J. Functional Programming **10** (1): 1–18, January 2000. Printed in the United Kingdom

Server side web scripting in Haskell

ERIK MEIJER

Department of Computer Science, Utrecht University,
PO Box 80089, 3508 TB Utrecht, The Netherlands
(*e-mail:* erik@cs.uu.nl)

Abstract

The Common Gateway Interface (CGI) for generating dynamic documents on web servers imposes much accidental complexity on the programmer. The Haskell/CGI library documented in this paper hides all this unpleasantness by using the common sense 'design pattern' of separating model and presentation. Low-level query string requests are represented by association lists, and primitive HTTP responses are easily constructed using a set of HTML generating combinators. The CGI programmer only needs to write a worker function that maps an abstract request into an abstract response. A (higher-order) wrapper function then transmutes the worker into a real low-level CGI script that deals with the exact format of concrete requests and responses as required by the CGI standard.

Capsule Review

This paper proposes a method for using Haskell as a CGI scripting language. The system allows the programmer to avoid the messy details of the various protocols and data representations of HTTP and HTML, by creating abstract interfaces that are much higher level and easier to use. The ideas are novel and the methods are sound, representing a nice application of functional programming techniques. The system has been implemented and is available as a Haskell library for Hugs. Overall the system promises to be a useful and practical alternative to using Perl or other language for CGI scripting.

1 Introduction

Many documents on the web are *static*, i.e. they are the same each time the server returns them in response to a client's request. *Dynamic* documents, on the other hand, are generated on-the-fly by running an external CGI script on the server.

The script receives the information embedded in the client's request from the server via environment variables and the standard input, and communicates its result back to the server via the standard output. Hence, any programming language that can access environment variables, read from the standard input, and write to the standard output, is in principle suited for writing CGI scripts (Khare, 1997). At present, most CGI scripts are written in Perl (Wall *et al*., 1996) because it has rich features for manipulating text (regular expressions).

The *Common Gateway Interface* (CGI) specifies exactly *how* the request and response are passed between server and script, and as such presents script writers

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

with a lot of accidental complexity. This paper describes a library for writing CGI scripts in the lazy functional language Haskell (Peyton Jones & Hughes, 1999). The library tries to make programming server-side scripts as simple as possible (section 3). Input and output coding of CGI scripts is handled by a *wrapper*, a higher-order function that decodes the incoming request, passes it to the *worker*, and then encodes the worker's response. Thus, the main part of the application, the worker function inside the wrapper, need not be at all concerned with the idiosyncrasies of the CGI protocol. Requests are parsed into association lists, and HTML responses are represented by a simple tree type which comes with a set of combinators from which complicated HTML pages (Ragget *et al.*, 1997) can be assembled easily. Most CGI scripts written in Perl do not fully separate model and presentation; requests are parsed into Perl associative arrays, but scripts usually print raw HTML string directly on the standard output.

Before we start writing scripts (sections 4–6) and explain how to maintain state across different invocations of a script (section 7), we briefly sketch the architecture of the Web (section 2) and the underpinnings of our library (section 3). Finally, we reflect on what we learned from this exercise, discuss some alternative designs (section 8), and give a few pointers to relevant literature (section 9).

2 The World Wide Web

The World Wide Web is an instance of the familiar *client-server* model of computation. In this model a *server* provides resources to potentially many *clients*. Server and clients communicate via the *HTTP protocol* (Fielding *et al.*, 1997). The client sends a *request* to the server, to which the server replies by sending a *response* back to the client. Once such a cycle is completed, the client and server are no longer in contact; the HTTP protocol is *stateless*. Section 7 discusses several ways to maintain state across invocations.

The HTTP protocol transmits all requests and responses as plain ASCII text, which makes it possible to play client manually. To get a better feeling for the HTTP protocol, we will establish a connection with an HTTP server by telnetting on port 80:

```
% telnet www.cse.ogi.edu 80
Trying ....
Connected to www.cse.ogi.edu.
Escape character is '^]'
```

Now we can type in an HTTP request, in exactly the right syntax and terminated by an empty line

```
GET /index.html HTTP/1.0
```

If we did not make any typos, the server will eventually respond with a HTTP reply (in this case it contains an HTML document) and close the connection:

```
HTTP/1.0 200 OK
```

