Week 7

Ch 11: Applicative Functors

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

Overview

- functor design
- IO actions as functors
- functions as functors
- functor laws
- breaking the functor laws
- using applicative functors
- `Maybe` the applicative functor
- the applicative style
- lists (as applicative functors)
- IO (as an applicative functor)
- functions (as applicative values)
- `ZipList` Applicative Functor
- Applicative Laws
- Useful Functions for Applicatives

Interface-style Design

The Haskell programming language is:

- bigger on interface-style design
- than on classes- and subclass-hierarchy design
  as in other object-oriented programming languages.

- some value in our programs can act as many different kinds of things, described by different type classes

A thing can be categorized into many type classes, not just one hierarchy.

Functor Type Class

Recall type classes such as:

- `Eq` for describing types with values we can check for equality, and
- `Ord` for describing types with values that are orderable.

These examples of type classes demonstrate the usefulness of abstract descriptions

Recall the `Functor` type class describes:

- types with nested values
- that can have a function applied
- and maintain the parent structure.

Functionality

`Functor` type class: types that can be mapped over.

Applying functions on elements of:

- an input set (a domain)…
- …to an output set (a range)

- (there could be repetition both in the input set and in the output set)

- this may seem overkill when the input set is a singleton (like with the `Maybe` type)

- but it allows you to focus your work on the nested values

Realize that `Functors` allow you to begin to think of things such as lists, `Maybe`, binary trees, etc., as having similar possible behaviour.

— Functor Design —

Functor versus Function

The `Functor` type class has only one method that must be implemented on any instances called `fmap`, which we have already seen.

- again, its description is `fmap :: (a -> b) -> f a -> f b`

- see how the description fits within the context of `f` to the nested values `a` and `b`

- the function passed in to `fmap` is NOT `f`, but the parameter function `(a -> b)`
  - `(a -> b)` is the function applied to the nested values, where as `f` maintains itself as parent context

Functors and Type Parameters

To describe an instance of `Functor` as a type constructor, it must be of kind `* -> *`:

- give one type parameter as input, and the type constructor will evaluate to one concrete type
  - e.g. `Maybe` takes one type parameter, such as `Maybe Int`, to describe a concrete type

Then with a type constructor such as `Either` that takes two type parameters:

- we must additionally supply exactly one type parameter, `Either a`
  - cannot write `instance Functor Either where`
  - must write `instance Functor (Either a) where`

`Either a` as a Functor

To implement `fmap` with the `Either a` type constructor would then be described as:


```
fmap :: (b -> c) -> Either a b -> Either a c
```


- in the above `Either a` remains as a fixed type constructor
  - the context is always a type constructor taking exactly one parameter

— I/O Actions as Functors —

`IO` as a Functor

Notice that the `IO a` type has the one parameter `a`, where `IO` has been implemented as a Functor.

A description for how it is implemented already:


```
instance Functor IO where
    fmap g action = do
        let result <- action
        return (g result)
```


The input parameter `g` is
NOT the parent context `f` (in this case `IO`)!

*Textbook Caveat

The textbook often uses the same letter `f` for both functor and function:

- `g` is some function passed in as a parameter of `fmap`

- the context is an I/O action, suppose `IO String` (which is NOT `g`)

Note that `return` wraps the IO parent context:

- this requires an I/O action in the process, so it must be bound with `<-` assignment (unless it is the last line of the `do` block)
- this must be done within a `do` block as part of multiple I/O actions

`IO` Functor Example (1)

This has many layers of concepts, so a few examples, first without, and then with:

```
main = do
    line <- getLine
    let line' = reverse line
    putStrLn $ "You said " ++ line' ++ " backwards!"
    putStrLn $ "Yes, you said " ++ line' ++ " backwards!"
```


Then `IO` as a functor where the type parameter is `String`:

```
main = do
    line <- fmap reverse getLine
    putStrLn $ "You said " ++ line ++ " backwards!"
    putStrLn $ "Yes, you said " ++ line ++ " backwards!"
```

