Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #12

Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

Important: Please remember to register your attendance in Moodle!

Today's lecture is about Functors, Applicative functors and Monads.

These may be difficult concepts to grasp. We strongly suggest complementing your study with the recommended bibliography.

- Programming in Haskell, 2nd Edition, by Graham Hutton. September 2016, Cambridge University Press, ISBN-13 978-1316626221. Available via a-z.lu

Chapter 12

- Learn You a Haskell for Great Good!, by Miran Lipovaca. April 2011, No Starch Press, ISBN-13 978-1593272838. Free online

Chapters 11 and 12

Functors

- Functors abstract the idea of mapping a function over each element of a structure, while preserving the structure itself.
- An notable example of Functor we've seen over and over is `[]`. We can map functions over lists with `map`.

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

- The Functor type class is defined in the Haskell standard library as follows:

In [1]:
```
:opt no-pager
```

```
:opt no-lint
```

In [2]:
```
:info Functor
```

```
type Functor :: (* -> *) -> Constraint
class Functor f where
fmap :: (a -> b) -> f a -> f b
(<$) :: a -> f b -> f a
{-# MINIMAL fmap #-}
-- Defined in ‘GHC.Base’
instance Functor ((,) a) -- Defined in ‘GHC.Base’
instance Functor ((,,) a b) -- Defined in ‘GHC.Base’
instance Functor ((,,,) a b c) -- Defined in ‘GHC.Base’
instance Functor ((,,,,) a b c d) -- Defined in ‘GHC.Base’
instance Functor ((,,,,,) a b c d e) -- Defined in ‘GHC.Base’
instance Functor ((,,,,,,) a b c d e f) -- Defined in ‘GHC.Base’
instance Functor ((->) r) -- Defined in ‘GHC.Base’
instance Functor IO -- Defined in ‘GHC.Base’
instance Functor [] -- Defined in ‘GHC.Base’
instance Functor Maybe -- Defined in ‘GHC.Base’
instance Functor Solo -- Defined in ‘GHC.Base’
instance Functor (Either a) -- Defined in ‘Data.Either’
```

Instances of Functor

- Some instances of Functor from Prelude.

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

instance Functor [] where
    fmap = map

instance Functor IO where
    -- fmap :: (a -> b) -> IO a -> IO b
    fmap g mx = do { x <- mx; return (g x) }


instance Functor (Either a) where
    -- fmap :: (b -> c) -> Either a b -> Either a c
    fmap _ (Left error) = Left error
    fmap g (Right y) = Right (g y)
```

- Custom instances of Functor.

```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
    fmap g (Leaf x) = Leaf (g x)
    fmap g (Node left right) = Node (fmap g left) (fmap g right)
```

Exercise #1 🧐

- Write a program that reads a `String` from `stdin` and prints to `stdout` the `String` in capital letters. Use `fmap`!

In [3]:
```haskell
import Data.Char (toUpper)

:t toUpper
toUpper 'a'

main :: IO ()
main = do
    putStrLn "Write a string to caps"
    str <- fmap (map toUpper) getLine
    putStrLn str

getLine :: IO String
fmap :: (String -> String) -> IO String -> IO String
toUpper :: Char -> Char
```

```
toUpper :: Char -> Char
```

```
'A'
```

Functor laws ⚖️

1. **Identity law**

   ```
   fmap id mx = mx
   ```

2. **Composition law**

   ```
   fmap (g . h) mx = fmap g (fmap h mx)
   ```

These laws have several important implications:

- **Preserving Structure**: The identity law ensures that applying fmap doesn't alter the underlying structure of the functor. It should only modify the values inside while maintaining the same shape.
- **Composability**: The composition law guarantees that fmap plays well with function composition. It allows chaining multiple transformations on a functor using fmap and expect a consistent behavior.
- **Reasoning**: The functor laws enable equational reasoning, which means we can replace expressions that are equal according to the laws and obtain equivalent results. This ability to reason about code algebraically facilitates understanding, correctness verification, and code optimization.
- **Interoperability**: By adhering to the functor laws, different implementations of functors become interchangeable. We can write generic code that relies on the functor laws, and it will work correctly with any lawful Functor instance.

Example of an instance of a Functor typeclass, but not following Functor laws:

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

instance Functor CMaybe where
    --fmap :: (a -> b) -> f a -> f b
        fmap g CNothing = CNothing
        fmap g (CJust n x) = CJust (n+1) (g x)
```

