WASH—Web Authoring System in Haskell User Manual

Peter Thiemann

August 18, 2004

Abstract

This user manual fulfills two purposes. First, it contains a gentle introduction to the concepts of WASH by guiding through the construction of an example application. After studying this introduction, the reader should be ready to understand the example applications shipped with the system and to get further ideas from studying the automatically generated reference manual.

Second, it gives some hints how to tackle frequently occurring tasks. Most of them are also covered by one or more example applications.

1 Overview

Currently, WASH consists of five parts, four of which are "visible".

- WASHCGI, a package that contains the functionality to develop CGI-based interactive Web applications. It requires the packages WASHHTML and Utility.
- WASHHTML, an API for constructing XHTML output. It requires the package Utility.
- WASHMail, an API for composing and decomposing (multi-part) emails and for sending mails.
- Wash2hs, a preprocessor that provides a JSP-style surface syntax for WASHHTML.

The preprocessor should be considered a front end for XHTML-generation with WASHHTML. The surface syntax that it provides simplifies the latter task considerably. It is particularly helpful when designing forms with more than a few input fields. In addition, using the preprocessor is not a yes or no question. Modules using the WASH surface syntax can be freely interchanged with ordinary modules. On the down side, errors get a bit harder to localize with the preprocessor, but this is far outweighed by the simplified interface.

WASH is built on top of the purely functional programming language Haskell. The embedding is fairly unobtrusive and the presence of the preprocessor allows the construction of simple applications without knowing much about Haskell. Thus, simple task are simple and complicated task require some knowledge of the language.

WASH relies heavily on monads. A monad is an abstraction that allows side effects to be introduced in Haskell programs. For understanding WASH, the simple intuition of a monad as packaging a sequence of actions is sufficient. However, it is recommended to acquire some background knowledge on monads to get the full potential of the APIs and to progress beyond casual use.

An essential prerequisite to using WASH is knowledge of XHTML. An appropriate characterization of WASH is "programming the Web like writing an interactive application with an XHTML GUI". While WASH takes away many tedious tasks like administering names of input elements and managing forms, there is still the work of creating the actual layout.

2 Getting started

This section covers the essential steps to create a simple interactive application. The first step comprises the magic words for creating a WASH program. The next step is the generation of XHTML output.

2.1 The main program

First and foremost, the WASH main program is a Haskell main program. That is, there must be a module named `Main` residing in file `Main.hs` and it must define a value `main :: IO ()`. For a WASH program, the following template is used to define `main`:

```
module Main where

import CGI hiding (div, head, map, span)

main =
  run mainCGI

mainCGI =
  ...
```

- Every module using WASH needs the line `import CGI ...`
- The `hiding` clause in the import avoids name clashes between names defined in the `CGI` module and the standard prelude (`div` is integer division, `head`, `map`, and `span` are frequently used functions on lists).
- The value `mainCGI` is a *CGI action* of type `CGI ()`. CGI actions can be constructed with the functions described below and composed with the standard monad operations (expressing sequencing of actions).

2.2 Generating XHTML output

The standard way of creating an XHTML output page is using the function

```
ask :: WithHTML x CGI a -> CGI ()
```

Its argument is a *document action*. Its type looks rather complicated, but for the moment it is sufficient to note that this is exactly the type of value created by the WASHHTML API and also by the element notation provided through the preprocessor. The latter is easier on persons familiar with XHTML and is applicable in most occasions. Hence, we defer the discussion of the API because its direct use is only required in exceptional cases.