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
Content-type: text/html

<HTML>
....
</HTML>
Connection closed by foreign host.
```

If the world consisted of only computer scientists, we would still be surfing the web in this way. Fortunately, the physists at CERN recognized that communicating with an HTTP server using a tty interface is rather tedious, and instead is best done via a graphical user interface. Contemporary web browsers such as *Netscape Navigator* do exactly this. They take an HTML document with embedded requests, or *hyperlinks*, and render it on the user's screen.

HTML is a domain specific language for programming the client side of the HTTP protocol, and a browser is an interpreter for this language. Requests for inline images (indicated by the `<IMG SRC="...">` tag) are evaluated eagerly and displayed inline. Requests for other pages (indicated by the `<A HREF="...">` tag) are evaluated lazily, i.e. only when the user clicks on them. When the user clicks on the visual rendering of the link `<A HREF="www.cse.ogi.edu">OGI CSE Homepage</A>`, the browser connects to the server `www.cse.edu.edu` and issues the same request `GET /index.html HTTP/1.0` as we did manually. The browser then replaces the current document by the new one.

The client sees no difference between hyperlinks to static or dynamic documents. Links to CGI scripts are the same as any other URL. For example, when the following (hypothetical) document is requested

```
http://www.cse.ogi.edu/~erik/cgi-bin/helloHTML.cgi
```

the "Hello World!" script of section 5 is executed and returns the response:

```
HTTP/1.0 200 OK
Content-type: text/html

<HTML>
<H1>Hello World</H1>
</HTML>
```

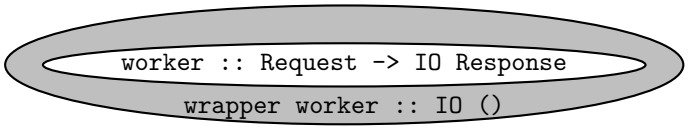
The client couldn't care less how the server obtains this piece of HTML.

There are two common ways for a *server* to know that it has to execute a script and not return its content as a static document. The first option is that scripts reside in a special directory, usually called `cgi-bin`, and the server will treat any URL involving `cgi-bin` as an executable script. The second possibility is that the server uses a particular filename extension, usually `.cgi`, to distinguish scripts from static documents. In this case, scripts can be stored anywhere within the directory hierarchy of the server.

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

3 Programming CGI scripts

The architecture of our CGI library provides the programmer with the illusion of an idealized HTTP client (the *wrapper*), which interacts with an idealized HTTP server (the *worker*) to be supplied by the programmer:



All the low-level details of the communication between the actual HTTP server and the script are handled by the wrapper function. As far as the server is concerned, the wrapper is a standard CGI script. As far as the wrapper is concerned, the worker is an abstract server, a function that produces a result of type `Response` from an argument of type `Request`. Functional programmers might recognize the wrapper as a monadic *representation changer* (Hutton & Meijer, 1996), object oriented programmers will say that we use a variation of the *proxy* design pattern (Gamma *et al*., 1994), Visual Basic programmers will see an instance of the *n*-tier client-server model, and Smalltalk programmers will note that it is an extremely primitive use of MVC. Whatever you want to call it, it is just a matter of good programming practice.

The work done by the wrapper is the same for all CGI scripts; the interesting bits are performed by the worker. The wrapper decodes a low-level HTTP request into an abstract value of type `Request`, passes it to the worker to obtain an abstract value of type `Response`, and encodes this back into a low-level HTTP response. If anything goes wrong we return an error status code depending on the error that occured (section 3.4):

```
wrapper :: (Request -> IO Response) -> IO ()
wrapper = \worker ->
 do{ request <- getRequest
   ; response <- worker request
   ; putResponse response
   }
 `catch`
   (\ioerror -> do{ putResponse (status ioerror) })
```

The worker function of type `Request -> IO Response` need not be concerned with any of the gory details of the CGI standard at all, it only has to produce an abstract response when given an abstract request. Most other CGI libraries do not decouple abstract and concrete requests and responses. To understand the logic of a CGI script, we can study the worker function in isolation. This is impossible if the decoding and encoding of requests and responses is intertwined with the actual computation of responses from requests.

Because the wrapper function abstracts all details of the CGI standard from the worker, it is easy to adapt to a platform such as Windows, which prefers a nonstandard interaction (ISAPI or ASP) between servers and scripts. In that case,

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

we only have to change the wrapper function once, instead of having to modify *every* worker (= CGI script) we have written.

3.1 Requests and responses

The set of abstract HTTP requests is modelled by the data type `Request`. A client can either request to retrieve a document using `GET`, or deposit some Mime content (section 3.2) using `POST`:

```
data Request = GET QueryString | POST Mime
```

The set of abstract HTTP responses is modelled by the data type `Response`. The server can return either some Mime content (using `Content`), a redirection to another location (using `Location`), or an error message (using `Status`):

```
data Response = Content Mime | Location Url | Status Code Reason
```

The types `Request` and `Response` are a sound, but not complete abstract syntax for concrete HTTP requests and responses, they are adequate for most of the CGI scripts that return parsed HTTP headers.

Besides the constructor functions for the data types `Request` and `Response`, we only assume that we can read requests from the standard input via function `getRequest :: IO Request` (section 3.3) and write responses to the standard output via function `putResponse :: Response -> IO ()` (section 3.4).

3.2 MIME

Mime types are the standard way of 'typing' binary data transmitted over the internet (Borenstein & Freed, 1993). Examples include plain text `text/plain` (section 4), url-encoded query strings `x-application/url-encoded` (section 6), HTML documents `text/html` (section 5), GIF pictures `image/gif`, and MPEG movies `video/mpeg`. In Haskell we represent Mime types by an algebraic data type with constructors that hold the Haskell representations of the respective data format (section 8 discusses an alternative representation using type classes):

```
data Mime
  = ...
  | TextPlain  String
  | TextHtml   HTML
  | UrlEncoded Query
  | ...
```

Section 5 defines the abstract Haskell representation `HTML` of HTML document of Mime type `text/html` while section 6 defines the Haskell representation `Query` of query strings of Mime type `x-application/url-encoded`.

To implement functions `getRequest` and `putResponse`, we must be able to read Mime types from the standard input and write them to the standard output, hence we also need functions `getMime` and `putMime`. Function `mimeType` returns the Mime

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

type description of a value of type `Mime`, for instance `mimeType (TextPlain "...")` yields `"text/plain"`:

```
getMime  :: IO Mime
putMime  :: Mime -> IO ()
mimeType :: Mime -> String
```

To implement functions `getMime` (section 3.2) and `putMime`, we need to read and show the various alternatives (`TextPlain`, `TextHtml`, `UrlEncoded`) of data type `Mime`:

```
showHTML  :: HTML -> String
...
readQuery :: String -> Query
```

We won't discuss the function `showHTML` in this paper, but it is worth mentioning that `showHTML` nicely formats its output in human readable form using John Hughes' pretty printing combinators (Hughes, 1995), which is extremely convenient for debugging HTML-generating programs. Section 8 explains why we don't use Haskell's `Show` and `Read` classes to overload `showHTML` and `readQuery`.

3.3 Decoding requests

Recall from section 2 that the server passes data about the HTTP request to a script via environment variables and the standard input. The four most important environment variables that the HTTP server sets for the wrapper are:

`REQUEST_METHOD` The method field of the client's request
`QUERY_STRING` The query string segment of a `GET` request.
`CONTENT_LENGTH` The number of bytes of the body of a `POST` request.
`CONTENT_TYPE` The MIME type of the body of a `POST` request.

By inspecting these variables, the wrapper can retrieve and decode the incoming request.

Function `getRequest` does a simple case analysis on the environment variable `REQUEST_METHOD` to find out what sort of request has been made by the client:

```
getRequest :: IO Request
getRequest =
 do{ method <- getEnv "REQUEST_METHOD"
   ; case method of
       { "GET"     -> getGET
       ; "POST"    -> getPOST
       ; otherwise -> userError "501 Not Implemented"
       }
   }
```

If `REQUEST_METHOD` equals `"GET"`, then the environment variable `QUERY_STRING` contains an url-encoded query string, which we decode using function `readQuery` (section 6):

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
getGET :: IO Request
getGET =
 do{ query <- getEnv "QUERY_STRING"
   ; return $ GET (readQuery query)
   }
```

If `REQUEST_METHOD` equals `"POST"`, then function `getMime` decodes the Mime content of the request from the standard input:

```
getPOST :: IO Request
getPOST =
 do{ mime <- getMime
   ; return $ POST mime
   }
```

The function `getMime` takes the first `CONTENT_LENGTH` bytes of the standard input, and then reads a value of Mime type `CONTENT_TYPE`.

```
getMime :: IO Mime
getMime =
  do{ contentLength <- getEnv "CONTENT_LENGTH"
    ; stdin <- getContents
    ; let mime = take (read contentLength) stdin
    ; contentType <- getEnv "CONTENT_TYPE"
    ; case contentType of
        { ...
        ; "application/x-url-encoded"
            -> do{ return $ UrlEncoded (readQuery mime) }
        ; ....
        }
    }
```

This code shows nicely how Mime types are used to dynamically type data transmitted as raw byte streams between client and server. Note that we use the overloaded function `read` here to coerce the `contentLength` string into a number.

