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 "* -> *":

- 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
    return :: a -> m a
    fail :: String -> m a
  ```

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

Syntactic Mystery Unveiled

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

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

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

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

do putStr "Hello" >> -- by rule (3a)
  getChar >> c <-
    do return c

do putStr "Hello" >> -- by rule (1)
  getChar >> c <-
    do return c

do putStr "Hello" >> -- by currying
  getChar >>
    return c
```
Functor and Monad Laws

- Functor laws:
  \[ \text{fmap id} = \text{id} \]
  \[ \text{fmap} (f \cdot g) = \text{fmap} f \cdot \text{fmap} g \]

- Monad laws:
  \[ \text{return } a \triangledown \triangleright k = k a \]
  \[ a \triangledown \triangleright \text{return } = a \]
  \[ a \triangledown \triangleright (b \triangledown \triangleright k) = (a \triangledown \triangleright b) \triangledown \triangleright k \]
  \[ \text{fmap} f \triangledown \triangleright (a \triangledown \triangleright b) = (\text{fmap} f \triangledown \triangleright a) \triangledown \triangleright b \]

- Connecting law:
  \[ \text{fmap} f \triangledown \triangleright \text{xs} = \text{xs} \triangledown \triangleright (\text{return} \cdot f) \]

Monad Laws Expressed using “do” Syntax

- \[ \text{do } x \leftarrow \text{return } a ; k x \]
  \[ \text{return } x \]
  \[ \text{do } x \leftarrow a ; y \leftarrow k x ; h y \]
  \[ \text{do } x \leftarrow a ; y \leftarrow k x \]
  \[ \text{do } x \leftarrow a ; \text{return } b \]

- Note special case of last law:
  \[ a \triangledown \triangleright (b \triangledown \triangleright c) = (a \triangledown \triangleright b) \triangledown \triangleright c \]

- For example, using the second rule above, the example given earlier can be simplified to just:
  \[ \text{do putStr } \text{"Hello" } \text{getChar} \]

  or, after desugaring: \[ \text{putStr } \text{"Hello" } \text{>> getChar} \]

The Maybe Monad

- Recall the Maybe data type:
  \[ \text{data Maybe } a \equiv \text{Just } a \mid \text{Nothing} \]
  \[ \text{instance Monad Maybe where} \]
  \[ \text{Just } x \triangledown \triangleright k = k x \]
  \[ \text{Nothing} \triangledown \triangleright k = \text{Nothing} \]

- It is both a Functor and a Monad:
  \[ \text{instance Functor Maybe where} \]
  \[ \text{fmap } f \text{ Nothing} = \text{Nothing} \]
  \[ \text{fmap } f \text{ Just } x = f x \]

- These instances are indeed “law abiding”.

Using the MaybeMonad

- Consider the expression \[ g (f \ x) \]. Suppose that both \( f \) and \( g \) could return errors that are encoded as \( \text{Nothing} \). We might do:
  \[ \text{case } f \ x \text{ of} \]
  \[ \text{Nothing} \rightarrow \text{Nothing} \]
  \[ \text{Just } y \rightarrow \text{case } g \ y \text{ of} \]
  \[ \text{Nothing} \rightarrow \text{Nothing} \]
  \[ \text{Just } z \rightarrow \ldots \text{proper result using } z \ldots \]

- But since Maybe is a Monad, we could instead do:
  \[ \text{do } y \leftarrow f \ x \]
  \[ \text{return } y \leftarrow g \ y \]
  \[ \text{return } \ldots \text{proper result using } y \ldots \]

Simplifying Further

- Note that the last expression can be desugared and simplified as follows:
  \[ f x \triangledown \triangleright \text{Just } y \rightarrow g y \triangledown \triangleright \text{return } x \rightarrow f x \triangledown \triangleright \text{Just } y \rightarrow f x \triangledown \triangleright g \rightarrow g y \rightarrow \text{return } \]

- So we started with \( g (f \ x) \) and ended with \( f x \triangledown \triangleright g \).

The List Monad

- The List data type is also a Monad:
  \[ \text{instance Monad } [] \text{ where} \]
  \[ \text{return } a = [a] \]
  \[ \text{fail } x = [\text{Nothing}] \]

- For example:
  \[ \text{do } x \leftarrow [1,2,3] \rightarrow [4,5] \]
  \[ \text{return } (x,y) \rightarrow [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)] \]

- Note that this is the same as:
  \[ [(x,y) | x \leftarrow [1,2,3], y \leftarrow [4,5]] \]

  Indeed, list comprehension syntax is an alternative to \text{do} syntax, for the special case of lists.
Useful Monad Operations

- `sequence :: Monad m => [m a] -> m [a]`
  - `sequence = foldr mcons (return [])`
    - `mcons p q = do x <- p; xs <- q; return (x:xs)`
- `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)`
- `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)`
  - `writeFile :: String -> String -> IO ()`
- 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 a`” has kind `* -> *`.