2.2.1 A Hello World page

The simplest Web script just produces a static output page whenever it is run. Thanks to the preprocessor, it suffices to apply the `ask` combinator to an XHTML document:

```
mainCGI =
  ask  <html>
         <head><title>Hello World</title></head>
         <body>
         <h1>Hello World</h1>
         </body>
       </html>
```

To run this program, the definition of `mainCGI` should be entered in the template for the `Main` module and placed in a file named `Main.hs`. The transformation to Haskell proper is performed by the compiler with the command:

```
ghc -package WASH --make Main
```

The resulting binary program file is called `Main`. It may be run from the command line to look at the HTML output or the binary may be copied into a CGI-enabled directory in the directory tree of a Web server (often `cgibin`). Start it from your favorite Web browser by entering its url

```
http://myserver/cgibin/Main
```

The name `myserver`, the path `cgibin`, and the name (in particular the suffix) you have to use for the program itself depends on the configuration of the Web server you are using. Contact the server's administrator if in doubt.

2.2.2 Digression: the XML document model

Proficient use of WASH requires a little knowledge of the XML document model. An XML document is represented by a tree where the nodes of the tree contain different kinds of information. The DOM standard distinguishes twelve different types, but for purposes of XHTML generation four types are sufficient.

Element An element node represents an XML element, it corresponds to the text between a start tag and its matching end tag.

Each element node has attached to it a number of attribute nodes and an ordered list of child nodes. Child nodes may be of type Text, Element, or Comment.

Attribute An attribute node represents an XML attribute occurrence.

Text A text node represents a piece of text without markup.

Comment A comment node represents a comment in XHTML, which is not displayed.

Many XML standards (*e.g.*, XPath) deal with documents in terms of *sequences of document nodes* and this notion also provides the right approach for WASH.

2.2.3 Document templates

The above Hello World program is extremly unflexible because the entire text is hard coded into the program. Fortunately, it is easy to abstract over parts of the document's text by using a Haskell function to construct the document and a *text escape* `<%=...%>` in the constructed page. The text escape may contain an arbitrary Haskell expression of type `String`.

```
mypage ttl =
  <html>
    <head><title><%=ttl%></title></head>
    <body>
      <h1><%=ttl%></h1>
    </body>
  </html>

mainCGI =
  ask (mypage "Hello World")
```

To add further contents to such a page requires abstraction over *sequences of document nodes* because such a sequence is expected in the body of the document after the header `h1`. The preprocessor supports it through a *code escape* `<%...%>`. The code escape may contain arbitrary Haskell code for document actions using Haskell syntax or XHTML notation. Of course, such nested XHTML fragments may contain arbitrarily nested escapes themselves.

```
yourpage ttl contents =
  <html>
    <head><title><%=ttl%></title></head>
    <body>
      <h1><%=ttl%></h1>
      <%contents%>
    </body>
  </html>

mainCGI =
  ask (yourpage "Hello World" <#></#>)
```

Since we don't have a useful content for the body of the template at this point, we pass an empty sequence as parameter to `yourpage`. In general, the markers `<#>` and `</#>` delimit a nameless container for a sequence of document nodes.

2.2.4 A standard query operation

The combination of `yourpage` and `ask` is so universally useful that WASH already provides it as a function.

```
standardQuery :: String -> WithHTML x CGI a -> CGI ()
```

Its first argument is a title string which is put in the title line of the browser window and which makes up the header line of the page. The second argument is a document action.

In addition, `standardQuery` wraps the body of the document into a `form` tag so that input elements may be used freely within the page.

Using `standardQuery`, the Hello World for WASH completes the template above by one line:

```
mainCGI =
  standardQuery "Hello World" <#></#>
```

2.2.5 Manipulating document actions

Often it is necessary to build documents from document fragments. While code escapes provide the means of parameterizing one fragment by another, sometimes documents need to be glued together, for example, to construct lists in the output.

Since a document action may be regarded as a sequence of document nodes, WASHHTML provides the operations for creating an empty sequence or a one-element sequence (a singleton) and for concatenating sequences. There is intentionally no way of examining document sequences once they are constructed. They can only be converted to (the textual representation of) a sequence of XHTML elements.

The empty sequence may be created in two ways. Either with `<#></#>` or with `empty`.

Any use of an XHTML element creates a singleton sequence. For example, `<h1>Hello World</h1>` creates a sequence with exactly one `h1` node.

