUSSION

In finished products, it is sometimes unclear how and why design decisions were made. The present design evolved through a number of steps. The first two subsections discuss two obsolete approaches and explain their shortcomings.

5.1 No monads

In the first design, the type `Element` of HTML elements was freely accessible. Furthermore, `Element` had to be parameterized over the type of return values of a form. In this approach, the type of a few representative combinators was as follows.

```
ask :: String -> [Element (Collect m)] -> CGI m
makeForm :: m -> [Element (Collect m)] ->
        Element (Collect m)
makeInputField :: (Read a) =>
        FieldConfig a -> (m -> a -> m) ->
        Element (Collect m)
```

The CGI action `ask title elems` displays a HTML page with title `title` where `elems` may contain as many forms as desired. The action returns the actual typed values for the input fields of the submitted form. The type `Collect m` is defined as follows:

```
data Collect m
  = I m
  | T (String -> m -> Maybe m)
```

The idea is that the type `m` is a state that is threaded through the whole form. Each input field updates it. In the end, it becomes the form's return value. The value `I m` specifies the value of the initial state. The value `T f` specifies a parsing function that maps the input string and the current state to the next state (if the input string was parsable).

Hence, the `makeForm q elems` action is parameterized over an initial value `q` for the state and the list of elements that make up the contents of the form. Finally, `makeInputField conf upd` takes a list of configuration options for the field and an update function that integrates the value of the field into the state. Typically, the type `m` is a record type and the update functions are record-field updates.

This approach has a number of drawbacks. First, it is not possible to re-use an existing HTML library because none provided the necessary parameterization. Second, because of the parameterization forms and form elements cannot be freely combined into a web page. All elements in a page must share a single type. This is quite restrictive and discourages re-use of form components. Third, programming and use of the update functions is error-prone, so that different input fields may accidentally update the same component. Fourth, it is not possible to use a callback-style of programming.

5.2 Imperative approach

In some GUI toolkits, in particular TkGofer [5] and its descendant TclHaskell, the creation of a widget returns a handle to the widget, which is later on used to access the widget. In our corresponding approach to the CGI library, there were actions in the CGI monad to create input fields. The HTML code was created separately from the resulting handles.

Putting the widget creation directly into the CGI monad is unsafe because the handle of one field may be used multiple times to generate its HTML representation. Even worse, it is possible to pass such a handle into a different form. Clearly, this plays havoc to our parameter-passing mechanism, which relies on the uniqueness and the locality of the names assigned to input elements. These issues could have been addressed using an advanced type system, but the requirements (linearity could enforce that each handle is used exactly once) go well beyond Haskell's capabilities.

5.3 Screen layout

It might be argued that the monadic organization of the HTML generation forces particular screen layouts, which is certainly a bad thing from the GUI designer's point of view. For example, it is easy to create input fields and refer to them "from the right" or "from below":

```
do f1 <- inputField (fieldSIZE 10)
   f2 <- inputField (fieldSIZE 10)
   submitField (someAction f1 f2) empty
```

But what if some adventurous soul wants to position the submit button to the left or above the input fields that it relates to?

Fortunately, both monads `CGI` as well as `WithHTML CGI` admit a fixpoint operator, since they are just specialized state monads [6]. Hence, it is possible to return the handles from the monad and feed them back in. This way, it is possible to place the `submitField` at an arbitrary place. In fact, there is a proposal for a variant of the `do` notation that provides recursive bindings using the keyword `mdo` [6]. Using this notation, we could simply reverse the order in the above example.

```
mdo submitField (someAction f1 f2) empty
    f1 <- inputField (fieldSIZE 10)
    f2 <- inputField (fieldSIZE 10)
```

6. RELATED WORK

Meijer's CGI library [12] 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. However, 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 [11] 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 [10]). Furthermore, it extends the functionality offered by the arrow CGI-library with a novel representation of HTML, compositional forms, and graphics. Also, the callback-style of programming advocated here is not encouraged by the arrow library.

Hanus's library [8] for server-side scripting in the functional-logic language Curry comes close to the functionality that we offer. In particular, its design inspired our switching to 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.

Bigwig [16] 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 [2]. 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 template 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 [1] is a domain specific language for specifying form-based interactions. It provides a subset of Bigwig's functionality, but it was the first language to offer a typing of forms against the code that received its input from the form. In particular, the facilities for document templates are much more limited.

