CSE399: Advanced Programming

Handout 17

Course Projects

Projects

- Assignment: Design and implement a full-featured web application on top of WASH/CGI.
- Grading will be based on
    - creativity and usefulness of your application
    - elegance of design
    - beauty of code
- Workload:
    - Roughly 30 hours of work
    - Due last day of class (April 22)

Project Proposals

As we decided on Monday, the due date for initial project proposals will be two weeks from this Friday, March 18th.

Your proposal must include all of the following (around 6-8 pages total):

- A short (less than one page) description of the application you intend to build
- Several (two or three pages worth) use cases, each sketching the “story” of a particular kind of interaction with the system, from the client’s point of view. (Feel free to include pictures of possible screenshots, but don’t spend a long time on them!)
- A (preliminary) list of the most important modules, with a brief description of what each one will do.
- A time budget for the project, adding up to about 30 hours.

Possible Project Ideas

- A blog server
- An E-vite server
- A more sophisticated Adventure game (perhaps something like Kingdom of Loathing)
- etc.

Details are up to you!

Miscellaneous Points on WASH/CGI

HTML Concrete Syntax

As described in the WASH/CGI User Manual (it was implemented after the tutorial paper was written), WASH provides an HTML-like concrete syntax for the API we have been discussing:

```
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?</#>
```

Feel free to use it or not, as you prefer.

Variants of the HTML commands

We noticed last time that there were three different versions of all the basic HTML constructors: `p` vs. `p_T` vs. `p_S`.

The differences between them are not essential for our purposes. Just use the un-annotated ones.

Quasi-Validation

Review: Type Constructors

Recall the general form of a type constructor declaration (with a `newtype`—`type` and `data` can also be used for declaring type constructors, but the trick we are about to see only works with `newtype` and `data`):

```
newtype C x y z = CC body
```

- `C` is the name of the type constructor being defined
- `x`, `y`, and `z` are its parameters
- `body` is its definition
- `CC` is the name of the (value) constructor used to build values of type `C`

Example:

```
newtype C x y z = CC [(x,y)->z]
```

Phantom Types

Note that we do not have to use all of the parameters in the body. For example, it is perfectly legal to write:

```
newtype P x y = PP [y]
```

or even:

```
newtype Q x = QQ [Int]
```

A phantom type is a parameter to a type constructor that does not appear in its body (like x here).

Why would such a thing be useful????

A Tiny Example of Phantom Types

```
newtype Q x = QQ [Int]

data Even = Even
data Odd = Odd

nil :: Q Even
nil = QQ []

conse :: Int -> Q Even -> Q Odd
conse i (QQ l) = QQ (i:l)

conso :: Int -> Q Odd -> Q Even
conso i (QQ l) = QQ (i:l)

myList = conso 5 (conse 3 nil)
```

Tiny Example Continued

But trying to write

```
myList’ = conso 5 nil
```

results in this complaint from the compiler:

```
Couldn’t match ‘Odd’ against ‘Even’
    Expected type: Q Odd
    Inferred type: Q Even
In the second argument of ‘conso’, namely ‘nil’
In the definition of ‘myList’’: myList’ = conso 5 nil
```

I.e., the type system now “understands” the difference between even- and odd-length lists.

Tiny Example, Continued

Moreover, we can define functions on these new lists that are polymorphic in the phantom type parameter—i.e., that work for both even and odd lists:

```
mapQ :: (Int->Int) -> (Q a) -> (Q a)

mapQ f (QQ l) = QQ (map f l)

myList'' = mapQ (+ 3) myList
```

Phantom Types in WASH

Phantom types are a very useful trick that has been exploited many times to achieve an astonishing range of effects.

It is used in various places in WASH. Two that we've seen are:

- quasi-validation of HTML
- `VALID` vs. `INVALID` input handles

Quasi-validity

WASH/CGI enforces quasi validity through (phantom type parameters in) the types of the construction functions for document nodes.

Read `WithHTML e m a` as “The type of (sequences of) HTML content [attributes and elements]...

- that can be used in a context described by the phantom parameter `e`
- whose construction involves side effects side effects in monad `m` (typically `CGI`)
- whose construction yields a value of type `a` (typically either `()` or some kind of input handle).”

Quasi-validity

Now:

```
text :: (Monad m, AdmitChildCDATA e)
     => String -> WithHTML e m ()

ul :: (Monad m, AdmitChildUL e)
   => WithHTML UL m a -> WithHTML e m a
```

I.e., if `s` is some string, then `text s` can be used in any context `e` that belongs to the type class `AdmitChildCDATA`.

Similarly, `UL` can be used in any context `e` that belongs to the type class `AdmitChildUL`. Moreover, `ul` takes as an argument some HTML content that can be used in the context `UL`.

Which types of HTML content are those??

Quasi-validation

From the `HTMLMonad98` module...

```
class AdmitChildUL e where
  ul :: (Monad m) => WithHTML UL m a -> WithHTML e m a
  ul = HTMLMonad.ul
...

data LI = LI
data DD = DD
data BODY = BODY
...

instance AdmitChildUL LI
instance AdmitChildUL DD
instance AdmitChildUL BODY
...
```

Input Handles

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

submit0 :: (AdmitChildINPUT context) =>
           CGI () -> HTMLField context ()

textInputField :: (AdmitChildINPUT context) =>
                  HTMLField context (InputField String INVALID)

submit1 :: (AdmitChildINPUT context) =>
           InputField a INVALID
           -> (InputField a VALID -> CGI ())
           -> HTMLField context ()

value :: InputField a VALID -> a
```

N.b.: `submit1` is subsumed by `activate` in the current version of WASH.

Validated Input

```
inputField :: (AdmitChildINPUT context, Reason a, Read a) =>
              HTMLField context (InputField a INVALID)

page1 =
  standardQuery "Numeric Input" $
  do inf <- p (do text "Enter a number "
     inputField (attr "size" "10"))
     submit inf page2 (attr "value" "Process")

page2 inf =
 let n :: Int
     n = value inf
 in
   standardQuery "Your Number" $
     p (text "Your number was " ## text (show n) ## text "!")
```