Two or more document sequences may concatenated using the `do` notation provided by Haskell:

```
do { seq1; seq2; ...; seqn }
```

or

```
do  seq1
    seq2
    ...
    seqn
```

In the second variant, the layout is essential: the first character of `seq1`, `seq2`, ... `seqn` *must* be aligned to the same starting column.

Alternatively, the binary operators `##` and `>>` may be used to concatenate sequences.

2.2.6 Attributes

Attributes nodes can be created "inline" using their standard XHTML syntax. Additionally the syntax `<[...]>` creates a sequence of attribute nodes from the bracketed list of name-value pairs. For example, the value of `twoAttrs` in

```
twoAttrs = <[bgcolor="red" onload="init()"]>
```

is a sequence of two attribute nodes. Also, `<[]>` provides a further way of constructing the empty sequence.

A code escape in child position attaches attributes to an element. For example, in

```
<body><%twoAttrs%> ... </body>
```

the two attributes are attached to the `body` element.

A code escape with attribute nodes may also be used in attribute position:

```
<body <%twoAttrs%>> ... </body>
```

In addition, only the attribute value may be replaced by a code escape as in

```
bgcolor= "red"
onload=  "init()"
b = <body bgcolor=<% bgcolor %> onload=<% onload %>> ... </body>
```

In this case, the type of the code escape must be `String`.

2.2.7 The API for constructing document nodes

We have seen XML notation for creating (singleton sequences of) element nodes, attribute nodes, and (implicitly) text nodes. All such nodes can be created by accessing the underlying API directly. This notation may be mixed arbitrarily with using the XML notation.

Element nodes for the tag *`name`* may be created with the function

```
name :: WithHTML x cgi a -> WithHTML x cgi a
```

The argument of the tag function is the sequence of child nodes for the element and the result is a singleton sequence with element *name*. A function *name* is predefined for every element name available in XHTML.

For example, the explicit Hello World program might look like this:

```
mainCGI =
  ask  (html
          (do   head (title (text "Hello World"))
                body (h1 (text "Hello World"))))
```

Attribute nodes may also be created with the function

```
attr :: String -> String -> WithHTML x cgi ()
```

The first argument is the attribute name and the second is its value. There are no restrictions on the value. The function takes care of all necessary escaping.

Text and comment nodes may be created with the functions

```
text    :: String -> WithHTML x cgi ()
comment :: String -> WithHTML x cgi ()
```

In both cases, the string argument is the text *as it should appear on the screen.* Both function automatically escape special characters like `<`, `>`, and `&` to avoid unwanted markup in the generated document. The `comment` function may insert extra characters to avoid the premature end of the comment.

2.3 Forms and input fields

The first step in making an application interactive, is defining input facilities. They are provided via the various input fields defined in XHTML. All input fields in an XHTML document must appear inside of a `form` element. Attributes of the latter specify an object (an URL) and a method (`GET` or `POST`) that should be invoked with the parameters entered in the input fields contained in the form.

2.3.1 Creating a form

The function `standardQuery` already wraps a form with a suitable specification of an object and a method around its body document. Usually, no further configuration of the form is required.

`standardQuery` uses the function `makeForm` to construct the form. It behaves like any other element constructor. It should not be used in the body of `standardQuery`.

2.3.2 Input philosophy

Each input element is created by a dedicated function that creates a singleton sequence with the desired element and returns a **handle for the input field**. The handle makes the usual naming of each input element with a string **unnecessary**. Instead, the handle is a value bound to a program variable and there are functions to access its value in the program.

Submission buttons are special input elements that cause the values of all input fields to be gathered and sent to the object and method specified in the form element. WASH attaches a *callback* and some input handles to each submission button. Pressing the button invokes the callback on the specified input handles.

In addition, it is possible to attach a validation to each input element. This validation is performed whenever the callback is invoked (but only for its input handles). The system automatically rejects invalid inputs and redisplays the previous page with the erronous input indicated and explained.

