Step 1: Storage

• Build up to the IO monad piece-by-piece

• Start with a way of viewing state as a dictionary

• Then just think of the world is a really big dictionary
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

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

instance Monoid a => Monad ((,) a) where
  return :: x -> (,) a x
  return x = (mempty,x)

  (>>=) :: m x -> (x -> m y) -> m y
  (>>=) :: (a,x) -> (x -> (a,y)) -> (a,y)
  (>>=) (xm,xv) f =
    let (xm’,xv’) = f xv in
    (mappend xm xm’,xv’)

class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

data Storage a b = Storage (a -> Maybe b)

instance Monoid (Storage a b) =
  mempty = Storage (\_ -> Nothing)
  mconcat (Storage f) (Storage g) =
    Storage (\x ->
      case g x of
        Nothing -> f x
        Just y -> Just y)
data Storage a b = Storage (a -> Maybe b)

instance Monoid (Storage a b) =
  mempty = Storage (\_ -> Nothing)
  mconcat (Storage f) (Storage g) =
    Storage (\x ->
      case g x of
        Nothing -> f x
        Just y -> Just y)

instance Monoid a => Monad ((,) a) where
  return x = (mempty,x)
  (>>=) (xm,xv) f =
    let (xm’,xv’) = f xv in
    (xm’,f xv xv’)

set :: (Eq a) => a -> b -> Storage a b
set k v =
  Storage (\k’ ->
    if k == k’ then
      Just v
    else
      Nothing)
Stateful Computation with Memory Addresses

data Storage a b = Storage (a -> Maybe b)
set :: (Eq a) => a -> b -> Storage a b

type ComputerMemory = Storage Int Int
callAndSet :: (Int -> (ComputerMemory,Int)) -> Int -> (ComputerMemory,())
callAndSet f i = do
  x <- f i
  y <- f x
  return ()

-- x has been set to f(i). f(i) may have also had computations that set other
Now, what if the storage Monoid is replaced by the world

data Storage a b = Storage (a -> Maybe b)

instance Monoid (Storage a b) =
  mempty = Storage (\_ -> Nothing)
  mconcat (Storage f) (Storage g) =
    Storage (\x ->
      case g x of
        Nothing -> f x
        Just y -> Just y)

data World = *The Universe*

instance Monoid World =
  mempty = A “empty” universe
  mconcat World1 World2 =
    *Apply the effects of world2 to world1*
    *Let propagate according to physics*
The IO Monad as Shorthand

```haskell
data World = *The Universe*

instance Monoid (Storage a b) =
  mempty = A “empty” universe
  mconcat World1 World2 =
  *Apply the effects of world2 to world1*
  *Let propagate according to physics*

instance Monad IO where
  return x = (x,*empty world*)

  bind :: IO a -> (a -> IO b) -> IO b
  bind (x,world) f =
    let (out,sideeffects) = f x in
    let outworld = mconcat world sideffects in
    (out,outworld)
```
High Level Takeaway

• The IO Monad encodes side-effects

• In other words: T1 -> T2 -> IO T3 means T3 is returned, and there are side effects in the computation

• Side effects can mean many things

• Arrays use the IO monad

• Writing to disk uses the IO monad

• Actuating a motor uses the IO monad
How is this functional?

• If you have IO x -> IO y, then the whole world is included in the input.

• In other words, if you read from disk at one time, and then read from disk a second time, it’s ok to get different results.

• If you somehow go back in time a perform the same read access in the same world, you will get the same result.

• Maybe this is not true with quantum physics???
Pure Side Effects

• What about a computation where there should be *no* return value

• You just want to perform a computation, which has a side effect

  • OO analogue: public void PushElement(int x);

• This is done with (), aka Unit, Top, True, and The Terminal Object
• () is a type with one value, ()

• There is exactly one function from every type to unit
  • \_ -> ()
Usages in non-IO Monad

• Creating a “thunk” a piece of computation that hasn’t yet been evaluated

• \() \rightarrow *really lengthy computation*

• The really lengthy computation is not performed immediately, but can be chosen to be performed or not

• ifEqElse :: (Eq a) => a -> a -> (() -> b) -> (() -> b) -> b
  ifEqElse a1 a2 f1 f2 = if a1 == a2 then f1 () else f2 ()
Usages in IO Monad

• Not much information contained in something of type ()

• So it’s basically pure side-effect
IO Functions

- putStrLn :: String -> IO ()
- Prints string to stdOut
- Nothing to return!
IO Functions

- `writeFile :: FilePath -> String -> IO ()`

- `writeFile "/path/to/file" "MySavedString"`
Writing is all well and good, what about reading?

- Say I want to read from a file
- It should take in knowledge of the outside world
- So the input should be IO String
- And the output should be a String
  - But by reading a file you might change things in the OS
  - So you’d think readFile :: IO FilePath -> IO String
This is not the case

• Pragmatism gets in the way of mathematical purity again!

• It’s really pragmatic to enable all effectful computation to happen in IO monad in do notation
  
  • But you can’t use do notation if readFile takes in IO FilePath

• readFile :: FilePath -> IO String
This makes me mad
Let's do some IO in Haskell

```haskell
main :: IO () = do
  _ <- putStrLn "Give input"
  s <- getLine
  x = performComputation s
  putStrLn (show x)
```
How do I get out of the IO Monad

- In Lists, you can go from List a -> a if the list is nonempty.
- In Maybe, you can go from Maybe a -> a if the Maybe is not Nothing.
- You cannot go from IO a -> a. Ever.
Using do notation + IO monad works quite well

Provides some nice guarantees

If you *ever* interact with the real world, or use stateful code, the result will be in the IO monad

Pragmatically, you only use readFile s when you are doing stateful code, so it’s ok for readFile :: FilePath -> IO String
Ok, I lied about one other thing

unsafePerformIO :: IO a -> a
Don’t Use unsafePerformIO

- You really have to know what you’re doing to use it
  - I have never used it
- You need to know deep internals about how lazy computation works, how compilation works, etc
- Only really done by library makers
What should unsafePerformIO be used for

• Say you have a library that *looks* purely functional, but in the internals isn’t

• Example: caching library

  • Caches have no side effects that change the values, but just store sub-computations

  • However, storing the values requires the IO monad

  • But consumers don’t need to know this

  • Good opportunity for unsafePerformIO