`IO` Functor Example (2)

See how the function `reverse` passed in to `fmap` must work with types `String`:

- input of `reverse` is String
  (the type for nested `getLine` output)

- the output of `reverse` is also a String
  (determining the type for `line`)

- but, we passed `reverse` in to `fmap`, which returns an `IO` context,
  so `fmap reverse getLine` result is of type `IO String`

- the `<-` operation removes the `IO` context
  and stores the nested `String` value in `line`

Point-free versus Nesting (1)

- if you are wanting to perform I/O action and *then* a function on the result…
- …instead consider using `fmap` and pass the function in together with the I/O action
- then the function passed in can be a composition using point-free notation, or a lambda function, etc.


```
main = do
    line <- fmap (intersperse '-' . reverse . map toUpper) getLine
    putStrLn line
```


The equivalent function passed to `fmap` written without using point-free is:

```
(\xs -> intersperse '-' (reverse (map toUpper xs)))
```

— Functions as Functors —

Functions as Functors

The syntax we have seen for descriptions of functions is `a -> b`:

- notice it is written similar to a binary operator
- consider it written as `(->) a b`
- if we omit the last parameter, we get `(->) a`
  - this describes the syntax for a constructor of a function that takes one parameter
  - this is used to implement an instance of `Functor`

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

*Equivalent to Composition

The textbook just demonstrates how the composition operator `.` is equivalent to `fmap` when implementing a function as a `functor`.

- function composition suffers from the notation of mathematics where they are applied in backward order from their evaluation

- piping would be much easier to read as code, similar to a `do` block

Signature of `fmap`

The above is just function composition, which could be written more concisely as:

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


The implementation exists in `Control.Monad.Instances` module. Consider some re-writing of types:


```
fmap :: (a -> b) -> f a -> f b
fmap :: (a -> b) -> (->) r a -> (->) r b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```


- then see in this instance, `fmap` takes two functions as parameters
- the composition would be, mathematically `r -> a` then `a -> b`, so that altogether the result is `r -> b`

Demonstrations of Functions as Functors

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a

ghci> fmap (*3) (+100) 1
303

ghci> (*3) `fmap` (+100) $ 1
303

ghci> (*3) . (+100) $ 1
303

ghci> fmap (show . (*3)) (+100) 1
"303"
```


Note that the order of operations will first compose the functions and then apply the one resulting function.

*A Few More Examples

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]

ghci> fmap (replicate 3) (Right "blah")
Right ["blah","blah","blah"]

ghci> fmap (replicate 3) Nothing
Nothing

ghci> fmap (replicate 3) (Left "foo")
Left "foo"
```

— Functor Laws —

The Functor Laws

There are properties and behaviours of functors we call laws:

- they are *not* checked by Haskell automatically

- however, all the library functors implement them

- we must check these laws when implementing our own functors

1. the function `id` mapped over a functor must return the same functor value
2. `fmap` distributes across composition

Details of Functor Laws

1. the function `id` mapped over a functor must return the same functor value
   - i.e.: `fmap id = id`
     - e.g.: `fmap id (Just 3)` vs `id (Just 3)`

2. `fmap` distributes across composition

   - i.e.: `fmap (f . g) = (fmap f . fmap g)`
   - i.e.: `fmap (f . g) x = fmap f (fmap g x)`

   - ultimately, nothing about applying the functor as a type changes the behaviour of other functions applied over it
   - for example, there is nothing about lists that changes how a function will operate on its elements

— Breaking the Functor Laws —

Breaking Functor Laws

We will consider an example that breaks the laws, just to see what happens.

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```


- the `C` stands for counter

- the first field in the `CJust` constructor will always have type `Int`

  - this is similar to `Maybe a`, but will just be used as a counter

- the second field is of type `a` and will depend on the concrete type we choose later for `CMaybe a`

Using CMaybe

```
ghci> CNothing
Cnothing

ghci> CJust 0 "haha"
CJust 0 "haha"
```



```
ghci> :t CNothing
CNothing :: CMaybe a
```



```
ghci> :t CJust 0 "haha"
CJust 0 "haha" :: CMaybe String

ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

CMaybe an Instance of Functor

Now we will implement `CMaybe a` as a functor.

