CSE399: Advanced Programming

Handout 14

Where We're Going

Where We've Been

So far in this course, we have

- covered pretty much all the features of the Haskell language
- gotten significant practical experience with Haskell programming in homework assignments
- seen some of the most common idioms used in Haskell programs and libraries
  - higher-order programming (functions as data)
  - programming with type classes
  - monadic style

Where We're Going

My aim in the rest of the course is to use all of this as a foundation for discussing and experimenting with a number of topics:

- modularity (modular decomposition, design by contract, unit testing, documentation, etc.)
- programming as writing (programming for people, not computers)
- more examples of interesting libraries and abstractions (pretty printing combinators, "arrows," etc.)
- larger applications of Haskell's idioms and features to practical programming problems
    - in particular: web site construction

Concretely

- The term projects (which will take up about the last 1/3 of the semester) will involve web site construction on top of the WASH/CGI framework.
- Most of the remaining homework assignments will be devoted to building up infrastructure for this (exploring various parts of the WASH/CGI framework, etc.)
- Class time will be divided between
  - examining various aspects of WASH/CGI
  - discussion of people's solutions to assignments
  - discussion of programming from various angles
  - other miscellaneous examples and topics (suggestions welcome!)

Warning: All this will be a little less structured than what we've done up to this point!

Additional Textbook

As fodder for the programming discussions, we'll be reading selections from

The Pragmatic Programmer
by Andrew Hunt and David Thomas

You'll need a copy of this book in about a week. (Should we get the bookstore to order it?)

XML Processing Combinators

XML in Haskell

A number of people have built high-level libraries (a.k.a. embedded languages) for manipulating and transforming XML in Haskell.

- HaXml (Wallace and Runciman)
- HXML (English)
- etc.

We'll look at the first one today, and (hopefully) come back to the second later in the course.

Running Example

```
<album>
  <title>Time Out</title>
  <artist>Dave Brubeck Quartet</artist>
  <recording date="June-August 1959" place="NYC"/>
  <coverart style='abstract'>
    <location thumbnail='pix/small/timeout.jpg' fullsize='pix/covers/timeout.jpg'/></coverart>
  <catalogno label='Columbia' number='CL 1397' format='mono'/>
  <catalogno label='Columbia' number='CS 8192' format='stereo'/>
  <catalogno label='Columbia' number='CPK 1181' format='LP' country='Korea'/>
  <catalogno label='Sony/CBS' number='Legacy CK 40585' format='CD'/>
  <personnel>
    <player name='Dave Brubeck' instrument='piano'/>
    <player name='Paul Desmond' instrument='alto sax'/>
    <player name='Eugene Wright' instrument='bass'/>
    <player name='Joe Morello' instrument='drums'/>
  </personnel>
  <track title='Blue Rondo &agrave; la Turk' credit='Brubeck' timing='6m42s'/>
  <track title='Strange Meadow Lark' credit='Brubeck'  timing='7m20s'/>
  <track title='Take Five' credit='Desmond'  timing='5m24s'/>
  <track title='Three To Get Ready' credit='Brubeck'  timing='5m21s'/>
  <track title="Kathy's Waltz" credit='Brubeck'  timing='4m48s'/>
  <track title="'Everybody's Jumpin'" credit='Brubeck'  timing='4m22s'/>
  <track title='Pick Up Sticks' credit='Brubeck'  timing='4m16s'/>
  <notes>
    Possibly the DBQ's most famous album, this contains
    <trackref link='#3'>Take Five</trackref>, the most famous jazz track
    of that period.  These experiments in different time signatures are
    what Dave Brubeck is most remembered for.  Recorded Jun-Aug 1959 in
    NYC.  See also the sequel,
    <albumref link='cbs-timefurthout'>Time Further Out</albumref>.
  </notes>
</album>
```

Representation of XML Data

Essentially the same as the one we have been using:

```
data Element   = Elem Name [Attribute] [Content]
type Attribute = (Name, AttValue)
data Content   = CElem Element
               | CText CharData

type Name = String
type AttValue = String
type CharData = String
```

(N.b.: This is slightly simplified from the real implementation.)

XML Filters

All document transformations are content filters. A filter takes a single XML `Content` value and returns a sequence of `Content` values (possibly empty).

```
type CFilter = Content -> [Content]
```

Selection Filters

Selecting Children

Throw away current node and return its children.

```
children :: CFilter

children (CElem (Elem _ _ cs)) = cs
children _ = []
```

Selecting by Position

Select the `n`'th positional result of a filter.

```
position :: Int -> CFilter -> CFilter

position n f = (\cs-> [cs!!n]) . f
```

(N.b.: not described in the paper.)

