Functional programming from its fundamentals

Whoami

I am

- **Software engineer at OpenValue**
- **Co-organizer of FP Ams**
- **Working at ING Bank**

The goal of today

Content

Fundamentals of functional programming

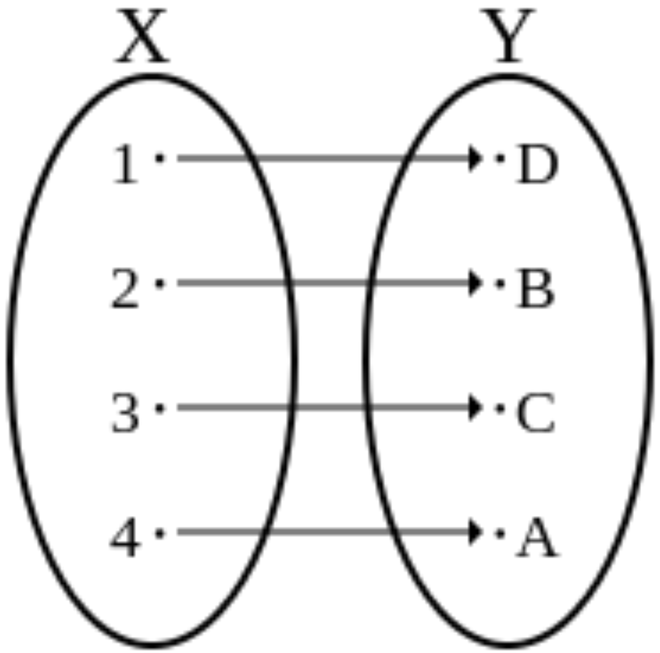
Why do we need functional programming

Common constructs from the fundamentals

What is functional programming

Programming with mathematical functions

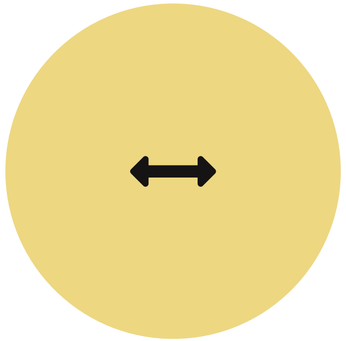
A function is a mapping between a domain and a codomain. The only effect of a function is computing the result



Properties of a function

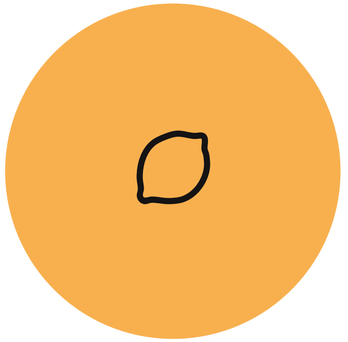
Totality

A function must yield a value for every possible input



Determinism

A function must yield the same result for the same input



Purity

A function’s only effect must be the computation of its return value

Referential transparency

Expressions can be replaced with their values without changing the meaning of the program

X

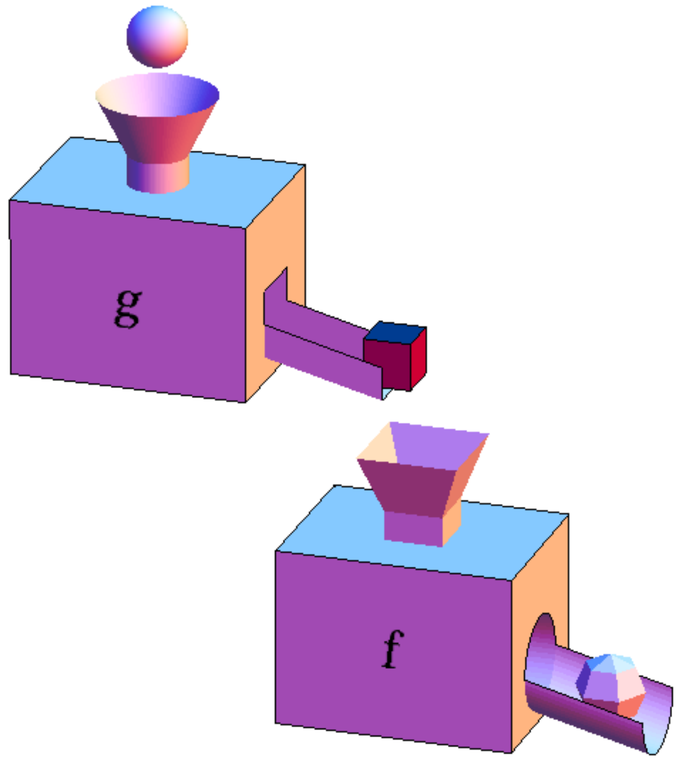
```
Integer x = 1

new Pair<Integer, Integer>(x, x)
```

```
new Pair<Integer, Integer>(1, 1)
```

```
public Integer random() { … }

Integer x = random();

new Pair<Integer, Integer>(x, x)
```

```
public Integer random() { … }

new Pair<Integer, Integer>(random(), random())
```

