FIT2102
Programming Paradigms
Lecture 8

Point-free style code
Functor
Applicative

Faculty of Information Technology

MONASH University

Learning Outcomes

- Apply the reduction rules we learned for Lambda Calculus and the compose combinator to write *point-free* (tacit) style haskell code
- Apply the fmap function from the Functor typeclass to different types
- Apply pure and <*> functions from the Applicative typeclass to different types
- Combine these functions to achieve common coding patterns

A polymorphic function

changes things into sheep when invoked

has multiple type arguments

has a concrete type

may resolve values of different types, depending on inputs

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

n f has the type Ord a => a -> a -> Bool and we apply it to one nume

What is the type now?

Ord a => a -> Bool

Num -> Num -> Bool

Ord a => a -> a -> Integer

(Ord a, Num a) => a -> Bool

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Some fundamental Haskell equivalences (≣)

Eta reduction

```
f x  =  g x
f    =  g
```

Operator Sectioning

```
x + y  ≣  (+) x y
       ≣  ((+) x) y
```

Compose

```
(f . g) x  ≣  f (g x)
```

Point-free Style - applied lambda calculus!

```
sort [] = []
sort (pivot:rest) = lesser ++ [pivot] ++ greater
  where
    lesser = sort $ filter (<pivot) rest
    greater = sort $ filter (>=pivot) rest
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p l = part (<p) l
   above p l = part (>=p) l
   part test l = sort (filter test l)
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)          ⇦ eta reduction
   above p l = part (>=p) l
   part test l = sort (filter test l)
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)                      ⇦ eta reduction
   part test l = sort (filter test l)
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test l = sort ((filter test) l)
```

⇦ because filter is a curried binary function and function application is left associative

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test l = (sort . (filter test)) l   ⇦ because (f.g) x = f(g x)
```

```
ghci> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

functions named f and g have types Char -> String and String -> [St
respectively. The composed function g . f has the type

Char -> String

Char -> [String]

[[String]]

Char -> String -> [String]

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test l = (sort . (filter test)) l
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test = sort . (filter test)
```

⇦ eta reduction

Point-free Style

```haskell
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test = sort . filter test
```

⇦ precedence function application > precedence (.)

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test = (sort .) (filter test)
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part test = ((sort .) . filter) test     ⇦ because (f.g) x = f(g x)
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (<p)
   above p = part (>=p)
   part = (sort .) . filter
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part (p>)        ⇦ because it's the same!
   above p = part (>=p)
   part = (sort .) . filter
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = part ((>) p)      ⇦ because (>) makes > a regular (non-infix) function
   above p = part (>=p)
   part = (sort .) . filter
```

Point-free Style

```haskell
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below p = (part . (>)) p      ⇦ because f.g x = f (g x)
   above p = part (>=p)
   part = (sort .) . filter
```

Point-free Style

```haskell
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below = part . (>)              ⇦ eta reduction
   above p = part (>=p)
   part = (sort .) . filter
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ [pivot] ++ above pivot rest
 where
   below = part . (>)
   above = part . (<=)
   part = (sort .) . filter
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = below pivot rest ++ pivot : above pivot rest
 where
   below = part . (>)
   above = part . (<=)
   part = (sort .) . filter
```

Point-free Style

```haskell
sort [] = []
sort (pivot:rest) = rest `below` pivot ++ pivot : rest `above` pivot
  where
    below = flip $ part . (>)
    above = flip $ part . (<=)
    part = (sort .) . filter
```

⇧ `operator` syntax makes it read more like english

Point-free Style

```
sort [] = []
sort (pivot:rest) = rest `below` pivot ++ pivot : rest `above` pivot
  where
    below = part (>) ; above = part (<=)
    part = flip . (((sort .) . filter) .)    ⇦ move flip into part
```

Point-free Style

```
sort [] = []
sort (pivot:rest) = lower rest ++ pivot : upper rest
  where lower = part (pivot>) ; upper = part (pivot<=)
        part = (sort .) . filter
```

My favourite:

+ concise
+ explicit with the pivot>
- references function arg in the where clause

Exercise 1:

… to be announced

Recap: Maybe

```haskell
data Maybe a = Just a | Nothing
    deriving (Eq, Ord)

phonebook :: [(String, String)]
phonebook = [ ("Bob", "0481 665 242"), ("Fred", "0421 556 442"), ("Alice", "011 985 333") ]

> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b

> lookup "Fred" phonebook
Just "0421 556 442"

> lookup "Tim" phonebook
Nothing
```

lookup is a *partial function*
A partial function is not defined over all the elements of its input set (domain)