Here is a simple example:

```
mainCGI =
  standardQuery "Your name please"
    <div>
      <p>Enter your name <% iName <- textInputField empty %></p>
      <p><% submit iName showResult empty %></p>
    </div>

showResult iName =
  let name = value iName in
  standardQuery "Hello"
    <#>Hello <%=name%>, how are you?</#>
```

In the example, `textInputField empty` creates a standard textual input field. In place of `empty`, attributes for the input element may be supplied. The notation `iName <-` in the code escape binds the handle to the input field to the variable `iName`. The function `submit` creates a submission button on the web page. Its first parameter, `iName`, is the handle processed by this button. The second parameter, `showResult` is the callback function that is applied to the handle when the button is clicked. The third parameter, `empty`, supplies further attributes to the submission button (none, in this case).

The callback function applies the function `value` to the handle to extract its string value into the variable `name`. Then it constructs and displays an answer page using the input value in a string escape.

Due to typing restrictions, it is not possible to extract the value from a handle before it is passed to a callback function.

Finally, to pass more than one handle to a callback requires the use of special tuple constructors `F2`, `F3`, `F4`, and so on (for two, three, four, etc handles). For example

```
submit (F2 factor rept) multiply (fieldVALUE "START")
```

applies the callback `multiply` to two handles, `factor` and `rept`. The same pattern must be used in the definition of the callback to take the handles apart, again:

```
multiply (F2 factor rept) =
  let n = value factor
      r = value rept
  in  ...
```

Often, a button just serves as a continuation button so that its callback does not require a handle argument. Such a button is created with

```
submit0 :: CGI () -> HTMLField x y ()
```

2.3.3 Unvalidated input fields

The types of all constructors of input fields follow the same pattern:

```
type HTMLField x y inputhandle =
  WithHTML  x CGI () -> WithHTML y CGI inputhandle
```

That is, they take a sequence of attribute nodes as their argument and return a singleton document action for the input element. The latter action yields the input handle of type `inputhandle`. The main difference between the fields is the type of the `inputhandle`.

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

The value returned by the field's handle is a `String`.

```
makeTextarea :: String -> HTMLField x y (InputField String INVALID)
```

This function creates a multi-line textual input box with a preset string specified with the function's argument. Its handle returns a value of type `String`.

```
checkboxInputField :: HTMLField x y (InputField Bool INVALID)
```

A checkbox generates an input field of type `Bool`, indicating whether the box was checked.

```
makeButton :: HTMLField x y (InputField Bool INVALID)
```

Similar to the above, but using a `button` element instead of `input`. In consequence, the argument is not restricted to attributes but nested XHTML elements may be used to specify the surface of the button.

```
fileInputField :: HTMLField x y (InputField FileReference INVALID)
```

The returned handle contains a value of type `FileReference`. Such a value indicates a file that has been uploaded and it contains its temporary location and some information about it (*e.g.*, its mediatype). There are no guarantees about the lifetime of this temporary file, so an application is advised to move the file to a safe location as soon as possible.

The attentive reader may be wondering why the type of each input field contains `INVALID`. The type `INVALID` indicates that the handle is not yet filled with data. Hence, the function `value` (which requires a `VALID` in place of the `INVALID` in the type of the handle) is not applicable to a freshly created handle. Only the `submit` function converts an `INVALID` handle into a `VALID` one. This setup ensures that handles are not read out before they are filled and it also provides a convenient place where additional checks on the input may be performed.

2.3.4 Wrapped input fields

For a few types of input fields WASH provides a more high-level way of specifying them than XHTML. These are radio buttons and selection boxes.

