IT5100A

Industry Readiness:
Typed Functional Programming

Monads

Foo Yong Qi
yongqi@nus.edu.sg

```haskell
f :: [Int] -> [Int]
f [] = []
f (x : xs) =
    let r = f xs
    in r ++ [x]

main :: IO ()
main = do
    let x = [1..10]
    print $ f x
```

Monads are **everywhere**!

Today we will learn more about monads and some typical patterns of monads, and how we can compose monads themselves!

More on Monads

Monad Operations

`Control.Monad` module defines more operations on monads; how do we use them?

Basic Monad functions

```
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
```
Source

Map each element of a structure to a monadic action, evaluate these actions from left to right, and collect the results. For a version that ignores the results see `mapM_`.

▸ Examples

```
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
```
Source

Map each element of a structure to a monadic action, evaluate these actions from left to right, and ignore the results. For a version that doesn't ignore the results see `mapM`.

`mapM_` is just like `traverse_`, but specialised to monadic actions.

```
forM :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
```
Source

`forM` is `mapM` with its arguments flipped. For a version that ignores the results see `forM_`.

```
forM_ :: (Foldable t, Monad m) => t a -> (a -> m b) -> m ()
```
Source

`forM_` is `mapM_` with its arguments flipped. For a version that doesn't ignore the results see `forM`.

`forM_` is just like `for_`, but specialised to monadic actions.

```
sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
```
Source

Ignoring Values

```python
def my_function(x):
    print(x) # standalone statement
    return x
```

Often we want to write standalone statements to perform some monadic action; makes no sense to bind result to a variable

```python
def my_function(x):
    z = print(x) # why?
    return x
```

Ignoring Values

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

`(>>)` method is similar to `>>=` except we ignore the previous monad value

Ignoring Values

```
ghci> [1, 2] >>= (\x -> [(x, 3)])
[(1, 3), (2, 3)]
ghci> [1, 2] >>= (\_ -> [3])
[3, 3]
ghci> [1, 2] >> [3]
[3, 3]
```

`(>>)` on lists does not seem so useful at the moment; we will see more on it shortly

do Notation

do notation	Translation	Description
`do e`	`e`	Expression
`do v <- e` `   s`	`e >>= (\v -> do s)`	Monadic bind
`do e` `   s`	`e >> (do s)`	Monad composition without bind
`do let v = e` `   s`	`let x = e in do s`	Pure bind

Monadic Functions

Standard functions have monadic equivalents with suffix `M`, function with ignored values have suffix `_`

```haskell
map       ::                 (a -> b)    -> [a] -> [b]
mapM @[]  :: Monad m => (a -> m b) -> [a] -> m [b]
filter  ::                 (a -> Bool)    -> [a] -> [a]
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

```haskell
ghci> map (+2) [1, 2, 3]
[3, 4, 5]
ghci> map (Just . (+2)) [1, 2, 3]
[Just 3, Just 4, Just 5]
ghci> mapM (Just . (+2)) [1, 2, 3]
Just [3, 4, 5]
```

Monadic Functions

`let x' = map f x` becomes `do x' <- mapM f x` if `f` is now in context!

```
ghci> import Text.Read
ghci> :{
ghci| toInt :: String -> Int
ghci| toInt = read
ghci| :}
ghci> toInt "123"
123
```

```
ghci> :{
ghci| toIntM :: String -> Maybe Int
ghci| toIntM = readMaybe
ghci| :}
ghci> toIntM "123"
Just 123
ghci> toIntM "hello"
Nothing
```

Monadic Functions

`let x' = map f x` becomes `do x' <- mapM f x` if `f` is now in context!

```
ghci> let x = map toInt ["1","2","3"]
      in  x
[1,2,3]
ghci> do x <- mapM toIntM ["1","2","3"]
         return x
Just [1,2,3]
```

Monadic Controls

```python
def f(x):
    if x > 10:
        print(x)
    return x
```

```haskell
f x = do
    if x > 10
    then someAction x
    else return () -- basically does nothing
    return x
```

Sometimes we want to express monadic controls more elegantly, e.g. only run monadic action when condition is met

Use functions in `Control.Monad` module!

Monadic Controls

```haskell
when :: Applicative f => Bool -> f () -> f ()
```

```haskell
import Control.Monad
f x = do
    when (x > 10) (someAction x)
    return x
```

Monadic Controls

```haskell
guard :: Alternative f :: Bool -> f ()
```

```haskell
safeDiv1 :: Int -> Int -> Maybe Int
safeDiv1 x y = if y == 0
               then Nothing
               else Just (x `div` y)

safeDiv2 :: Int -> Int -> Maybe Int
safeDiv2 x y
    = do guard (y /= 0)
         return $ x `div` y
```

Monadic Controls

```haskell
guard :: Alternative f => Bool -> f ()
guard True = pure ()
guard False = empty
```

An `Alternative` is something that could be empty

```
ghci> Just () >> (return 1)
Just 1
ghci> Nothing >> (return 1)
Nothing
```

`empty` bound with another monad should also give `empty`, `return () >> m` gives `m`

Monadic Controls

```
ghci> import Control.Monad
ghci> ls = [-2, -1, 0, 1, 2]
ghci> :{
ghci> ls2 = do x <- ls
ghci|          guard (x > 0)
ghci|          return x
ghci| :}
ghci> ls2
[1, 2]
```

Lists are also `Alternative`s; `guard` places a guard on each element in the list, basically a filter!

Monadic Controls

```
ghci> import Control.Monad
ghci> ls = [-2, -1, 0, 1, 2]
ghci> :{
ghci> ls2 = do x <- ls
ghci|          guard (x > 0)
ghci|          return $ x * 2
ghci| :}
ghci> ls2
[2, 4]
```

Familiar?

Monadic Controls

```
ghci> import Control.Monad
ghci> ls = [-2, -1, 0, 1, 2]
ghci> :{
ghci> ls2 = do x <- ls
ghci|           guard (x > 0)
ghci|           return $ x * 2
ghci| :}
ghci> ls2
[2, 4]
```

