

Monads

Edward Z. Yang

(w/ help from Simon Peyton Jones,
Kathleen Fisher and John Mitchell)

Beauty

FP is mathematics; it
is beautiful!


Elegant & powerful abstractions

the "Awkward Squad":
input/output, error recovery,
concurrency, FFI

and the *Beast*

Direct approach

putchar 'x' + putchar 'y'



I/O via "functions"
with side effects

~~Direct approach~~

Laziness!

xy?

putchar 'x' + putchar 'y'

yx?

depends on evaluation order

~~Direct approach~~ Laziness!

ls = [putchar 'x', putchar 'y']

> length ls \rightsquigarrow nothing

> head ls \rightsquigarrow x

()

the "hair shirt"



Laziness forces us to take
a different, more principled
approach

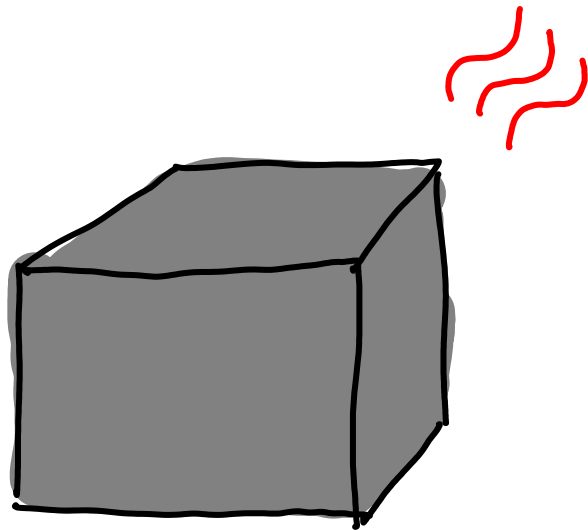
(lessons applicable to strict languages too)

The tension

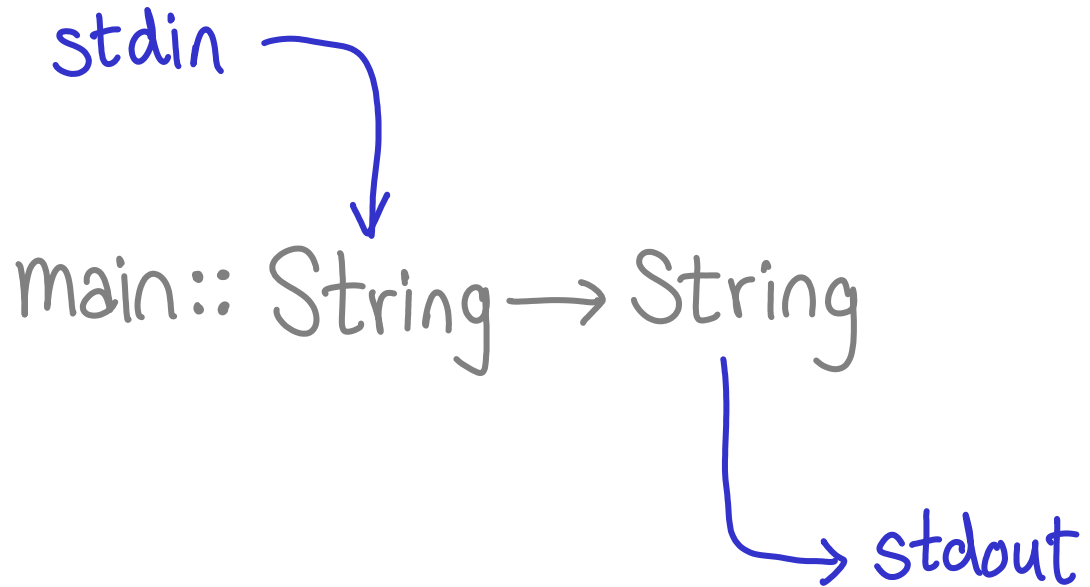
Functional programs
define pure functions
with no side effects



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



Functional I/O



Functional I/O

lazy streams

main :: [Response] → [Request]



```
graph TD; A[lazy streams] --> B[Response]; A --> C[Request];
```

Functional I/O

lazy streams

main :: [Response] → [Request]



```
graph TD; A[lazy streams] --> B[Response]; A --> C[Request];
```

Functional I/O

$\text{main} :: [\text{Response}] \longrightarrow [\text{Request}]$

↳ ReadFile :

...