We can pattern match Just a or Nothing to give default behaviour

Recap: Pattern matching Maybes

```
reportNumber name = msg $ lookup name phonebook
  where
      msg (Just number)   = show $ "the number is " ++ number
      msg Nothing         = show $ name ++ " not found in database"
```

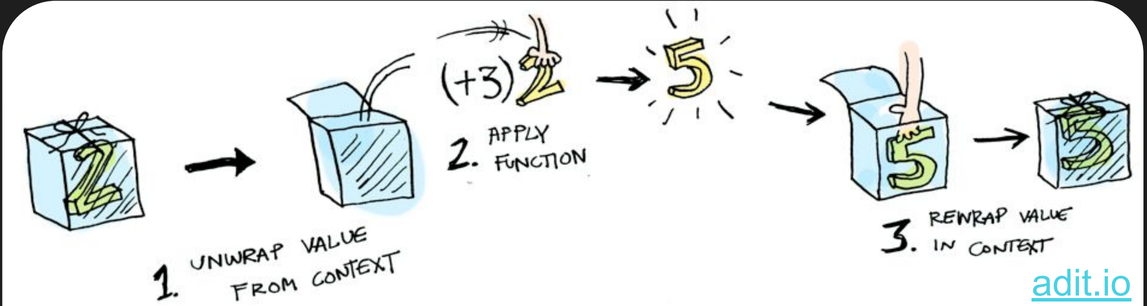
```
*GHCi> printNumber "Fred"

"The number is 0421 556 442"

*GHCi> printNumber "Tim"

"Tim not found in database"
```

The Functor Typeclass

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

instance Functor Maybe where
 fmap f (Just x) = Just (f x)
 fmap f Nothing = Nothing
```

```
ghci> fmap (+3) (Just 2)
Just 5
```

```
person = Just "Tim"
nobody = Nothing

greet p = fmap ("Hello "++) p
ghci> greet person
Just "Hello Tim"
ghci> greet nobody
Nothing
```



adit.io

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'
```

Aliases for fmap: <$>

```
ghci> :i <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
        -- Defined in `Data.Functor'
infixl 4 <$>
```

```
person = Just "Tim"
nobody = Nothing

greet p = ("Hello "++) <$> p
ghci> greet person
Just "Hello Tim"
ghci> greet nobody
Nothing
```

Aliases for fmap: map

List is a functor:

```haskell
instance Functor [] where
    fmap = map
```

```
ghci> (*2) <$> [1..5]
[2,4,6,8,10]
```

Source on Hackage

Applying functions inside contexts

So we can fmap a function over a Maybe without having to unpackage it:

```
GHCi> fmap (+1) (Just 6)
Just 7
```

This is such a common operation that there is an operator alias for fmap: <$>

```
GHCi> (+1) <$> (Just 6)
Just 7
```

Which also works over lists:

```
GHCi> (+1) <$> [1,2,3]
[2,3,4]
```

Applying functions inside contexts

Lists of Maybes frequently arise. For example, the “mod” operation on integers (e.g. mod 3 2 == 1) will throw an error if you pass 0 as the divisor:
GHCi> mod 3 0
*** Exception: divide by zero

We might define a safe mod:

```
safeMod :: Integral a => a-> a-> Maybe a
safeMod _ 0 = Nothing
safeMod numerator divisor = Just $ mod numerator divisor
```

```
GHCi> map (safeMod 3) [1,2,0,4]
[Just 1,Just 2,Nothing,Just 4]
```

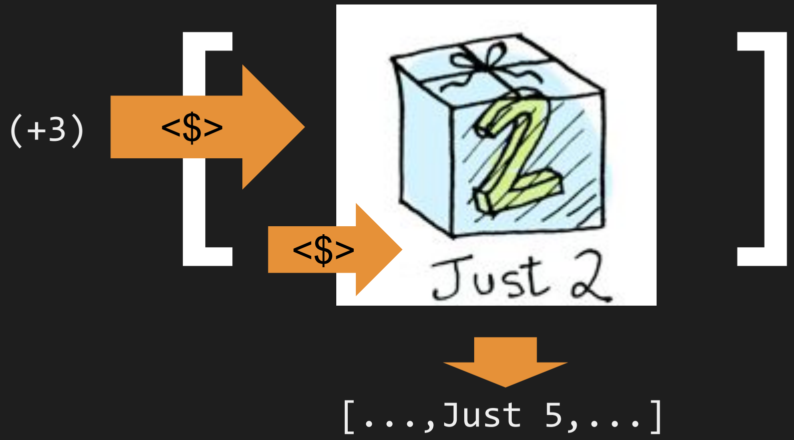
Applying functions inside contexts inside contexts

But how do we keep working with such a list of Maybes? We can map an fmap over the list:

```
GHCi> map ((+3) <$>) [Just 0,Just 2,Nothing,Just 3]
[Just 3,Just 5,Nothing,Just 6]
```

Or equivalently:

```
GHCi> ((+3) <$>) <$> [Just 0,Just 2,Nothing,Just 3]
[Just 3,Just 5,Nothing,Just 6]
```



Exercise 2

To be announced...