```
ghci> ls = [-2, -1, 0, 1, 2]
ghci> ls2 = [x * 2 | x <- ls, x > 0]
ghci> ls2
[2, 4]
```

List comprehension is just a specialized form of monadic binds and guards! Do notation allows us to express monadic operations in any order, giving you maximum control!

Commonly Used Monads

Commonly Used Monads

These types act as **contexts** or **notions of computation**:

`Maybe a` — an **a** or nothing

`Either a b` — either **a** or **b**

`[a]` — a list of possible **a**s (nondeterminism)

`Maybe` `Int`

`Either Int` `String`

`[]` `Char`

Other common **notions of computation**:

- Reading from state
- Writing to, or editing state

Reader

`env -> a`

An **a** that depends on an environment **env**

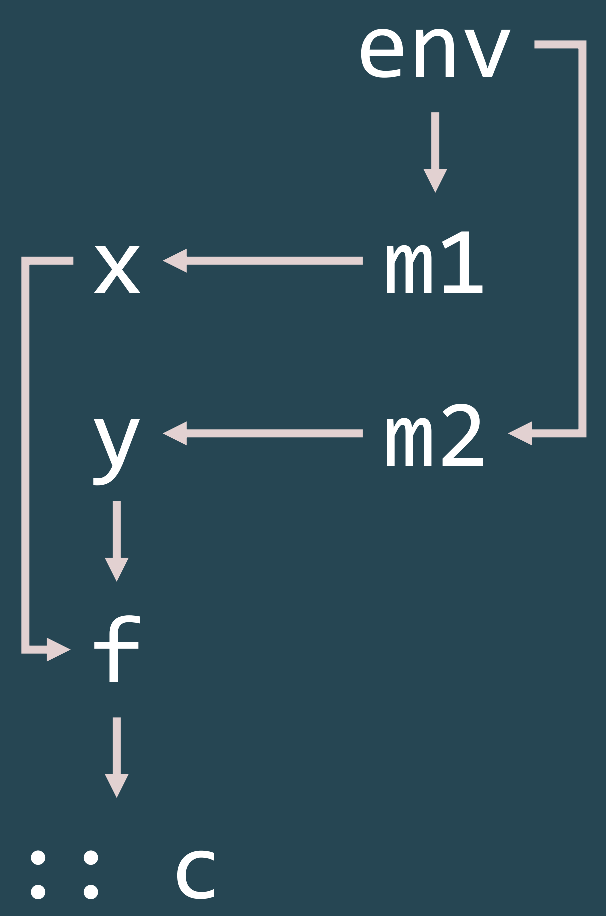
`env ->` is a monad!

```haskell
instance Functor ((->) env) where
    fmap :: (a -> b) -> (env -> a) -> (env -> b)
    fmap f x = f . x

instance Applicative ((->) env) where
    pure :: a -> (env -> a)
    pure = const
    (<*>) :: (env -> (a -> b)) -> (env -> a) -> env -> b
    (<*>) f g x = f x (g x)

instance Monad ((->) env) where
    return :: a -> (env -> a)
    return = pure
    (>>=) :: (env -> a) -> (a -> (env -> b)) -> env -> b
    (>>=) m f x = f (m x) x
```

Reader: Intuition

```haskell
m1 :: env -> a
m2 :: env -> b
f :: a -> b -> c
```

```haskell
do x <- m1
   y <- m2
   return $ f x y
```



Example: obtaining the neighbouring nodes in a graph

```haskell
type Reader = (->)
type Node = Int
type Graph = [(Node, [Node])]

getNeighbours :: Node -> Reader Graph [Node]
getNeighbours x = do
    neighbours <- lookup x
    return $ concat neighbours
```

```haskell
lookup :: a -> [(a, b)] -> Maybe b
lookup :: Node -> Reader Graph (Maybe [Node])

concat :: Foldable t => t [a] -> [a]
concat :: Maybe [Node] -> [Node]
```

Now we can perform DFS; graph is not mentioned anywhere!

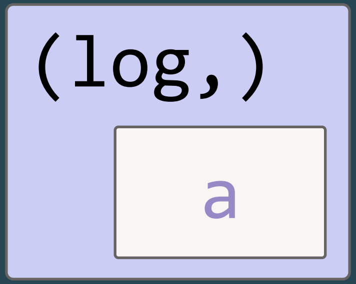
```haskell
dfs :: Node -> Node -> Reader Graph Bool
dfs src dst = aux [] src where
  aux :: [Node] -> Node -> Reader Graph Bool
  aux visited current
    | arrived          = return True
    | alreadyVisited   = return False
    | otherwise        = do
      neighbours <- getNeighbours current
      ls <- mapM (aux (current : visited)) neighbours
      return $ or ls
      where arrived = current == dst
            alreadyVisited = current `elem` visited
```

```
ghci> my_map = [(1, [2, 3])
              , (2, [1])
              , (3, [1, 4])
              , (4, [3])
              , (5, [6])
              , (6, [5])]
ghci> dfs 5 6 my_map
True
ghci> dfs 5 2 my_map
False
ghci> dfs 1 2 [] -- empty
False
```

Reader monad abstracts the environment from programmers; however, can directly get the environment using `ask` function

```haskell
ask :: Reader env env
ask = id

getNeighbours :: Node -> Reader Graph [Node]
getNeighbours x = do
    my_graph <- ask -- gets the graph directly
    let neighbours = lookup x my_graph
    return $ concat neighbours
```

Writer



An `a` that also has some `log`

`(log,)` is a monad!

