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.

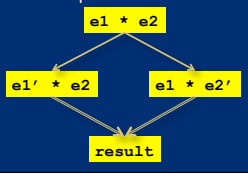
Why Monads?

- 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)



The compiler can choose the best order in which to do evaluation, including skipping a term if it is not needed.

...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 := !z + 1;
f(z);
w = !z     (* 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

The Problem

A functional program defines a pure function, with no side effects.

Tension

The whole point of running a program is to have some side effect.

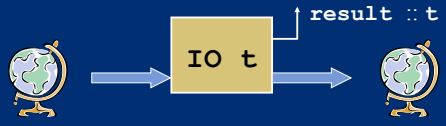
Monadic I/O: The Key Idea

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

A Helpful Picture

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

```
type IO t = World -> (t, World)
```



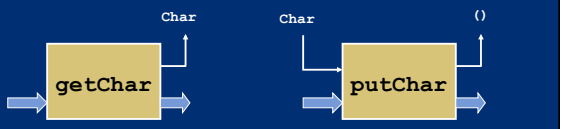
Actions are First Class

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

```
type IO t = World -> (t, World)
```

- "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



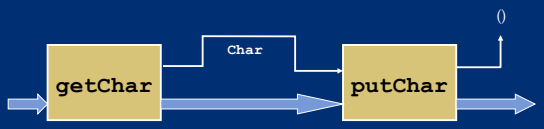

```
getChar :: IO Char
putChar :: Char -> IO ()

main :: IO ()
main = putChar 'x'
```

Main program is an action of type IO ()

Connection Actions

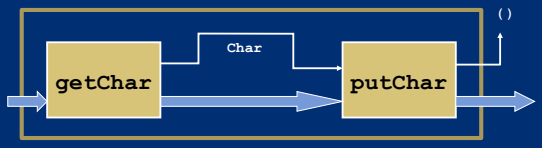
To read a character and then write it back out, we need to connect two actions.



The "bind" combinator lets us make these connections.

The Bind Combinator (>>=)

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

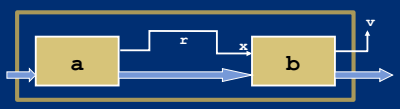


- We have connected two actions to make a new, bigger action.

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

The (>>=) Combinator

- 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->b`:
  - performs action `a`, to yield value `r`
  - applies function `\x->b` to `r`
  - performs the resulting action `b{x <- r}`
  - returns the resulting value `v`



Printing a Character Twice

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

- 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.

The (>>) Combinator

- The "then" combinator (>>) does sequencing when there is no value to pass:

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

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

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

Getting Two Characters

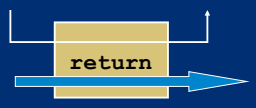
```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar     >>= \c1 ->
              getChar     >>= \c2 ->
              ????
```

- 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 `return` Combinator

- The action (`return v`) does no IO and immediately returns `v`:

```
return :: a -> 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.

```
-- 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) }
```

- Do syntax designed to look imperative.

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
do {let ds; es}	**=**	**let ds in do {es}**

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 <- p1; ...; xn <- pn; q }
```

```
do   x1 <- p1; ...; xn <- pn; q
```

```
do x1 <- p1
   ...
   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:

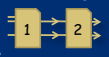
```
getLine :: IO [Char]
getLine = do { c <- getChar ;
               if c == '\n' then
                    return []
               else
                    do { cs <- getLine;
                         return (c:cs) }}
```

Note the "regular" code mixed with the monadic operations and the nested "do" expression.

An Analogy: Monad as Assembly Line

- Each action in the IO monad is a possible stage in an assembly line.

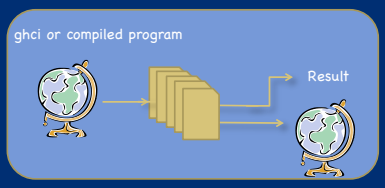
- For an action with type `IO a`, the type
  - tags the action as suitable for the IO assembly line via the `IO` type constructor.
  - indicates that the kind of thing being passed to the next stage in the assembly line has type `a`.
- The bind operator "snaps" two stages s1 and s2 together to build 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.

Powering the Assembly Line

- Running the program turns on the IO assembly line.
- The assembly line gets "the world" as its input and delivers a result and a modified world.
- The types guarantee that the world flows in a single thread through the assembly line.



Control Structures

- 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')
```

For Loops

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

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

- Example use:

```
Main> for [1..10] (\x -> putStr (show x))
```

Sequencing

A list of IO actions.

An IO action returning a list.

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

- Example use:

```
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:

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

The IO Monad as ADT

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b

getChar :: IO Char
putChar :: Char -> IO ()
... more operations on characters ...

openFile :: [Char] -> IOMode -> IO Handle
... more operations on files ...

newIORef :: a -> IO (IORef a)
... more operations on references ...
```

- 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!

Implementation

- GHC uses world-passing semantics for the IO monad:

```
type IO t = World -> (t, World)
```

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

```
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'
```

- 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."

Monads

- 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.

Monad Laws

return x >>= f = f x

m >>= return = m

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

Derived Laws for (>>) and done

```
(>>) :: 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.

```
putStr :: String -> IO ()
putStr [] = done
putStr (c:s) = putChar c >> putStr s
```

Proposition:

```
putStr r >> putStr s = putStr (r ++ s)
```

```
putStr :: String -> IO ()
putStr [] = done
putStr (c:cs) = putChar c >> putStr cs
```

Proposition:

```
putStr r >> putStr s = putStr (r ++ s)
```

Proof: By induction on r.

Base case: r is []

```
   putStr [] >> putStr s
 = (definition of putStr)
   done >> putStr s
 = (first monad law for >>)
   putStr s
 = (definition of ++)
   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 (>>=) 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

```
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))))
```

Recit Example: Monads in 15 minutes

- Using definition of bind operator (>>=)

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

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

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