Functional I/O

ReadOK str:

...



main :: [Response] → [Request]



ReadFile:

...

Functional I/O

ReadOK str :

...



main :: [Response] → [Request]



ReadFile :
WriteFile :
...

Functional I/O

ReadOK str:

WriteOK:

...



main :: [Response] → [Request]

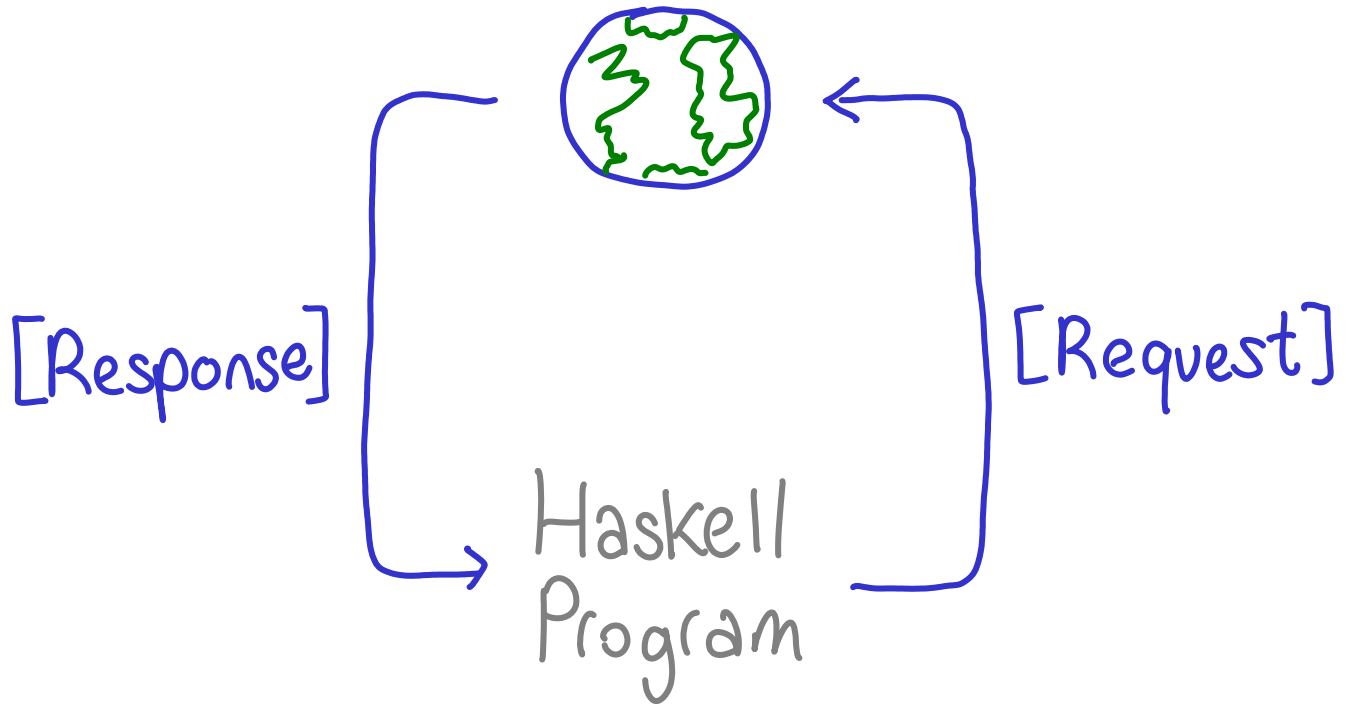


ReadFile:

WriteFile:

...

Functional I/O



Functional I/O: Awkward!

- ▶ Hard to extend

- ▶ Hard to use
Deadlock!

