Advanced Programming
Handout 12

Higher-Order Types
(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 used in useful ways, especially when used with type classes.
The Functor Class

- The Functor class demonstrates the use of high-order types:
  
  ```haskell
  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 "* \( \rightarrow \) *".

- For example:
  ```haskell
  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:
  ```haskell
  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:

  ```haskell
  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:

  - \(\text{do } e \rightarrow e\)
  - \(\text{do } e_1; e_2; \ldots; \text{en } \rightarrow e_1 >> \text{do } e_2 ; \ldots; \text{en}\)
  - \(\text{do } \text{pat } <- e_1 ; e_2 ; \ldots; \text{en } \rightarrow\)
    - \(\text{let ok pat } = \text{do } e_2 ; \ldots; \text{en}\)
    - \(\text{ok } = \text{fail "..."}\)
    - \(\text{in } e_1 >>= \text{ok}\)
  - \(\text{do let decllist } ; e_2 ; \ldots; \text{en } \rightarrow\)
    - \(\text{let decllist in do } e_2 ; \ldots; \text{en}\)

- Note special case of rule 3:
  - 3a. \(\text{do } x <- e_1 ; e_2 ; \ldots; \text{en } \rightarrow\)
    - \(e_1 >>= \lambda x \rightarrow \text{do } e_2 ; \ldots; \text{en}\)
Example Involving IO

- “do” syntax can be completely eliminated using these rules:

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

⇒ putStr "Hello" >>
do c <- getChar
   return c

⇒ putStr "Hello" >>
   getChar >>= \c ->
do return c

⇒ putStr "Hello" >>
   getChar >>= \c ->
   return c

⇒ putStr "Hello" >>
   getChar >>= return
```

-- by rule (2)
-- by rule (3a)
-- by rule (1)
-- by currying
Functor and Monad Laws

- **Functor laws:**
  
  \[
  \begin{align*}
  \text{fmap \ id} &= \text{id} \\
  \text{fmap} \ (f \ . \ g) &= \text{fmap} \ f \ . \ \text{fmap} \ g
  \end{align*}
  \]

- **Monad laws:**
  
  \[
  \begin{align*}
  \text{return} \ a \ >>= \ k &= k \ a \\
  m \ >>= \ \text{return} &= m \\
  m \ >>= \ (\lambda x \rightarrow k \ x \ >>= \ h) &= (m >>= k) >>= h
  \end{align*}
  \]

  **Note special case of last law:**

  \[
  m1 \ >> (m2 \ >> \ m3) = (m1 \ >> m2) \ >> \ m3
  \]

- **Connecting law:**

  \[
  \text{fmap} \ f \ xs = xs >>= (\text{return} \ . \ f)
  \]
Monad Laws Expressed using “do” Syntax

- \[ \text{do } x \leftarrow \text{return } a \ ; \ k x \] = \[ k a \]
- \[ \text{do } x \leftarrow m \ ; \ \text{return } x \] = \[ m \]
- \[ \text{do } x \leftarrow m \ ; \ y \leftarrow k x \ ; \ h y \] = \[ \text{do } y \leftarrow (\text{do } x \leftarrow m \ ; \ k x) \ ; \ h y \]
- \[ \text{do } m_1 \ ; m_2 \ ; m_3 \] = \[ \text{do } (\text{do } m_1 \ ; m_2) \ ; m_3 \]
- \[ \text{fmap } f \ xs \] = \[ \text{do } x \leftarrow xs \ ; \ \text{return } (f x) \]

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

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

or, after desugaring: \texttt{putStr "Hello" >> getChar}
The Maybe Monad

召回 Maybe 数据类型:

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

它是双 Functo 以及 Mono 体:

```haskell
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)
```

这些实例确实是“守法的”。
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:

  \[
  f \ x \ggg \ \lambda y \rightarrow \\
  g \ y \ggg \ \lambda z \rightarrow \ \text{return} \ \ z
  \]

  \[
  \Rightarrow \\
  f \ x \ggg \ \lambda y \rightarrow \\
  g \ y
  \]

  \[
  \Rightarrow \\
  f \ x \ggg \ \lambda y \rightarrow \\
  g (\ f \ x \ )
  \]

  \[
  \Rightarrow \\
  f \ x \ggg \ g
  \]

- So we started with \( g ( f \ x) \) and ended with \( f \ x \ggg \ g \).
The List Monad

- The List data type is also a Monad:

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

- For example:

  ```haskell
  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:

  ```haskell
  [(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:

```haskell
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
  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:

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

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

```hs
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:

```haskell
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 sml = fsm1 a1
        (s2, a2) = sml s1
    in (s2, a2)
```

Note the partial application of the type constructor SM in the instance declaration. This works because SM has kind \( * \rightarrow * \rightarrow * \), so “SM s” has kind \( * \rightarrow * \).