Advanced Programming Handout 7

Monads and Friends

(SOE Chapter 18)

The Type of a Type

- In previous chapters we discussed:
  - Monomorphic types such as `Int`, `Bool`, etc.
  - Polymorphic types such as `[a]`, `Tree a`, etc.
  - Monomorphic *instances* of polymorphic types such as `[Int]`, `Tree Bool`, etc.
- `Int`, `Bool`, etc. are *nullary type constructors,* whereas `[]`, `Tree`, etc. are *unary type constructors.* `FiniteMap` is a *binary type constructor*.
- The "type of a type" is called a *kind*. The kind of all monomorphic types is written "`*`":

```
Int, Bool, [Int], Tree Bool  ::  *
```

- Therefore the type of unary type constructors is:

```
[], Tree  ::  * -> *
```

- These "higher-order types" can be useful in various ways, especially with type classes.

The Functor Class

- The Functor class demonstrates the use of high-order types:

```
class Functor f where
      fmap :: (a -> b) -> f a -> f b
```

- Note that `f` is applied here to one (type) argument, so should have kind "`* -> *`".
- For example:

```
instance Functor Tree where
    fmap f (Leaf x)       = Leaf (f x)
    fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

- Or, using the function mapTree previously defined:

```
instance Functor Tree where
    fmap = mapTree
```

- Exercise: Write the instance declaration for *lists.*

The Monad Class

- *Monads* are perhaps the most famous (infamous?) feature in Haskell.
- They are captured in a type class:

```
class  Monad m  where
  (>>=)   :: m a -> (a -> m b) -> m b    -- "bind"
  (>>)    :: m a -> m b -> m b           -- "sequence"
  return  :: a -> m a
  fail    :: String -> m a

  -- default implementations:
  m >> k       =  m >>= (\_ -> k)
  fail s       = error s
```

- The key operations are `(>>=)` and `return`.

Syntactic Mystery Unveiled

- The "`do`" syntax in Haskell is shorthand for Monad operations, as captured by these rules:

```
do e → e
do e1; e2; ...; en → e1 >> (do e2 ; ...; en)
do pat <- e1 ; e2 ; ...; en →
            let ok pat = do e2 ; ...; en
                ok _   = fail "..."
            in e1 >>= ok
do let decllist ; e2 ; ...; en →
            let decllist in (do e2 ; ...; en)
```

- Note special case of rule 3:

  3a. `do x <- e1 ; e2 ; ...; en →`
  `e1 >>= \x -> do e2 ; ...; en`

Example Involving IO

- "`do`" syntax can be completely eliminated using these rules:

```
do putStr "Hello"
     c <- getChar
     return c

➔    putStr "Hello" >>    -- by rule (2)
     do c <- getChar
          return c

➔    putStr "Hello" >>    -- by rule (3a)
     getChar >>= \c ->
     do return c

➔    putStr "Hello" >>    -- by rule (1)
     getChar >>= \c ->
     return c

➔    putStr "Hello" >>    -- by currying
     getChar >>=
     return
```

Functor and Monad Laws

- Functor laws:

```
fmap id         = id
fmap (f . g)    = fmap f . fmap g
```

- Monad laws:

```
return a >>= k               = k a
m >>= return                 = m
m >>= (\x -> k x >>= h)      = (m >>= k) >>= h
```

Note special case of last law:

```
m1 >> (m2 >> m3)     = (m1 >> m2) >> m3
```

- Connecting law:

```
fmap f xs  =  xs >>= (return . f)
```

Monad Laws Expressed using "do" Syntax

- `do x <- return a ; k x      =  k a`
- `do x <- m ; return x        =  m`
- `do x <- m ; y <- k x ; h y = do y <- (do x <- m ; k x) ; h y`
- `do m1 ; m2 ; m3             = do (do m1 ; m2) ; m3`
- `fmap f xs                   = do x <- xs ; return (f x)`

- For example, using the second rule above, the example given earlier can be simplified to just:

```
do putStr "Hello"
   getChar
```

or, after desugaring: `putStr "Hello" >> getChar`

The Maybe Monad

- Recall the Maybe data type:

```
data Maybe a = Just a
             | Nothing
```

- It is both a Functor and a Monad:

```
instance  Monad Maybe  where
     Just x  >>= k       =  k x
     Nothing >>= k       =  Nothing
     return x            =  Just x
     fail s              =  Nothing

instance  Functor Maybe  where
     fmap f Nothing      =  Nothing
     fmap f (Just x)     =  Just (f x)
```

- These instances are indeed “law abiding”.

Using the Maybe Monad

- Consider the expression "`g (f x)`". Suppose that both `f` and `g` could return errors that are encoded as "`Nothing`". We might do:

```
case f x of
    Nothing -> Nothing
    Just y   -> case g y of
       Nothing -> Nothing
       Just z   -> …proper result using z…
```

- But since Maybe is a Monad, we could instead do:

```
do y <- f x
     z <- g y
     return …proper result using z…
```

Simplifying Further

- Note that the last expression can be desugared and simplified as follows:

```
f x >>= \y ->            ➔     f x >>= \y ->
g y >>= \z ->                  g y >>= return
return z

➔  f x >>= \y ->         ➔     f x >>= g
   g y
```

- So we started with `g (f x)` and ended with `f x >>= g`.

The List Monad

- The `List` data type is also a Monad:

```
instance  Monad []  where
    m >>= k   =  concat (map k m)
    return x  =  [x]
    fail x    =  [ ]
```

- For example:

```
do x <- [1,2,3]
     y <- [4,5]
     return (x,y)
```

  ➔ `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

- Note that this is the same as:

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
```

  Indeed, list comprehension syntax is an alternative to `do` syntax, for the special case of lists.

Useful Monad Operations

```
sequence     :: Monad m => [m a] -> m [a]
sequence     =  foldr mcons (return [])
                   where mcons p q = do x  <- p
                                        xs <- q
                                        return (x:xs)

sequence_    :: Monad m => [m a] -> m ()
sequence_    =  foldr (>>) (return ())

mapM         :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as    =  sequence (map f as)

mapM_        :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as   =  sequence_ (map f as)

(=<<)        :: Monad m => (a -> m b) -> m a -> m b
f =<< x      =  x >>= f
```

State Monads

- State monads are perhaps the most common kind of monad: they involve updating and threading *state* through a computation. Abstractly:

```
data SM a = SM (State -> (State, a))

instance Monad SM where
     return a          = SM $ \s -> (s,a)
     SM sm0 >>= fsm1   = SM $ \s0 ->
                         let (s1,a1)  = sm0 s0
                              SM sm1  = fsm1 a1
                              (s2,a2) = sm1 s1
                         in (s2,a2)
```

- Haskell's *IO monad* is a state monad, where State corresponds to the "state of the world".
- But state monads are also commonly user defined. (For example, tree labeling – *see text.*)

IO is a State Monad

- Suppose we have these operations that implement an association list:

```
lookup :: a -> [(a,b)] -> Maybe b
update :: a -> b -> [(a,b)] -> [(a,b)]
exists :: a [(a,b)] -> Bool
```

- A file system is just an association list mapping file names (strings) to file contents (strings):

```
type State = [(String, String)]
```

- Then an extremely simplified IO monad is:

```
data IO a = IO (State -> (State, a))
```

  whose instance in `Monad` is exactly as on the preceding slide, replacing "`SM`" with "`IO`".

State Monad Operations

- All that remains is defining the domain-specific operations, such as:

```
readFile :: String -> IO (Maybe String)
readFile s = IO (\fs -> (fs, lookup s fs) )

writeFile :: String -> String -> IO ()
writeFile s c = IO (\fs -> (update s c fs, ()) )

fileExists :: String -> IO Bool
fileExists s = IO (\fs -> (fs, exists s fs) )
```

- Variations include generating an error when `readFile` fails instead of using the `Maybe` type, etc.

Polymorphic State Monad

- The state monad can be made polymorphic in the state, in the following way:

```
data SM s a = SM (s -> (s, a))

instance Monad (SM s) where
      return a          = SM $ \s -> (s,a)
      SM sm0 >>= fsm1   = SM $ \s0 ->
                            let (s1,a1)  = sm0 s0
                                 SM sm1  = fsm1 a1
                                 (s2,a2) = sm1 s1
                            in (s2,a2)
```

- Note the partial application of the type constructor SM in the instance declaration. This works because SM has kind `* -> * -> *`, so "`SM s`" has kind `* -> *`.