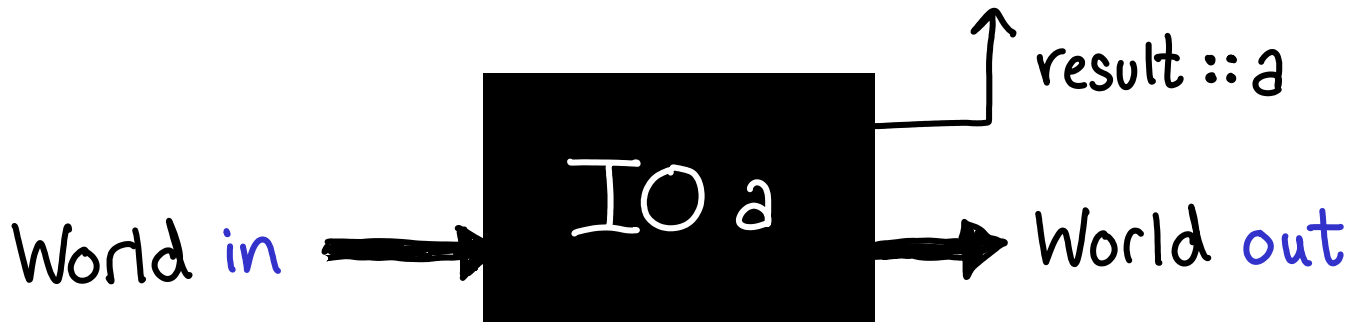
Monadic I/O: the key idea

A value of type $(\text{IO } a)$ is an "action" that, when performed, may do some I/O before delivering a result of type a .

Monadic I/O: the key idea

A value of type (**IO a**) is an "**action**" that, when performed, may do some I/O before delivering a result of type a .

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



Actions are first class

A value of type (**IO a**) is an "**action**" that, when performed, may do some I/O before delivering a result of type **a**.

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

Evaluating an action has no effect;
performing the action has an effect

Evaluating

vs.

