The Type of a Type

- In previous chapters we discussed:
  - Monomorphic types such as \texttt{Int}, \texttt{Bool}, etc.
  - Polymorphic types such as \texttt{[a]}, \texttt{Tree a}, etc.
  - Monomorphic instances of polymorphic types such as \texttt{[Int]}, \texttt{Tree Bool}, etc.
  - \texttt{Int}, \texttt{Bool}, etc. are nullary type constructors, whereas \texttt{[]}, \texttt{Tree}, etc. are unary type constructors. \texttt{FiniteMap} is a binary type constructor.
- The "type of a type" is called a kind. The kind of all monomorphic types is written "*":
  \[
  \text{Int}, \text{Bool}, \texttt{[Int]}, \texttt{Tree Bool} :: \ast
  \]
- Therefore the type of unary type constructors is:
  \[
  [], \text{Tree} :: \ast \to \ast
  \]
- 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:
  ```haskell
  class Functor f where
    fmap :: (a -> b) -> f a -> f b
  ```
- Note that \texttt{f} is applied here to one (type) argument, so should have kind "* \to *".
- 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 \texttt{mapTree} previously defined:
  ```haskell
  instance Functor Tree where
    fmap = mapTree
  ```
- Exercise: Write the instance declaration for \texttt{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
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
  ```
  -- default implementations:
  \[
  m >> k = m >>= (\_ -> k)
  fail s = error s
  \]
- The key operations are (\texttt{>>=}) and \texttt{return}.

Syntactic Mystery Unveiled

- The "do" syntax in Haskell is shorthand for Monad operations, as captured by these rules:
  ```haskell
  do e \equiv e
  do e1; e2; ...; en \equiv e1 >> (do e2 ; ...; en)
  do let decllist ; e2 ; ...; en \equiv
test
  do let declist in (do e2 ; ...; en)
  ```
- Note special case of rule 3:
  3a. \texttt{do x <- e1 ; e2 ; ...; en \equiv \texttt{do x <- e1 ; e2 ; ...; en}}

Example Involving IO

- "do" syntax can be completely eliminated using these rules:
  ```haskell
  do putStrLn "Hello"
      c <- getChar
      return c
  do putStrLn "Hello" >> -- by rule (2)
      c <- getChar
      return c
  do putStrLn "Hello" >> -- by rule (3a)
      c <- getChar >> \c -> do return c
  do putStrLn "Hello" >> -- by rule (1)
  do putStrLn "Hello" >> -- by currying
      getChar >> \c -> return c
  ```
Functor and Monad Laws

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

**Monad laws:**
- \( \text{return a >>= k} = k a \)
- \( \text{m >>= \text{return}} = \text{m} \)
- \( \text{m >>= (\lambda x \rightarrow \text{f x} >>= h)} = \text{(m >>= f)} >>= h \)

Note special case of last law:
- \( m1 >> (m2 >> m3) = (m1 >> m2) >> m3 \)

**Connecting law:**
- \( \text{fmap f xs} = xs >>= (\text{return} \cdot f) \)

Monad Laws Expressed using “do” Syntax

- \( \text{do x <- return a ; k x} = k a \)
- \( \text{do x <- m ; return x} = m \)
- \( \text{do x <- m ; y <- \text{f x} ; h y} = \text{do y <- (do x <- m \cdot k x) ; h y} \)
- \( \text{do m1 ; m2 ; m3} = \text{do (do m1 ; m2) ; m3} \)
- \( \text{fmap f xs} = \text{do x <- xs ; return (f x)} \)

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

\[
\text{do putStr "Hello" \ overwhelmed by getChar}
\]
or, after desugaring:

\[
\text{putStr "Hello" >> getChar}
\]

The Maybe Monad

**Recall the Maybe data type:**
- \( \text{data Maybe a} = \text{Just a} \ |
\text{Nothing} \)

**It is both a Functor and a Monad:**
- \( \text{instance Monad Maybe where} \)
  - \( \text{Just x} >>= k = k x \)
  - \( \text{Nothing} >>= k = \text{Nothing} \)
  - \( \text{return x} = \text{Just x} \)
  - \( \text{fail x} = \text{Nothing} \)

**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:

```
\text{do y <- f x} \\
\text{z <- g y} \\
\text{return ... proper result using z...}
```

Simplifying Further

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

- \( f x >>= \\lambda y \rightarrow \text{\_} = f x >>= \\lambda y \rightarrow \text{\_} \rightarrow \text{return z} \)
- \( g y >>= \\lambda z \rightarrow \text{\_} = g y >>= \text{\_} \rightarrow \text{return}\ \text{\_} \rightarrow g y \)

So we started with \( g (f x) \) and ended with \( f x >>= \text{\_} \).

The List Monad

The List data type is also a Monad:

- \( \text{instance Monad [\_] where} \)
  - \( \text{m >>= k} = \text{concat (map k m)} \)
  - \( \text{return x} = [x] \)
  - \( \text{fail x} = [] \)

For example:

```
\text{do x <- [1,2,3]} \\
\text{y <- [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:

\( [(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]] \)

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]
  
  ```haskell```
  `sequence = foldr mcons (return [])`
  ```

  where mcons p q = do x <- p
                      xs <- q
                      return (x:xs)
  ```

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

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

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

- **(=<<)**: Monad m => (a -> m b) -> m a -> m b
  
  ```haskell```
  `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
                                  (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:
  
  ```haskell```
  `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):
  
  ```haskell```
  `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:
  
  ```haskell```
  `readFile :: String -> IO (Maybe String)`
  ```

  `writeFile :: String -> String -> IO ()`
  ```

  `fileExists :: String -> IO Bool`
  ```

- 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 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 * -> *.