```haskell
instance Functor (log,) where
    fmap :: (a -> b) -> (log, a) -> (log, b)
    fmap f (log, a) = (log, f a)

instance Monoid log => Applicative (log,) where
    pure :: a -> (log, a)
    pure = (mempty,)
    (<*>) :: (log, a -> b) -> (log, a) -> (log, b)
    (<*>) (log1, f) (log2, x) = (log1 `mappend` log2, f x)

instance Monoid log => Monad (log,) where
    return :: a -> (log, a)
    return = pure
    (>>=) :: (log, a) -> (a -> (log, b)) -> (log, b)
    (log, a) >>= f = let (log2, b) = f a
                     in  (log1 `mappend` log2, b)
```

Monoids

Our log must be combine-able and have an empty value ε that behaves in the most obvious way

$$E \oplus \varepsilon = \varepsilon \oplus E = E$$
$$E_1 \oplus (E_2 \oplus E_3) = (E_1 \oplus E_2) \oplus E_3$$

Make the log type a **Monoid, which are Semigroups with empty values**!

```
class Semigroup a => Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

Monoids

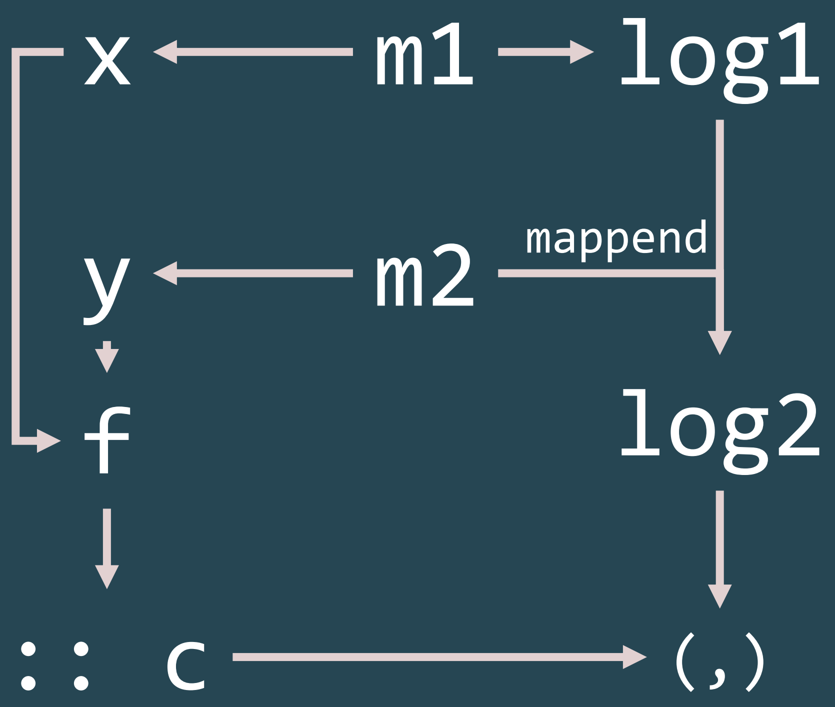
As per usual, lists are monoids with respect to the empty list and list concatenation!

```haskell
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Writer: Intuition

```
m1 :: (log, a)
m2 :: (log, b)
f :: a -> b -> c
```

```
do x <- m1
   y <- m2
   return $ f x y
```



Writer

`ask` for `Reader`s retrieve environment, `write` for `Writer`s writes to log

```haskell
write :: w -> (w, ())
write = (,())
```

Writer

Example of logged addition:

```
type Writer = (,)
type Log = [String]

loggedAdd :: Int -> Int -> Writer Log Int
loggedAdd x y = do
    let z = x + y
    write [show x ++ " + " ++ show y ++ " = " ++ show z]
    return z
```

Writer

We can use `loggedAdd` to define a logged sum!

```
loggedSum :: [Int] -> Writer Log Int
loggedSum [] = return 0
loggedSum (x:xs) = do
    sum' <- loggedSum xs
    loggedAdd x sum'
```

```
ghci> y = loggedSum [1, 2, 3]
ghci> snd y
6
ghci> fst y
["3 + 0 = 3","2 + 3 = 5","1 + 5 = 6"]
```

State

Stateful operation combines `Reader` and `Writer`; is a function that receives state and produces something with a new state

```haskell
randomInt :: Seed -> (Int, Seed)
```

State

```haskell
newtype State s a = State { runState :: s -> (a, s) }
```

`newtype` declaration: same as `data` declaration with one constructor over one field; introduces no operational overhead

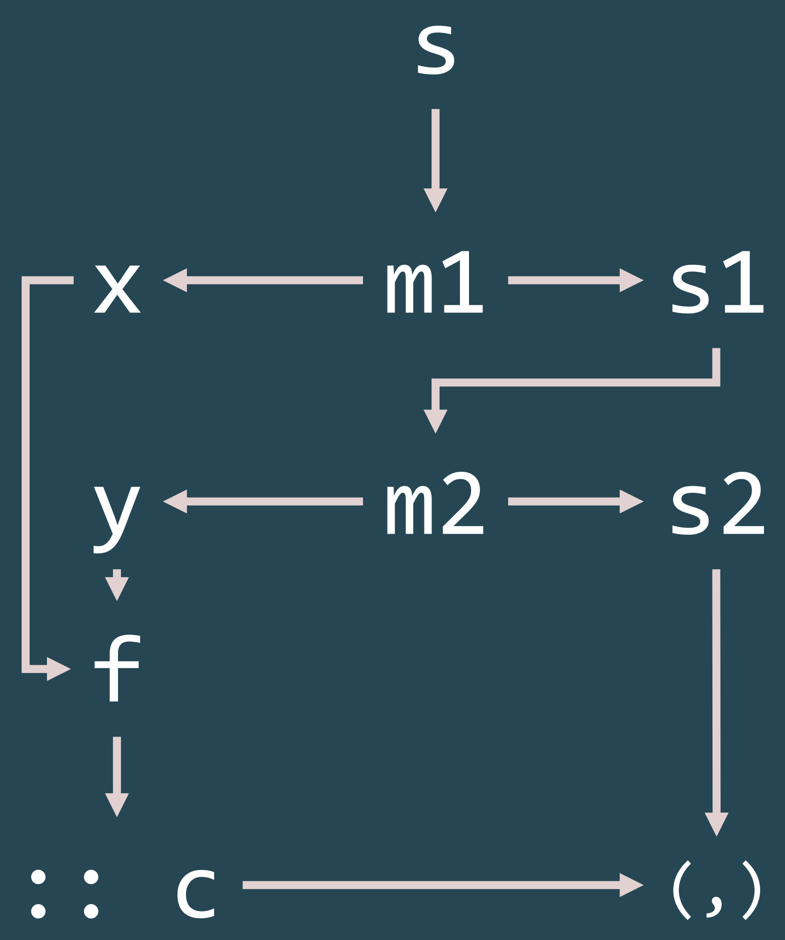
```haskell
instance Functor (State s) where
    fmap :: (a -> b) -> State s a -> State s b
    fmap f (State f') = State $
        \s -> let (a, s') = f' s
              in  (f a, s')
instance Applicative (State s) where
    pure :: a -> State s a
    pure x = State (x,)
    (<*>) :: State s (a -> b) -> State s a -> State s b
    (<*>) (State f) (State x) = State $
        \s -> let (f', s') = f s
                  (x', s'') = x s'
              in  (f' x', s'')
instance Monad (State s) where
    return :: a -> State s a
    return = pure
    (>>=) :: State s a -> (a -> State s b) -> State s b
    (State f) >>= m = State $
        \s -> let (a, s') = f s
                  State f' = m a
              in  f' s'
```

basic instances

State: Intuition

