IO and Monads
Introduction

To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.
However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.
The Problem: Haskell functions are pure mathematical functions

Haskell programs have no side effects.

Referential transparency: called with the same arguments, a function always returns the same value

However, reading from the keyboard and writing to the screen are side effects:

Interactive programs have side effects.
IO is Awkward

Simon Peyton Jones: “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell”

“The call-by-value (or strict) family of functional languages have generally taken a pragmatic approach to these questions, mostly by adopting a similar approach to that taken by imperative languages. You want to print something? No problem; we’ll just have a function printChar that has the side effect of printing a character...”
IO is Awkward

Simon Peyton Jones: “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell”

“Of course, printChar isn’t really a function any more (because it has a side effect), but in practice this approach works just fine, provided you are prepared to specify order of evaluation as part of the language design — and that is just what almost all other programming languages do, from FORTRAN and Java to mostly-functional ones like Lisp, and Standard ML.”
IO is Awkward

“Call-by-need (or lazy) languages, such as Haskell, wear a hair shirt because their evaluation order is deliberately unspecified. Suppose that we were to extend Haskell by adding side-effecting “functions” such as `printChar`. Now consider this list

```haskell
xs = [printChar 'a', printChar 'b']
```

....What on earth might this mean? ... in Haskell, the calls to `printChar` will only be executed if the elements of the list are evaluated. For example, if the only use of `xs` is in the call (length `xs`), then nothing at all will be printed, because length does not touch the elements of the list.”
IO is Awkward because of laziness

“The bottom line is that laziness and side effects are, from a practical point of view, incompatible. If you want to use a lazy language, it pretty much has to be a purely functional language; if you want to use side effects, you had better use a strict language.”
IO is Awkward because of laziness

“[I]t's a bit cheeky to call input/output "awkward" at all. I/O is the raison d'être of every program—a program that had no observable effect whatsoever (no input, no output) would not be very useful.”
The Solution - The IO Type

Interactive programs can be viewed as a pure function whose domain and codomain are the current *state of the world*:

\[
\text{type IO} = \text{World} \to \text{World}
\]

However, an interactive program may return a result value in addition to performing side effects:

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

What if we need an interactive program that takes an argument of type \(b\)?

Use currying:

\[
b \to \text{World} \to (a, \text{World})
\]
The Solution (Cont.)

Now, interactive programs (impure actions) can be defined using the IO type:

IO a  
The type of actions that return a value of type a

For example:

IO Char  
The type of actions that return a character

IO ()  
The type of actions that return the empty tuple (a dummy value); purely side-effecting actions
Basic Actions (defined in the standard library)

1. The action `getChar` reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

   ```haskell
   getChar :: IO Char
   ```

2. The action `putChar c` writes the character `c` to the screen, and returns no result value:

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

3. The action `return v` simply returns the value `v`, without performing any interaction:

   ```haskell
   return :: a -> IO a
   ```
Sequencing
A sequence of actions can be combined as a single composite action using the >>= (then) or >> (binding) operators.

\[ (\gg\gg) :: IO \, a \rightarrow (a \rightarrow IO \, b) \rightarrow IO \, b \]
\[ (act1 \gg\gg act2) = \\lambda \text{world} . \text{case act1 world of} \]
\[ (v, \text{world'}) \rightarrow act2 \, v \, \text{world'} \]

Compare it with:

\[ (\gg) :: IO \, a \rightarrow IO \, b \rightarrow IO \, b \]
\[ (act1 \gg act2) = \\lambda \text{world} . \text{case act1 world of} \]
\[ (v, \text{world'}) \rightarrow act2 \, \text{world'} \]

Apply action1 to world, get a new action (action2 v), and apply that to the modified world
Derived Primitives

• Reading a string from the keyboard:

```haskell
getLine :: IO String
getLine = getChar >>= \x ->
    if x == '\n' then return []
    else (getLine >>= \xs -> return (x:xs))
```

• Writing a string to the screen:

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = putChar x >> putStr xs
```

• Writing a string and moving to a new line:

```haskell
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```
Derived Primitives (do Notation)

• Reading a string from the keyboard:

    getLine :: IO String
    getLine  = do x <- getChar
                  if x == ‘\n’ then return []
                  else do xs <- getLine
                          return (x:xs)

• Writing a string to the screen:

    putStr :: String → IO ()
    putStr []     = return ()
    putStr (x:xs) = do putChar x
                        putStr xs

• Writing a string and moving to a new line:

    putStrLn :: String → IO ()
    putStrLn xs = do putStr xs
                     putChar ‘\n’
Building More Complex IO Actions

We can now define an action that prompts for a string to be entered and displays its length:

```haskell
strlen :: IO ()
strlen = putStrLn "Enter a string: " >>
  getLine >>= \xs ->
  putStrLn "The string has " >>
  putStrLn (show (length xs)) >>
  putStrLn " characters."
```
Building More Complex IO Actions (do)

Using the do notation:

```haskell
strlen :: IO ()
[strlen  = do  putStrLn "Enter a string: 
    xs <- getLine
    putStrLn "The string has 
    putStrLn (show (length xs))
    putStrLn " characters."
```
IO Monad As An Abstract Data Type

Consider:

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
getChar :: IO Char
putChar :: Char -> IO ()
openFile :: [Char] -> IOMode -> IO Handle
```

• All primitive IO operations return an IO action
• IO monad is sticky: all functions that take an IO argument, return an IO action
• `return` offers a way *in* to an IO action, but no function offers a way *out* (you can bind a variable to the IO result by use of “<-”)
The Type of main

A complete Haskell program is a single IO action. For example:

```
main :: IO ()
main = getLine >>= \cs ->
    putLine (reverse cs)
```

Typically, IO “contaminates” a small part of the program (outermost part), and a larger portion of a Haskell program does not perform any IO. For example, in the above definition of main, reverse is a non-IO function.
IO is Awkward (cont/)

“For a long time this situation was rather embarrassing for the lazy community: even the input/output story for purely-functional languages was weak and unconvincing, let alone error recovery, concurrency, etc. Over the last few years, a surprising solution has emerged: the monad. I say "surprising" because anything with as exotic a name as "monad"—derived from category theory, one of the most abstract branches of mathematics—is unlikely to be very useful to red-blooded programmers.”
IO is Awkward (cont/)

“But one of the joys of functional programming is the way in which apparently-exotic theory can have a direct and practical application, and the monadic story is a good example. Using monads we have found how to structure programs that perform input/output so that we can, in effect, do imperative programming where that is what we want, and only where we want. Indeed, the IO monad is the unifying theme...”
Monad (Roughly)

- Monad is a strategy for combining computations into more complex computations
- No language support, besides higher-order functions, is necessary
  - But Haskell provides the do notation
- Monads play a central role in the I/O system
  - Understanding the I/O monad will improve your code and extend your capabilities
Monad Example: Maybe

```
data Maybe a = Nothing | Just a
```

Reminder:
• Maybe is a type constructor and Nothing and Just are data constructors
• The polymorphic type Maybe a is the type of all computations that may return a value or Nothing – properties of the Maybe container
• For example, let f be a partial function of type a -> b, then we can define f with type:
  \[ f :: a \rightarrow \text{Maybe} \ b \text{ -- returns Just } \ b \text{ or Nothing} \]
Example Using Maybe

Consider the following function querying a database, signaling failure with Nothing

```haskell
doQuery :: Query -> DB -> Maybe Record
```

Now, consider the task of performing a sequence of queries:

```haskell
r :: Maybe Record
r = case doQuery q1 db of
    Nothing -> Nothing
    Just r1 -> case doQuery (q2 r1) db of
        Nothing -> Nothing
        Just r2 -> case doQuery (q3 r2) db of
            Nothing -> Nothing
            Just r3 -> . . .
```
Capture the pattern into a combinator

\[
\text{thenMB} :: \text{Maybe } a \to (a \to \text{Maybe } b) \to \text{Maybe } b
\]

\[
mB \ `\text{thenMB}` \ f = \text{case } mB \text{ of}
\]

\[
\text{Nothing } \to \text{Nothing}
\]

\[
\text{Just } a \to f \ a
\]

This allows the following rewrite to doQuery

\[
r :: \text{Maybe Record}
\]

\[
r = \text{doQuery } q1 \ \text{db} \ `\text{thenMB}` \ r1 \to
\]

\[
\text{doQuery } (q2 \ r1) \ \text{db} \ `\text{thenMB}` \ r2 \to
\]

\[
\text{doQuery } (q3 \ r2) \ \text{db} \ `\text{thenMB}` \ . . .
\]
Another Example: The List Monad

The common Haskell type constructor, [], (for building lists), is also a monad that encapsulates a strategy for combining computations that can return 0, 1, or multiple values:

```
instance Monad [] where
    m >>= f = concatMap f m
    return x = [x]
```

The type of (>>=):

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

The binding operation creates a new list containing the results of applying the function to all of the values in the original list.

```
concatMap :: (a -> [b]) -> [a] -> [b]
```
Combinators controlling parameter passing and computational flow

Many uses for the kind of programming we just saw

• Data Structures: lists, trees, sets
• Computational Flow: Maybe, Error Reporting, non-determinism
• Value Passing: state transformer, environment variables, output generation
• Interaction with external state: IO, GUI programming
• Other: parsing combinators, concurrency, mutable data structures

There are instances of Monad for all of the above situations
Monad Definition

Monad is a triple \((M, \text{return}, \triangleright=)\) consisting of a type constructor \(M\) and two polymorphic functions:

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

\[
(\triangleright=) :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b
\]

which satisfy the monad laws (note, checking these is up to the programmer):

\[
\text{return}\ x \triangleright= f \quad == \quad f\ x \quad -- \text{left identity}
\]

\[
m \triangleright= \text{return} \quad == \quad m \quad -- \text{right identity}
\]

\[
(m \triangleright= f) \triangleright= g \quad == \quad m \triangleright= (\lambda x \rightarrow f\ x \triangleright= g) \quad -- \text{associativity}
\]
What is the practical meaning of the monad laws?

Let us rewrite the laws in do-notation:

**Left identity:**

```haskell
do { x' <- return x;
    f x'                    ==  do { f x }
}
```
What is the practical meaning of the monad laws?

Let us rewrite the laws in do-notation:

**Right identity:**

```haskell
do { x <- m;
    return x == do { m }
}
```
What is the practical meaning of the monad laws?

Let us rewrite the laws in do-notation:

**Associativity:**

```haskell
do { y <- do { x <- m; f x; } } == do { x <- m; y <- f x; g y; } g y
```
The Monad Type Class

class Monad m where
  >>= :: m a -> (a -> m b) -> m b
  >> :: m a -> m b -> m b
  return :: a -> m a
  m >> k = m >>= \_ -> k

- >> is a shorthand for >>= ignoring the result of first action
- Any type with compatible combinators can be made to be an instance of this class. For example:

data Maybe a = Just a | Nothing
thenMB :: Maybe a -> (a -> Maybe b) -> Maybe b
instance Monad Maybe where
  (>>>=) = thenMB
  return a = Just a
Utilizing the Monad Type Class

```haskell
class Monad m where
    >>= :: m a -> (a -> m b) -> m b
    >> :: m a -> m b -> m b
    return :: a -> m a

    m >> k = m >>= \_ -> k

sequence       :: Monad m => [m a] -> m [a]
sequence []     = return []
sequence (c:cs) = c >>= \x ->
                      sequence cs >>= \xs ->
                      return (x:xs)
```

- The type class gives a common interface for all monads.
- Thus, we can define functions operating on all monads.
- For example, execute each monadic computation in a list:
Running a Monad

- Most monadic computations (such as IO actions) are functions of some sorts

- Combining computations with bind creates ever more complex computations, where some state/world/... is threaded from one computation to another, but essentially a complex computation is still a function of some sorts

- A monadic computation is “performed” by applying this function