Functional Web Programming

Peter Thiemann
Universität Freiburg
Web Programming

• Early Web pages:
  static, contents of files transported over the network

• Today’s Web pages:
  – highly dynamic
  – composed from document templates, database accesses, computed elements
  – parameterized wrt. language, image quality, user profiles, …

⇒ must be programmed
  – either on client-side (applets, JavaScript, VB, …)
  – or on server-side (SSI, CGI, NSAPI, ISAPI, Servlets, JSP, …)
The WASH/CGI Approach

- Server-side Web scripting
- Embedded DSL hosted by Haskell
- Based on CGI (portability)
- Raw CGI functionality accessible
- Advanced high-level functionality
1 Preliminaries

1.1 Definitions

- program: defines a number of values (possibly functions)

- \( v = e \)
  
  *define* the value of variable \( v \) as the value of expression \( e \)

- \( f \ v_1 \ldots v_n = e \)

  *define* the function \( f \) which takes \( n \) arguments; expression \( e \) is the body of the function

- let definitions in e

  establishes *definitions* local to expression \( e \)

- e where definitions

  establishes *definitions* local to expression \( e \)
1.2 Types

- \( v :: t \)
  a type signature; asserts that the value of variable \( v \) has type \( t \)

- Built-in Types
  - \( \text{Int} \) integers
  - \( \text{Char} \) characters
  - \([t]\) lists of value of type \( t \)
  - \( \text{String} \) lists of characters
  - \( t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \) functions that expect \( n \) arguments of type \( t_1, \ldots, t_n \) and return a result of type \( t \)
  - \( \text{IO } t \) an I/O action that returns a result of type \( t \) (later)
2 Generating Web Pages

• Webpages-as-text is not appropriate
  – phase errors (headers, main message)
  – structural errors (well-formedness, validity)
  – requires too much low-level knowledge

• WASH/CGI’s approach
  – Web pages represented by data structures
  – constructed functionally
  – automatic conversion to text on output
import CGI  -- indicate it's using CGI
main =    -- main program (fixed)
  run $  -- starts a CGI script
  ask $  -- delivers a Web page
  standardPage "Hello" $  -- constructs a Web page
text "This is my first CGI program!"  -- contents of page
• $ is function application;
  write “f $ a” for “f (a)” or “f a”
  “f $ g $ a” means “f (g a)”

• main is an I/O action of type “IO ()”

• run is a function that maps a CGI action to an I/O action
  \[
  \text{run :: CGI () -> IO ()}
  \]

• ask maps a document to a CGI action
  \[
  \text{ask :: WithHTML CGI () -> CGI ()}
  \]

• standardPage is a \textit{parameterized document} of type
  \[
  \text{String -> WithHTML CGI a -> WithHTML CGI a}
  \]
More on Documents

- **WithHTML CGI a**
  - type of sequences of document nodes
    (elements, attributes, or text nodes)
  - corresponds to contents of a HTML element
  - also computes a value of type a (*later*)
- **text :: String -> WithHTML CGI ()**
  - creates a singleton sequence with one text node
- for each HTML tag \( t \), there is a constructor function
  - \( t :: \) WithHTML CGI a \( \rightarrow \) WithHTML CGI a
    - it takes a sequence of child elements and attributes
    - creates an element with tag \( t \)
    - returns it in a singleton sequence
- Example: \( \text{p (text "This is my first CGI program!")} \)
Document Node Sequences

- the empty sequence

\[
\text{empty}
\]

- concatenation of sequences

\[
\text{seq1} \#\# \text{seq2} \quad \text{or} \\
\text{seq1} \gg \text{seq2} \quad \text{or} \\
\text{do} \{ \text{seq1; seq2; ...; seqn} \} \quad \text{or}
\]

\[
\text{do seq1} \\
\text{seq2} \\
\text{...} \\
\text{seqn}
\]
Example

ask $
standardPage "Hello" $
do p (text "This is my second CGI program!"
  
    p (do text "My hobbies are"
        
        ul (do li (text "swimming")
            
            li (text "music")
            
            li (text "skiing")))
  
)
HTML With Style: Composable Style Attributes

- style operators are `:=:`, `^::`, and `using`

- style attributes (cf. CSS2)
  ```plaintext
  fgRed = "color" :=: "red"
  bgGreen = "background" :=: "green"
  ```

- combining style attributes
  ```plaintext
  styleImportant = fgRed ^:: bgGreen
  ```