```haskell
m1 :: s -> (a, s)
m2 :: s -> (b, s)
f :: a -> b -> c
```

```haskell
do x <- m1
   y <- m2
   return $ f x y
```



State

Helper functions to retrieve and update state just like ask and write

```
put :: s -> State s ()
put s = State $ const ((), s)

get :: State s s
get = State $ \s -> (s, s)

modify :: (s -> s) -> State s ()
modify f = do s <- get
              put (f s)
```

State

Example stateful operation: memoized fibonacci

```haskell
type Memo = [(Integer, Integer)]
getMemoized :: Integer -> State Memo (Maybe Integer)
getMemoized n = lookup n <$> get
```

```haskell
fib :: Integer -> Integer
fib n = fst $ runState (aux n) [] where
  aux :: Integer -> State Memo Integer
  aux 0 = return 0
  aux 1 = return 1
  aux n = do
    x <- getMemoized n
    case x of
        Just y -> return y
        Nothing -> do
            r1 <- aux (n - 1)
            r2 <- aux (n - 2)
            let r = r1 + r2
            modify ((n, r) :)
            return r
```

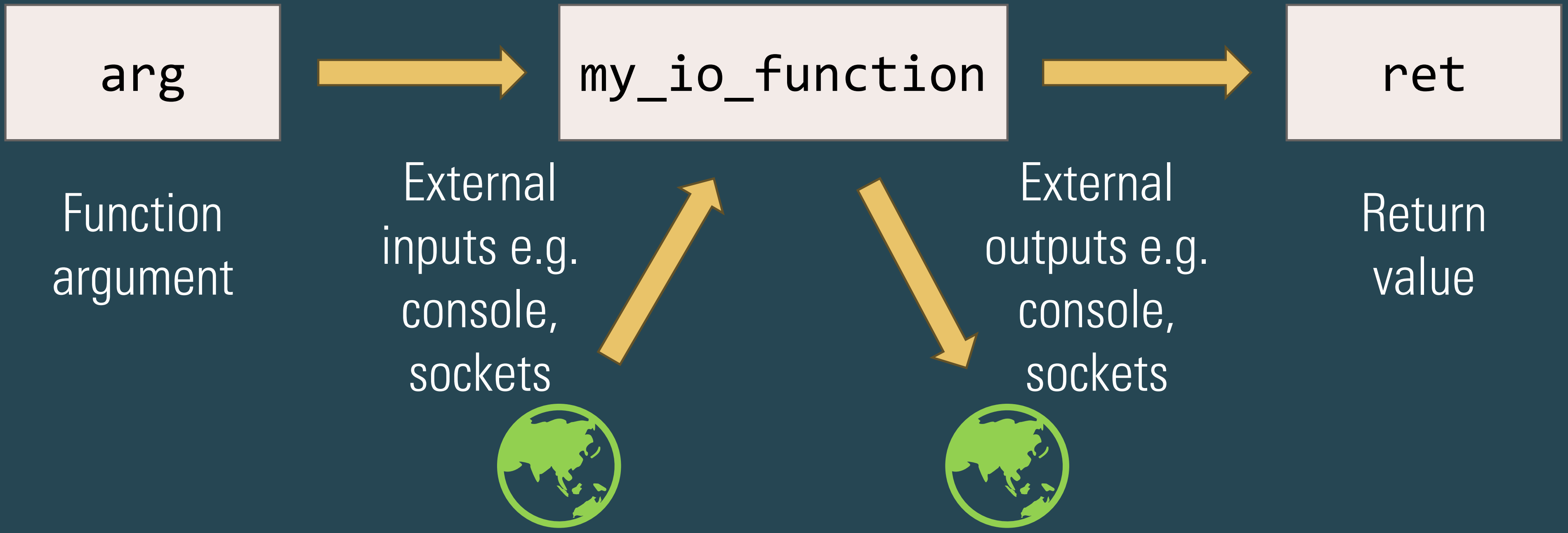
State

Example stateful operation: memoized fibonacci

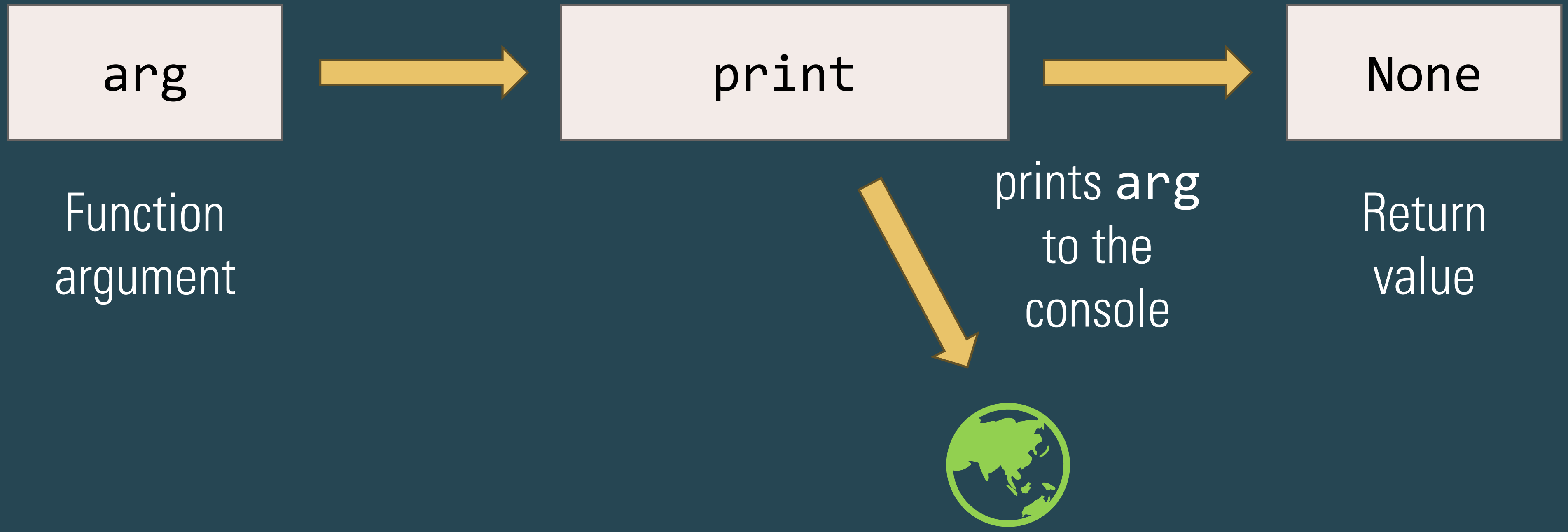
I/O

Finally, we shall learn how to write a Hello World! Program. However, how does a purely functional program produce side effects?

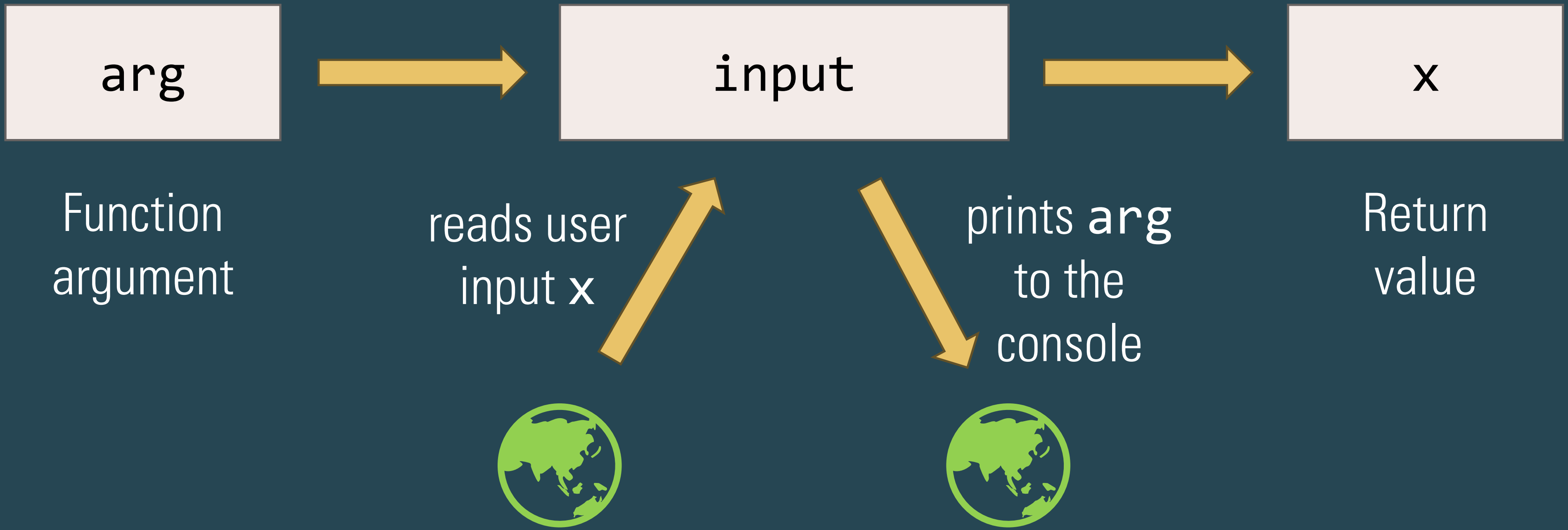
I/O in Other Languages



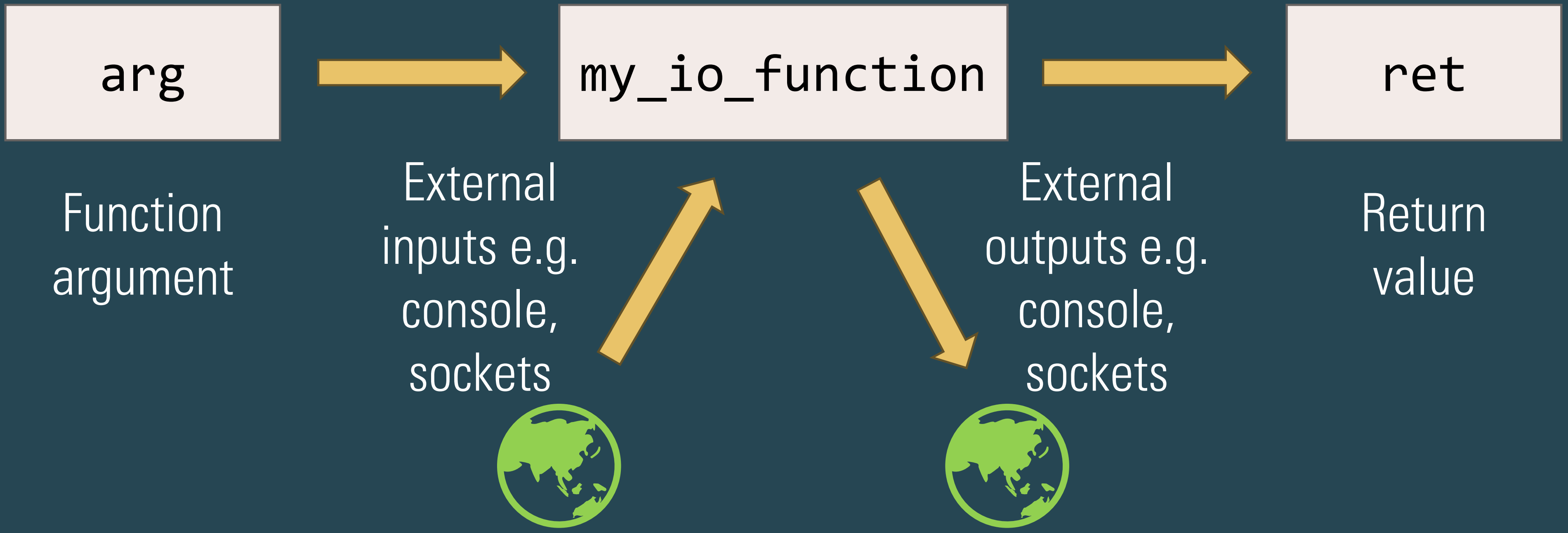
