FIT2102
Programming Paradigms
Lecture 10

File IO
Monads

Faculty of Information Technology

MONASH University

Learning Outcomes

- Create programs that perform IO operations such as reading or writing from stdio and files.
- Describe how Monads enable effectful programming while allowing Haskell to still be considered a pure language.

Review: Functor Typeclass

```
ghci> :i Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
      -- Defined in `GHC.Base'
instance Functor (Either a) -- Defined in `Data.Either'
instance Functor [] -- Defined in `GHC.Base'
instance Functor Maybe -- Defined in `GHC.Base'
instance Functor IO -- Defined in `GHC.Base'
instance Functor ((->) r) -- Defined in `GHC.Base'
instance Functor ((,) a) -- Defined in `GHC.Base'
```

```
ghci> (+1) <$> Nothing
Nothing
ghci> (+1) <$> Just 2
Just 3
ghci> (+1) <$> [1, 2, 3]
[2,3,4]
ghci> (+1) <$> Node 7 [Node 1 [], Node 2 [], Node 3 [Node 4 []]]
Node 8 [Node 2 [],Node 3 [],Node 4 [Node 5 []]]
```

Review: Applicative Typeclass

```
> :i Applicative
class Functor f => Applicative (f :: * -> *) where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b

       -- Defined in `GHC.Base'
instance Applicative (Either e) -- Defined in `Data.Either'
instance Applicative [] -- Defined in `GHC.Base'
instance Applicative Maybe -- Defined in `GHC.Base'
instance Applicative IO -- Defined in `GHC.Base'
instance Applicative ((->) a) -- Defined in `GHC.Base'
instance Monoid a => Applicative ((,) a) -- Defined in `GHC.Base'
```

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
ghci> replicate <$> [1,2,3] <*> ['a','b','c']
["a","b","c","aa","bb","cc","aaa","bbb","ccc"]
```

Exercise 1: Warm up

Consider type of fmap:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

The implementation of fmap for the Maybe instance of Functor:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

The implementation of map for List (which is also also a Functor, i.e. map = fmap):

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

Compare to the TypeScript type for Array.map:

```
Array<T>.map<U>(fn: (value: T) => U): U[]
```

A direct translation of the TypeScript type to Haskell might look like:

```
map :: Array t -> (t->u) -> Array u
```

Computational Contexts

Some instances of Functor and Applicative:

- Maybe - a computation that can fail
- List - a computation that can return 0 or many results

These are our old, familiar friends.

What we can already do with our friends

Map over them (Functor):

```
fmap (*2) (Just 10) ⇒ Just 20

fmap (++"!") ["a","b","c"] ⇒ ["a!", "b!", "c!"]
```

What we can already do with our friends

Combine them in a limited way (Applicative):

```
(+) <$> Just 1 <*> Just 2 ⇒ Just 3

(*) <$> [2,3] <*> [10,100] ⇒ [20,200,30,300]

Card <$> [Spade .. Heart] <*> [Two .. Ace]
   ⇒ [S2,S3,S4,S5,S6,S7,S8,S9,S10,SJ,SQ,SK,SA,C2,C3,C4, ...]

filter (\(i,j)->i<j) $ (,) <$> [1..5] <*> [1..5]
   ⇒ [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
```

Interlude: list comprehensions

Source for [`instance Applicative []`]:

```
pure x    = [x]
fs <*> xs = [f x | f <- fs, x <- xs] -- list comprehension
```

English: “the set (Haskell list) of all functions in `fs` applied to all values in `xs`”

Mathematical set builder notation: $\{ f(x) \mid f \in fs \wedge x \in xs \}$

You can also put in conditions to filter by:

```
GHCi> [ (i,j) | i <- [1..5], j <- [1..5], i < j ]
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
```

A new friend

Our newest friend is IO.

- Maybe - a computation that can fail
- List - a computation that can return 0 or many results
- IO - a computation that uses "the outside world"

A value of type "`IO a`" produces a value of type "a", while interacting with the outside world.

Often called an "IO action" or just "action".

Some simple IO actions

