THE IO MONAD
Kathleen Fisher for cs242 at Tufts
Lightly edited with permission, Michelle Strout 4/13/15

Reading: "Tackling the Awkward Squad," Sections 1-2
"Real World Haskell," Chapter 7: I/O

Thanks to Simon Peyton Jones for many of these slides.

Why Monads?

- Predictive parser
  - Passed around a list of tokens while processing.
- PA3 MeggyJava compiler
  - Passed around a number to create unique labels for code generation.
- PA4 MeggyJava compiler
  - Passing around a symbol table with parameter and method type and code generation information.
- Monads will help us abstract away some of that passing around. Stay tuned til Wednesday.

Beauty...

Functional programming is beautiful:
- Concise and powerful abstractions
  - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
- Close correspondence with mathematics
  - Semantics of a code function is the math function
  - Equational reasoning: if \( x = y \), then \( f(x) = f(y) \)
  - Independence of order-of-evaluation (Church-Rosser)

...and the Beast

- But to be useful as well as beautiful, a language must manage the "Awkward Squad":
  - Input/Output
  - Imperative update
  - Error recovery (eg, timing out, catching divide by zero, etc.)
  - Foreign-language interfaces
  - Concurrency

The whole point of a running a program is to affect the real world, an "update in place."
The Direct Approach

- Do everything the “usual way”:
  - I/O via “functions” with side effects:
    ```
    putchar 'x' + putchar 'y'
    ```
  - Imperative operations via assignable reference cells:
    ```
    z = ref 0; z := t + 1;
    f(z);
    w = t   (* What is the value of w? *)
    ```
  - Error recovery via exceptions
  - Foreign language procedures mapped to “functions”
  - Concurrency via operating system threads
  - Ok if evaluation order is baked into the language.

The Lazy Hair Shirt

In a lazy functional language, like Haskell, the order of evaluation is deliberately undefined, so the “direct approach” will not work.

- Consider: ```res = putchar 'x' + putchar 'y'```  
  - Output depends upon the evaluation order of (+).
- Consider: ```ls = [putchar 'x', putchar 'y']```  
  - Output depends on how the consumer uses the list. If only used in length ls, nothing will be printed because length does not evaluate elements of list.

Tackling the Awkward Squad

- Laziness and side effects are **incompatible**.
- Side effects are **important!**
- For a long time, this tension was embarrassing to the lazy functional programming community.
- In early 90’s, a surprising solution (**the monad**) emerged from an unlikely source (**category theory**).
- Haskell’s **IO monad** provides a way of tackling the awkward squad: I/O, imperative state, exceptions, foreign functions, & concurrency.

Monadic Input and Output
A functional program defines a pure function, with no side effects. The whole point of running a program is to have some side effect.

A value of type \((\text{IO} \ t)\) is an "action." When performed, it may do some input/output before delivering a result of type \(t\).

\[
\text{type IO } t = \text{World} \to (t, \text{World})
\]

A value of type \((\text{IO} \ t)\) is an "action." When performed, it may do some input/output before delivering a result of type \(t\).

- "Actions" are sometimes called "computations."
- An action is a first-class value.
- Evaluating an action has no effect; performing the action has the effect.
**Simple I/O**

- `getChar` :: IO Char
- `putChar` :: Char -> IO ()
- `main` :: IO ()
  ```
  main = putChar 'x'
  ```

Main program is an action of type IO ()

To read a character and then write it back out, we need to connect two actions.

- `getChar` :: IO Char
- `putChar` :: Char -> IO ()

The "bind" combinator lets us make these connections.

**The Bind Combinator (>>=)**

- `getChar` :: IO Char
- `putChar` :: Char -> IO ()
- `echo` :: IO ()
  ```
  echo = getChar >>= putChar
  ```

We have connected two actions to make a new, bigger action.