3.4 Encoding responses

The function `putResponse` puts a response to the standard output in the exact format that is required by the CGI standard for a *parsed header* response; a header and an optional body, separated by a blank line:

```
putResponse :: Response -> IO ()
putResponse = \response ->
  case response of
    { Content mime
        -> do{ putStr $ "Content-type: "++mimeType mime++"\n\n"
             ; putMime mime
```

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
            }
    ; Location url
        -> do{ putStr $ "Location: "++url++"\n\n" }
    ; Status code reason
        -> do{ putStr $ "Status: "++code++" "++ reason++"\n\n" }
    }
```

Function `putMime` then prints a value of type `Mime` as a raw byte stream on the standard output:

```
putMime :: Mime -> IO ()
putMime = \mime ->
  case mime of
    { TextHtml html    -> do{ putStr $ showHTML html }
    ; TextPlain s      -> do{ putStr $ s }
    ; UrlEncoded query -> do{ putStr $ showQuery query }
    }
```

4 MIME type `text/plain`

4.1 Hello world!

Using Mime type `text/plain` we can write our first CGI script, the cut-and-dried "Hello World!" program. Remember from section 3 that we are only required to write a worker function of type `Request -> IO Response`. In this case, the worker function ignores its request argument, and greets the user in a rather static way:

```
helloWorld :: IO ()
helloWorld = wrapper $ \request ->
 do{ return $ Content (TextPlain "Hello World!") }
```

4.2 Greetings

One of the environment variables that is passed to a CGI script is `REMOTE_HOST`, which contains the fully qualified domain name of the client that has sent the request to the server. An example of a fully qualified domain name is `www.cse.ogi.edu`. The top-level domain, the name after the rightmost dot (`edu` in the example), gives us some information about the country or type of organization where the client is located (`edu` indicates a university or educational institution in the US), and we will use that to generate a personalized greeting:

```
type Domain = String

greeting :: Domain -> String
greeting = \domain ->
  case (top domain) of
    { "edu"     -> "Hi there!"
```

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
    ; "com"     -> "Can you find everything OK today?"
    ; "nl"      -> "Hoi, hoe gaat het?"
    ; "uk"      -> "Good afternoon!"
    ; otherwise -> "Hello!"
    }
```

We can extract the top-level domain by first splitting a fully qualified domain name at every `'.'` and then taking the last element of the resulting list:

```
top :: Domain -> Domain
top = last.split (== '.')
```

The standard function `split (== '.') "www.cse.ogi.edu"` returns the list `["www", "cse", "ogi", "edu"]`.

The script proper is straightforward. We lookup the `REMOTE_HOST` variable in the environment, and return a greeting in the appropriate language for the given domain of the client:

```
helloWorld :: IO ()
helloWorld = wrapper $ \query ->
  do{ host <- getEnv "REMOTE_HOST"
    ; let message = greetings host
    ; return $ Content (TextPlain message)
    }
```

5 MIME type `text/html`

In practice, most CGI scripts return an HTML page of Mime type `text/html`. Typically, CGI scripts written in C or Perl print raw, concrete HTML directly on the standard output. In Perl, the HTML variant of the "Hello World!" CGI script would look something like:

```
print << EOF;
Content-type: text/html

<HTML>
<HEAD>
  <TITLE>Hello world in Perl</TITLE>
</HEAD>
<BODY>
  <H1>Hello, world!</H1>
</BODY>
</HTML>
EOF
```

This is not very flexible and rather error-prone, especially when we want to generate more complicated HTML pages. Just for comparison, using the combinators from Table 1, the script:

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
helloHTML =
  let { hi = page "Hello world in Haskell" [h 1 "Hello, world!"] }
  in cgi $ \query -> do{ return hi }
```

generates the same HTML content as the Perl script above, modulo the page title.

5.1 Modelling HTML

An HTML document consists of a number of nested *elements* such as headers (page title, section headings), paragraphs, lists (ordered, unordered), logical markup (citation, computer code), visual markup (italic, bold), hypertext links, images, fill-in forms, etc.

Every HTML element is delimited by begin- and end-*tags* of the form <*tag*> respectively </*tag*>. Most elements take (optional) attributes, which are given as *name* = *value* pairs in the start tag. Boolean attributes can be set by just giving their name, without a value. Tags and attribute names in HTML are not case sensitive, so for example `<HTML>` is equivalent to `<html>` or `<HtMl>`. Some tags such as `<HR>` and `<BR>` do not have a closing tag.

We represent HTML by a simple universal tree type. An HTML value is either an ordinary text string, or a complex element with a tag, a list of attributes, and an embedded list of HTML values:

```
data HTML
  = Text    String
  | Element Tag [(Name,Value)] [HTML]
```

For simplicity, all HTML related types such as `Tag`, `Name`, `Value`, etc. are synonyms for `String`. We don't expect programmers to use the concrete constructors of the HTML data type however. Instead we provide a set of combinators from which complicated HTML pages can be assembled easily.

The basic HTML combinators `set`, `attributedElement`, `element`, and `text` provide an abstract interface to construct values of type `HTML`:

```
set :: [(Name,Value)] -> (HTML -> HTML)
attributedElement :: Tag -> [(Name,Value)] -> [HTML] -> HTML
element :: Tag -> [HTML] -> HTML
text :: String -> HTML
```

By hiding the construction of concrete HTML elements we can always decide to change the representation of the HTML data type.

The set of combinators in Table 1 capture patterns that we have found convenient for generating complex HTML.

The `span` combinator is especially useful as it is the only combinator that turns a list of HTML elements into a single HTML element in a way that does not influence the rendering of the resulting HTML on the user's screen.

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

Table 1. *Some compound HTML combinators*

```
page :: String -> [HTML] -> HTML

h :: Int -> String -> HTML
p :: [HTML] -> HTML

href :: URL -> [HTML] -> HTML
name :: String -> [HTML] -> HTML

table :: [[ [HTML] ]] -> HTML

ul, ol :: [[HTML]] -> HTML
dl :: [(String,[HTML])] -> HTML

span, div :: [HTML] -> HTML

hr, br :: HTML
```

5.2 Printing the environment

Our first script that employs HTML combinators, maps the list of environment variables that are set by the server, such as

```
[ ("SERVER_NAME","www.cse.ogi.edu")
, ("REQUEST_METHOD","GET")
, ..
]
```

into an HTML definition list, i.e.

```
dl [ ("SERVER_NAME",   [text "www.cse.ogi.edu"])
   , ("REQUEST_METHOD",[text "GET"])
   , ...
   ]
```

As we see from the example, we have to construct a pair `(dt,[text dd])` for every `(dt,dd)` pair in the environment and then wrap the whole resulting list in a definition list:

```
envPage = \env ->
  page "Environment"
    [ h 1 "Environment"
    , dl (map (\(dt,dd) -> (dt,[text dd])) env)
    ]
```

The complete script first gets the list of all environment variables using function `getWholeEnv`, and then returns the requested HTML page:

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
envPassed :: IO ()
envPassed = wrapper $ \query ->
  do{ env <- getWholeEnv
    ; return $ Content (HTML (envPage env))
    }
```

6 MIME type `application/x-url-encoded`

When surfing the web, everyone has encountered strangely encoded query strings such as

```
http://www.altavista.digital.com/cgi-bin/query
?pg=q&what=web&kl=XX
&q=haskell%2B%22cgi+programming%22
&search.x=34&search.y=7
```

In general, an url-encoded query string of Mime type `application/x-url-encoded` consist of a sequence of zero or more url-encoded *name*=*value* pairs separated by ampersands `&`:

$$query \quad ::= \quad [name{=}value\{\&name{=}value\}]$$

We model query strings by a simple association list of (name,value) pairs (remember that `Name` and `Value` are just synonyms for `String`):

```
type Query = [(Name, Value)]
```

The grammar for query strings is readily transliterated into a parser using standard parser (monadic) combinators (Hutton & Meijer, 1998):

```
readQuery :: String -> Query
readQuery = parse query []

query :: Parser Query
query =
  do{ name <- urlEncoded; string "="; value <- urlEncoded
    ; return (name,value)
    } `sepby` (string "&")
```

Names and values are url-encoded, which the parser will decode:

```
urlEncoded :: Parser String
urlEncoded =
  many (alphanum ++ extra ++ safe ++ space ++ hexencoded)
```

Alphanumeric characters and ‘safe’ and ‘special’ characters are not encoded:

```
extra :: Parser Char              safe :: Parser Char
extra = sat (`elem` "@!*'(),")    safe = sat (`elem` "$-_.")
```