- `CMaybe` does not follow functor laws:

1. `fmap id (CJust 0 'a') = CJust (0+1) (id 'a') = CJust 1 'a' =/= Cjust 0 'a'`
2. A. `fmap ((+1) . (*2)) (CJust 0 1) = CJust (0+1) (((+1) . (*2)) 1) = CJust 1 3`
   B. `fmap (+1) (fmap (*2) (CJust 0 1)) = fmap (+1) (CJust 1 2) = CJust 2 3`

Functors are a generic abstractions

- Function `fmap` can be used to process elements of any structure that is functorial.
- We can use the same name for functions that are essentially the same, rather than having to come up with separate names for each instance.
- We can define generic functions that can be used with any functor.

In [6]:
```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
    fmap g (Leaf x) = Leaf (g x)
    fmap g (Node left right) = Node (fmap g left) (fmap g right)

-- examples
inc :: Functor f => f Int -> f Int
inc = fmap (+1)

inc (Just 2)
inc [1,2,3,10]
inc (Node (Leaf 1) (Leaf 2))
```

```
Just 3
[2,3,4,11]
Node (Leaf 2) (Leaf 3)
```

Infix version of `fmap`

In [13]:
```haskell
:t fmap
:t (<$>)

fmap (^2) [1,2,3]
(^2) <$> [1,2,3]
```

```
(+10) <$> (Just 27)
```

```
fmap :: forall (f :: * -> *) a b. Functor f => (a -> b) -> f a -> f b
(<$>) :: forall (f :: * -> *) a b. Functor f => (a -> b) -> f a -> f b
[1,4,9]
[1,4,9]
Just 37
```

Applicative functors

- `fmap` maps a function with a single argument over each element of a structure.

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

- How to we combine two Functors?

  ```
  fmap (+) (Just 3) (Just 5)
  ```

This does not work because types don't match!

- One solution would be to create a special version of `fmap` that accepts functions with 2 arguments.

  ```
  fmap2 :: (a -> b -> c) -> f a -> f b -> f c
  ```

fmap2 does not exist in Prelude because there's a better way...

- How to apply functions with multiple arguments?

  ```
  fmap2 :: (a -> b -> c) -> f a -> f b -> f c
  fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
  fmap4 :: (a -> b -> c -> d -> e) -> f a -> f b -> f c -> f d ->
  f e
  ...
  ```

How many arguments are needed? It's really not convenient to define all these versions of `fmap`.

- What if the function itself already has some context?

  ```
  let a = Just (3*)
  let b = Just 5
  ```

How to combine `a` and `b`?

- With normal Functors, we can only map normal functions (not *wrapped* functions) over existing functors.
- Using the idea of currying, it turns out that a version of `fmap` for functions with any desired number of arguments can be constructed in terms of two basic functions, `pure` and `<*>` (frequently named *apply*).

Meet the Applicative typeclass!

In [8]:
```
:info Applicative
```

```
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
(*>) :: f a -> f b -> f b
(<*) :: f a -> f b -> f a
{-# MINIMAL pure, ((<*>) | liftA2) #-}
-- Defined in ‘GHC.Base’
instance Monoid a => Applicative ((,) a) -- Defined in ‘GHC.Base’
instance (Monoid a, Monoid b) => Applicative ((,,) a b) -- Defined in
‘GHC.Base’
instance (Monoid a, Monoid b, Monoid c) => Applicative ((,,,) a b c) --
Defined in ‘GHC.Base’
instance Applicative ((->) r) -- Defined in ‘GHC.Base’
instance Applicative IO -- Defined in ‘GHC.Base’
instance Applicative [] -- Defined in ‘GHC.Base’
instance Applicative Maybe -- Defined in ‘GHC.Base’
instance Applicative Solo -- Defined in ‘GHC.Base’
instance Applicative (Either e) -- Defined in ‘Data.Either’
instance Applicative Q -- Defined in ‘Language.Haskell.TH.Syntax’
```

- A typical use of `pure` and `<*>` has the following form:

  `pure g <*> x1 <*> x2 <*> ... <*> xn`

In [14]:
```
pure (+) <*> (Just 3) <*> (Just 7)

(+) <$> (Just 3) <*> (Just 7)

pure (*2) <*> [1,2,3,4,5]

pure (*) <*> [1,2] <*> [3,4]

pure (*) <*> [1,2]
```