Example

Applying

```
ex2 = position 1 children
```

to our sample XML yields:

```
<artist>Dave Brubeck Quartet</artist>
```

Selecting an Attribute

Throw away current node and return one of its attributes.

```
showAttr :: String -> CFilter

showAttr key c@(CElem (Elem _ as _)) =
  [CText (lookfor key as)]
  where
    lookfor x as =
       case (lookup x as) of
         Nothing ->
           (error ("missing attribute: "++show x))
         Just v -> v

showAttr _ _ = []
```

(N.b.: the current library does this a little differently — see the on-line documentation.)

Renaming

Rename an element tag.

```
replaceTag :: String -> CFilter

replaceTag n (CElem (Elem _ _ cs)) =
  [CElem (Elem n [] cs)]
replaceTag n _ =
  []
```

(Note that one might quibble with the details of this!)

Construction Filters

Literal

The filter `literal s` ignores its argument and yields `s`:

```
literal :: String -> CFilter

literal s = \_ -> [CText s]
```

Also (in the paper, but not in the current library)...

```
(!) = literal
```

...so we can write `("hello"!)` instead of `(literal "hello")`, using Haskell's "section" syntax.

mkElem

Build an element with the given tag name. Its content is formed by concatenating the results of the given list of filters.

```
mkElem :: String -> [CFilter] -> CFilter
mkElem h cfs = \t-> [ CElem (Elem h [] (cat cfs t)) ]

cat :: [CFilter] -> CFilter
cat fs = \e-> concat [ f e | f <- fs ]
```

Similarly, `mkElemAttr` builds an element with the given name, attributes, and content. The attributes are specified by a list of (string,filter) pairs.

Example

```
ex1 =
  mkElem "html"
    [mkElem "body"
       [literal "Hello world"]]
```

Running this on our sample XML (or any other XML document!) yields:

```
<html><body>Hello world</body></html>
```

Another Example

```
ex4 =
  mkElem "html"
    [mkElem "body"
       [position 1 children,
        position 2 children]]
```

Running this on our sample XML yields:

```
<html><body>
  <artist>Dave Brubeck Quartet</artist>
  <recording date="June-August 1959" place="NYC"/>
</body></html>
```

Predicate Filters

Success and Failure Filters

The `keep` filter takes any content and returns it; the `none` filter takes any content and returns nothing.

```
keep, none :: CFilter

keep = \x->[x]
none = \x->[]
```

Content tests

These filters either keep or throw away some content based on a simple test: `elm` keeps only a tagged element, `txt` keeps only non-element text, `tag` keeps only an element with the named tag, `attr` keeps only an element with the named attribute, `attrval` keeps only an element with the given attribute value, `tagWith` keeps only an element whose tag name satisfies the given predicate.

```
elm, txt    :: CFilter
tag         :: String -> CFilter
attr        :: Name -> CFilter
attrval     :: Attribute -> CFilter
tagWith     :: (String->Bool) -> CFilter
```

```
elm x@(CElem _)   = [x]
elm _             = []

txt x@(CText _) = [x]
txt _           = []

tag t x@(CElem (Elem n _ _)) | t==n  = [x]
tag t _  = []

tagWith p x@(CElem (Elem n _ _)) | p n  = [x]
tagWith p _  = []

attr n x@(CElem (Elem _ as _)) | n `elem` (map fst as)  = [x]
attr n _  = []

attrval av x@(CElem (Elem _ as _)) | av `elem` as  = [x]
attrval av _  = []
```

Combinators

“Irish Composition”

Apply the left filter to the results of the right filter.

```
o :: CFilter -> CFilter -> CFilter

f `o` g = concat . map f . g
```

Example

Running

```
ex5 = tag "personnel" `o` children
```

on our sample XML yields:

```
<personnel>
  <player name="Dave Brubeck" instrument="piano"/>
  <player name="Paul Desmond" instrument="alto sax"/>
  <player name="Eugene Wright" instrument="bass"/>
  <player name="Joe Morello" instrument="drums"/>
</personnel>
```

Example

The filter

```
ex3 = txt `o` children `o` tag "title"
```

means “only the plain-text children of the current element, provided the current element has the tag `title`.”

Parallel Composition

Binary parallel composition. Each filter gets a copy of the input, rather than one filter using the result of the other.

```
union :: CFilter -> CFilter -> CFilter

f `union` g = \c -> f c ++ g c
```

(Called `|||` in the paper.)