Function composition



Creates a new function from two other functions where the **second** function will be called with the result of evaluating the first function

Function composition

```
public static Integer length(String s) {
    return s.length();
}

public static Integer plus2(Integer x) {
    return x + 2;
}

public static Integer lengthPlus2(String s) {
    return plus2(length(s));
}
```

Function composition

```
public static <IN, MID, OUT> Function<IN, OUT> compose(Function<IN, MID> fst, Function<MID, OUT> snd) {
    return (x -> snd.apply(fst.apply(x)));
}
```

X

Function composition

```
Main.compose(Main::length, Main::plus2);
```

X


Disclaimer

Function composition

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

(.) g f = \a -> g (f a)
```

Function composition

```
length :: String -> Int

plus2 :: Int -> Int

lengthPlus2 :: String -> Int -- lengthPlus2 "Hello Crowd!" -- 14
lengthPlus2 = plus2 . length
```

Function composition

```
String -> Int
          Int -> Int
```

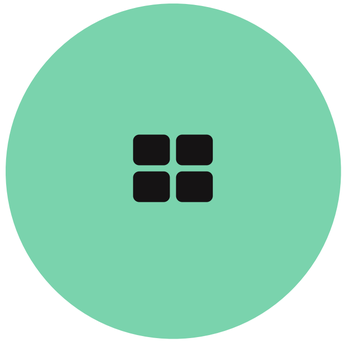
Recap

Referential transparency

Function composition

Why do we need functional programming

Why do we need functional programming



Concurrency & Parallelism

Reducing cognitive load

Easier to reason about

Example 1

```
public class Class1 {

    public String head(List<String> list) {
        if (list.size() <= 0) {
            return null;
        }

        return list.get(0);
    }
}
```

X

Example 1

```
public class Class2 {

    public Integer program(List<String> list) {

        // can produce null pointer if list is empty
        return head(list).length();
    }
}
```

X

Example 2

```
public class Class1 {

    public Optional<String> head(List<String> list) {
        if (list.size() <= 0) {
            return Optional.empty();
        }

        return Optional.of(list.get(0));
    }
}
```

X

Example 2

```
public class Class2 {

    public Optional<Integer> program(List<String> list) {
        if (head(list).isPresent()) {
          return Optional.of(head(list).get().length());
        } else {
          return Optional.empty();
        }
    }
}
```

X

Common constructs from the fundamentals

Type classes

A tool to perform ad-hoc polymorphism in a functional programming language

X

Code duplication

```
-- sortString [“ba”, “ab”, “cd”]  == [“ab”, “ba”, “cd”]
sortString :: [String] -> [String]
sortString [] = []
sortString (p:xs) =
    (sortString lesser) ++ [p] ++ (sortString greater)
        where
            lesser  = filter (< p) xs
            greater = filter (>= p) xs
```

Code duplication

```haskell
-- sortInteger [1, 3, 2] == [1, 2, 3]
sortInteger :: [Integer] -> [Integer]
sortInteger [] = []
sortInteger (p:xs) =
    (sortInteger lesser) ++ [p] ++ (sortInteger greater)
        where
            lesser  = filter (< p) xs
            greater = filter (>= p) xs
```

Ord typeclass