Use <$>

Found: **Why Not:**

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

Redundant bracket

Found: **Why Not:**

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

Redundant bracket

Found: **Why Not:**

```
pure (+) <*> (Just 3) <*> (Just 7)pure (+) <*> (Just 3) <*> Just 7
```

Redundant bracket

Found: **Why Not:**

```
(+) <$> (Just 3)(+) <$> Just 3
```

Redundant bracket

Found: **Why Not:**

```
(+) <$> (Just 3) <*> (Just 7)(+) <$> (Just 3) <*> Just 7
```

Use <$>

Found: **Why Not:**

```
pure (* 2) <*> [1, 2, 3, 4, 5](* 2) <$> [1, 2, 3, 4, 5]
```

Use <$>

Found: **Why Not:**

```
pure (*) <*> [1, 2]((*) <$> [1, 2])
```

Use <$>

Found: **Why Not:**

```
pure (*) <*> [1, 2](*) <$> [1, 2]
```

```
Just 10
Just 10
[2,4,6,8,10]
[3,4,6,8]
```

```
<interactive>:1:1: error: [GHC-39999]
    • No instance for ‘Show (Integer -> Integer)’ arising from a use of ‘print’
        (maybe you haven't applied a function to enough arguments?)
    • In a stmt of an interactive GHCi command: print it
```

List comprehension in applicative style

- How should we understand these examples?
  - View type `[a]` as a generalisation of `Maybe a` that permits multiple results in the case of success.
  - The empty list as representing failure.
  - A non-empty list as representing all the possible ways in which a result may succeed.

```
prods :: [Int] -> [Int] -> [Int]
prods xs ys = [x*y | x <- xs, y <- ys]
```

Function `prod` can be defined in *applicative style* as:

```
prods' :: [Int] -> [Int] -> [Int]
prods' xs ys = (*) <$> xs <*> ys
```

Instances of Applicative

- `Maybe`:

```
instance Applicative Maybe where
    pure x = Just x

    Nothing <*> _ = Nothing
    Just g <*> mx = fmap g mx
```

- `[]`:

```
intance Applicative [] where
    pure x = [x]

    gs <*> xs = [g x | g <- gs, x <- xs]
```

- `Either a`:

```
instance Applicative (Either a) where
    pure x = Right x

    Left e <*> _ = Left e
    Right g <*> x = fmap g x
```

- `IO`:

```
instance Applicative IO where
    -- pure :: a -> IO a
    pure = return
    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    mg <*> mx = do g <- mg
                   x <- mx
                   return (g x)
```

Applicative functor laws ⚖️

1. **Identity law**: The identity law says that applying the pure id morphism does nothing, exactly like with the plain id function.

   ```
   pure id <*> x = x
   ```

2. **Preserving function application**: Applying a "pure" function to a "pure" value is the same as applying the function to the value in the normal way and then using pure on the result

   ```
   pure (g x) = pure g <*> pure x
   ```

3. **Interchange law**: When an effectful function is applied to a pure argument, the order in which we evaluate the two components doesn't matter.

   ```
   x <*> pure y = pure (\g -> g y) <*> x
   ```

4. **Composition law**: modulo the types that are involved, the operator `<*>` is associative.

   ```
   x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
   ```

Monads

Recap

- **Functors** allow mapping a function over each element of a structure, while preserving the structure itself.

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

- **Applicatives** extend Functors by allowing to map functions with multiple arguments. Arguments are no longer plain arguments, but may in fact have effects such as *the possibility of failure* (`Maybe` or `Either`), *having many ways to succeed* (lists), or *performing IO actions* (`IO`).

  ```
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  ```

What else we might need?

- While functors and applicatives provide powerful abstractions for working with computational effects, there are scenarios where additional capabilities are required.
- One such scenario arises when computations depend on previous results or require sequential execution. This is where Monads come into play.

Monads by way of example

- Consider the following data type for expressions that are built up from integers and divisions.

  ```
  data Expr = Val Int | Div Expr Expr
  ```

- We can evaluate an expression with `eval`.