I/O in Other Languages



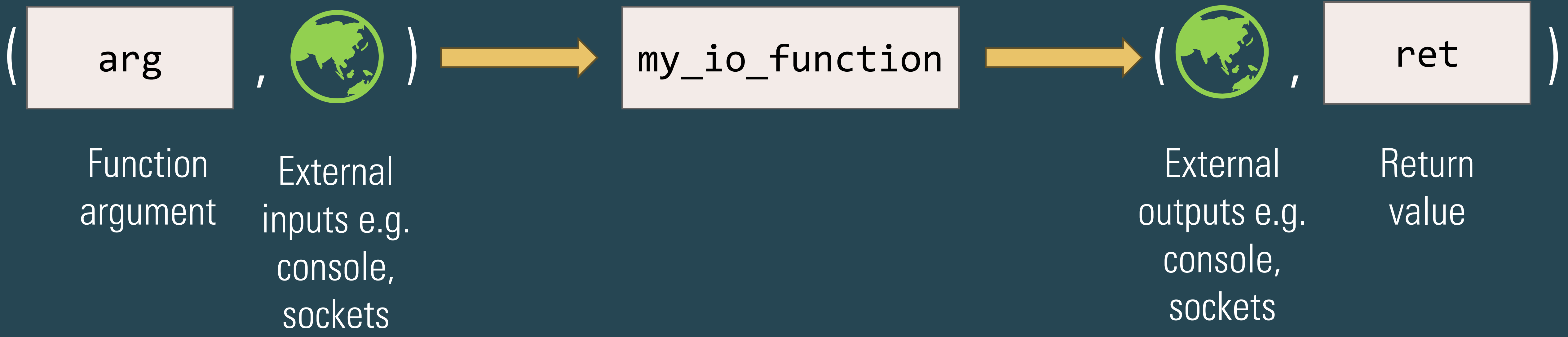
I/O in Other Languages



What if the world (external environment) was a term? :O



Group inputs as tuple, outputs as tuple



Currying

`arg` → → `my_io_function` → (, `ret`)

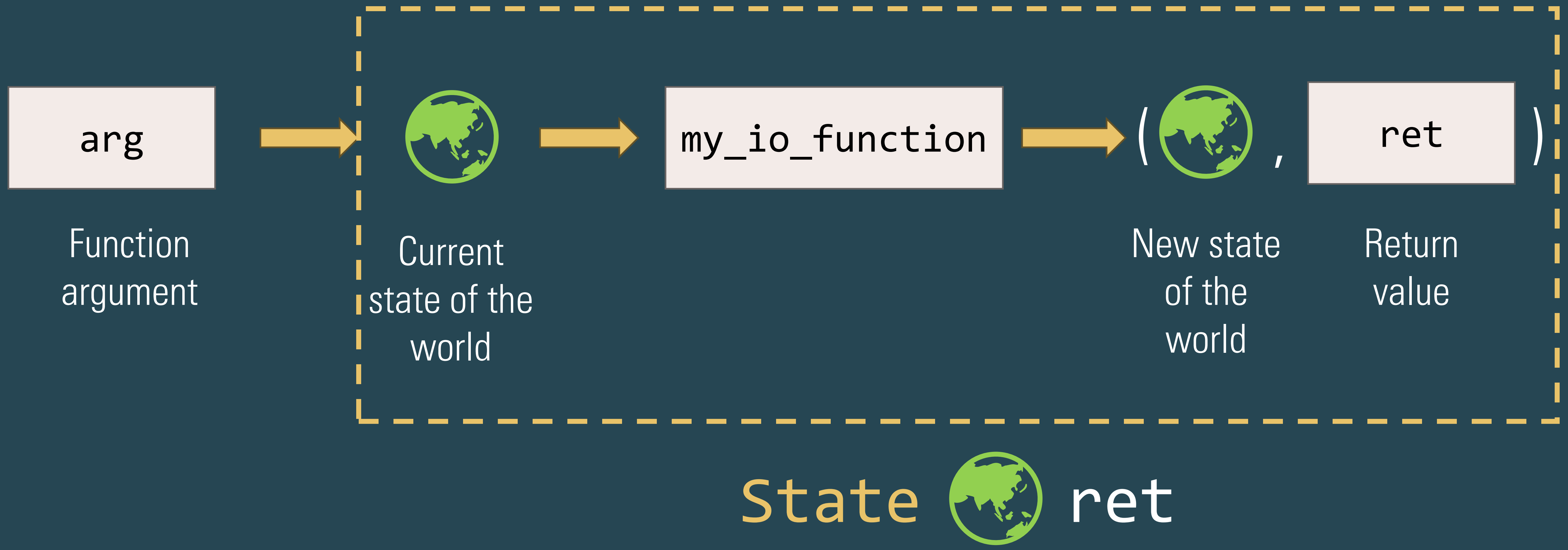
Function argument

External inputs e.g. console, sockets

External outputs e.g. console, sockets

Return value

State monad?

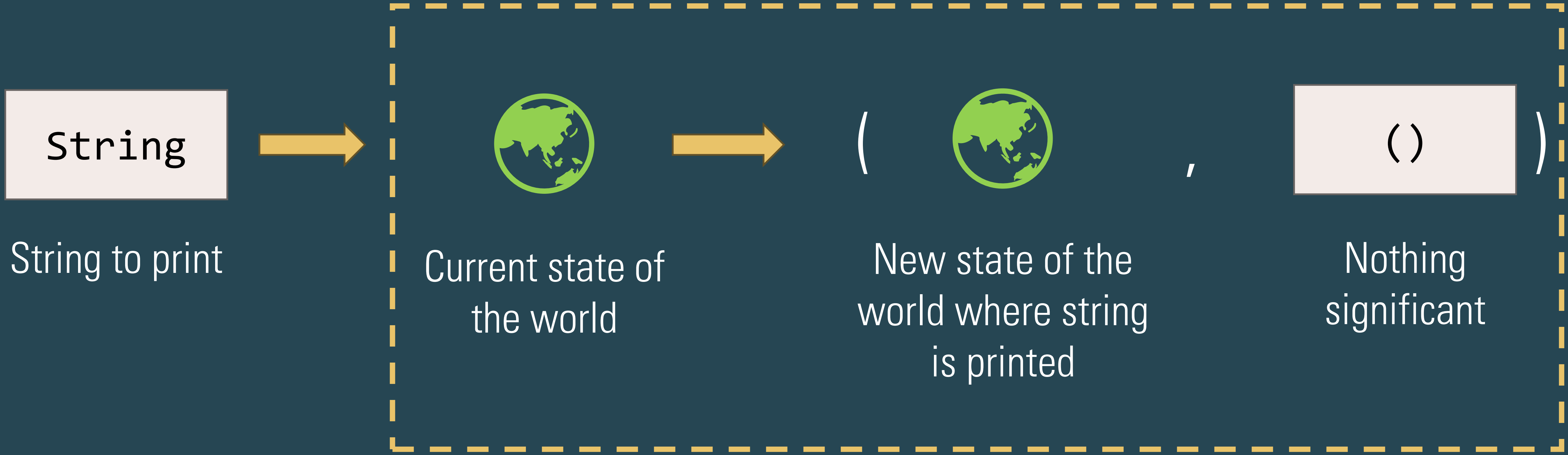

arg
Function argument
Current state of the world
my_io_function
(, ret)
New state of the world
Return value
State ret

I/O

`IO` monad is the `State` monad whose state is the real world

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