- so `fmap`
  - applies the function passed in to the second field of `CJust`
  - and increments the first field,
- and otherwise, a `CNothing` is left alone:


```
instance Functor CMaybe where
    fmap g CNothing = CNothing
    fmap g (CJust counter x) = CJust (counter + 1) (g x)
```

- (in ghci, no need for `let` with `instance` and can be multiline)

First Functor Law Broken

See how we can apply fmap now:

```
ghci> fmap (++ "ha") (CJust 0 "ho")
CJust 1 "hoha"

ghci> fmap (++ "he") (fmap (++ "ha") (CJust 0 "ho"))
CJust 2 "hohahe"

ghci> fmap (++ "blah") CNothing
CNothing
```


But the first law does not hold:

```
ghci> fmap id (CJust 0 "haha")
CJust 1 "haha"

ghci> id (CJust 0 "haha")
CJust 0 "haha"
```

Second Functor Law Broken

And neither does the second law hold:


```
ghci> fmap (++ "he") . fmap (++ "ha") $ (CJust 0 "ho")
CJust 2 "hohahe"
ghci> fmap ((++ "he") . (++ "ha")) $ (CJust 0 "ho")
CJust 1 "hohahe"
```

Code Independent of Context

The functor laws are necessary to ensure they do not obfuscate the use of our other functions we may write.

- i.e.: we should not get confused about how a function will be applied to nested elements depending on context

- this makes our code easier to read

- in turn, many of the other "-ities" become supported, such as extensibility, maintainability, etc.

— Using Applicative Functors —

Functions in Context

Functors can be taken to a more general context by partially applying the function passed in to `fmap`:


```
fmap (*) Just 3
```


The above results in `Just ((*) 3)` or `Just (3 *)`.

- the nested value becomes a partially applied function

Nested Partially Applied Functions (1)

```
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])

ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)

ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]

ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: Fractional a => [a -> a -> a]
```

Nested Partially Applied Functions (2)

In the expressions involving `compare` function

- the type for compare is `compare :: (Ord a) => a -> a -> Ordering`

- `fmap compare "A LIST OF CHARS"`

  - the first `a` in the type description for `compare` is inferred to be `Char`

  - then the second `a` must be type `Char`
  - the combined partially-applied compare function and the functor together generate a list of functions of type `Char -> Ordering`

Lists of Multiparameter Functions

- you may wonder how to work with the last expression
  - assign the expression result to a variable: `functions` (see below)
    - each function is missing two parameters: `y` and `z`
    - these correspond to the `[ a -> a ->` part of the type description
  - apply the element functions `fmap (\f -> f 1 2) $ functions`
    - this adds `0.5 = y / z` to each of the already supplied values of `x` in the original list [3,4,5,6]


```
functions = (fmap (\x y z -> x + y / z) [3,4,5,6])
ghci> fmap (\f -> f 1 2) functions
```

— `Maybe` the Applicative Functor —

Applicative Functors (1)

We have seen how to use functions on the nested elements of functors.

- "functor value" just means some context with nested elements

Applicative functors go one step more abstract and allow us to define operations between functor values.

Consider the following situation:

- we have a functor full of nested partially applied functions
- we have another functor full of nested elements
- we want the corresponding nested functions and nested elements to be calculated together

Applicative Functors (2)

Consider such an operation:

```
ghci> Just (+3) <*> Just 9
Just 12
```

We need the `Applicative` type class:

- we must then implement the `pure` function, and
- we must implement the `<*>` function

The `Applicative` type class
(remember, `f` is likely NOT a function!):

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

Maybe as an Applicative Functor

A function named with **all** special characters is automatically a binary operator.