More instances of Functor: function!

```haskell
instance Functor ((->) r) where
    fmap = (.)
```

So function composition (.) is just mapping a function over another function!

```
ghci> ((+1).(*2)) 3
7
ghci> ((+1)<$>(*2)) 3
7
```

More instances of Functor: Either

```
> :i Either
data Either a b = Left a | Right b        -- Defined in `Data.Either'
```

```
instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Source on Hackage

```
countdown::(Int->Int)->Int -> Either Int String
countdown _ 0 = Right "Blastoff"
countdown _ 1 = Left 1
countdown dec n = countdown dec $ dec n
```

```
ghci> (++"!") <$> countdown (\i -> i - 1) 10
Left 1
ghci> (++"!") <$> countdown (\i -> i - 2) 10
Right "Blastoff!"
```

More instances of Functor: Either

```
> :i Either
data Either a b = Left a | Right b        -- Defined in `Data.Either'
```

[Source on Hackage]

```
instance Functor (Either a) where
   fmap _ (Left x) = Left x
   fmap f (Right y) = Right (f y)
```

```
countdown::(Int->Int)->Int -> Either Int String
countdown _ 0 = Right "Blastoff"
countdown _ 1 = Left 1
countdown dec n = countdown dec $ dec n
```

```
ghci> (++"!") <$> countdown ((+) (-1)) 10
Left 1
ghci> (++"!") <$> countdown ((+) (-2)) 10
Right "Blastoff!"
```

More instances of Functor: Either

```
> :i Either
data Either a b = Left a | Right b      -- Defined in `Data.Either'
```

```
instance Functor (Either a) where
   fmap _ (Left x) = Left x
   fmap f (Right y) = Right (f y)
```

[Source on Hackage]

```
countdown::(Int->Int)->Int -> Either Int String
countdown _ 0 = Right "Blastoff"
countdown _ 1 = Left 1
countdown dec n = countdown dec $ dec n
```

```
ghci> (++"!") <$> countdown pred 10
Left 1
ghci> (++"!") <$> countdown (pred.pred) 10
Right "Blastoff!"
```

More instances of Functor: Tuple

```
> :i (,)
data (,) a b = (,) a b  -- Defined in `ghc-prim-0.5.0.0:GHC.Tuple'
```

```
instance Functor ((,) a) where
   fmap f (x,y) = (x, f y)
```

[Source on Hackage]

```
ghci> (+1) <$> (1,2)
(1,3)
```

More instances of Functor: IO

```haskell
instance  Functor IO where
   fmap f x = x >>= (pure . f)
```

[Source on Hackage]

```haskell
> :i getLine
getLine :: IO String    -- Defined in `System.IO'

reverseInput::IO ()
reverseInput = do
   line <- reverse <$> getLine
   putStrLn $ "You said " ++ line ++ " backwards!"
   putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```

```
ghci> reverseInput
hello there
You said ereht olleh backwards!
Yes, you really saidereht olleh backwards!
```

First Law of Functor - law of identity

fmap id = id

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

Second Law of Functor - law of composition

```
fmap (f . g) = fmap f . fmap g
```

These two laws basically state that fmap is a combinator with no hidden extras

- It does nothing but map the function over the type

Different ways to apply functions

`g x` ⇦ apply function g to argument x

`g $ x` ⇦ apply function g to argument x

`g <$> f x` ⇦ apply function g to argument x which is inside Functor f

`f g <*> f x` ⇦ apply function g in Applicative context f to argument x inside context f