  ```
  eval :: Expr -> Int
  eval (Val x) = x
  eval (Div x y) = (eval x) `div` (eval y)
  ```

- However, devision by zero is undefined and therefore `eval` might fail. Recall the fail-safe version from Lecture #5:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv n m = Just (n `div` m)
```

- Can we use this version instead?

```
evalV2 :: Expr -> Maybe Int
evalV2 (Val x) = Just x
evalV2 (Div x y) = pure safeDiv <*> x <*> y
```

- No because definition is not type correct.
  - `safeDiv` has type `Int -> Int -> Maybe Int`.
  - Definition requires type `Int -> Int -> Int`, which does not provide any means to indicate failure when the second integer argument is zero.
- We don't have the right tools.

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

- Let's solve the problem with a case analysis.

```
evalV3 :: Expr -> Maybe Int
evalV3 (Val x) = Just x
evalV3 (Div x y) = case evalV3 x of
                     Nothing -> Nothing
                     Just m  -> case evalV3 y of
                                  Nothing -> Nothing
                                  Just n  -> m `safeDiv` n
```

- Looking for patterns...
- Function `evalV3` essentially does the following:
  1. Evaluate `x` and check if it fails;
  2. Evaluate `y` and check it it fails;
  3. If both evaluations succeed, compute `safeDiv`.
- Our `evalV3` is essentially doing the same thing twice.
- How can we abstract this computation?

- Let's look ate typeclass `Monad`

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
               Nothing -> Nothing
               Just x -> f x
```

- Let's use `>>=`.

```
evalV4 :: Expr -> Maybe Int
evalV4 (Val x) = return x
evalV4 (Div x y) = evalV4 x >>= (\m ->
                   evalV4 y >>= (\n -> m `safeDiv` n))
```

- Remember the `do` notation from Lecture #6? We can use it with any Monad, not just `IO`. Let's improve readability.

```
evalV5 :: Expr -> Maybe Int
evalV5 (Val x) = return x
evalV5 (Div x y) = do m <- evalV5 x
                      n <- evalV5 y
                      safeDiv m n
```

Generalizing the pattern

- Generalizing from the above example, a typical expression that is built using the `>>=` (pronounced *bind*) operator has the following structure:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
.
.
.
mn >>= \xn ->
f x1 x2 ... xn
```

- We evaluate each of the expressions `m1 ... mn` in turn, and then combine their result values `x1 ... xn` by applying the function `f`. The definition of the `>>=` operator ensures that such an expression only succeeds if every component `mi` in the sequence succeeds.

Monads in Haskell

- A Monad is a type class with `return` (no longer needed) and `>>=`.
- We've rediscovered the Maybe Monad.
- We learned how to sequence operations that may fail
- This generalization works with other effects, such as `Maybe`, `Either`, `IO`, `State`, etc.
- Monads allow to do impure things in a pure functional programming language, like Haskell.
- The use of effects is explicit in types.
- Effect polymorphism: we can write generic functions that work with effects of any kind.

Monad laws ⚖️

1. **Left identity**: Applying the monadic `return` function to a value and then using a monadic operation `>>=` (pronounced *bind*) should be equivalent to simply applying the monadic operation directly to the value. This law ensures that the return function does not introduce any additional effects or changes to the value.

   ```
   return x >>= f = f x
   ```

2. **Right identity**: Using the bind operation `>>=` to combine a monadic value with `return` should be equivalent to the original monadic value. This law ensures that applying return after a monadic operation has no effect on the computation.

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

3. **Associativity**: Chaining together multiple monadic operations with bind should produce the same result, regardless of how the operations are grouped. This law ensures that the order of applying monadic operations does not affect the final result.

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

Exercise #2 🎯

- Your task is to guess a number between 1 and 100 using only 7 tests.
- You are given a function `greaterThan :: Monad m => Int -> m Bool`

https://www.codewars.com/kata/5c607bb95a40bc7b4b9ecc27

In [ ]:
```haskell
guess :: Monad m => (Int -> m Bool) -> m Int
guess gt = guessPlus gt 1 100
  where
    guessPlus :: Monad m => (Int -> m Bool) -> Int -> Int -> m Int
    guessPlus gt minValue maxValue
        | minValue == maxValue = return minValue
        | otherwise            = let midValue = (maxValue - minValue) `div` 2 +
                                 in  do b <- gt midValue
                                        if b then guessPlus gt (midValue+1) max
                                        else      guessPlus gt minValue midValu
```