- using the style
  ```plaintext
  using ⟨style⟩ ⟨elem⟩ ⟨sequence⟩
  using styleImportant p (text "This is important!")
  ```
A Complete Example

import CGI

fgRed = "color" :=: "red"
bgGreen = "background" :=: "green"
styleImportant = fgRed :^: bgGreen

important = using styleImportant

main =
  run $
  ask $
  standardPage "Hello" $
  important p (text "This is important!")
3 Simple Interaction

Let’s personalize our program:

• ask for the name

• send a personalized greeting

For programming this interaction, we need to specify

• a form

• an input field

• an action taken on input
Creating a Form

- "raw" constructor for form element not available
- the "cooked" constructor
  
  ```haskell
  makeForm :: WithHTML CGI a -> WithHTML CGI ()
  ```

  creates form with standard attributes preset
- for convenience, we wrap this into a parameterized document:

  ```haskell
  standardQuery :: String -> WithHTML CGI a -> WithHTML CGI a
  standardQuery ttl elems =
      ask (standardPage ttl (makeForm elems))
  ```
Creating an Input Field

• “raw” constructor for input element not available

• the “cooked” constructor

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

using the type definition

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

• `textInputField` is a function that maps
  
  – a sequence of attributes for the input field to
  
  – a singleton sequence containing the input field
Input Handles

• *in addition* to constructing the HTML element,
  the constructor returns a *handle* to the input field

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

• the type of the handle is `InputField String INVALID`
  – *String*  the field contains a string
  – *INVALID* the field does not contain valid information, yet
Attaching an Action to an Input Field

Simple method for activating one input field

```haskell
activate actionFun inpField elems
```

- **actionFun :: a -> CGI ()**
  maps contents of input field to a CGI action
  activated when data is entered into the field

- **inpField :: HTMLField (InputField a INVALID)**

- **elems :: WithHTML CGI ()**
  sequence of attributes for the input field

- in our example: a is String
import CGI

standardQuery ttl cont =
    ask (standardPage ttl (makeForm cont))

main = run $ standardQuery "What’s your name?" $
    p (do text "Hi there! What’s your name?"
        activate greeting textInputField empty)

greeting :: String -> CGI ()
greeting name =
    standardQuery "Hello" $
    do text "Hello "
       text name
       text ". This is my first interactive CGI program!"

19
4 Typed Input and Tabular Output

Let’s extend the previous example to print a multiplication table. After the greeting

• ask for a multiplier

• print its multiplication table
Replace greeting by mtable

```haskell
mtable name =
    standardQuery "Multiplication Table" $
    do p (text ("Hello " ++ name ++ "!"))
        p (text "Let’s see a multiplication table!")
        p (text "Give me a multiplier " >>
            activate ptable inputField empty)

• ++ is string and list concatenation

• given that `ptable :: Int -> CGI ()`

• the input field has type `InputField Int INVALID`

⇒ an input field of this type refuses all inputs that are not integers!
```
Tabular Output

ptable :: Int -> CGI ()
ptable mpy =
    standardQuery "Multiplication Table" $
    table (mapM_ pLine [1..12])
where
    align = attr "align" "right"
pLine i = tr (do td (text (show i) ## align)
            td (text "*")
            td (text (show mpy))
            td (text "=")
            td (text (show (i * mpy)) ## align))

• [1..12] is list of integers 1, 2, 3, ..., 12
• mapM_ pLine [1..12] applies pLine to each element of [1..12]
• attr "align" "right" creates the attribute align="right"
5 Interaction with Multiple Inputs

Let’s modify the previous example to a teaching program for exercising multiplication:

- Ask for a multiplier
- Ask for a number of exercises
- Present exercise questions one at a time
- Display summary evaluation at the end
Replace greeting by mdrill

mdrill name =
  standardQuery "Multiplication" $
  do p (text ("Hello " ++ name ++ "!
  
  p (text "Let’s exercise some multiplication!
  
  mpyF <- p (text "Give me a multiplier " >>
    inputField (attr "value" "2"))
  
  rptF <- p (text "Number of exercises " >>
    inputField (attr "value" "10"))
  
  submit (F2 mpyF rptF) (firstExercise name) empty
Extended do Notation

Recall that construction of a sequence also computes a value.

The notation

\[
do \ldots
\]

\[
\text{var} \leftarrow \text{seq}
\]

\[
\ldots
\]

extracts the value (e.g., an input handle) computed while constructing \text{seq} into variable \text{var}.