```haskell
putStrLn :: String -> IO ()
```



IO ()

```haskell
getLine :: IO String
```

→ (, `String`)

Current state of the world

New state of the world where program waits for user input

Whatever the user entered

`IO String`

Composing I/O actions is as simple as using the `State` monad! `RealWorld` term never needs to be accessed or used

```haskell
-- Main.hs
main :: IO ()
main = do
    name <- getLine
    putStrLn $ "Hello " ++ name ++ "!"
```

`IO` monad abstracts `RealWorld` state and prevents impurity in pure functions. All functions performing I/O actions must be done within the `IO` monad… enforced by type system!

```haskell
main :: IO ()
```

`IO ()`

When a program is run, the Haskell runtime "passes in the current state of the world" into main, and "returns the new state of the world after program execution"

Monad Transformers

Length of path between two connected edges in graph, where each node only has one neighbouring node

```haskell
type Node = Int
type Graph = [(Node, Node)]
dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst gph = aux src [] gph where
    aux :: Node -> [Node] -> Graph -> Maybe Int
    aux current visited gph'
      | arrived = return 0
      | alreadyVisited = Nothing
      | otherwise  = do
          n <- lookup current gph
          (+1) <$> aux n (current : visited) gph'
     where arrived = current == dst
           alreadyVisited = current `elem` visited
```

Currently using the **Maybe** monad, can we also add **Reader**?

```haskell
type Node = Int
type Graph = [(Node, Node)]
dfs :: Node -> Node -> Reader Graph (Maybe Int)
dfs src dst = aux src [] where
    aux :: Node -> [Node] -> Reader Graph (Maybe Int)
    aux current visited
      | arrived = return 0
      | alreadyVisited = Nothing
      | otherwise  = do
          n <- lookup current
          (+1) <$> aux n (current : visited)
     where arrived = current == dst
           alreadyVisited = current `elem` visited
```

Naively changing the type to use **`Reader`** and **`Maybe`** and removing occurrences of graph will not type check as **`do`** block works in **`Reader`** context, not **`Maybe`**!

Enriching Readers with Maybe

```haskell
newtype ReaderMaybe env a = ReaderMaybe { runReaderMaybe :: Reader env (Maybe a) }

instance Monad (ReaderMaybe env) where
  return :: a -> ReaderMaybe env a
  return = pure
  (>>=) :: ReaderMaybe env a -> (a -> ReaderMaybe env b) -> ReaderMaybe env b
  (ReaderMaybe ls) >>= f = ReaderMaybe $ do
    m <- ls
    case m of
      Just x -> runReaderMaybe $ f x
      Nothing -> return Nothing
```

`ReaderMaybe` monad is a combination of `Reader` and `Maybe`!

Enriching Readers with Maybe

```haskell
dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst = runReaderMaybe (aux src []) where
    aux :: Node -> [Node] -> ReaderMaybe Graph Int
    aux current visited
      | arrived = return 0
      | alreadyVisited = ReaderMaybe $ return Nothing
      | otherwise  = do
          n <- ReaderMaybe $ lookup current
          (+1) <$> aux n (current : visited)
     where arrived = current == dst
           alreadyVisited = current `elem` visited
```

Now we can use the **`ReaderMaybe`** monad in our **`dfs`** function; graph not mentioned anywhere and we still have **`Maybe`** behaviour!

Enriching IO with Maybe