Radio buttons In XHTML, radio buttons are implicitly grouped through sharing a common name. Since WASH does not require naming of input fields, such grouping is not possible. Hence, WASH has an abstract notion of a `radioGroup`, an invisible widget which is created first and to which all buttons of the group are attached later on.

```
radioGroup  :: Read a => WithHTML x CGI (RadioGroup a INVALID)
radioButton :: Show a => RadioGroup a INVALID -> a -> HTMLField x y ()
```

A handle is only created for the radio group. Each individual button only specifies a value of a suitable type `a`. This type `a` may be anything that is `Read`able and `Show`able, for exampe, numbers, strings, etc. However, all buttons of a radio group must yield values of the same type.

Here is an example using radio buttons, where `PayCredit` and `PayTransfer` are two values of an enumerated type:

```
<table>
  <% rg <- radioGroup %>
  <tr><td><%radioButton rg PayCredit empty%></td>
      <td>Pay by Credit Card</td>
  </tr>
  <tr><td><%radioButton rg PayTransfer empty%></td>
      <td>Pay by Bank Transfer</td>
  </tr>
  <tr><td></td>
      <td><% submit rg paymentOption empty %></td>
  </tr>
</table>
```

Selection boxes A simple selection box supplies a choice of one from many items. It is implemented by the function `selectSingle`.

```
selectSingle :: Eq a
             => (a -> String)
             -> Maybe a
             -> [a]
             -> HTMLField x y (InputField a INVALID)
```

The type `a` is the type of values returned from the selection box. It only needs to provide an equality (`Eq a`). The first argument is a function that turns a selection value into a string, for displaying the choices on the screen. The second argument specifies an optional preselection: `Nothing` means no item is preselected, `Just x` means that value `x` is preselected. The third argument specifies the list of all choices. The result is an input field, as before.

Here is an example use:

```
displayCard 0 = "MasterCard"
displayCard 1 = "Visa"
displayCard 2 = "American Express"

<tr><td>My Credit Card is a
        <% ccF <- selectSingle displayCard Nothing [0,1,2] empty %>
    </td>
</tr>
```

There is a shortcut, `selectBounded`, for simple selection boxes when the choice type is a (finite) enumerated type.

Multiple selection boxes are accessible through `selectMultiple`. Their interface is slightly more complicated, but the essence is as with `selectSingle`.

2.3.5 Validated input fields

The textual input fields come in a variant that validates all input and converts it to a specific type. That is, a field can be restricted to only accept numbers, booleans, dates, credit card numbers, or email addresses. Also simple restrictions (like non-emptiness of a field) can be enforced. The validation mechanism is extensible in the sense that new fields with customized restrictions and return types may be specified in the program.

Validated and typed input fields are created by

```
inputField         :: (Reason a, Read a)
                   => HTMLField x y (InputField a INVALID)
passwordInputField :: (Reason a, Read a)
                   => HTMLField x y (InputField a INVALID)
```

with `passwordInputField` supplying the non-echoing variant. Apart from their typing properties they are used in exactly the same way as their unvalidated

variants. In addition, instead of yielding a string value, those handles yield a value of type `a`.

It is important that the callback function that consumes such a typed input handle accesses its value in a way that makes its type obvious. This may require a type specification in the callback function, as the following example demonstrates:

```
mainCGI =
  standardQuery "Multiplication Drill"
    (table
      (do factor <- question "Choose a factor: " 2
          rept   <- question "Number of exercises: " 20
          tr (td (submit (F2 factor rept) multiply (fieldVALUE "START")))))

multiply (F2 factor rept) =
  let n, r :: Int
      n = value factor
      r = value rept
  in  multiplyBy n r 1 [] []
```

Often the accessor functions for the value fixes the type sufficiently. This is the case in the example below for the (predefined) `Email` and `NonEmpty` field types.

```
  <#>
    <table>
      <tr><td>Email Adresse</td>
          <td><% emailH <- inputField (fieldSIZE 40) %></td>
      </tr>
      <tr><td>Vorname</td>
          <td><% firstH <- inputField (fieldSIZE 40) %></td>
      </tr>
      <tr><td>Nachname</td>
          <td><% lastH  <- inputField (fieldSIZE 40) %></td>
      </tr>
      <tr><td><% submit (F3 emailH firstH lastH) collectOne empty %></td>
      </tr>
    </table>
  </#>

collectOne (F3 emailH firstH lastH) =
  let email     = unEmailAddress (value emailH)
      firstname = unNonEmpty (value firstH)
      lastname  = unNonEmpty (value lastH)
  in
```