Example:

\[
do \ldots
\]

\[
\text{mpyF} \leftarrow p \ (\text{text } "\text{Give me a multiplier }" \ >> \ 
\text{inputField (attr } "\text{value}" "2"))
\]
Value Propagation

• inputField occurs nested within p

⇒ must specify how value of inputField becomes value of p (...)

• Propagation rules
  - \( \text{elem (seq)} \) returns the value of seq
    (elem an element constructor)
  - \( \text{seq1} \gg \text{seq2} \) returns the value of seq2
  - \( \text{seq1} \#\# \text{seq2} \) returns the value of seq1
  - \( \text{do \{seq1; \ldots; seqn\}} \) returns value of seqn

• Example:

\[
\text{p (text "Give me a multiplier " } \gg \text{inputField (attr "value" "2")})
\]

returns the input handle created by the inputField.
Specifying Actions

- Creation of a separate submit button

  ```
  submit handle action attrs
  ```

  - `handle invalid` handle for input fields
  - `action` function that maps `valid` handles to a CGI action
  - `attrs` further attributes for the input field

- `submit validates` the input handles and passes them to `action`

  ```
  ⇒ handle :: h INVALID
  ⇒ action :: h VALID -> CGI ()
  ⇒ attrs :: WithHTML GCI ()
  ```

- where `h` is any input handle
Combining Input Handles

Different handle types must be used:

- \( h = F_0 \)  
  - no input handles
  - \textbf{submit} \( F_0 \) action

- \( h = \text{InputField} \ a \)  
  - a single input handle for values of type \( a \)
  - \textbf{do} \textbf{inF} <- \text{inputField} \ empty
    \textbf{submit} \textbf{inF} action

- \( h = F_2 \ h_1 \ h_2 \)  
  - a pair of two input handles, \( h_1 \) and \( h_2 \)
  - \textbf{do} \textbf{inF}1 <- \text{inputField} \ empty
    \textbf{inF}2 <- \text{inputField} \ empty
    \textbf{submit} (F_2 \textbf{inF}1 \textbf{inF}2) action

- and so on...
Accessing Input Handles

- \( \text{value} :: \text{InputHandle} \ a \ \text{VALID} \rightarrow \ a \)  
  if the handle is valid, then contents can be directly accessed

- In the example:
  
  \[
  \text{firstExercise name (F2 mpyF rptF) = runExercises 1 [] [] where mpy, rpt :: Int mpy = value mpyF rpt = value rptF}
  \]

- \( \text{mpy, rpt :: Int} \)  
  fixes type of input to integer
Interaction Logic (in Haskell)

runExercises nr successes failures =
  if nr > rpt then
    finalReport
  else
    let msg = "Question " ++ show nr ++ " of " ++ show rpt
do factor <- io (randomRIO (0,12))
  standardQuery msg $
    do text (show factor ++ " * " ++ show mpy ++ " = ")
      activate (checkAnswer factor) inputField empty

• io lifts an I/O action into a CGI action

• randomRIO (0,12) is I/O action that returns a random number between 0 and 12 (from Haskell standard library Random)

• still nested inside where (to access rpt and mpy)
Further Interaction Logic

where

checkAnswer factor answer =
  let result = factor * mpy
  correct = answer == result
  message = if correct then "correct! " else "wrong! "
  continue F0 = if correct
    then runExercises (nr+1) (factor:successes) failures
    else runExercises (nr+1) successes (factor:failures)
in standardQuery ("Answer " ++ show nr ++ " of " ++ show rpt) $
do p (text (show factor ++ " * " ++ show mpy ++ " = " ++ show result))
text ("Your answer " ++ show answer ++ " was " ++ message)
submit F0 continue (attr "value" "CONTINUE")

• continue takes no input handles ⇒ F0
6 Specifying Input Fields

So far, we have seen

- `textInputField`
  unconstrained text input

- `inputField`
  input in Haskell read syntax

But often, more restrictions apply

- select from a fixed set of alternatives

- further consistency checks (non-empty fields, email addresses, ...)

32
6.1 Selector Boxes

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

selectSingle showFunction maybeDefault options

- a is type of selected values
- Eq a states that values must be comparable
- showFunction :: a -> String
  - maps a value to its menu entry (a string)
- maybeDefault is either Nothing or Just defaultValue
- options is the list of values from which to choose
Application in mdrill

do ...  
  mpyF <- p (text "Give me a multiplier " >>  
              selectSingle show Nothing [2..12] empty)  
...  