```
getLine :: IO String

putStrLn :: String -> IO ()

getCurrentTime :: IO UTCTime

readFile :: FilePath -> IO String
```

Note: `putStrLn "hello"` is the action, not `putStrLn` alone.

Values of type "IO a" are proper values!

We can already do IO!

Because IO is an instance of Functor and Applicative, we can already do a lot.

ghci will "execute" IO actions for you.

ghci time: io1.hs & io2.hs

What can't we do?

If we can already use IO, what is this lecture for?

What do we gain from Monad?

Shortcoming of Functor and Applicative

You have these two:

```haskell
readName :: IO String
readName = getLine
greet :: String -> IO ()
greet name = putStrLn ("Nice to meet you, " ++ name ++ ".")
```

You want to execute readName and pass the input string to greet.

Cannot be done with Functor and Applicative.

(see io3.hs)

New operator!

Let's invent the function we want.

```
IO String -> (String -> IO ()) -> IO ()
```

Generalised and given a name:

```
(>>=) :: f a -> (a -> f b) -> f b
```

In essence, that's all there is to monads.

Introducing: Monad

```
ghci> :i Monad
class Applicative m => Monad (m :: * -> *) where
 (>>=) :: m a -> (a -> m b) -> m b
 (>>) :: m a -> m b -> m b              -- special case of (>>=)
 return :: a -> m a                     -- just "pure" from Applicative
 fail :: String -> m a                  -- shouldn't be here, historical reasons
      -- Defined in `GHC.Base'

instance Monad (Either e) -- Defined in `Data.Either'
instance Monad [] -- Defined in `GHC.Base'
instance Monad Maybe -- Defined in `GHC.Base'
instance Monad IO -- Defined in `GHC.Base'
instance Monad ((->) r) -- Defined in `GHC.Base'
instance Monoid a => Monad ((,) a) -- Defined in `GHC.Base'
```

Typeclasses for mapping and binding over contexts

Functor
`fmap (<$>)`

Applicative
`pure`
`(<*>)`

Monad
`(=<<)`

Functor - things that can be mapped over

```
(<$>) :: Functor f     =>    (a -> b)  ->  f a  ->  f b
(+1) <$> [1..9]
```

Applicative - Functors that support function application within their contexts

```
(<*>) :: Applicative f =>  f (a -> b)  ->  f a  ->  f b
[(+1)] <*> [1..9]
```

Monad - Applicative Functors that additionally support function “bind”

```
(=<<) :: Monad m       =>  (a -> m b)  ->  m a  ->  m b
pure . (+1) =<< [1..9]
```

Typeclasses for mapping and binding over contexts

Functor

```
fmap  (<$>)
```

↓

Applicative

```
pure
(<*>)
```

↓

Monad

```
(>>=)  (=<<)
```

Functor - things that can be mapped over

```haskell
(<$>) :: Functor f     =>    (a -> b) -> f a -> f b
(+1) <$> [1..9]
```

Applicative - Functors that support function application within their contexts

```haskell
(<*>) :: Applicative f =>  f (a -> b) -> f a -> f b
[(+1)] <*> [1..9]
```

Monad - Applicative Functors that additionally support function “bind”

```haskell
(=<<) :: Monad m       =>  (a -> m b) -> m a -> m b
pure . (+1) =<< [1..9]

(>>=) :: Monad m       =>  m a -> (a -> m b) -> m b
[1..9] >>= pure . (+1)
```

Different ways to apply functions:

```
($)   ::                           (a -> b)  ->    a  ->    b
(<$>) :: Functor f     =>    (a -> b)  ->  f a  ->  f b
(<*>) :: Applicative f =>  f (a -> b)  ->  f a  ->  f b
(=<<) :: Monad m       =>  (a -> m b)  ->  m a  ->  m b
(>>=) :: Monad m       =>  m a  ->  (a -> m b)  ->  m b
```

Bind

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

Examples with maybe...