```
class Ord a where
  (<)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  (>)  :: a -> a -> Bool
  (>=) :: a -> a -> Bool
```

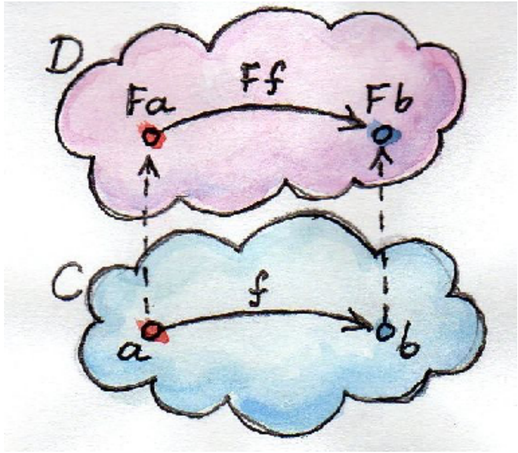
Ord instance for Integer

```
instance Ord Integer where
  (<)  a b = …
  (<=) a b = …
  (==) a b = …
  (/=) a b = …
  (>)  a b = …
  (>=) a b = …
```

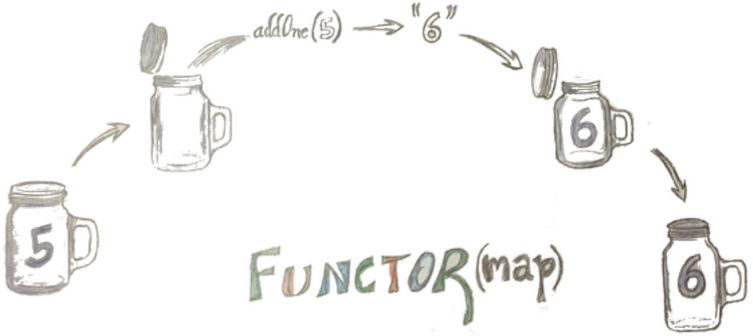
Generalized

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort (p:xs) = (sort lesser) ++ [p] ++ (sort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Functor



Abstract over type constructors that can be mapped over



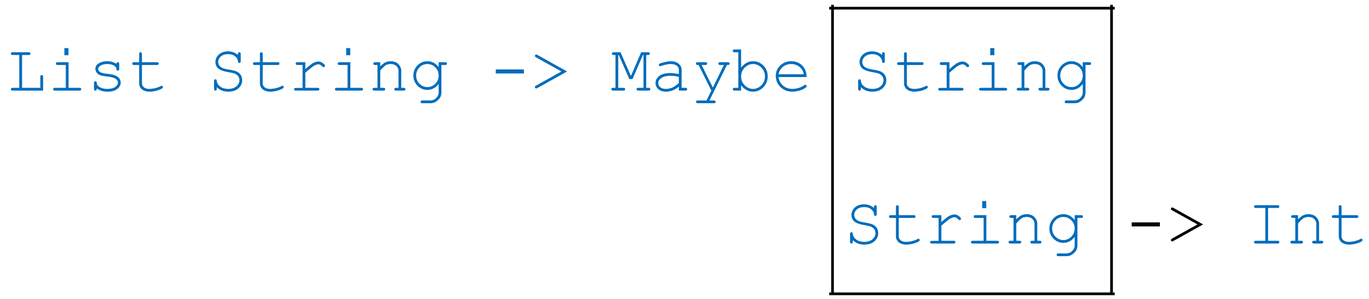
X

When function composition doesn't fit

```
-- head [“a”, “b”, “c”] == “a”
head :: List String -> Maybe String

-- Length “hello” == 5
length :: String -> Int
```

X

When function composition doesn't fit



X

Mapping over the value in a context

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

Mapping over the value in a context

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

```
head :: List String -> Maybe String

length :: String -> Int

-- composed [“test”, “b”] == Just 4
composed :: List String -> Maybe Int
composed = fmap length . head
```

Lifting a function in a context

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