Performing

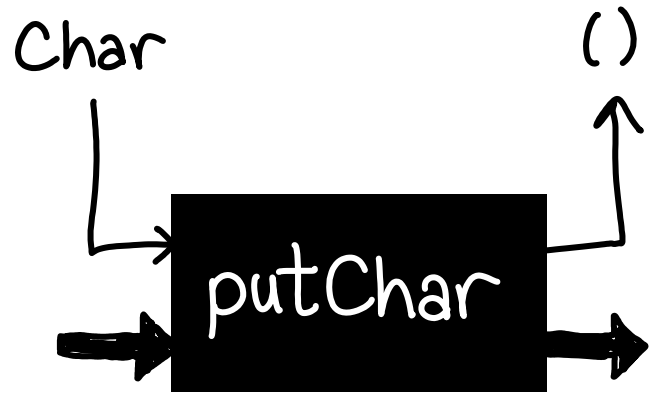
Recipe for
Swedish Meatballs

1. ~~~~~
2. ~~~~~
3. ~~~~~
~~~~~  
~~~~~



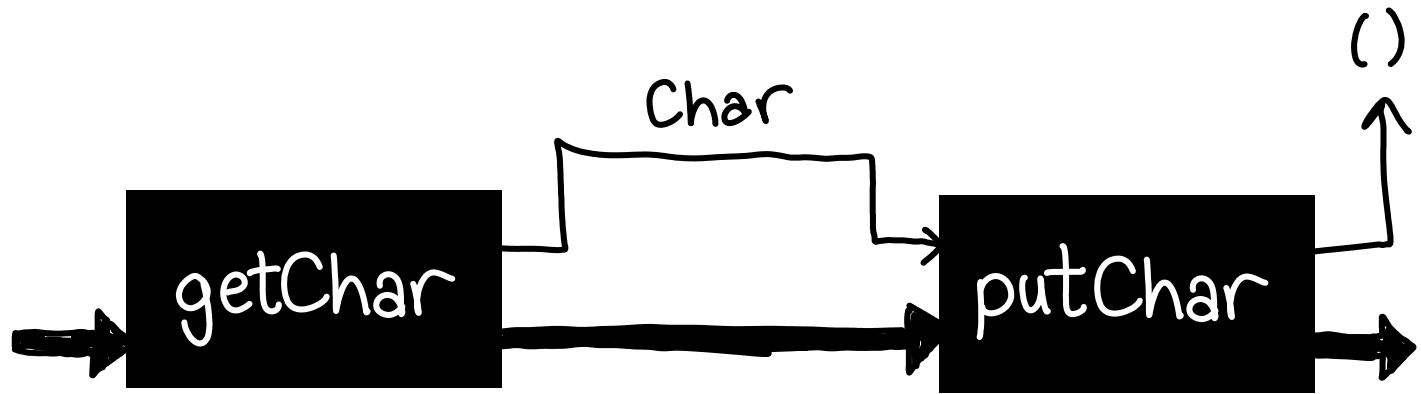


`getChar :: IO Char`



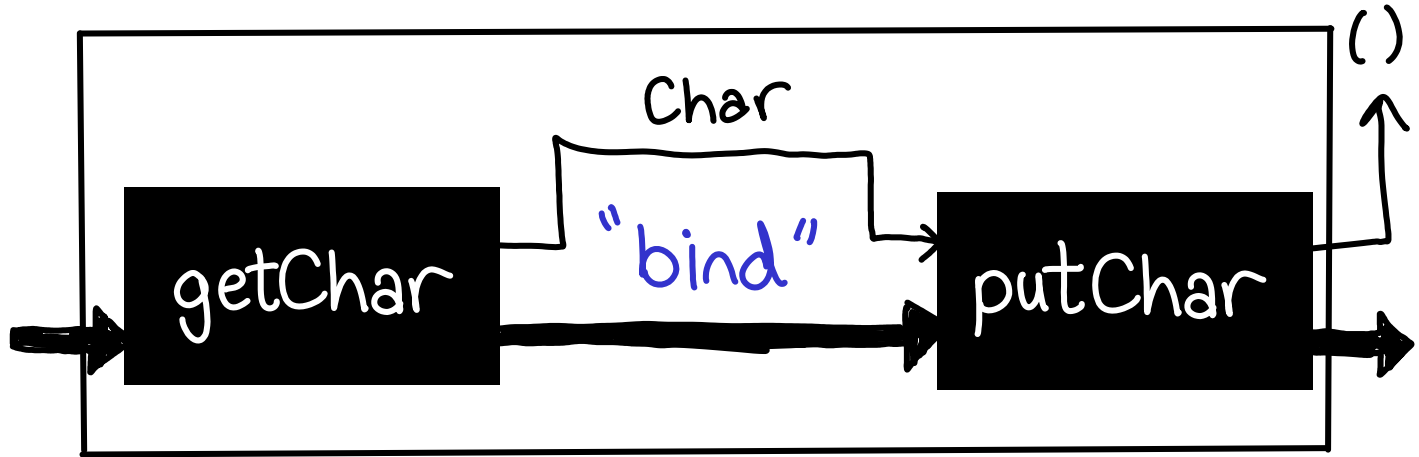
`putChar :: Char → IO()`

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



Read a character and write it back out

$(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$



a new, bigger
action!

`echo :: IO ()`

`echo = getChar >>= putChar`

Print a character twice

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

↑ nothing interesting

(parentheses optional)

The (\gg) combinator

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

$(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$

$m \gg n = m \gg= _ \rightarrow n$

The **return** combinator

↖ not short circuiting

getTwoChars :: IO (Char, Char)

getTwoChars = getChar >>= \c1 →

getChar >>= \c2 →

???

The **return** combinator

↖ not short circuiting

getTwoChars :: IO (Char, Char)

getTwoChars = getChar >>= \c1 →
getChar >>= \c2 →

X (c1, c2)

Got (Char, Char), expecting IO (Char, Char)

The **return** combinator

↖ not short circuiting

getTwoChars :: IO (Char, Char)

getTwoChars = getChar >>= \c1 →
getChar >>= \c2 →
return (c1, c2)



return :: a → IO a

Do notation

$\left\{ \begin{array}{l} \text{getTwoChars} :: \text{IO (Char, Char)} \\ \text{getTwoChars} = \text{getChar} \gg= \backslash c1 \rightarrow \\ \quad \text{getChar} \gg= \backslash c2 \rightarrow \\ \quad \text{return (c1, c2)} \end{array} \right\}$