2.4 Menu-style interaction

Many programs are structured around menus. The interaction with such a program comprises navigation through (nested) menus, selection of a task in the menu, and then returning to the menu, again. A long interaction sequence with this kind of program may lead to an inefficiency in the current implementation of WASH. The implementation stores an "interaction log" of all input values and the results of all I/O operations in an invisible field in the generated Web page. If this log grows too big, the response time may suffer considerably.

To avoid this inefficency, WASH provides two operators:

```
once    :: (Read a, Show a) => CGI a -> CGI a
forever :: CGI () -> CGI ()
```

The operator `once` summarizes a sequence of CGI actions into a single one, which only contributes a single value to the log. A typical use is the following:

```
userInteraction user =
  do once (oneAction user)
     userInteraction user
```

where the function `oneAction` presents the menu and then performs the selected choice. The end of processing the selected choice is indicated in the program by `return ()`. Executing this action returns the `()` through `once`. Example `ex3-2-a.hs` contains a small, but full example.

The above pattern occurs so frequently that it has been abstracted into the function `forever`:

```
userInteraction' user =
  forever (oneAction user)
```

Its implementation is even more efficient than the recursive definition of `userInteraction`.

3 Miscellaneous

This section collects various comments and properties of WASH that do not fite lsewhere.

3.1 Accessing I/O

Since the main thread of execution in a WASH program takes place in a CGI action, the standard Haskell I/O operations are not directly applicable. They need to be wrapped into the operator `io`:

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

The typing of I/O actions suggests that they be used *before* using `ask`, `standardQuery`, or similar operators.

The use of I/O actions with bulky output is possible but strongly discouraged[1]. If at all possible, multiple I/O actions should be gathered to a single action and the value returned by the action should be as small as possible. Sometimes a value can be made smaller by incorporating part of the computation into the I/O action.

For example, a naive implementation of password verification is

```
pwfile <- io (readFile "passwords")
let logged_in = checkpasspw pwfile login passwd
```

where `checkpasspw` returns `True` or `False`. A much more efficient implementation is

```
logged_in <- io (do pwfile <- readFile "passwords"
                    return (checkpasspw pwfile login passwd))
```

3.2 Bulky I/O

Sometimes, it cannot be avoided to have an I/O action return a large amount of data. For example, the results of a database query may be expressed with an I/O action that returns a list of lists of strings where the nested list comprises the tuples selected by the query.

The problem with an I/O action is that the rest of the interaction may depend on the result of the action. This potential dependency requires the current implementation of WASH to store results of I/O actions. In many cases, however, this assumption is overly pessimistic. Often data is only displayed and the rest of the interaction depends on a single tuple at most. Such cases are served best with an abstract table.

An abstract table is created by a variant of the `io` operator:

```
table_io :: IO [[String]] -> CGI AT
```

It converts an I/O action that returns a list of database tuples (encoded as above) into a CGI action that returns an abstract table of type `AT`.

The value stored in an abstract table cannot be read out directly. However, each component in a abstract table can be turned into a text node and thus incorporated into a generated XHTML page. This is accomplished with the functions

```
as_rows :: AT -> Int
as_cols :: AT -> Int
getText :: Monad m => AT -> Int -> Int -> WithHTML x m ()
```

for determining the number of rows and columns and for obtaining a text node that corresponds to an entry in a specific row and column of the table.

Furthermore, there are two kinds of selection facilities that may be used to extract entire rows (tuples) from the table.