**The (>>=) Combinator**

- `a` -> `b`

- `Operator is called bind because it binds the result of the left-hand action in the action on the right.`

- `Performing compound action a >>= \(x \rightarrow b\):`
  - Performs action `a`, to yield value `x`
  - Applies function `\(x \rightarrow b\)` to `x`
  - Performs the resulting action `b(x <- r)`
  - Returns the resulting value `v`
The parentheses are optional because lambda abstractions extend “as far to the right as possible.”

The `putChar` function returns unit, so there is no interesting value to pass on.

---

We want to return \((c1, c2)\).

But, \((c1, c2) :: (Char, Char)\)

And we need to return something of type `IO(Char, Char)`

We need to have some way to convert values of “plain” type into the I/O Monad.

---

The “then” combinator \((>>\)) does sequencing when there is no value to pass:

\[
\text{m} \gg \text{n} = \text{m} \gg (\_ \rightarrow \text{n})
\]

```
> echoDup :: IO ()
> echoDup = getChar >>= (\c ->
> getChar >>= (\c2 ->
> putChar c >>= (\() ->
> putChar c))
```

```
> echoTwice :: IO ()
> echoTwice = echo >> echo
```

---

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= (\c1 ->
getChar >>= (\c2 ->
return (c1,c2))
```

---

The `return` combinator

The action \((\text{return } v)\) does no IO and immediately returns \(v\):

\[
\text{return} :: a \rightarrow \text{IO } a
\]

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= (\c1 ->
getChar >>= (\c2 ->
return (c1,c2))
```
The “do” Notation

- The “do” notation adds syntactic sugar to make monadic code easier to read.

- Do syntax designed to look imperative.

```
-- Plain Syntax
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>\> \c1 \-> getChar >>\> \c2 \->
              return (\c1,\c2)

-- Do Notation
getTwoCharsDo :: IO (Char,Char)
getTwoCharsDo = do {\c1 \<- getChar ;
                      \c2 \<- getChar ;
                      return (\c1,\c2) }
```

Desugaring “do” Notation

- The “do” notation only adds syntactic sugar:

```
do { x\<-e; es } = e \>>\> x \-> do { es }
do { e; es } = e \>>> do { es }
do { e } = e
```

- The scope of variables bound in a generator is the rest of the “do” expression.

- The last item in a “do” expression must be an expression.

Syntactic Variations

- The following are equivalent:

```
do { x1 \<- pl; \ldots; xn \<- pn, q }
do x1 \<- pl; \ldots; xn \<- pn, q
```

- If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.

Bigger Example

- The `getLine` function reads a line of input:

```
getLine :: IO [Char]
getLine = do { c \<- getChar ;
              if c == '\n' then
                return []
              else
                do { cs \<- getLine;
                     return (c:cs) })
```

Note the “regular” code mixed with the monadic operations and the nested “do” expression.
Each action in the IO monad is a possible stage in an assembly line.

For an action with type `IO a`, the type tag `IO` as suitable for the IO assembly line via the `IO` type constructor.

Indicates that the kind of thing being passed to the next stage in the assembly line has type `a`.

The `bind` operator "snaps" two stages `s1` and `s2` together to build a compound stage.

The `return` operator converts a pure value into a stage in the assembly line.

The assembly line does nothing until it is turned on.

The only safe way to "run" an IO assembly is to execute the program, either using `ghci` or running an executable.

Values of type `(IO t)` are first class, so we can define our own control structures.

```
forever :: IO () -> IO ()
forever a = a >>= forever a

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = a >>= repeatN (n-1) a
```

Example use:
```
Main> repeatN 5 (putChar 'h')
```

Values of type `(IO t)` are first class, so we can define our own control structures.

```
for :: [a] -> (a -> IO b) -> IO ()
for [] = return ()
for (x:xs) fa = fa x >>= for xs fa
```

Example use:
```
Main> for [1..10] (\x -> putStrLn (show x))
```
**Sequencing**

A list of IO actions.