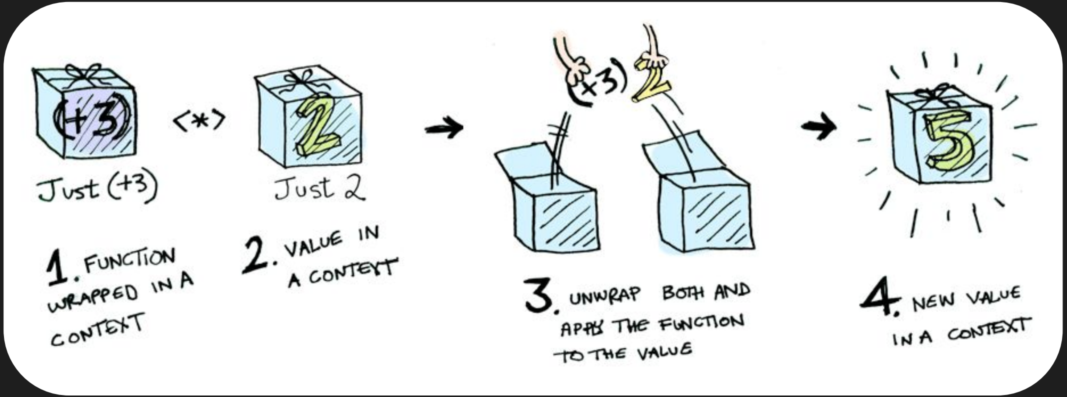
The Applicative Typeclass

```
> :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"]
```

Applicative in pictures

```
GHCi> (Just (+3)) <*> (Just 2)

Just 5
```



adit.io

fmap and applicative

```
(*) <$> [2,5,10] <*> [8,10,11]
   == [(*2),(*5),(*10)] <*> [8,10,11]

[16,20,22,40,50,55,80,100,110]

replicate <$> [1,2,3] <*> ['a','b','c']
   == [replicate 1, replicate 2, replicate 3] <*> ['a','b','c']

["a","b","c","aa","bb","cc","aaa","bbb","ccc"]

ghci> (*) <$> Just 3 <*> Just 4
Just 12

ghci> (*) <$> Nothing <*> Just 4
Nothing
```

Playing cards

```haskell
data Suit = Spade|Club|Diamond|Heart
 deriving (Eq,Ord,Enum,Bounded)

instance Show Suit where
 show Spade = "^"
 show Club = "&"
 show Diamond = "O"
 show Heart = "V"

data Rank = Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|Jack|Queen|King|Ace
 deriving (Eq,Ord,Enum,Show,Bounded)

data Card = Card Suit Rank
 deriving (Eq, Ord, Show)
```

Playing cards

```
GHCi> [Spade ..]

[^,&,O,V]

GHCi> [(minBound :: Suit) ..]

[^,&,O,V]

GHCi> [(minBound :: Rank) ..]

[Two,Three,Four,Five,Six,Seven,Eight,Nine,Ten,Jack,Queen,King,Ace]

GHCi> Card Spade Two

Card ^ Two
```

Playing cards

```haskell
sortedDeck :: [Card]
sortedDeck = Card <$> [Spade ..] <*> [Two ..]
```

```
GHCi> sortedDeck
[Card ^ Two,Card ^ Three,Card ^ Four,Card ^ Five,Card ^ Six,Card ^ Seven,Card ^
Eight,Card ^ Nine,Card ^ Ten,Card ^ Jack,Card ^ Queen,Card ^ King,Card ^ Ace,Card &
Two,Card & Three,Card & Four,Card & Five,Card & Six,Card & Seven,Card & Eight,Card &
Nine,Card & Ten,Card & Jack,Card & Queen,Card & King,Card & Ace,Card O Two,Card O
Three,Card O Four,Card O Five,Card O Six,Card O Seven,Card O Eight,Card O Nine,Card O
Ten,Card O Jack,Card O Queen,Card O King,Card O Ace,Card V Two,Card V Three,Card V
Four,Card V Five,Card V Six,Card V Seven,Card V Eight,Card V Nine,Card V Ten,Card V
Jack,Card V Queen,Card V King,Card V Ace]
```

```
GHCi> length sortedDeck
52
```

Applicative Laws

In addition to the laws inherited from Functor:

`pure f <*> x = fmap f x`	*fmap*
`pure id <*> v = v`	*Identity*
`pure f <*> pure x = pure (f x)`	*Homomorphism*
`u <*> pure y = pure ($ y) <*> u`	*Interchange* — note: `($ y) = \f -> f y` `because operator sectioning`
`pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`	*Composition*

Like the laws for Functor, these laws ensure you use <*> and pure safely and let you reason about applications where the Applicative is applicable.

It's not necessary to memorise these - just know that they exist.

More details: [WikiBook on Applicative Functors]

Conclusions

- Making your code point free can make it either cleaner and clearer - or more cryptic: use it wisely
- Functor gives you fmap (<$>)
- Applicative gives you pure and <*>
- These functions can be applied to lots of common types (or your own instances of Functor and Applicative) to capture many common coding patterns