```haskell
newtype IOMaybe a = IOMaybe { runIOMaybe :: IO (Maybe a) }

instance Monad IOMaybe where
  return :: a -> IOMaybe a
  return = pure
  (>>=) :: IOMaybe a -> (a -> IOMaybe b) -> IOMaybe b
  (IOMaybe m) >>= f = IOMaybe $ do
    maybe_m <- m
    case maybe_m of
      Just x -> runIOMaybe $ f x
      Nothing -> return Nothing
```

Combining `IO` and `Maybe` requires us to go through the same process!

Enriching IO with Maybe

```haskell
instance Monad (ReaderMaybe env) where
  return = pure

  (ReaderMaybe ls) >>= f = ReaderMaybe $ do
    m <- ls
    case m of
      Just x -> runReaderMaybe $ f x
      Nothing -> return Nothing
```

```haskell
instance Monad IOMaybe where
  return = pure

  (IOMaybe m) >>= f = IOMaybe $ do
    maybe_m <- m
    case maybe_m of
      Just x -> runIOMaybe $ f x
      Nothing -> return Nothing
```

Notice any similarities?

Monad Transformers

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
    return = MaybeT . return . Just
    x >>= f = MaybeT $ do
        v <- runMaybeT x
        case v of
            Nothing -> return Nothing
            Just y  -> runMaybeT (f y)
```

MonadT enriches m with Monad

Monad Transformers

```haskell
dfs :: Node -> Node -> Graph -> Maybe Int
dfs src dst = runMaybeT (aux src []) where
    aux :: Node -> [Node] -> MaybeT (Reader Graph) Int
    aux current visited
      | arrived = return 0
      | alreadyVisited = MaybeT $ return Nothing
      | otherwise  = do
        n <- MaybeT $ lookup current
        (+1) <$> aux n (current : visited)
     where arrived = current == dst
           alreadyVisited = current `elem` visited
```

Directly use the `MaybeT` monad transformer in our `dfs` implementation!

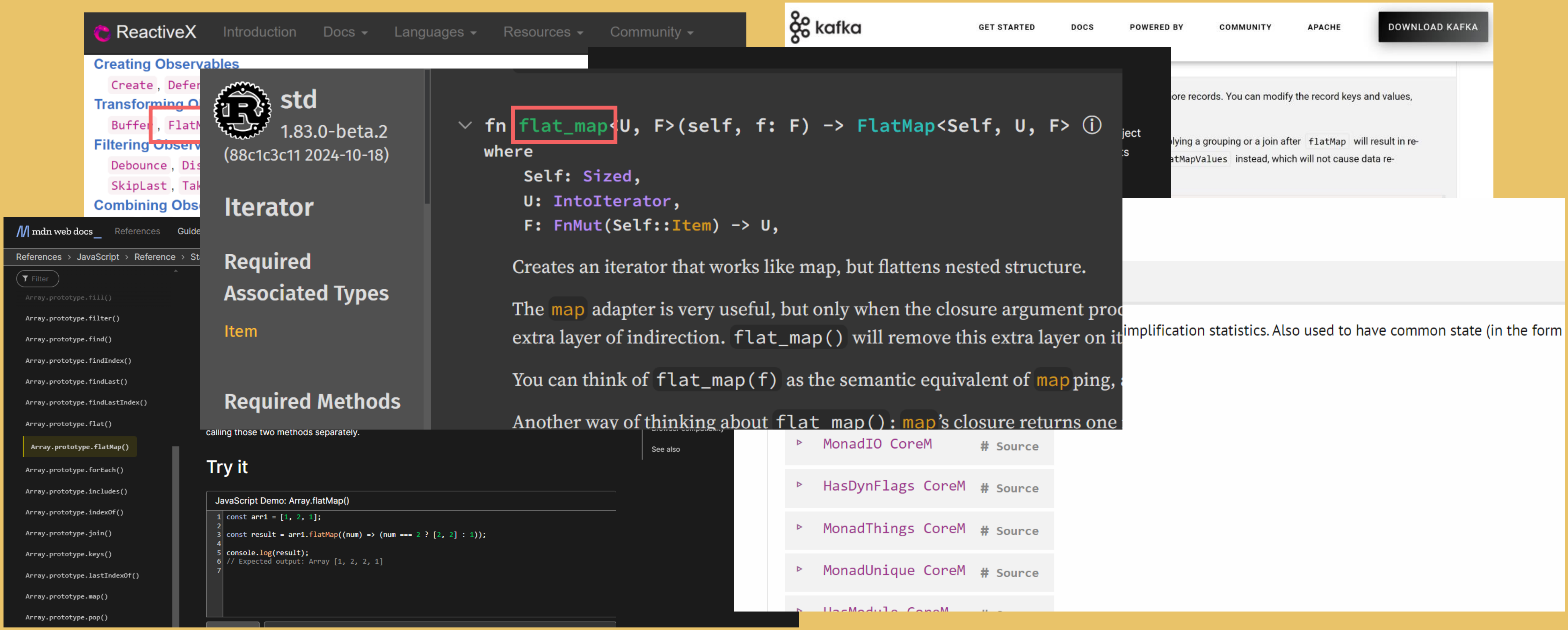
Monad Transformer Libraries

Monads are so common—`transformers` and `mtl` libraries contain common monad transformers. Download dependencies and use them!

To work with projects with dependencies easily, use a package manager/build tool like `cabal` or `stack`!

Monads in the Wild

Monads Are Everywhere



Creating Monads?

- Does your library involve nondeterminism or streams/lists of data?
- Does your library perform I/O?
- Does your library produce potentially empty computation?
- Does your library potentially fail?
- Does your library read from an environment?
- Does your library write to state?
- Does your library process state?

If yes, create a monad!

Thank you

Foo Yong Qi

yongqi@nus.edu.sg

https://yongqi.foo/