```
length :: String -> Int -- length “example” == 7

-- maybeLength (Just “example”) == Just 7
maybeLength :: Maybe String -> Maybe Int
maybeLength = fmap length
```

Abstracting over fmap

```haskell
-- fproduct (Just 5) ((+) 1) == Just (5, 6)
fproduct :: Maybe a -> (a -> b) -> Maybe (a, b)
fproduct ma f = fmap (\a -> (a, f a)) ma
```

Abstracting over fmap

```
-- fproduct [1, 2, 3] ((+) 1) == [(1,2), (2, 3), (3, 4)]
fproduct :: List a -> (a -> b) -> List (a, b)
fproduct list f = fmap (\a -> (a, f a)) list
```

Abstracting over fmap

```
fproduct :: IO a -> (a -> b) -> IO (a, b)
fproduct ma f = fmap (\a -> (a, f a)) ma
```

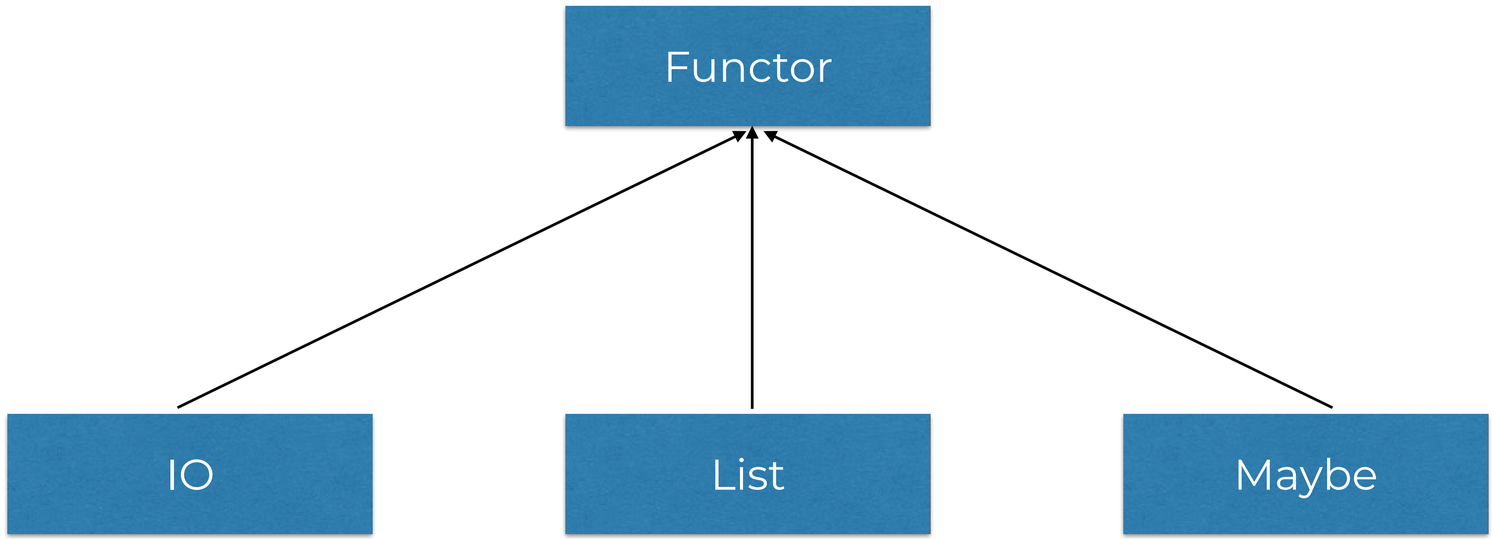
Code duplication

```
fproduct :: List a -> (a -> b) -> List (a, b)
fproduct ma f = fmap (\a -> (a, f a)) ma

fproduct :: Maybe a -> (a -> b) -> Maybe (a, b)
fproduct ma f = fmap (\a -> (a, f a)) ma

fproduct :: IO a -> (a -> b) -> IO (a, b)
fproduct ma f = fmap (\a -> (a, f a)) ma
```

Functor typeclass

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

Hierarchy



X

Functor for optionality

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just x) = Just $ f x
  fmap f Nothing = Nothing
```