Spaces “ ” are encoded by plus signs “+”:

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

Table 2. *Widget combinators*

```
form      :: URL -> [HTML] -> HTML
post, get :: [HTML] -> HTML

checkbox :: Name -> HTML

radio :: (Name,Value) -> HTML
menu  :: (Name,[Value]) -> HTML

textfield :: (Name,Value) -> HTML
textarea  :: (Int,Int) -> (Name,Value) -> HTML

hidden :: (Name,Value) -> HTML

button :: (Name,Value) -> HTML
reset  :: Value -> HTML
```

```
space :: Parser Char
space = do{ char '+' ; return ' '}
```

Nonalphanumeric characters such as '%' are hex-encoded via an escape sequence that consists of a percent character % followed by two hexadecimal digits, for example the hex-encoding of % itself is %25.

```
hexencoded :: Parser Char
hexencoded =
  do{ char '%'; d1 <- hexit; d2 <- hexit
    ; return $ chr (readHex [d1,d2])
    }
```

6.1 *Generating* `application/x-url-encoded` *data with forms*

Up to now, none of our scripts have really used their request argument. In fact, we don't even know how to create query strings on the client! HTML forms give users a way to dynamically generate `GET` or `POST` request with varying query strings to provide true interaction between client and server. An HTML form can contain standard GUI widgets such as text-fields, various kinds of buttons, menus, etc. Table 2 gives a useful interface to HTML forms in Haskell.

Most widgets take a (name,value)-argument whose given initial value can be changed interactively by the user. When the user clicks on a submit button or enters a value in a single line textfield all the (name,value)-pairs of the form are url-encoded and combined into a query string. This query is then included in the request which the client issues to the server.

The first argument of the `form` combinator specifies an explicit URL to which the query is submitted. The `post` and `get` combinators submit the form to the URL of the page that contains the form itself and thus make scripts location independent.

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

6.2 Specializing the wrapper for HTML generating forms

In practice, most HTTP requests contain url-encoded data and most responses HTML documents. We can capture this special case by providing a specialized wrapper function `cgi :: (Query -> IO HTML) -> IO ()` that coerces a simplified worker function into a proper worker as expected by function `wrapper`:

```
cgi :: (Query -> IO HTML) -> IO ()
cgi = \script -> wrapper $ \request ->
  do{ html <- script (request2query request)
    ; return $ Content (HTML html)
    }
```

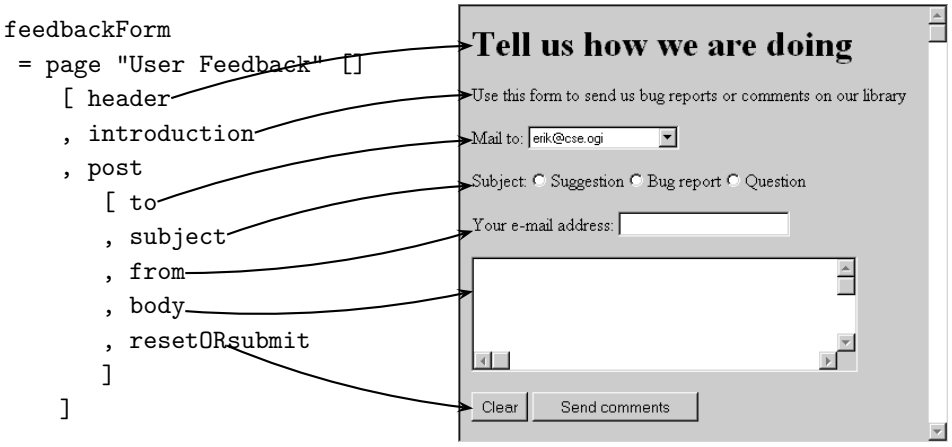
Function `request2query` extracts the query string from either a `GET` or `POST` request:

```
request2query :: Request -> Query
request2query = \request
  case request of
    { GET query                -> query
    ; POST (UrlEncoded query) -> query
    ; POST (_)                 -> []
    }
```

At this point we want to stress once more that in order to write a CGI script, we only have to provide a function of `Query -> IO HTML` and everything else is taken care of by the various wrapper functions; CGI scripting can hardly be more convenient than this.

