The IO Monad

John Mitchell
previously formulated by K Fisher
some slides from Simon Peyton Jones

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

Announcements

• Midterm exam
  – Wed 10/26, 7-9PM, Nvidia Auditorium
  – Local SCPD students are required to come to campus
  – Closed book, one page of notes allowed (by vote)
• Homework 3
  – Due Wed 5PM
• Homework 4
  – Ungraded problems with solutions Mon 10/24

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 mathematical function
    • Equational reasoning: if x = y, then f x = f y
  – Independence of order-of-evaluation (Confluence, aka 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, timeout, divide by zero, etc.)
  – Foreign-language interfaces
  – Concurrency control

The Direct Approach

• Just add imperative constructs “the usual way”
  – I/O via “functions” with side effects:

  ```haskell
  putStrLn 'x' + putStrLn 'y'
  ```
  – Imperative operations via assignable reference cells:

  ```haskell
  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
• Can work if language determines evaluation order
  – Ocaml, Standard ML are good examples of this approach

But what if we are “lazy”?

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

• Example: `res = putStrLn 'k' + putStrLn 'y'`
  – Output depends upon the evaluation order of (+).
• Example: `ls = [putchar 'x', putStrLn 'y']`
  – Output depends on how list is used
  – If only used in `length ls`, nothing will be printed because `length` does not evaluate elements of list
Fundamental question

• Is it possible to regard pure Haskell as the basic programming paradigm, and add imperative features without changing the meaning of pure Haskell expressions?

Tackling the Awkward Squad

• Basic conflict
  – Laziness and side effects are incompatible
  – Historical aside: “Jensen’s device” in Algol 60; see book (p96)
  – Side effects are important!
• History
  – This conflict 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 IO monad tackles the awkward squad
  – I/O, imperative state, exceptions, foreign functions, concurrency
  – Practical application of theoretical insight by E. Moggi

Web Server Example

• The reading uses a web server as an example
• Lots of I/O, need for error recovery, need to call external libraries, need for concurrency

Client 1  Client 2  Client 3  Client 4

Web server  1500 lines of Haskell  700 connections/sec

Writing High-Performance Server Applications in Haskell, by Simon Marlow

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

The term “side effect” itself is misleading

Before Monads

• Streams
  – Program sends stream of requests to OS, receives stream of responses
• Continuations
  – User supplies continuations to I/O routines to specify how to process results (will cover continuations Wed)
• World-Passing
  – The “State of the World” is passed around and updated, like other data structures
  – Not a serious contender because designers didn’t know how to guarantee single-threaded access to the world
• Haskell 1.0 Report adopted Stream model
  – Stream and Continuation models were discovered to be inter-definable
Stream Model: Basic Idea

- Move side effects outside of functional program
- Haskell `main :: String -> String`

<table>
<thead>
<tr>
<th>Haskell main program</th>
<th>Wrapper Program, written in some other language</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard input location (file or stdin)</td>
<td>standard output location (file or stdout)</td>
</tr>
</tbody>
</table>

- Gets more complicated...
  – But what if you need to read more than one file? Or delete files? Or communicate over a socket? ...

Stream Model

- Move side effects outside of functional program
- If Haskell `main :: [Response] -> [Request]`

<table>
<thead>
<tr>
<th>[Response]</th>
<th>[Request]</th>
</tr>
</thead>
</table>

- Laziness allows program to generate requests prior to processing any responses.

Stream Model is Awkward!

- Hard to extend
  – New I/O operations require adding new constructors to Request and Response types, modifying wrapper
- Does not associate Request with Response
  – easy to get “out-of-step,” which can lead to deadlock
- Not composable
  – no easy way to combine two “main” programs
- ... and other problems!!!

Stream Model

- Enrich argument and return type of `main` to include all input and output events.

<table>
<thead>
<tr>
<th><code>main :: [Response] -&gt; [Request]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data request =</code> ReadFile Filename</td>
</tr>
<tr>
<td>WriteFile FileName String</td>
</tr>
<tr>
<td><code>data response =</code> RequestFailed</td>
</tr>
<tr>
<td>ReadOK String</td>
</tr>
<tr>
<td>WriteOk</td>
</tr>
<tr>
<td>Success</td>
</tr>
</tbody>
</table>

- Wrapper program interprets requests and adds responses to input.

Example in Stream Model

- Haskell 1.0 program asks user for filename, echoes name, reads file, and prints to standard out

