**THE IO MONAD**

Kathleen Fisher

Thanks to Simon Peyton Jones for many of these slides.

---

---

**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.

Monadic Input and Output

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.

The Problem

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

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.

- “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

Main program is an action of type IO ()

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

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)

World in → IO t → result :: t → World out
Connection Actions

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

\[
\text{getChar} \rightarrow \text{Char} \rightarrow \text{putChar} \rightarrow 0
\]

The (\(\gg\gg\)) Combinator

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

\[
\text{echo} \equiv \text{getChar} \gg\gg \text{putChar}
\]

The (\(\gg\gg\)) Combinator

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

- Performing compound action \(a \gg\gg \lambda x. b\):
  - performs action \(a\), to yield value \(r\)
  - applies function \(\lambda x. b\) to \(r\)
  - performs the resulting action \(b[x \leftarrow r]\)
  - returns the resulting value \(v\)

\[
\text{a} \rightarrow r \rightarrow x \rightarrow v
\]

Printing a Character Twice

- 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:

\[
(\gg\gg) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b
\]

\[
m \gg\gg n = m \gg\gg (\_ \rightarrow n)
\]

\[
\text{echoDup :: IO ()}
\]

\[
\text{echoDup = getChar} \gg\gg \backslash c \rightarrow
\text{putChar } c \gg
\text{putChar } c
\]

\[
\text{echoTwice :: IO ()}
\]

\[
\text{echoTwice = echo} \gg\gg \text{echo}
\]

Getting Two Characters

- We want to return \((c_1, c_2)\).
  - But, \((c_1, c_2) :: \text{(Char, Char)}\)
  - And we need to return something of type \text{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 (\text{return } v) does no IO and immediately returns \(v\):

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

\[
\text{return}
\]

\[
\text{getTwoChars :: IO (Char,Char)}
\]

\[
\text{getTwoChars = getChar} \gg\gg \backslash c_1 \rightarrow
\text{getChar} \gg\gg \backslash c_2 \rightarrow
\text{return } (c_1, c_2)
\]

The “do” Notation

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

--- Plain Syntax

\[
\text{getTwoChars :: IO (Char,Char)}
\]

\[
\text{getTwoChars = getChar} \gg\gg \backslash c_1 \rightarrow
\text{getChar} \gg\gg \backslash c_2 \rightarrow
\text{return } (c_1, c_2)
\]

--- Do Notation

\[
\text{getTwoCharsDo :: IO(Char,Char)}
\]

\[
\text{getTwoCharsDo = do \{ c_1 \leftarrow \text{getChar} ;
\text{c2 \leftarrow \text{getChar} ;
\text{return } (c_1, c_2) \}}
\]

- 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
  ```

  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 == '
' 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 (\(\text{IO } t\)) are first class, so we can define our own control structures.

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

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

- Example use:

```haskell
Main> drepeatN 5 (putChar 'h')
```

For Loops

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

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

- Example use:

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

Sequencing

- A list of IO actions.
- An IO action returning a list.

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

References

- The IO operations let us write programs that do I/O in a strictly sequential, imperative fashion.
- Idea: We can leverage the sequential nature of the IO monad to do other imperative things!

```
data IORef a -- Abstract type
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

A value of type `IORef a` is a reference to a mutable cell holding a value of type `a`.

Example Using References

```
import Data.IORef  -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  { r <- newIORef 0;
    loop r 1 }
where
  loop :: IORef Int -> Int -> IO Int
  loop r i | i > n     = readIORef r
             | otherwise = do
          { v <- readIORef r;
            writeIORef r (v + i);
            loop r (i+1)}
```

But this is terrible! Contrast with: `sum [1..n]`. Claims to need side effects, but doesn’t really.
Example Using References

```haskell
import Data.IORef -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  { r <- newIORef 0;
    loop r 1 }
where
  loop :: IORef Int -> Int -> IO Int
  loop r i | i > n     = readIORef r
            | otherwise = do
              { v <- readIORef r;
                writeIORef r (v + i);
                loop r (i+1) }
```

Just because you can write C code in Haskell, doesn’t mean you should!