6.3 Feedbackform

As an example application of HTML forms, we sketch a design of a script that processes user feedback:



The script first parses its input into a mail message using function `checkMsg :: Query -> Maybe MailMsg`. If this fails it just returns the original form so that the

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

user can try again. Otherwise, it mails the message using function `sendMail :: MailMsg -> IO ()` and returns an acknowledgment form.

```
feedback = cgi $ \query ->
  case checkMsg query of
    { Nothing  -> do{ return feedbackForm }
    ; Just msg -> do{ sendMail msg; return (acknowledgeForm msg) }
    }
```

Returning an initial page whenever the user input is incorrect is a widely used technique for CGI scripts.

7 Maintaining state

The fact that the HTTP protocol is stateless, presents a nasty problem for CGI programmers. If an application, such as an electronic shopping cart, requires multiple interactions between client and server, we must arrange somehow that intermediate state persists across these interactions.

Ideally, the underlying operating system or run-time system should implement a stateful protocol on top of HTTP. In principle there is no difference between standard IO where the 'server' program issues responses to the operating system using

```
putChar :: Char -> IO ()
```

and waits for requests from the operating system using

```
getChar :: IO Char
```

and the situation where the server issues responses to the client using

```
putResponse :: Response -> IO ()
```

and waits for requests from the client using

```
getRequest :: IO Response
```

Languages such as Mawl (Ladd & Ramming, 1995), or the arrow-combinators CGI library of John Hughes (Hughes, 1998) take this route, but such sophisticated techniques are outside the scope of this paper.

The straightforward solution is to encode the state in the response returned to the client and decode it from the subsequent request. This can be the complete state, or only part of the state (for instance a session key) from which the rest of the state, which is kept on the server, can be retrieved.

7.1 *<INPUT TYPE="hidden">*

There several commonly used tricks to encode state in responses and requests. The simplest one is to use hidden fields such as `hidden ("state",show state)` to record the state information `state` in the page returned to the client. Then, on the

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

next request, the old state is available by doing a `read (lookup "state")` on the query string. A very similar technique is to append state as pathinfo on the URL that will be used by the client to do the next request. In this case, the old state is available in the environment variable `PATH_INFO`.

7.2 Cookies

The disadvantage of these two methods is that the state is tightly coupled to the outgoing response and the subsequent request. This makes it hard to maintain state over multiple sessions, which can be useful, for instance, to avoid having users to register themselves over and over when they visit a particular site. In such a situation, we can use *cookies*. Cookies are simple name-value pairs that are transmitted from the server to the client with the response (Krisol & Montulli, 1997). The browser stores any cookies it receives on the client, and whenever it will perform another request on the server, it will add the cookies to the outgoing request. We provide two basic functions to set and retrieve cookies as global variables within a CGI script:

```
setCookie :: (Read a, Show a) => Name -> a -> IO ()
getCookie :: (Read a, Show a) => Name -> IO a
```

Function `getCookie name` raises an IO exception when there is no cookie named `name`. In reality, cookies have more attributes such as `domain`, `path`, `expires`, and `secure`, but we do not need those for most common scripts.

Since cookies are transmitted in HTTP requests and responses, we can easily use the plain `getRequest` and `putResponse` functions as hooks to get and put cookies into the communication stream between client and server.

8 Lessons learned

Polishing the Haskell/CGI library took surprisingly long. Originally, we designed the library as an exercise for a compilers course, but along the way we explored many different design alternatives. For instance, we experimented with a strongly typed representation for HTML terms that reflects the HTML DTD more closely. In the end, however, we decided to take simplicity (for both the user as well as for ourselves) as the leading design principle. In this section, we reflect on an alternative representation for Mime types using type classes that we rejected for exactly this reason.

Experienced Haskell programmers know the duality between algebraic data types and type classes. For example, instead of defining an algebraic type `Mime` with alternatives for each individual Mime type such as `String`, `HTML`, `Query`, etc. as we did in section 3.2:

```
data Mime
 = ...
 | TextPlain  String
 | TextHtml   HTML
```

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

```
  | UrlEncoded Query
  | ...
```

and then using a big case statement inside function `putMime :: Mime -> IO ()` and wrapping the results of `getMime :: IO Mime` inside constructors, we could have defined a type class `Mime` with two members `getMime :: Mime a => IO a` and `putMime :: Mime a => a -> IO ()` as follows

```
class Mime a where { putMime :: a -> IO (); getMime :: IO a }
```

and make `String`, `HTML` and `Query` instances of class `Mime`.