- show is Haskell-provided printing function  
- Nothing: no default specified ⇒ form *insists* on an entry  
- [2..12] list of options  
- empty — no attributes for the selection box
6.2 Radio Buttons

- **radioGroup attrs**
  - creates a radio group (an invisible widget)
  - attrs are common attributes for all members
  - the function value extracts the value from a radio group
  - hence, all members have the same type

- **radioButton radiogroup val**
  attaches a button returning val to radiogroup

- **radioError radiogroup**
  specifies the location of the error indicator for radiogroup
do ...

rptF <- radioGroup empty
p (text "Number of exercises " >>
   text " 5 " ## radioButton rptF 5 empty >>
   text " 10 " ## radioButton rptF 10 empty >>
   text " 20 " ## radioButton rptF 20 empty >>
   radioError rptF)

...
6.3 Constrained Textual Input Fields

For application-specific input formats like

- non-empty string
- email address
- amount of money

we can define customized input fields by

- creating application-specific datatypes
- defining a read syntax
- giving an explanatory text

(requires skill in Haskell programming)
Example: EmailAddress

- the application-specific datatype

```haskell
newtype EmailAddress = EmailAddress unEmailAddress :: String
```

unEmailAddress extracts the string value from EmailAddress.

- the explanatory text

```haskell
instance Reason EmailAddress where
  reason _ = "email address \n  \{must contain @ and no special characters except . - _}"
Example: EmailAddress — continued

- defining a read syntax (not quite RFC2822)

