...and the Beast

- But to be useful as well as beautiful, a language must manage the “Awkward Squad”:
  - Input/Output
  - Imperative update
  - Error recovery (e.g., 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 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 = putStrLn 'x' + putStrLn 'y'`
  - Output depends upon the evaluation order of (+).
- Consider: `ls = ['x' | putStrLn 'x', putStrLn '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 90s, 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 Direct Approach

- Do everything the “usual way”:
  - I/O via “functions” with side effects:
    ```haskell```
    ```
    putStrLn 'x' + putStrLn 'y'
    ```
  - Imperative operations via assignable reference cells:
    ```haskell```
    ```
    let z = ref 0, x = !z + 1, f(x) = !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 reading uses a web server as an example.

Lots of I/O, need for error recovery, need to call external libraries, need for concurrency.

Web server

Client 1
Client 2
Client 3
Client 4

1500 lines of Haskell
700 connections/sec

Writing High-Performance Server Applications in Haskell
by Simon Marlow

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 Web Server

Monadic Input and Output

The Problem

Before Monads

- Streams
  - Program issues a stream of requests to OS, which responds with a stream of inputs.
- Continuations
  - User supplies continuations to I/O routines to specify how to process results.
- World-Passing
  - The "World" is passed around and updated, like a normal data structure.
  - Not a serious contender because designers didn’t know how to guarantee single-threaded access to the world.
- Stream and Continuation models were discovered to be inter-definable.
- Haskell 1.0 Report adopted Stream model.

Stream Model: Basic Idea

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

Wrapper Program, written in some other language

Wrapper Program

- Standard input location (file or stdin)
- Haskell main program
- Standard output location (file or stdout)

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

Stream Model

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

main :: [Response] -> [Request]

data Request = ReadFile Filename
| WriteFile FileName String
| ...

data Response = RequestFailed
| ReadOK String
| WriteOk
| Success ...

- Wrapper program interprets requests and adds responses to input.

- Streams
  - Program issues a stream of requests to OS, which responds with a stream of inputs.

- Continuations
  - User supplies continuations to I/O routines to specify how to process results.

- World-Passing
  - The "World" is passed around and updated, like a normal data structure.
  - Not a serious contender because designers didn’t know how to guarantee single-threaded access to the world.
  - Stream and Continuation models were discovered to be inter-definable.

- Haskell 1.0 Report adopted Stream model.
Move side effects outside of functional program  

If Haskell main :: [Response] -> [Request]

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

Stream Model

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

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

Example in Stream Model

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

```
main :: [Response] -> [Request]
main ~(Success : ~(((Str userInput) : ~(Success : ~(r4 : _)))))
  = [ AppendChan stdout "enter filename\n", 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.

Stream Model is Awkward!

- Hard to extend: new I/O operations require adding new constructors to Request and Response types and modifying the wrapper.
- No close connection between a Request and corresponding Response, so easy to get "out-of-step," which can lead to deadlock.
- The style is not composable: no easy way to combine two "main" programs.
- ... and other problems!!!

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.

```
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.

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)
```

```
result :: t
World in = IO t
World out
```

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 'a'`

Connection Actions

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

```
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`

```
a    r    b
```

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.