```haskell
instance Monad Maybe where
   return x = Just x
   Nothing >>= f = Nothing
   Just x >>= f  = f x
   fail _ = Nothing
```

Mathematical basis for Monad

The scary name "monad" comes from category theory.

No category theory in this lecture.

You don't need to know any category theory to use Haskell.

But in Haskell we represent natural transformations as polymorphic functions (the components of categorical natural transformations). So η becomes `return :: a -> m a`, and μ becomes `join :: m (m a) -> m a`, where `m` is the endofunctor for the monoidal structure of a monad is thus hidden

Let's try out our new operator

ghci time: maybe2.hs

Exercise

Implement this function:

```
sumFile :: FilePath -> IO Int
```

which reads the given text file and adds up all the integers in it. There is one integer per line. Don't worry about handling invalid input.

Useful functions:

```
readFile :: FilePath -> IO String
read :: Read a => String -> a
lines :: String -> [String]
```

Do-notation

ghci time: io4.hs & maybe3.hs

Do-notation

Remember:

- Do-notation can be used with any Monad
- It is just an alternative syntax for >>= and >>

```
putStrLn "Hi, what's your name?" >> getLine >>= \name->putStrLn $"Hi "++name++"!"
 ==
do
  putStrLn "Hi, what's your name?"
  name <- getLine
  putStrLn $ "Hi "++name++"."
```

Do-notation

Remember:

- Do-notation can be used with any Monad
- It is just an alternative syntax for >>= and >>

```
putStrLn "Hi, what's your name?" >> getLine >>= \name->putStrLn $"Hi "++name++"!"
```

```
do
  putStrLn "Hi, what's your name?"
  name <- getLine
  let greeting = "Hi "++name++"."
  putStrLn greeting
```

Equivalent definitions of Monad

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

```
join :: m (m a) -> m a
```

These two are equivalent to bind (>>=).

Maybe example

```
join :: Maybe (Maybe a) -> Maybe a
join Nothing = Nothing
join (Just Nothing) = Nothing
join (Just (Just a)) = Just a
```

Maybe example

```
(>=>) :: (a -> Maybe b)
      -> (b -> Maybe c)
      -> (a -> Maybe c)
f >=> g = \a -> case f a of
                  Nothing -> Nothing
                  Just b -> g b
```

>>= in terms of >=>

```
(>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
(>>=) :: m b -> (b -> m c) -> m c
mb >>= f =
    ??? >=> ???
```

Follow the types!

>>= in terms of >=>

```
(>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
(>>=) :: m b -> (b -> m c) -> m c
mb >>= f =
    (\a -> mb) >=> ???
```

>>= in terms of >=>

```
(>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
(>>=) :: m b -> (b -> m c) -> m c
mb >>= f =
    (\a -> mb) >=> f
```

>>= in terms of >=>

```
(>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
(>>=) :: m b -> (b -> m c) -> m c
mb >>= f =
   ((\a -> mb) >=> f) ()
```

>>= in terms of >=>

```
(>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
(>>=) :: m b -> (b -> m c) -> m c
mb >>= f =
     (const mb >=> f) ()
```

Conclusion

Monad: a natural extension to Functor & Applicative that anyone could have invented!

Can be viewed with different perspectives:

- allows you to "squash" nested contexts: m (m a) -> m a
- allows you to "chain" contextual computations together

A convenient way to treat IO in a pure functional language, but with other uses too.

Extra Information About Monad

The following are not examinable in this unit but may be of interest…

Random Numbers

Most random number generators you come across are *pseudo-random number generators*.

- Fully deterministic
  - Not actually random, just random-looking
- Have a seed value and some internal state

Bad Random Number Generator

Let's write our own random number generator.

Bad in two ways:

- mathematically (we won't fix this)
- hard to use (we will tackle this)

ghci time: rng1.hs

Threading state

We have to pass the internal state of the RNG around.

This is error-prone and a huge inconvenience.

We want a new computational context: "computations that depend on some internal state".

Data type for our new computation

Type of a computation that reads from some state and produces some new state, and some regular output:

```
s -> (a, s)
```

Where s is the type of the state value, and a is the type of the output value. Wrap this in a new data type:

```
data State s a = State (s -> (a, s))
```

Stateful computation

Look at these types:

```
nextRandom :: Int -> Int -> InternalState -> (Int, InternalState)

d6 :: InternalState -> (Int, InternalState)

diceRolls :: Int -> InternalState -> ([Int], InternalState)
```

They all have the form `s -> (a, s)`.
So our data type looks like the right choice.

Adapting our RNG to State

ghci time: rng2.hs

Properties of `State s`

Is it a Functor?

Is it an Applicative?

Is it a Monad?

If so, what is the intuition behind these definitions?

Implementing the instances

ghci time: rng3.hs

State Monad's bind

```haskell
instance Monad (State s) where
  return = pure
  (>>=) :: State s a -> (a -> State s b) -> State s b
  State sa >>= f = State (\s -> let (a, s2) = sa s
                                    State g = f a
                                in g s2)
```

previous state's function

previous Internal state

Apply previous state's function on next internal state

Result of bind will be a new State whose function uses f to transform the input value

You don't need to memorise or particularly grok this.

The point is, it implements the tricky part of managing state so that you don't have to!

Rewriting nextInteger

Now that the state-threading details are in the Functor/Applicative/Monad instances, we can make our use of the RNG simpler.

ghci time: rng4.hs

Running a stateful computation

Once we're in State, can we escape?

We can "run" a stateful computation by providing an initial value for the state.

```
runState :: State s a -> s -> (a, s)
```

"Run this stateful computation, with this initial state, and give me the result and the final state."

Running a stateful computation

Once we're in State, can we escape?

We can "run" a stateful computation by providing an initial value for the state.

```
runState :: State s a -> s -> (a, s)
runState (State f) i = ???
```

"Run this stateful computation, with this initial state, and give me the result and the final state."

Wrapping up State

See rng-final.hs.

Already available in the Control.Monad.State module.

Related to State (and potentially useful for Assignment 2):

- Reader
- Writer
- Parser

Pattern: defined over a data type with a function

Monad Laws

As for Functor and Applicative, instances of Monad must obey some laws.

Right Identity:

```
m >>= return  ===  m

do a <- m      ===  m
   return a
```

Monad Laws

```
Left Identity:

return x >>= f      ===  f x

do a <- return x    ===  f x
   f a
```

Monad Laws

Associativity:

```
(m >>= f) >>= g    ===  m >>= (\x -> f x >>= g)

do a <- do x <- m  ===  do x <- m
           f x             a <- f x
   g a                     g a
```

Monad Laws

```
do response <- do input <- getLine
                  process input
   write response

do input <- getLine
   response <- process input
   write response
```

IO and sequencing

When we chain IO actions, they have to be executed in order.

```
getLine >>= putStrLn
```

The getLine's IO will certainly happen before putStrLn's.

However: see io6.hs

IO and laziness

The IO monad sequences the I/O part, but not the ordinary evaluation.

```
x <- timeAction (pure (slowFib n))
```

Here, the value "inside" the IO action is a not-yet-evaluated one.

It is evaluated when needed: when we print it.

"Wow I wish we had time for more"

- *Learn You a Haskell* chapter on monads
- https://en.wikibooks.org/wiki/Haskell/Understanding_monads
- Look at what's available in Control.Monad
- *Typeclassopedia*: https://wiki.haskell.org/Typeclassopedia

(These are still useful even if your reaction was less enthusiastic.)