```haskell
instance Read EmailAddress where
  readsPrec i str =
    let isAddressChar c = isAlpha c || isDigit c || c `elem` ".-_"
        (name, atDomain) = span isAddressChar (dropWhile isSpace str)
    in case atDomain of
      '@' : domainPart ->
        let (domain, rest) = span isAddressChar domainPart in
          if null name || null domain
            then []
            else [(EmailAddress (name ++ '@' : domain)
                ,dropWhile isSpace rest)]
      _ -> []
```
Example: EmailAddress — in use

main = run $
  standardQuery "Enter Your Email Address" $
  p (do text "Hi there! What’s your email address?"
      activate getEmail inputField empty)

getEmail email =
  standardQuery "Process Email" $
  do p (text ("Hello " ++ unEmailAddress email ++ ")")

• created using inputField

• extract and fix type using

  unEmailAddress :: EmailAddress -> String
7 Server-Side State

For the final report, we would like to have a “hall of fame” that displays the best results for each student.

• Keep a mapping from names and multipliers to correct results on the server

• Mapping is generally accessible from all clients

⇒ concurrency control required

(invisible for programmer)
Considerations for Server-Side State

- data is stored in textual format
  ⇒ conversion done using built-in Read and Show classes
- type safety across program boundaries
  ⇒ class Types
    (using problem-specific types requires Haskell expertise)
- provide abstract datatype of persistent values
  ⇒ only indirectly accessible through handles
- each handle has notion of current value
  ⇒ accessible throughout lifetime of handle
Initializing Server-Side State

• **import Persistent2**
  
  import API for persistent values

• **init externalName initialValue**

  a CGI action
  
  – allocates/accesses a persistent value named externalName
  
  – initialized with initialValue
    
    only if persistent value is freshly created
  
  – returns Nothing if the value existed but had a different type
  
  – returns Just handle where the persistent value of type a is accessible through handle of type T a
Suppose \texttt{handle :: T a} is a handle to a persistent value of type \texttt{a}

- \texttt{get handle} retrieves the persistent value
- \texttt{set handle newValue} updates the persistent value
  if successful, return a \texttt{Just newHandle} for the current value
  returns \texttt{Nothing} if the handle is not current (if it was modified by a concurrent process)
- \texttt{add handle additionalValue} handle refers to a value of list type
  adds \texttt{additionalValue} to the persistent list of values
- \texttt{current handle} returns a \texttt{newHandle} that refers to the current persistent value
### Process A

<table>
<thead>
<tr>
<th>h &lt;- init p v0</th>
<th>PV p</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- h == 0</td>
<td>0 -&gt; v0</td>
</tr>
<tr>
<td>x &lt;- get h</td>
<td>hb &lt;- init p v1</td>
</tr>
<tr>
<td>-- x == v0</td>
<td>-- hb == 0</td>
</tr>
<tr>
<td></td>
<td>-- v1 discarded</td>
</tr>
</tbody>
</table>

### Process B

<table>
<thead>
<tr>
<th>mha &lt;- set h v2</th>
<th>0 -&gt; v0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- mha == Nothing</td>
<td>1 -&gt; v2</td>
</tr>
<tr>
<td>-- h not current</td>
<td>mhb &lt;- set hb v2</td>
</tr>
<tr>
<td>curh &lt;- current h</td>
<td>-- mhb == Just 1</td>
</tr>
<tr>
<td>-- curh == 1</td>
<td></td>
</tr>
<tr>
<td>x1 &lt;- get h</td>
<td>0 -&gt; v0</td>
</tr>
<tr>
<td>-- x1 == v0</td>
<td>1 -&gt; v2</td>
</tr>
<tr>
<td>x2 &lt;- get curh</td>
<td>2 -&gt; v3</td>
</tr>
<tr>
<td>-- x2 == v2</td>
<td></td>
</tr>
<tr>
<td>set curh v3</td>
<td></td>
</tr>
<tr>
<td>-- successful</td>
<td></td>
</tr>
</tbody>
</table>
import qualified Persistent2 as P

-- abbreviate Persistent2 to P

finalReport =
do Just initialHandle <- P.init ("multi-" ++ name) []
currentHandle <- P.add initialHandle (mpy, lenSucc, rpt)
hiScores <- P.get currentHandle
standardQuery "Final Report" $
do p (text "Here are your recent scores."
   ul (mapM_ pItem hiScores)
where lenSucc = length successes

pItem (m, l, r) = li (text ("Multiplier " ++ show m ++
   " : " ++ show l ++ " correct out of " ++ show r))
API Summary: Persistent2

init :: (Read a, Show a, Types a) =>
   String -> a -> CGI (Maybe (T a))
get :: (Read a) =>
   T a -> CGI a
set :: (Read a, Show a) =>
   T a -> a -> CGI (Maybe (T a))
add :: (Read a, Show a) =>
   T [a] -> a -> CGI (T [a])
current :: (Read a) =>
   T a -> CGI (T a)
8 Client-Side State

A user should only be required to enter his name once

- store user name on client side

⇒ store on client

- implemented using “cookies”

- . . . but type-safe!
  (errh, type-indexed)

- interface similar to Persistent2

- but no history maintained
import qualified Cookie as C

main = run $
  do nameC <- C.init "name" Nothing
     mname <- C.get nameC
     case mname of
       Just name ->
         mdrill name
       Nothing ->
         standardQuery "What’s your name?" $
         p (do text "Hi there! What’s your name?"
                 activate (mdrillCookie nameC) textInputField empty)

mdrillCookie nameC name =
  do C.set nameC (Just name)
     mdrill name
Tour of Cookie API

- (Read a, Show a, Types a) =>
  required for all storable types (cf. Persistent2)

- init cookieName initialValue
  a CGI action that
  - creates a handle to client-side variable cookieName
  - initializes to initialValue if the variable must be created
  - always successful (names are type-indexed)
  - returned handle is current
Tour of Cookie API, Part 2

- **get handle**
  
a CGI action that
  
  - returns value associated to handle
  
  - **fails** if handle is not current
    
    usually due to improper behavior of user or programming error

- **set handle newValue**
  
  - if handle is current, then overwrite with newValue and return
    
    Just the new current handle
  
  - if handle is not current, then return Nothing
API Summary: Cookie

init :: (Read a, Show a, Types a) =>
   String -> a -> CGI (T a)
get :: (Read a, Show a, Types a) =>
   T a -> CGI a
set :: (Read a, Show a, Types a) =>
   T a -> a -> CGI (Maybe (T a))
delete :: (Types a) =>
   T a -> CGI ()
9 Advanced Topics

9.1 Uploading Files

fileInputField :: HTMLField (InputField FileReference INVALID)

- value of type FileReference is a record
  - fileReferenceName, a local file path (on server)
  - fileReferenceContentType, content type of the file
  - fileReferenceExternalName, provided by submitter

- FileReference is only temporary

- script responsible for renaming or copying to safe location
Example Uploader

main = run $ 
  standardQuery "Upload File" $ 
  do text "Enter file to upload " 
     fileH <- fileInputField empty 
     submit fileH display (fieldVALUE "UPLOAD")

display :: InputField FileReference VALID -> CGI ()
display fileH = 
  let fileRef = value fileH in 
  standardQuery "Upload Successful" $ 
  do text "Check file contents " 
     submit F0 (const (tell fileRef)) (fieldVALUE "GO")

  Warning! Security problems may lurk!
9.2 Non-textual Responses

- **tell :: CGIOutput data => data -> CGI ()**

- transform data to CGI action that returns data to browser

- examples for data
  - FileReference
  - Element (HTML elements)
  - String (generates text/plain document)
  - Status messages
  - Location (redirection)
  - FreeForm contents:
    - FreeForm fileName contentType rawContents
Example: File Downloader

main = run $ standardQuery "SendFile" $ table $ do
  pcNameF <- tr (td (text "File name") >>
      td (textInputField (fieldSIZE 20)))
  passwordF <- tr (td (text "Password") >>
      td (passwordInputField (fieldSIZE 20)))
  tr (td (submit (F2 pcNameF passwordF) sendFile (fieldVALUE "SEND"))) >> td empty)

sendFile (F2 fileNameF passwordF) =
  let fileName = value (unNonEmpty fileNameF)
      password = value (unNonEmpty passwordF)
  in if validPassword fileName password then tell
      FileReference { fileName = storeDirectory ++ fileName
          , fileReferenceContentType = guessContentType fileName
      }
  else htell $ standardPage "Login incorrect" $ backLink
9.3 Inlined Downloading

- standard link (no download button)
- still return arbitrary files
  - accessible to script
  - not necessarily accessible to Web server

⇒ install a translator

- \texttt{translator :: [String] \rightarrow CGI ()}
  maps path name to CGI action
Using a Translator

• replace run with runWithHook translator

• create a reference to a named item with `makRef name attrs`

• example:

```haskell
translator (name:_ ) =
  let fileName = storeDirectory ++ name in
do ex <- unsafe_io (doesFileExist fileName)
  if ex
    then tell FileReference
      { fileReferenceName = fileName
        , fileReferenceContentType = guessContentType name
      }
  else fallbackTranslator [name]
```
9.4 Sending Email

- Deja Vue: message-as-text not appropriate

⇒ create record data types for email contents and messages

- Email contents: data type DOC

```
mediatype :: String, -- type
subtype   :: String, -- subtype
parameters :: [KV], -- parameters
filename   :: String, -- suggested filename
-- depending on mediatype only one of the following is relevant:
messageData :: String, -- data
textLines   :: [String], -- lines
parts       :: [DOC]  -- data
```
Actual Interface

- `textDOC :: String -> [String] -> DOC`
  
  `textDOC subty docLines`

  create a text document with content type `text/subty`

- `binaryDOC :: String -> String -> String -> DOC`

  `binaryDOC mediaty subty bindata`

  arbitrary document with content type `mediaty/subty`

- `multipartDOC :: [DOC] -> DOC`

  `multipartDOC subdocs`

  collect a list `subdocs` of documents into one

- further possibilities (alternative, external, …)
Datatype for Messages

- Mail is a record

  to :: [String],
  subject :: String,
  cc :: [String],
  bcc :: [String],
  headers :: [Header],
  contents :: DOC

- convenience function

  simpleMail recipients subj doc
Example of Sending Mail

notifyAccept submission reports = do
  instr <- io (readFile instructionsFile)
  let opening = textDOC "plain"
      
      ["Dear " ++ itemAuthor submission ++ 
      
      ,"
      
      ,"I am pleased to inform you that your paper"
      
      ,"  " ++ itemTitle submission
      
      ,"has been accepted for presentation ..."
      
      instructions =
        (textDOC "plain" (lines instr))
        [ filename= "AuthorInstructions" ]
  notify [opening, instructions] submission reports
notify frontmatter submission reports = do
    let doReport report nr =
        (textDOC "plain" (lines (reportForAuthor report)))
        { filename= "Review#" ++ show nr }
    doc = multipartDOC (frontmatter ++ zipWith doReport reports [1..])
    message = (simpleMail [itemEmail submission] "Notification" doc)
        {cc= [chairperson],headers= [Header ("From", chairperson)]}
    exitcode <- io (sendmail message)
    htell (standardPage ("Message sent. Exitcode = " ++ show exitcode) empty)
10 Conclusion

• simple, declarative approach to Web-based user interfaces
• types and type safety essential
• GUI-style programming interface
• natural interface to HTML
• ideas not tied to CGI
• applications: submission software, generic time table, ...
• available from
  http://www.informatik.uni-freiburg.de/~thiemann/WASH