Functor for List

```
instance Functor List where
  fmap :: (a -> b) -> List a -> List b
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs
```

Generalizing fproduct function

```
fproduct :: Functor f => f a -> (a -> b) -> f (a, b)
fproduct ma f = fmap (\a -> (a, f a)) ma
```

Impossible in Java

```
class Functor<F<>> {
}
```

X

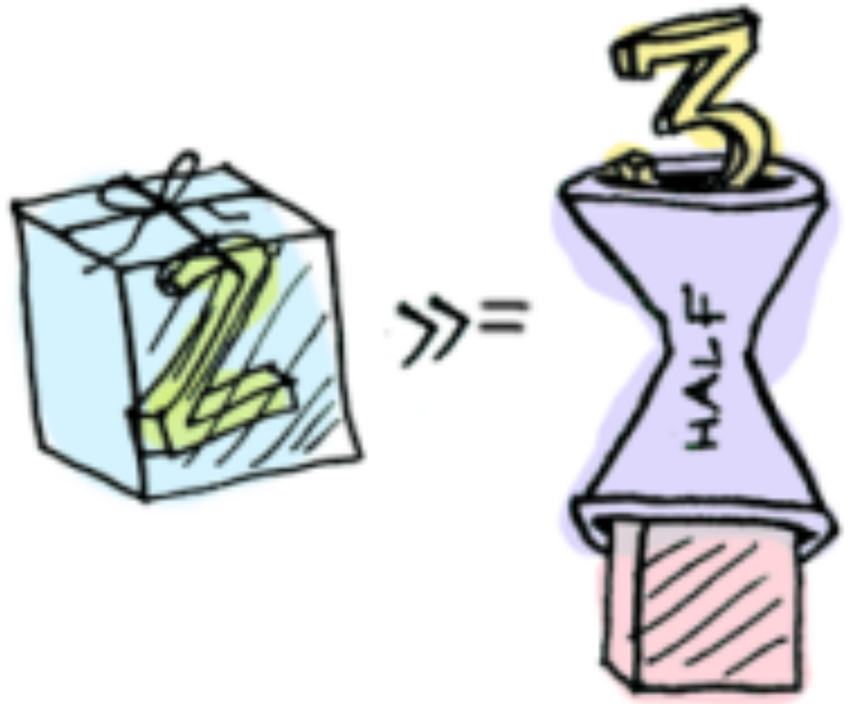
Recap

Transforming values in context

Functor typeclass allows polymorphism

Monad

Combine computations that depend on each other



Monad

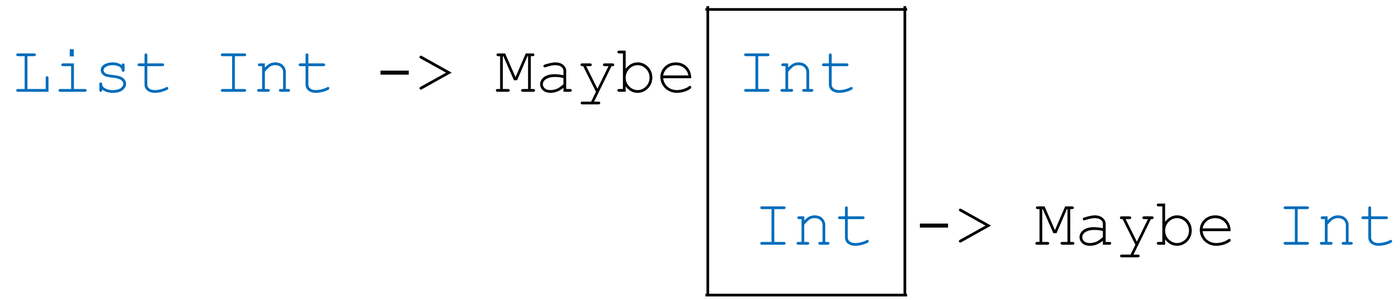
X

When functor is not enough

```haskell
head :: List Int -> Maybe Int
head [] = Nothing
head (x:xs) = Just x

-- fiveDivBy 2 == Just 2.5
-- fiveDivBy 0 == Nothing
fiveDivBy :: Int -> Maybe Int
fiveDivBy 0 = Nothing
fiveDivBy x = Just (5 / x)
```