A Second Example

- Track the number of chars written to a file.

```haskell
type HandleC = (Handle, IORef Int)
openFileC :: String -> IOMode -> IO HandleC
openFileC fn mode = do
  { h <- openFile fn mode;
    v <- newIORef 0;
    return (h,v)           }
hPutStrC :: HandleC -> String -> IO()
hPutStrC (h,r) cs = do
  { v <- readIORef r;
    writeIORef r (v + length cs);
    hPutStr h cs                 }
```

- Here it makes sense to use a reference.

The IO Monad as ADT

```haskell
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!

Irksome Restriction?

- Suppose you wanted to read a configuration file at the beginning of your program:

```haskell
configFileContents :: [String]
configFileContents = lines (readFile "config") -- WRONG!
useOptimisation :: Bool useOptimisation = "optimise" `elem` configFileContents
```

- The problem is that readFile returns an IO String, not a String.
- Option 1: Write entire program in IO monad. But then we lose the simplicity of pure code.
- Option 2: Escape from the IO Monad using a function from IO String -> String. But this is the very thing that is disallowed!
Taking off the Safety Helmet

- Reading a file is an I/O action, so in general it matters when we read the file relative to the other actions in the program.
- In this case, however, we are confident the configuration file will not change during the program, so it doesn’t really matter when we read it.
- This situation arises sufficiently often that Haskell implementations offer one last unsafe I/O primitive: `unsafePerformIO`.

```haskell
unsafePerformIO :: IO a -> a
configFileContents :: [String]
configFileContents = lines (unsafePerformIO (readFile "config"))
```

unsafePerformIO

- As its name suggests, `unsafePerformIO` breaks the soundness of the type system.

```haskell
r :: IORef c
r = unsafePerformIO (newIORef (error "urk"))
cast :: a -> b
cast x = unsafePerformIO (do {writeIORef r x; readIORef r})
```

So claims that Haskell is type safe only apply to programs that don’t use `unsafePerformIO`.

Similar examples are what caused difficulties in integrating references with Hindley/Milner type inference in ML.

unsafePerformIO

- The operator has a deliberately long name to discourage its use.
- Its use comes with a proof obligation: a promise to the compiler that the timing of this operation relative to all other operations doesn’t matter.

Implementation

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

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

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'
```

- It represents the “world” by an un-forgeable token of type `World`, and implements `bind` and `return`.

- 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 \( \text{bind} :: M a \to (a \to M b) \to M b \)
  - A function \( \text{return} :: a \to M a \)
- Plus:
  - Laws about how these operations interact.

Monad Laws

\[
\text{return } x \gg= f = f x
\]
\[
m \gg= \text{return} = m
\]
\[
m_1 \gg= (\lambda x. m_2 \gg= (\lambda y. m_3)) = (m_1 \gg= (\lambda x. m_2)) \gg= (\lambda y. m_3)
\]
x not in free vars of \( m_3 \)

Derived Laws for (\(\gg\)) and \textit{done}

\[
(\gg) :: \text{IO } a \to \text{IO } b \to \text{IO } b
\]
\[
m \gg n = m \gg= (\lambda \_ \to n)
\]

\[
\text{done} :: \text{IO } ()
\]

\[
\text{done} = \text{return } ()
\]

\[
\text{done} \gg m = m
\]

\[
m \gg \text{done} = m
\]

\[
m_1 \gg (m_2 \gg m_3) = (m_1 \gg m_2) \gg m_3
\]

Reasoning

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

Proposition:

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

\[
\text{putStr} :: \text{String } \to \text{IO } ()
\]

\[
\text{putStr } [] = \text{done}
\]

\[
\text{putStr } (c:s) = \text{putChar } c \gg \text{putStr } s
\]
**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 \((c:cs)\) ...

---

**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.

---

**Monads**

- So far, we have only seen one monad, but there are many more!

- We’ll see a bunch more of them on Wednesday.

---

**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.