```
main :: [Response] -> [Request]
main ~(Success : ~(Str userInput) : ~(Success : ~(r4 : _)))) = [ AppendChan stdout "enter filename
", ReadChan stdin, 
  AppendChan stdout name, 
  ReadFile name, 
  AppendChan stdout (case r4 of 
    Str contents -> contents 
    Failure ioerr -> "can’t open file") 
  ] where (name : _) = lines userInput
```

- The `~` denotes a lazy pattern, which is evaluated only when the corresponding identifier is needed.

Monadic I/O: The Key Idea

A value of type (IO t) is an “action”

When performed, an action may do some input/output and deliver a result of type t
Monads

- General concept from category theory
  - Adopted in Haskell for I/O, side effects, ...
- A monad consists of:
  - A type constructor $M$
  - A function bind :: $M a \rightarrow (a \rightarrow M b) \rightarrow M b$
  - A function return :: $a \rightarrow M a$
- Plus:
  - Laws about how these operations interact

Monad Examples

- Error handling $M(A) = A \cup \{\text{error}\}$
  - Add a special "error value" to a type
  - Define sequencing operator ";" to propagate error
- Information-flow tracking $M(A) = A \times \{\text{Labels}\}$
  - Add information flow label to each value
  - Define ";" to check and propagate labels
- Nontermination $M(A) = A \cup \{\infty\}$
  - Result of computation can be value or "run forever"
  - Define ";" to proceed when first computation halts
- State $M(A) = A \times \text{States}$
  - Computation produces value and new state
  - Define ";" to make output state of first to input state of second

Can write code to compute on $A$, but run it using $M(A)$

A Helpful Picture

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

$$\text{type } \text{IO } t = \text{World} \rightarrow (t, \text{World})$$

Simple I/O

- "Actions" are sometimes called "computations"
- An action is a first-class value
- Evaluating an action has no effect; performing the action has the 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 $t$.

$$\text{type } \text{IO } t = \text{World} \rightarrow (t, \text{World})$$

Main program is an action of type $\text{IO } ()$

getChar :: $\text{IO } \text{Char}$
putChar :: $\text{Char} \rightarrow \text{IO } ()$
main :: $\text{IO } ()$
main = putChar 'x'
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 (>>=) Combinator

- Operator is called bind because it binds the result of the left-hand action in the action on the right
- Performing compound action \( a >>= \lambda x \cdot b : \)
  - performs action \( a \), to yield value \( r \)
  - applies function \( \lambda x \cdot b \) to \( r \)
  - performs the resulting action \( b(x \leftarrow r) \)
  - returns the resulting value \( v \)

The (>>=) Combinator

\[
(\gg=) : \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b \\
\text{echoDup} :: \text{IO } () \\
\text{echoDup} = \text{getChar} \gg= (\lambda c \to \\
\text{putChar} c \gg \\
\text{putChar} c) \\
\text{echoTwice} :: \text{IO } () \\
\text{echoTwice} = \text{echo} \gg \text{echo}
\]

The Bind Combinator (>>=)

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

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.

Getting Two Characters

- We want to return \((c_1,c_2)\).
  - But, \((c_1,c_2) :: \text{(Char,Char)}\)
  - We need to return value of type \(\text{IO(Char,Char)}\)
- We need to have some way to convert values of “plain” type into the \(\text{I/O Monad}\).
The return Combinator

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

```
return :: a -> IO a
```

```
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
```

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 semicolons are omitted, then the generators must align. Indentation replaces 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 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 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.

```
1 2
```
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.

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

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

  ```hs
  Main> repeatN 5 (putChar 'h')
  hhhhh
  ```

For Loops

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

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

  ```hs
  Main> for [1..10] (\x -> putStr (show x))
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
  ```

First Class Actions

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

Sequencing

- A list of IO actions.

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

  ```hs
  Main> sequence [getChar, getChar, getChar]
  hhh
  ```

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:

  ```hs
  openFile :: FilePath -> 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
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  { r <- newIORef 0;
    addToN r 1 }
where
  addToN :: IORef Int -> Int -> IO Int
  | i > n = readIORef r
  | otherwise = do
    { v <- readIORef r
    ; writeIORef r (v + i)
    ; addToN r (i+1) }

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

Example Using References

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

But this is terrible! Contrast with: \[ \sum_{i=1}^{n} \] Claims to need side effects, but doesn't really.
```

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!

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
getChar :: IO Char
putChar :: Char -> IO ()
... more operations on characters ...
openFile :: FilePath -> IOMode -> IO Handle
... more operations on files ...
newIORef :: a -> IO (IORef a)
... more operations on references ...
```

A Second Example

• Track the number of chars written to a file.

```
type HandleC = (Handle, IORef Int)

openFileC :: FilePath -> IOMode -> IO HandleC
openFileC file mode = do
  { h <- openFile file 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 }
```

Irksome Restriction?

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

```
configFileContents :: [String]
configFileContents = lines (readFile "config") -- WRONG!
useOptimisation :: Bool
useOptimisation = "optimize" `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!
Type-Unsafe Haskell Programming

- Reading a file is an I/O action, so in general it matters when we read the file.
- But we know the configuration file will not change during the program, so it doesn’t 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
getConfigFileContents :: [String]
getConfigFileContents = lines(unsafePerformIO(readFile "config"))
```

- 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.

unsafePerformIO

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

```haskell
x :: forall a. IORef a  -- This is bad!
r = unsafePerformIO (newIORef (error "urk"))
```

- 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.

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 interact

```haskell
return x >>= f = f x
m >>= return = m
```

Implementation

- GHC uses “world-passing semantics” for the IO monad

```haskell
type IO t = World -> (t, World)
return :: a -> IO a
return a = \
  \w \\rightarrow (a,w)
(\v) = \w -> (a,a) -> IO b
(\a,b) = \w \\rightarrow case \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.”

Monad Laws
Derived Laws for (>>) and done

\((\triangleright\triangleright) : IO \rightarrow IO \rightarrow IO\)

\(\text{done} :: IO ()\)
\(\text{done} = \text{return} ()\)

\(\text{done} \triangleright\triangleright m = m\)
\(m \triangleright\triangleright \text{done} = m\)
\(m1 \triangleright\triangleright (m2 \triangleright\triangleright m3) = (m1 \triangleright\triangleright m2) \triangleright\triangleright m3\)

\(\triangleright\triangleright :: IO a \rightarrow IO b \rightarrow IO b\)

Reasoning

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

\(\text{putStr} :: \text{String} \rightarrow IO ()\)
\(\text{putStr} [] = \text{done}\)
\(\text{putStr} \ (c:cs) = \text{putChar} \ c \triangleright\triangleright \text{putStr} \ cs\)

**Proposition:**
\(\text{putStr} \ r \triangleright\triangleright \text{putStr} \ s = \text{putStr} \ (r \ ++ \ s)\)

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 (\(\triangleright\triangleright\)) 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.

Comparison

- 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.