[1]The current implementation cannot deal efficiently with bulky output. It works, but things become slow.

We will discuss the group of operations for selecting single rows, the operations for multiple rows work analogously. As with radio buttons (which are used in the implementation of selection groups), we first need to create a selection group using

```
selectionGroup :: WithHTML y CGI (SelectionGroup AR INVALID)
```

Later on, the function

```
selectionButton :: SelectionGroup AR INVALID -> AT -> Int -> HTMLField x y ()
```

takes a selection group, an abstract table, and a valid row number in the table and places a radio button on the page. Clicking the button sets the value of the selection group to the contents of the row associated with the button.

The function

```
selectionDisplay :: SelectionGroup AR INVALID -> AT -> Int
                 -> (WithHTML x CGI () -> [WithHTML x CGI ()] -> WithHTML x CGI a)
                 -> WithHTML x CGI a
```

combines the creation of the button with the extraction of the text nodes from the abstract table. The first three parameters determine the selection group, the abstract table, and the row number. The last parameter is a function that takes first the button and then the list of text nodes corresponding to the contents of the row. The job of this function is to combine these to an XHTML fragment for displaying the row's contents and the button.

Here is an example for using an abstract table.

```
page1 =
  do at <- table_io `some IO action returning [[String]]'
     standardQuery "Haskell Library Modules" $
       do sg <- selectionGroup
          let makeRow row = selectionDisplay sg at row dispRow
              dispRow button texts = tr (do td button
                                            sequence (Prelude.map td texts))
          p $ table (mapM makeRow [0..(as_rows at -1)])
          submit sg page2 empty

page2 sg =
  let row = unAR (value sg) in
  standardQuery "Selected row" (
  do text "You selected "
     text (show row))
```

The function `dispRow` takes the button and a list of texts and builds a row of an XHTML table from it.

3.3 Non-XHTML output

The combinator

```
tell :: CGIOutput a => a -> CGI ()
```

delivers any output that may sensibly be returned by a Web application. The most interesting of these are XHTML elements, strings, and `FileReference`s. Further output options are `Location` and `Status` responses as defined in the CGI standard. The reference manual contains a full listing.

3.4 Defining new types for validated input fields

A type suitable for entering through `inputField` must be a member of the type classes `Reason` and `Read`. The class `Reason` has a single member function:

```
class Reason a where
  reason :: a -> String
```

This member function should return a description of the input syntax for the type. Its value is used to generate the error message and the explanatory text for an input field of this type. To add new instances to `Reason` requires to import module `Fields`.

The class `Read` is predefined in Haskell. To create an instance of `Read a` requires the definition of a function

```
type ReadS a = String -> [(a, String)]
readsPrec :: Int -> ReadS a
```

`ReadS a` is the type of a parser that converts strings into values of type `a`. Such parsers may be created with functions from the module `Utility.SimpleParser`, which provides the usual monadic interface for parsing. Parsers may also be created by hand as the following example (excerpted from the library's source) demonstrates.

```
-- |Non-empty strings.
newtype NonEmpty = NonEmpty { unNonEmpty :: String }

instance Read NonEmpty where
  readsPrec i []  = []
  readsPrec i str = [(NonEmpty str, "")]

instance Reason NonEmpty where
  reason _ = "non empty string"
```

3.5 Character set encoding

Early versions of WASH have ignored this surprisingly tricky issue. The current version assumes that the I/O primitives use values of type `Char` between `\0` and

`\255` to read and write bytes. It generates output in UTF-8 format, whenever that is covered by an applicable standard (XHTML, Email), and it expects the form submissions from the browsers also in UTF-8 encoding.[2]

This choice causes older browsers to choke and, particularly, email using the UTF-8 encoding causes problems with newer email clients.

3.6 Form attributes

The standard setting of the form attributes is as follows.