Implementation for the `Maybe` type:


```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just g) <*> something = fmap g something
```

- `pure = Just` is equivalent to `pure x = Just x`

Implementation of <*>

```
(Just g) <*> something = fmap g something
```

- the last line may be difficult to imagine what is happening, but recall the example we are working toward

  - we want to get the function `g` out of the first functor `(Just g)`

  - apply the function `g` to the second functor
  - (`something` contains elements that can have `g` applied to them)

  - by implementation, we are forced to have the two functors in exactly this order with `<*>`
  - we cannot transpose the order for nested function and something

pure

These implementations are already part of Haskell, so give them a try:

```haskell
Just (+3) <*> Just 9
pure (+3) <*> Just 9
Just (++ "haha") <*> Nothing
Nothing <*> Just "woot"
```


- there are many kinds of applicative functors
- so, there are many kinds of results for `pure`
- `pure (+3)` takes advantage of Haskell's inference
  - what functor type will match with `Just 9`
    in order to match on the left an expression `Just (+3)`

— The Applicative Style —

Using <*>

The order of operations using `<*>` is from left-to-right

- when writing larger expressions of more than two functor values
- this is called left-associative
- then partially applied functions leftmost need *more* parameters

For example:


```
pure (+) <*> Just 3 <*> Just 5
```


- notice that the above expression is similar in syntax as `(+) 3 5`
- the given expression is equivalent to
  - `(pure (+) <*> Just 3) <*> Just 5`
  - …and result of `(pure (+) <*> Just 3)` is `Just (3+)`

Applicative Advantage

The advantage of applicative types:

- we can use functions on nested values within functors without having to worry about what those functors are

- `pure g <*> x <*> y <*> …`

  - the `g` can have as many parameters as desired
  - each successive evaluation of `<*>` applies one more parameter

- `pure g <*> x` is equivalent to `fmap g x`
  - (this is one of the applicative laws we will discuss later)

`fmap` Synonym `<$>`

Instead of writing `` `pure g <*> x <*> ...` `` we could just write `` `fmap g x <*> ...` ``

- however, there is an infix version of `` `fmap` `` to make expressions even more concise with `` `<$>` ``


```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
g <$> x = fmap g x
```


- so, we could instead write `` `g <$> x <*> y <*> ...` ``
- note that `` `g` `` is a function (a variable one)

Type Descriptions with <$> and <*>

Another example:


```
(++) <$> Just "Doctor Strange " <*> Just "and the Multiverse of Madness"
```


- recall the type for concatenation `(++) :: [a] -> [a] -> [a]`

- the `<$>` operation results in a partially applied function of type
  `Just ("Doctor Strange "++) :: Maybe ([Char] -> [Char])`

- can you work out the type of the last functor in the example?

Example of <$>

(Simranjit Singh)

```haskell
-- Presentation 3
-- Simranjit Singh

-- playing with random, IO, and <$>
import System.Random

getList :: String -> [Int]
getList xs = foldr (\n acc -> (read n :: Int) : acc) [] list
    where list = words xs
```

Example of <$>

(Simranjit Singh)

```haskell
genNewList :: [Int] -> StdGen -> IO ()
genNewList xs gen =
    do
        let
            (randNumber, newGen) =
                randomR (1,3) gen :: (Int, StdGen)
            secretCalc x
                | x == 1  = print $  (+75) <$> xs
                | x == 2  = print $ (*5) <$> xs
                | x == 3  = print $ (`div` 3) <$> xs
                | True    =
                    putStrLn "Something went terribly wrong"
        secretCalc randNumber
```