An IO action returning a list.

```haskell
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a:as) = do { r <- a; 
    rs <- sequence as; 
    return (r:rs) }
```

Example use:

```haskell
Main> sequence [getChar, getChar, getChar]
```

**First Class Actions**

**Slogan:** First-class actions let programmers write application-specific control structures.

**IO Provides Access to Files**

- The IO Monad provides a large collection of operations for interacting with the "World."
- For example, it provides a direct analogy to the Standard C library functions for files:

```haskell
openFile :: String -> IOMode -> IO Handle
hPutStr :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
hClose :: Handle -> IO ()
```

**The IO Monad as ADT**

- All operations return an IO action, but only bind (>>=) takes one as an argument.
- Bind is the only operation that combines IO actions, which forces sequentiality.
- Within the program, there is no way out!
GHC uses world-passing semantics for the IO monad:

It represents the "world" by an un-forgeable token of type World, and implements bind and return as:

```
return :: a -> IO a
return x = \w -> (x,w)
(`` >>=``) :: IO a -> (a -> IO b) -> IO b
(`` >>=``) m k = \w -> case m w of (r,w') -> k r w'
```

Using this form, the compiler can do its normal optimizations. The dependence on the world ensures the resulting code will still be single-threaded.

The code generator then converts the code to modify the world "in-place:"

```
return :: a -> IO a
return a = \w -> (a,w)
(`` >>=``) :: IO a -> (a -> IO b) -> IO b
(`` >>=``) m k = \w -> case m w of (r,w') -> k r w'
```

What makes the IO Monad a Monad?

A monad consists of:

- A type constructor M
- A function bind :: M a -> (a -> M b) -> M b
- A function return :: a -> M a

Plus:
Laws about how these operations interact.

```
return x >>= f = f x
m >>= return = m
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

```
(`` >>=``) :: IO a -> IO b -> IO b
m >>= n = m >>= (\_ -> n)
```

done :: IO ()
done = return ()

done >> m = m
m >> done = m
m_1 >> (m_2 >> m_3) = (m_1 >> m_2) >> m_3

```
```
Reasoning

Using the monad laws and equational reasoning, we can prove program properties.

Proposition:
\[ \text{putStr } r \gg \text{putStr } s = \text{putStr } (r ++ s) \]

Proof: By induction on \( r \).

Base case: \( r \) is \([\ ]\).

\[ \text{putStr } [\ ] \gg \text{putStr } s = (\text{definition of putStr}) \]
\[ \text{done } \gg \text{putStr } s = (\text{first monad law for } \gg) \]
\[ \text{putStr } s = (\text{definition of } ++) \]
\[ \text{putStr } ([\ ] ++ s) \]

Induction case: \( r \) is \([c : cs]\)...

Summary

- A complete Haskell program is a single IO action called `main`. Inside IO code is single-threaded.
- Big IO actions are built by gluing together smaller ones with bind (\( \gg \)) and by converting pure code into actions with `return`.
- IO actions are first-class.
  - They can be passed to functions, returned from functions, and stored in data structures.
  - So it is easy to define new "glue" combinators.
- The IO Monad allows Haskell to be pure while efficiently supporting side effects.
- The type system separates the pure from the effectful code.

A Monadic Skin

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because it is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- So it is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.
Recit Example: Monads in 15 minutes

- Removing syntactic sugar to help in understanding

```haskell
solveConstraints = do
  x <- choose [1,2,3]
  y <- choose [4,5,6]
  guard (x*y==8)
  return (x,y)

solveConstraint' :: Choice (Int,Int)
solveConstraint' =
  choose [1,2,3] >>= (\x ->
    choose [4,5,6] >>= (\y ->
      guard (x*y==8) >>= (\_ ->
        return (x,y)))))

choices >>= f = join (map f choices)
```

Recit Example: Monads in 15 minutes

- Using definition of bind operator (>>=)

```haskell
solveConstraint' =
  choose [1,2,3] >>= (\x ->
    choose [4,5,6] >>= (\y ->
      guard (x*y==8) >>= (\_ ->
        return (x,y)))))

solveConstraint'' =
  join (map (\x ->
    join (map (\y ->
      join (map (\_ ->
        return (x,y))
      (guard (x*y==8))))
    (choose [4,5,6])))
  (choose [1,2,3]))
```