x

Map over f with g resulting in context



When functor is not enough

```
composed list = fmap fiveDivBy (head list)
```

When functor is not enough

```
composed :: List Int -> Maybe (Maybe Int)
composed list = fmap fiveDivBy (head list)

-- composed [1, 2] == Just (Just 5)
-- composed [] == Nothing
-- composed [0, 3] == Just (Nothing)
```

When functor is not enough

```
composed :: List Int -> Maybe Int
composed list = flatten (fmap fiveDivBy (head list))

-- composed [1, 2] == Just 5
-- composed [] == Nothing
-- composed [0, 3] == Nothing
```

The Monad typeclass

```
class Monad m where
  return :: a -> m a
  fmap :: (a -> b) -> m a -> m b
  flatten :: m (m a) -> m a
```

The Monad typeclass

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

The Monad typeclass

```
class Monad m where
  return :: a -> m a
  fmap :: (a -> b) -> m a -> m b
  flatten :: m (m a) -> m a

  (>>=) :: m a -> (a -> m b) -> m b
  (>>=) ma f = flatten (fmap f ma)
```

x

The Monad typeclass

```
instance Monad Maybe where
  return :: a -> Maybe a
  return x = Just x

  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing

  flatten :: Maybe (Maybe a) -> Maybe a
  flatten (Just (Just x)) = Just x
  flatten _ = Nothing
```

x

Composing with Monads

```
head :: List Int -> Maybe Int
head [] = Nothing
head (x: xs) = Just x

fiveDivBy :: Int -> Maybe Int
fiveDivBy 0 = Nothing
fiveDivBy x = Just (5 / x)
```

Composing with Monads

Head

```
List Int -> Maybe Int
```

```
Int
Int -> Maybe Int
```

fiveDivBy

Composing with Monads

```
composed :: List Int -> Maybe Int
composed list = flatten (fmap fiveDivBy (head list))
```

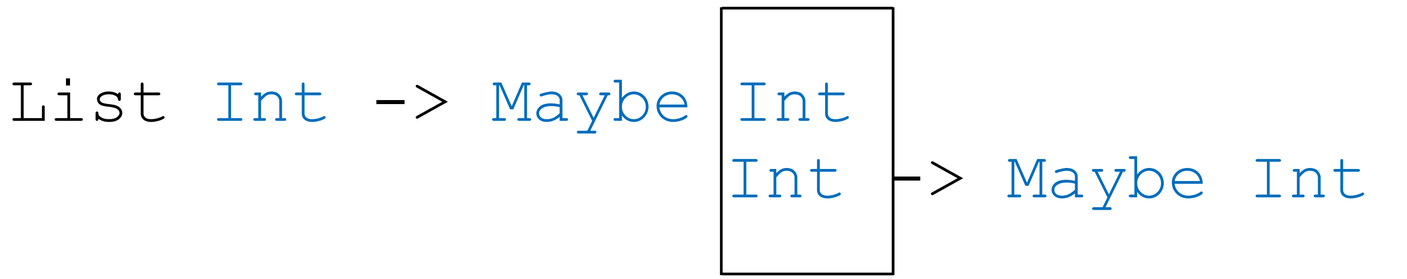
Composing with Monads

```
composed :: List Int -> Maybe Int
composed list = (head list) >>= fiveDivBy

-- composed [1, 2] == Just 5
-- composed [] == Nothing
-- composed [0, 3] == Nothing
```

X

Abstracting function composition with effects

Head



fiveDivBy

x

Abstracting function composition with effects

f

```
a -> Maybe b
b -> Maybe c
```

g

x

Abstracting function composition with effects

f

```
a -> List b
        b -> List c
```

g

x

Abstracting function composition with effects

f

```
a -> IO b
     b -> IO c
```

g

x

Abstracting function composition with effects

```
maybeComposition :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
maybeComposition f g = \a -> (f a) >>= g
```