Example of <$>

(Simranjit Singh)

```
main = do
    gen <- getStdGen
    putStrLn "Enter a list of numbers (no commas or brackets):"
    input <- getLine
    let list = getList input
    putStr "The list you entered was: "
    print(list) -- == putStrLn $ show list
    putStrLn "I have now done a secret operation on your list"
    putStr "Your new list is: "
    genNewList list gen
```

— Lists (as applicative functors) —

Lists as Applicative Functors

We have the implementation of lists as applicative functors:

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [g x | g <- fs, x <- xs]
```


- notice that `pure` creates a singleton list *always*

- also notice that the `g` and `x` in the above create ***ALL*** possible combinations of functions from `fs` and values from `xs`

  - the type of `<*>` restricted to lists:
    `(<*>) :: [a -> b] -> [a] -> [b]`
  - since there are potentially many functions,
    the implementation needs list comprehensions
    (to facilitate all possible combinations)

Practice with Lists and <>

Lists with `<*>` will remind you when you apply it that you will get every combination of result possible.

```
[(*0),(+100),(^2)] <*> [1,2,3]
```

The next example shows step-by-step evaluation of multiple operations:


```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[(+1),(+2),(*1),(*2)] <*> [3,4]
[4,5,5,6,3,4,6,8]
```


One more example:

```
ghci> (++) <$> ["ha", "he", "hm"] <*> ["?", "!", "."]
["ha?","ha!","ha.","he?","he!","he.","hm?","hm!","hm."]
```

Nondeterministic Computation

We can think of lists as nondeterministic computations.

- a value such as `"what"` or `100` is deterministic
- a value such as `[1,2,3]` may decide among its three elements
- the `<*>` presents us with all possible outcomes on lists

Notice how we can use `<*>` to replace list comprehensions:


```
ghci> [ x*y | x <- [1,2,3], y <- [4,5,6] ]
[4,5,6,8,10,12,12,15,18]

ghci> (*) <$> [1,2,3] <*> [4,5,6]
[4,5,6,8,10,12,12,15,18]
```

`filter` and `<$>`

Combining with `filter` is especially useful:


```
ghci> filter (> 10) $ (*) <$> [1,2,3] <*> [4,5,6]
[12,12,15,18]
```

— IO (as an applicative functor) —

IO as Applicative Functor

We look at the implementation of `IO` as an applicative functor:

```
instance Applicative IO where
    pure = return
    s <*> t = do
        g <- s
        x <- t
        return (g x)
```


- `pure = return` works as an IO action ignoring the value passed in
- `<*>` for `IO` has description `<*> :: IO (a -> b) -> IO a -> IO b`
  - implementation of `<*>` must then remove the `IO` context for both s and t parameter values
  - `do` is needed to glue together multiple I/O actions into one
  - `return` will place the result `(g x)` back into an `IO` context

getLine and <*>

```
:set +m

do
    x <- (++) <$> getLine <*> getLine
    putStrLn $ "two lines concatenated: " ++ x
```


- the nested result of a `getLine` I/O action is a `String`

  - the order of the performed I/O action of each `getLine` determines the order of the concatenated values

- the result of `(++) <$> getLine <*> getLine` is of type `IO b` where `b` in this case is `String`
  - this is altogether one I/O action and we can assign the yield to `x` as a `String` value

— Functions (as applicative values) —

Functions as Applicative Functors

The implementation for functions as applicatives:

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```


- the `pure` implementation creates a value of minimal context for the functor type
    - in this case, the result is a function that ignores its parameter and always evaluates to `x`
    - the type for `pure` is `pure :: a -> (r -> a)`

pure Behaviour

The default behaviour of `pure` is kind of strange here:

```
(pure 3) "blah"
```


- the result of the above is actually `3`
  - the parentheses `(pure 3)` create a function that always returns `3` which requires one parameter passed in
  - it is a partially applied function that will take "blah" as its one parameter
  - the result is `3`, as expected
- equivalently, because functions are left-associative, there is no need for parentheses: `pure 3 "blah"`

Function Composition

We look at a few examples:

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: Num b => b -> b
```



```
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```


- we want to work with two functions `(+3)` and `(*100)`

    - both functions take one parameter, and in the above we pass in `5`
        - `(5+3) = 8`
        - `(5*100) = 500`
    - add the results together (as if we had not passed in `5` yet)
    - the result of the entire function is `(5+3) + (5*100) = 508`

* <$> and <*> Operations First

Here is another wild one to read:

```
ghci> (\x y z -> [x, y, z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```


- the leftmost operand is a *function* that takes three parameters

- first, this function is placed within the context of the list elements, which are each then *one* parameter functions

- each next operand in the above expression fills in one parameter
  - in the order `x`, `y`, `z`
- this results in a function equivalent to `(\x -> [(x+3),(x*2),(x/2)])`
  - (arguably, the original expression is much more difficult to read)

— `ZipList` Applicative Functor —

Corresponding Elements

We will often want corresponding elements between lists to operate together, rather than combinations.

`ZipList` gives an alternative implementation of applicative functor (found in the module `Control.Applicative`):


```
instance Applicative ZipList where
    pure x = ZipList (repeat x)
    fs <*> xs = ZipList (zipWith (\f x -> f x) fs xs)
```

ZipList

- we can see that `zipWith` applies each function element of `fs` to its corresponding element of `xs`

- for `pure = ZipList (repeat x)` creates an infinite list, whereas `replicate` takes two parameters to create an finite list

  - we want "minimal context" as an infinite list, because `zipWith` will stop on the shorter list (which could be any length, even infinite...)

  - for example:
    `take 2 $ zipWith (\x y -> x + y) [1,2..] [3,4..]`

getZipList

The `ZipList` type is not implemented as an instance of `Show`, so we must use the `getZipList` function to return results as a list:

```
import Control.Applicative
```



```
ghci> getZipList $ (+) <$> ZipList [1,2,3]
        <*> ZipList [100,100,100]

[101,102,103]
```



```
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3]
        <*> ZipList [5,3,1,2]

[5,3,3,4]
```

```
ghci> getZipList $ (,,) <$> ZipList "dog"
        <*> ZipList "cat" <*> ZipList "rat"

[('d','c','r'),('o','a','a'),('g','t','t')]
```

Multiple Lists and Zip Functions

`(,,)` is a constructor for a triple,
equivalent to `(\x y z -> (x, y, z))`.

There are functions for zipping three lists, four lists, etc.:

- `zipWith3`
- `zipWith4`
- ...
- `zipWith7`

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,6] [7,8,9]
[12,15,18]
```

Multi-Parameter with `ZipList`

Equivalently:

```
:{
getZipList $ (\x y z -> x + y + z)
    <$> ZipList [1,2,3]
    <*> ZipList [4,5,6]
    <*> ZipList [7,8,9]
:}
```


It is a bit more writing, because of the redundant `ZipList` constructors.

ZipList Example

(David Semke)

```
import Control.Applicative
-- Multiply the first number in list1 by 1,
--   the second by 2, the third by 3, ...

list1 = take 10 (repeat 1)

incrementMult :: ZipList Int -> ZipList Int
incrementMult (ZipList xs) =
    let mults = ZipList $ fmap (*) $ take (length xs) [1, 2..]
    in mults <*> ZipList xs

listzip = ZipList list1

result = getZipList $ incrementMult listzip

ghci> getZipList $ incrementMult $ ZipList result
```

— Applicative Laws —

Applicative Laws (1)

1. `pure id <*> v = v`
2. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
3. `pure f <*> pure x = pure (f x)`
4. `u <*> pure y = pure ($ y) <*> u`

Applicative Laws:

Examples