At first sight, overloading seems to have the advantage that it is more extensible, we can always define new instances of class `Mime` such as GIF images, whereas in the non-overloaded style, we have to fix type `Mime` once and for all, otherwise all functions that pattern-match on values of type `Mime` break. But as usual life is not as easy as it looks.

Haskell style overloading is a powerful tool, but it can be dangerous in the hands of inexperienced programmers. For example many CGI scripts gave rise to ambiguous types such as `Mime a => IO ()`, akin to the ambiguity in the expression `(show.read) :: (Show a, Read a) => String -> String`.

Moreover, to make overloading really convenient we have to use a trick to simulate overlapping instances that is also used in Haskell's standard `Show` and `Read` classes. For example, to be able to use `putMime` on both `String = [Char]` and `Query = [(Name,Value)]` without making them abstract, we have to add two helper methods `putMimeList :: Mime a => [a] -> IO ()` and `getMimeList :: Mime a => IO [a]` to class `Mime` which will be used to overlap the instances for `String` and `Query`. But this also obliterates extensibility, since users might have to add members to the `Mime` class when they want introduce new Mime types.

9 References and further reading

Much of the material on web related technology is not available in traditional printed media such as conference proceedings, journals and books, and URLs have the dubious reputation of being pretty volatile so the URLs that we mention here might be musty by the time you read this paper. Moreover, it is a truly daunting task to find the needle you are looking for in the enormous haystack of documents that search-engines return. In the end, we found that the best sources of information are available on the W3C website `http://www.w3.org` and on the Internet-Drafts shadow directories such as `ftp://ftp.isi.edu`. The RFC-project page for the CGI standard is `http://web.golux.com/coar/cgi/`. Most commercial books are popularized versions of the online draft documents. The Haskell/CGI library is part of the standard Hugs distribution, which can be downloaded from `www.haskell.org`.

https://doi.org/10.1017/S0956796899003561 Published online by Cambridge University Press

Acknowledgements

Joost van Dijk contributed much to the first iteration of Haskell/CGI. Many thanks to Jim Hook, Tim Sheard, and Daan Leijen for reading draft versions of these notes, Byron Cook for inviting me to talk about Haskell/CGI at the CISE summer school, and Dino Oliva for convincing me to drop the all too clever use of overloading. Special thanks to Phil Wadler who encouraged me to write this paper, and to Simon Peyton Jones and two anonymous JFP referees for suggesting some major restructuring and pointing out numerous weakness in both the model and presentation of this paper. Last but not least, I thank the Nottingham and Yale Hugs implementors for creating the productive soil in which the library grew and for including it as one of the demos in the standard distribution.

A large part of this work was done during the author's sabbatical with the Pacsoft group at OGI funded by a contract with US Air Force Material Command (F19628-93-C- 0069).

References

Borenstein, N. and Freed, N. (1993) *Mime (Multipurpose Internet Mail Extensions) part one: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. RFC1521.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and Lee, T. (1997) *Hypertext Transfer Protocol – HTTP/1.1*. `http://www.w3.org/Protocols/rfc2068/rfc2068`.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hughes, J. (1995) The Design of a Pretty-Printing Library. *Advanced Functional Programming: Lecture Notes in Computer Science 925*. Springer-Verlag.

Hughes, J. (1998) *Generalising Monads to Arrows*. Submitted.

Hutton, G. and Meijer, E. (1996) Back to Basics: Deriving Representation Changers Functionally. *J. Functional Programming*, **6**(1).

Hutton, G. and Meijer, E. (1998) Monadic Parsing in Haskell. *J. Functional Programming*, **8**(4).

Khare, R. (ed) (1997) *Scripting Languages: Automating the Web*. World Wide Web Journal, vol. 2, no. 2. O'Reilly & Associates, Inc.

Krisol, D. M. and Montulli, L. (1997) *HTTP State Management Mechanisms*. `http://www.w3.org/Protocols/rfc2109/rfc2109`.

Ladd, D. A. and Ramming, J. C. (1995) Programming the Web: An Application-Oriented Language for Hypermedia Services. *4th Int. World Wide Web Conf*.

Peyton Jones, S. and Hughes, J. (eds) (1999) *Report on the Language Haskell'98*. `htpp://www.haskell.org/report`.

Ragget, D., Le Hors, A. and Jacobs, I. (1997) *HTML 4.0 Specification*. `http://www.w3.org/TR/REC-html40`.

Wall, L., Christiansen, T. and Schwartz, R. L. (1996) *Programming Perl (2nd ed)*. O'Reilly & Associates.