In comparison to a GUI library [3, 18, 7, 15] 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, even for a GUI, and Web-based user interfaces are ubiquitous, so there is a market for the kind of library that we are proposing.

7. CONCLUSIONS AND FUTURE WORK

The WASH/CGI library brings new power to CGI programmers. It offers an easy 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 and in the use of callback-actions 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 [13], with Bigwig's runtime system [2], or with proprietary APIs.

Here are some points for future work.

- Integration with WASH/HTML, a generic typed representation of HTML. This representation ensures that only well-formed documents are generated [17].
- The current collection of image combinators is ad-hoc and is mainly used for a proof-of-concept implementation. It should be redone, perhaps along the lines of the functional model for 2D graphics used in the GUI toolkit Haggis [7].
- Improve the efficiency of image generation. There are several avenues to consider here. The current representation of a PPM image is a functional one (*i.e.*, a function `Int -> Int -> Pixel`) and one of the main bottlenecks is the writing of the PPM file. The possibilities here range from improving the Haskell side of the code to moving more functionality into external programs, for example using the GD library.

8. REFERENCES

[1] David Atkinson, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. USENIX.

[2] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. A runtime system for interactive web services. *Journal of Computer Networks*, 1999.

[3] Magnus Carlsson and Thomas Hallgren. FUDGETS: A graphical interface in a lazy functional language. In Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pages 321–330, Copenhagen, Denmark, June 1993. ACM Press, New York.

[4] Cgi: Common gateway interface. `http://www.w3.org/CGI/`.

[5] Koen Claessen, Ton Vullinghs, and Erik Meijer. Structuring graphical paradigms in TkGofer. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

[6] Levent Erkök and John Launchbury. Recursive monadic bindings. In Wadler [19], pages 174–185.

[7] Sigbjørn Finne and Simon Peyton Jones. Composing Haggis. In *Proc. 5th Eurographics Workshop on Programming Paradigms in Graphics*, Maastricht, NL, September 1995.

[8] Michael Hanus. 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, 2001. Springer-Verlag.

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

[10] John Hughes. Private communication, September 2000.

[11] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[12] Erik Meijer. Server-side web scripting with Haskell. *Journal of Functional Programming*, 10(1):1–18, January 2000.

[13] Erik Meijer and Danny van Velzen. Haskell Server Pages, functional programming and the battle for the middle tier. In *Draft proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, pages 23–33, Montreal, Canada, September 2000.

[14] Jef Poskanzer. pbmplus – image file format conversion package. `http://www.acme.com/software/pbmplus`, 1991.

[15] Meurig Sage. FranTk — a declarative GUI language for Haskell. In Wadler [19], pages 106–117.

[16] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic web documents. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 290–301, Boston, MA, USA, January 2000. ACM Press.

[17] Peter Thiemann. Modeling HTML in Haskell. In *Practical Aspects of Declarative Languages, Proceedings of the Second International Workshop, PADL'00*, volume 1753 of *Lecture Notes in Computer Science*, pages 263–277, Boston, Massachusetts, USA, January 2000.

[18] Ton Vullinghs, Daniel Tuijnman, and Wolfram Schulte. Lightweight GUIs for functional programming. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, pages 341–356, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[19] Philip Wadler, editor. *International Conference on Functional Programming*, Montreal, Canada, September 2000. ACM Press, New York.

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

APPENDIX

To be read at the discretion of the reviewer.

A. INSTANCES OF CGIOUTPUT

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

instance CGIOutput (WithHTML IO a) where
  cgiPut hma =
    do elem <- build_document hma
       cgiPut elem

instance CGIOutput Element where
  cgiPut elem =
    do putStrLn "Content-Type: text/html"
       putStrLn ""
       putElement elem

instance CGIOutput String where
  cgiPut str =
    do putStrLn "Content-Type: text/plain"
       putStrLn ""
       putStr str

-- contents of a file
instance CGIOutput FileRef where
  cgiPut fr =
    do putStr "Content-Type: "
       putStrLn (fileRefContentType fr)
       putStrLn ""
       readFile (fileRefName fr) >>= putStr

-- a redirection
data Location = Location URL
instance CGIOutput Location where
  cgiPut (Location url) =
    do putStr "Location: "
       putStrLn url
       putStrLn ""
```