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

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

echoTwice :: IO ()
echoTwice = echo >> echo
```
We want to return \((c_1, c_2)\).

But, \((c_1, c_2) :: (\text{Char}, \text{Char})\)

And we need to return something of type \(\text{IO}(\text{Char}, \text{Char})\).

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

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

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

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

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

The "do" notation adds syntactic sugar to make monadic code easier to read.

\[
\begin{align*}
\text{Plain Syntax} & \\
\text{getTwoChars} :: \text{IO}(\text{Char}, \text{Char}) \\
\text{getTwoChars} & = \text{getChar} \gg \lambda c_1 \rightarrow \\
& \text{getChar} \gg \lambda c_2 \rightarrow \\
& \text{return} (c_1, c_2)
\end{align*}
\]

\[
\begin{align*}
\text{Do Notation} & \\
\text{getTwoCharsDo} :: \text{IO}(\text{Char}, \text{Char}) \\
\text{getTwoCharsDo} & = \text{do} \{ \\
& c_1 \leftarrow \text{getChar}; \\
& c_2 \leftarrow \text{getChar}; \\
& \text{return} (c_1, c_2) \}
\end{align*}
\]

Do syntax designed to look imperative.

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.

The following are equivalent:

\[
\begin{align*}
\text{do} & \{ x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \} \\
\text{do} & \quad x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \\
\text{do} & \quad x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q
\end{align*}
\]

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

The "do" notation only adds syntactic sugar:

\[
\begin{align*}
\text{do} \{ x \leftarrow e; es \} & = e \gg \lambda x \rightarrow \text{do} \{ es \} \\
\text{do} \{ e; es \} & = e \gg \text{do} \{ es \} \\
\text{do} \{ e \} & = e \\
\text{do} \{ \text{let } ds; es \} & = \text{let } ds \text{ in } \text{do} \{ es \}
\end{align*}
\]

The "do" notation

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.

The following are equivalent:

\[
\begin{align*}
\text{do} & \{ x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \} \\
\text{do} & \quad x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \\
\text{do} & \quad x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q
\end{align*}
\]

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

The "do" notation only adds syntactic sugar:

\[
\begin{align*}
\text{do} \{ x \leftarrow e; es \} & = e \gg \lambda x \rightarrow \text{do} \{ es \} \\
\text{do} \{ e; es \} & = e \gg \text{do} \{ es \} \\
\text{do} \{ e \} & = e \\
\text{do} \{ \text{let } ds; es \} & = \text{let } ds \text{ in } \text{do} \{ es \}
\end{align*}
\]

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.

The following are equivalent:

\[
\begin{align*}
\text{do} & \{ x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \} \\
\text{do} & \quad x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q \\
\text{do} & \quad x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n; q
\end{align*}
\]

If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.
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 \(\text{IO} \, a\), the type:
  - tags the action as suitable for the IO assembly line via the \(\text{IO}\) type constructor.
  - indicates that the kind of thing being passed to the next stage in the assembly line has type \(a\).
- The \texttt{bind} operator "snaps" two stages \(s_1\) and \(s_2\) together to build a compound stage.
- The \texttt{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 \texttt{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.

```
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 \((\text{IO} \, t)\) 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:

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

- 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!

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

Example Using References

```haskell
import Data.IORef

-- 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.

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);
      putStr h cs
```

- Here it makes sense to use a reference.

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!

- A value of type `IORef a` is a reference to a mutable cell holding a value of type `a`.
Suppose you wanted to read a configuration file at the beginning of your program:

```
configFileContents :: [String]
configFileContents = lines (readFile "config") -- WRONG!
useOptimization = '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!

```
configFileContents :: [String]
configFileContents = lines(unsafePerformIO(readFile"config"))
```

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 `unsafePerformIO` primitive:

```
unsafePerformIO :: IO a -> a
configFileContents = lines(unsafePerformIO(readFile"config"))
```

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.

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

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

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.

GHC uses world-passing semantics for the IO monad:

```
type IO b = World -> (b, 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
k >>= k' = \w -> case m x of (x, w') -> k x x'
```

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.”

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

\[
\text{return } x \triangleright\triangleright= f \quad \text{=} \quad f \cdot x \\
\text{m } \triangleright\triangleright= \text{return } \quad \text{=} \quad m \\
\text{m1 } \triangleright\triangleright= (\lambda x.\text{m2 } \triangleright\triangleright= (\lambda y.\text{m3})) \quad \text{=} \\
\quad (\text{m1 } \triangleright\triangleright= (\lambda x.\text{m2})) \triangleright\triangleright= (\lambda y.\text{m3}) \\
x \text{ not in free vars of m3}
\]

Derived Laws for (\triangleright\triangleright) and done

\[
(\triangleright\triangleright) : \text{IO } a \to \text{IO } b \to \text{IO } b \\
m \triangleright\triangleright a = a \triangleright\triangleright (\lambda x \to a) \\
\text{done } : \text{IO } () \\
\text{done } = \text{return } () \\
\text{done } \triangleright\triangleright m = m \\
m \triangleright\triangleright \text{done } = m \\
\text{m1 } \triangleright\triangleright (\text{m2 } \triangleright\triangleright \text{m3}) = (\text{m1 } \triangleright\triangleright \text{m2}) \triangleright\triangleright \text{m3}
\]

Reasoning

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

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

Proposition:

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

Proof: By induction on \(r\).

Base case: \(r []\)

\[
\text{putStr [] } \triangleright\triangleright \text{putStr } s \\
= (\text{definition of putStr}) \\
\quad \text{done } \triangleright\triangleright \text{putStr } s \\
= (\text{first monad law for } \triangleright\triangleright) \\
\quad \text{putStr } s \\
= (\text{definition of ++}) \\
\quad \text{putStr } ([] ++ s) \\
\text{Induction case: } r \text{ is } (c:s) ...
\]

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 (\(\triangleright\triangleright\triangleright\)) 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.

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.