```
1.  (trivial)
```



```
2.  (pure (.) <*> [(*3)] <*> [(+2)] <*> [1]) :: [Int]
```



```
3.  (pure (*3) <*> pure 4) :: [Int]
```



```
4.  pure ($ 4) <*> [(*3)]
```

Applicative Laws (2)

- `(.)` is the operation of composition
  - so for Law 2, note that it only makes sense when both `u` and `v` have functions nested inside

- the `($ y)` is any function you would like to apply to element `y` taken as another parameter (a function)

  - but we know `u` must have functions nested inside as elements in its context that should be applied (from the LHS of the equation of law (4))
  - so, the RHS will be fine to apply these functions with `($ y)`

— Useful Functions for Applicatives —

liftA2

The `Control.Applicative` module has a function called `liftA2`

- applies the applicative operations we have practiced so far
- the implementation for `liftA2` is as follows:


```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 g x y = g <$> x <*> y
```

Using `liftA2`

The name of the function `liftA2` is fitting:

- consider type description as `(a -> b -> c) -> (f a -> f b -> f c)`
- we see `liftA2` can promote a regular binary function and make that function operate within the context of two applicatives

For example:


```
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```



```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
```

*sequenceA

Now we would like to apply a similar operation to the above demonstration, but repeatedly:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```


- A base case is an empty list in default context as `pure []`
- `x` is a functor we can prefix as the first element within the context of the functor that contains a list `(sequenceA xs)`
  - `xs` is a list of functors


```
sequenceA [Just 1, Just 2]
(:) <$> Just 1 <*> sequenceA [Just 2]
(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])
(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])
(:) <$> Just 1 <*> Just [2]
Just [1,2]
```

*Equivalent to sequenceA

We can also implement the same `sequenceA` function with `foldr` instead:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```


- `(liftA2 (:))` is the function acting on accumulator and next element both processed inside the context of the functor `f`
    - it may help you to imagine the prefix `:` acting on the accumulator regardless of the functor context
- result is a nested list within the context of the passed in functor

*Using sequenceA

Take a moment to convince yourself of the following examples that the result matches the passed in context:

```
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
```



```
ghci> sequenceA [[1,2,3],[4,5,6]]
```



```
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```



```
ghci> sequenceA [[1,2,3],[4,5,6],[]]
[]
```

- in short, `sequenceA [(+3),(+2),(+1)]` has resulting context as a function that takes one parameter

*Compare with `sequenceA`


```
(\xs x -> fmap ($ x) xs) [(*2),(*5)] 3
```

[] Similar to `Nothing`

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]

ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
```


- applying `sequenceA` to a list of `Maybe` values results in a list nested inside as a `Maybe` value

- useful if we are interested in a list of `Maybe` values where we only care about the result when ***none*** of the input elements are `Nothing`

*Multiple Predicates

Suppose we have a number that we would like to check if it satisfies a list of predicates:


```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
```



```
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```


- recall that `and` returns `True` only when all of the elements in a list are `True`

*sequenceA Refactor

We can achieve the same result as above with the `sequenceA` function:

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
```



```
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```


- `sequenceA [(>4),(<10),odd]` generates a function that takes one parameter `7` and feeds it to the predicates
- it results in the list of `Bool` values
- the type for `[(>4),(<10),odd]` is `(Num a) => [a -> Bool]`
- the type of `sequenceA [(>4),(<10),odd]` is `(Num a) => a -> [Bool]`

*Mixed Function Types

Note that lists must have the same type for each element.

- we cannot make a list such as `[ord, (+3)]`
  - `ord` takes a character and returns a number
  - `(+3)` takes a number and returns a number

The last demonstration of `sequenceA` we consider is with the list of lists:


```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]

ghci> [[x, y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

*Using sequenceA with IO

One last useful application of `sequenceA` is on the context of `IO`:


```
ghci> sequenceA [getLine, getLine, getLine]
what
doing
?
["what","doing","?"]
```

Finally

Altogether, we have used `<$>` and `<*>` for:

- combining yields of I/O actions

- nondeterministic computations

- sets of computations that might have failed

Thank You!

Questions?