Versus

$\left\{ \begin{array}{l} c1 = \text{getchar}(); \\ c2 = \text{getchar}(); \\ \text{return (c1, c2)} \end{array} \right\}$

imperative 

Do notation

getTwoChars :: IO (Char, Char)

getTwoChars = do { c1 ← getChar;
 c2 ← getChar;
 return (c1, c2) }

syntax sugar

Do notation

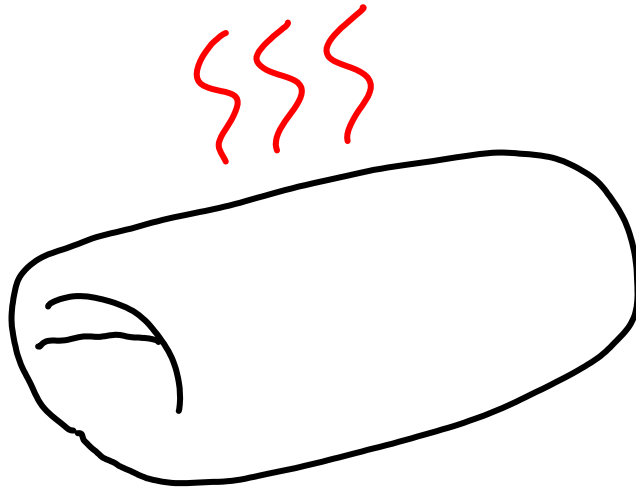
$$\text{do } \{ x \leftarrow e; s \} \equiv e \gg \lambda x \rightarrow \text{do } \{ s \}$$

$$\text{do } \{ e; s \} \equiv e \gg \text{do } \{ s \}$$

$$\text{do } \{ e \} \equiv e$$

GHCi examples

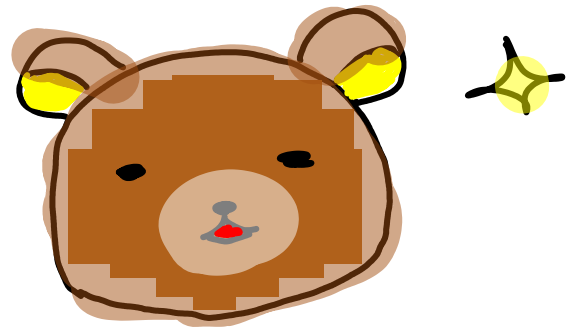
A MONAD IS A BURRITO



- Global state `configFile :: [String]`
- Configuration variables

Trouble?

unsafePerformIO :: IO a \rightarrow a



$r :: \text{IORef } a$ ← forall!

$r = \text{unsafePerformIO } (\text{newIORef undefined})$

$\text{cast} :: b \rightarrow \text{IO } c$

$\text{cast } x = \text{do}$
 $\text{writeIORef } r \ x$
 $\text{readIORef } r$



Implementing the IO monad

“`type IO a = World → (a, World)`”
... literally!
↑
unforgeable
token

Standard FP optimizations apply
(threading ensures linearity)

Comparison: Traditional Languages

All programs

Computation

I/O

Comparison: Haskell

Pure code

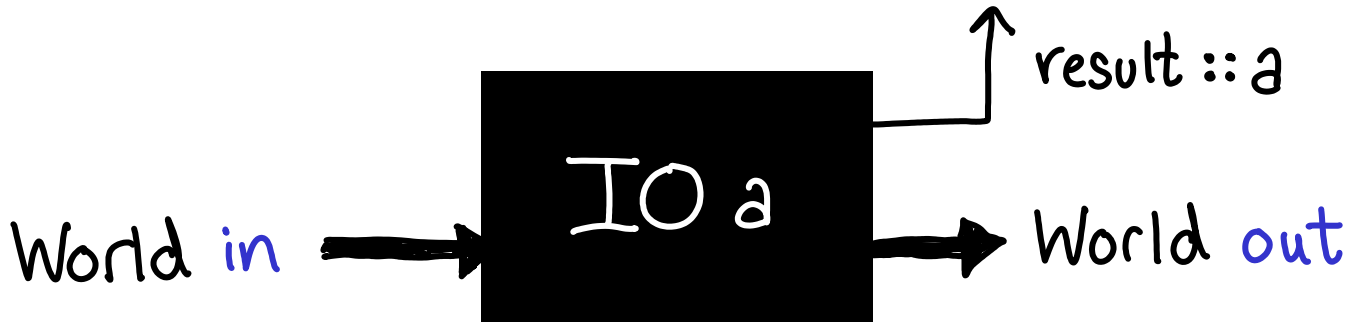
$:: a$

Effectful I/O code

$:: IO\ a$

Monadic I/O: the key idea

A value of type (**IO a**) is an "**action**" that, when performed, may do some I/O before delivering a result of type a .



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