Example

Running

```
ex6 =
    mkElem "stuff"
      [(tag "personnel" `union` tag "catalogno") `o` children]
```

on our XML yields:

```
<stuff>
  <catalogno label="Columbia" number="CL 1397" format="mono"/>
  <catalogno label="Columbia" number="CS 8192" format="stereo"/>
  <catalogno label="Columbia" number="CPK 1181" format="LP"
             country="Korea"/>
  <catalogno label="Sony/CBS" number="Legacy CK 40585" format="CD"/>
  <personnel>
    <player name="Dave Brubeck" instrument="piano"/>
    <player name="Paul Desmond" instrument="alto sax"/>
    <player name="Eugene Wright" instrument="bass"/>
    <player name="Joe Morello"
            instrument="drums"/>
  </personnel>
</stuff>
```

Example

On the other hand, running

```
ex7 =
   mkElem "stuff"
     [(tag "personnel" `o` children)
      `union`
      (tag "catalogno" `o` children)]
```

on our XML yields:

```
<stuff>
  <personnel>
    <player name="Dave Brubeck" instrument="piano"/>
    <player name="Paul Desmond" instrument="alto sax"/>
    <player name="Eugene Wright" instrument="bass"/>
    <player name="Joe Morello" instrument="drums"/></personnel>
  <catalogno label="Columbia" number="CL 1397" format="mono"/>
  <catalogno label="Columbia" number="CS 8192" format="stereo"/>
  <catalogno label="Columbia" number="CPK 1181" format="LP"
             country="Korea"/>
  <catalogno label="Sony/CBS" number="Legacy CK 40585"
             format="CD"/></stuff>
```

Pruning

`f ‘with‘ g` keeps (just) those results of `f` for which `g` also produces at least one result.

```
with :: CFilter -> CFilter -> CFilter

f ‘with‘ g = filter (not.null.g) . f
```

Example

Running

```
ex8 =
  (children `o` tag "personnel" `o` children)
  `with`
  (attrval ("instrument","bass"))
```

on our XML yields:

```
<player name="Eugene Wright" instrument="bass"/>
```

(N.b.: Needs to be modified slightly to run with the real library.)

Pruning

Similarly, `f ‘without‘ g` keeps (just) those results of `f` for which `g` does **not** produce any results.

```
without :: CFilter -> CFilter -> CFilter

f ‘without‘ g = filter (null.g) . f
```

Conditionals

Conditional

These definitions provide C-like conditionals, lifted to the filter level.

The `(cond ? yes : no)` notation in C becomes `(cond ?> yes :> no)` in Haskell.

```
-- Conjoin the two branches of a conditional.
data ThenElse a = a :> a

-- Select between the two branches of a joined conditional.
(?>) :: (a->[b]) -> ThenElse (a->[b]) -> (a->[b])
p ?> (f :> g) = \c-> if (not.null.p) c then f c else g c
```

Directional choice

`f |>| g` returns g's results only if there are no f-results

```
(|>|) :: (a->[b]) -> (a->[b]) -> (a->[b])

f |>| g =
  \x-> let fx = f x in if null fx then g x else fx
```

Alternatively: `f |>| g = f ?> f :> g`

Path-like Selection

Pronounced "slash", `f \/> g` means `g` inside `f`:

```
(/>) :: CFilter -> CFilter -> CFilter
f /> g = g `o` children `o` f
```

Pronounced "outside", `f \<\/ g` means `f` containing `g`

```
(</) :: CFilter -> CFilter -> CFilter
f </ g = f `with` (g `o` children)
```

Editing

Editing a Node

Process CHildren In Place. The filter is applied to any children of an element content, and the element rebuilt around the results.

```
chip :: CFilter -> CFilter

chip f (CElem (Elem n as cs)) =
  [ CElem (Elem n as (concat (map f cs))) ]
chip f c =
  [c]
```

Recursion

Recursive Editing

Recursive application of filters: a fold-like operator.

```
foldXml :: CFilter -> CFilter

foldXml f = f `o` chip (foldXml f)
```

Running

```
ex9 =
  foldXml
    ((replaceTag "MUSICIAN" `o` tag "player") |>| keep)
```

on our XML yields:

```
<album>
  <title>Time Out</title>
  <artist>Dave Brubeck Quartet</artist>
  <recording date="June-August 1959" place="NYC"/>
  <coverart style="abstract">
     <location thumbnail="pix/small/timeout.jpg"
               fullsize="pix/covers/timeout.jpg"/></coverart>
  <catalogno label="Columbia" number="CL 1397" format="mono"/>
  <catalogno label="Columbia" number="CS 8192" format="stereo"/>
  <catalogno label="Columbia" number="CPK 1181" format="LP"
              country="Korea"/>
  <catalogno label="Sony/CBS" number="Legacy CK 40585" format="CD"/>
  <personnel>
    <MUSICIAN/>
    <MUSICIAN/>
    <MUSICIAN/>
    <MUSICIAN/>
  </personnel>
  <track title="Blue Rondo &agrave; la Turk" credit="Brubeck"
         timing="6m42s"/>
  <track title="Strange Meadow Lark" credit="Brubeck"
          timing="7m20s"/>
```

Recursive Search

```
deep, deepest, multi :: CFilter -> CFilter

deep f     = f |>| (deep f `o` children)
deepest f  = (deepest f `o` children) |>| f
multi f    = f `union` (multi f `o` children)
```