- `enctype="application/x-www-form-urlencoded"`
- `method="post"`
- `action="`*`full url of the script itself`*`"`
- `accept-charset="UTF-8"` (implicit through the UTF-8 encoding of the document itself)

The presence of an input field for file uploading in the form changes the encoding type to `multipart/form-data`.

3.7 More than one form on a page

This is not necessary with WASH. Grouping your input fields by passing them in groups to the `submit` functions is sufficient.

3.8 Encryption of the interaction state

Each XHTML page delivered by WASH contains a log for reconstructing the current interaction state. In standard operation, this log is only superficially encoded using the Base64 encoding. If it is expected that the log contains sensitive information, then further encoding is advisable.

WASH can be switched to encode logs by providing a key file at

```
/tmp/KEYFILE
```

(This location is configurable in module `CGIConfig`). This file should be a large file with random content. For encryption, WASH generates a random offset into the key file and adds the log bytewise to the contents of the keyfile and to the accumulated sum of the log so far (see functions `encrypt1` and `decrypt1` in module `RawCGIInternal` for details).

The security of this mechanism relies entirely on the key file. If it is too small, its contents is not really random, or the same file is used for too long, then it is easy to break the encryption.

[2]The latter behavior is not fixed in any standard known to the author.

4 Installing and Trouble Shooting

4.1 Compiling a WASH program

This section assumes the following setup.

- All files are in one directory.
- The main program resides in file `Main.hs`.
- Each subsidiary modules reside in file *modname*`.hs`.

Place the following `Makefile` in the directory:

```
HC=             ghc
HCFLAGS=        -package WASHCGI

HMAKE=          $(HC) $(HCFLAGS) --make
RM=             rm -f

HSFILES=   $(wildcard *.hs)

Main: $(HSFILES)
        $(HMAKE) $@

clean:
        $(RM) Main *.o *.hi

%: %.hs
        $(HMAKE) $@

%.o: %.hs
        $(HC) $(HCFLAGS) -c $< -o $@

%.hi: %.o
        @\:
```

Type `gnumake` to compile the program `Main`. It requires

1. the preprocessor `wash2hs` installed somewhere in your search path,
2. the Haskell compiler `ghc` installed somewhere in your path.

The name of the executable program is `Main`. To use it as a CGI program, just copy it into a CGI enabled directory in the document tree of your web server. No further configuration is required, but certain directories need to be readable and writable by the effective user running the CGI programs (see Sec. 4.2).

To have more than one main program in one directory, each program needs to have its own main module. Each main module resides in a file by itself with an arbitrary file name (and extension `.hs`), for example `MyMod.hs`. The makefile above is equipped to compile such a program by entering

```
gnumake MyMod
```

This creates an executable named `MyMod` which may then be passed to the Web server in the usual way.

4.2 Files, directories, and permissions

WASHCGI uses the following directories, which can be modified in module `CGIConfig`. Wrong permissions for these directories is a frequent source of problems.

```
/tmp/Images/            (r/w)          if CGIGraphics is used
                                               temporary files
/tmp/Frames/            (r/w)          if frames are used (experimental)
                                               temporary files
/tmp/Persistent2/       (r/w)          if module Persistent2 is used
                                               *PERSISTENT FILES*
/tmp/REGISTRY/          (r/w)          *always*
                                               temporary files
/tmp/KEYFILE            r              if log encryption is used
                                               provided by user
```

If running under a real user (*e.g.*, when started from the command line), then the user's home directory is prefixed to the path name.

The following programs are assumed.

```
/usr/X11R6/bin/ppmtogif (x)            if CGIGraphics used
/usr/X11R6/bin/pbmtext  (x)            if CGIGraphics used
/usr/X11R6/bin/pnmcrop  (x)            if CGIGraphics used
/usr/X11R6/bin/pbmtopgm (x)            if CGIGraphics used
/usr/X11R6/bin/pgmtoppm (x)            if CGIGraphics used

/bin/cat                (x)            for displaying files and FileReferences

/usr/sbin/sendmail      (x)            if WashMail used for sending mail
```