Abstracting function composition with effects

```
listComposition :: (a -> List b) -> (b -> List c) -> (a -> List c)
listComposition f g = \a -> (f a) >>= g
```

Abstracting function composition with effects

```
ioComposition :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)
ioComposition f g = \a -> (f a) >>= g
```

```haskell
listComposition :: (a -> List b) -> (b -> List c) -> (a -> List c)
listComposition f g = \a -> (f a) >>= g

maybeComposition :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
maybeComposition f g = \a -> (f a) >>= g

ioComposition :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)
ioComposition f g = \a -> (f a) >>= g
```

X

```haskell
listComposition :: (a -> List b) -> (b -> List c) -> (a -> List c)
listComposition f g = \a -> (f a) >>= g

maybeComposition :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
maybeComposition f g = \a -> (f a) >>= g

ioComposition :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)
ioComposition f g = \a -> (f a) >>= g
```

x

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

maybeComposition :: (a -> m b) -> (b -> m c) -> (a -> m c)
maybeComposition f g = \a -> (f a) >>= g

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

x

Kleisli composition

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

Kleisli examples

```
head :: [a] -> Maybe a
head [] = Nothing
head (x:xs) = Just x

fiveDivBy :: Double -> Maybe Double
fiveDivBy 0 = Nothing
fiveDivBy x = Just (5 / x)
```

Kleisli examples

```
composed :: [Double] -> Maybe Double
composed = head >=> fiveDivBy
```

Kleisli examples

```
composed :: [Double] -> Maybe Double
composed = head >=> fiveDivBy

-- composed [1.0, 2.0, 3.0] == Just 5.0
-- composed [0.0, 1.0, 2.0] == Nothing
-- composed [] == Nothing
```

Kleisli examples

```
composed :: [Double] -> Maybe Double
composed list = case head list of
  Just x -> fiveDivBy x
  Nothing -> Nothing
```

Kleisli examples

```
composed :: [Double] -> Maybe Double
composed = head >=> fiveDivBy
```

```
another :: Double -> Maybe String
```

Kleisli examples

```
newComposed :: [Double] -> Maybe String
newComposed = head >=> fiveDivBy >=> another
```

Kleisli examples

```
newComposed :: [Double] -> Maybe String
newComposed list = case head list of
  Just x -> case fiveDivBy x of
    Just y -> another y
    Nothing -> Nothing
  Nothing -> Nothing
```

Recap

- Monad allows sequencing of computations
- Monad typeclass for abstracting over monads
- Kleisli composition as function composition with embellished types

Gabriel Gonzalez @GabrielG439 · 14 apr.

Using "the right tool for job" requires learning the available tools. Expand your horizons by trying a typed and/or functional language if you haven't already

Tweet vertalen

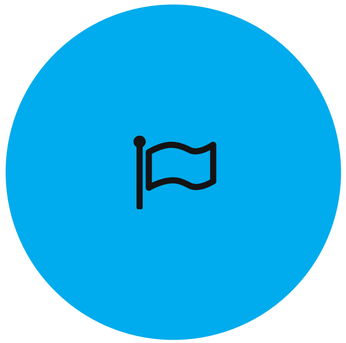
6 80 178

X

Beauty of functional programming

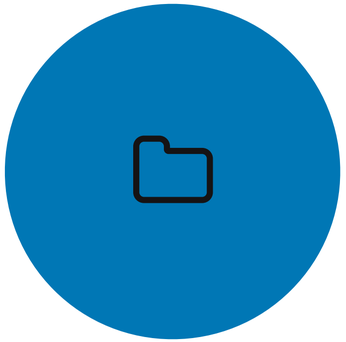
- We start with function composition
- Function composition doesn’t fit all requirements anymore
- Beautiful abstractions to fit the new requirements

Let’s connect



Twitter

@MauroPalsgraaf



LinkedIn

mauropalsgraaf

X