Programming Language Design

11/4 Functional Programming

What is functional programming?

- Use only pure functions
  - have no side effects
- And usually support
  - Higher-order functions
  - Currying
  - Lazy evaluation
  - Algebraic data types

No side effects

Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side-effects at all. A function call can have no effect other than to compute its result.

--- Why Functional Programming Matters. John Hughes, 1990.

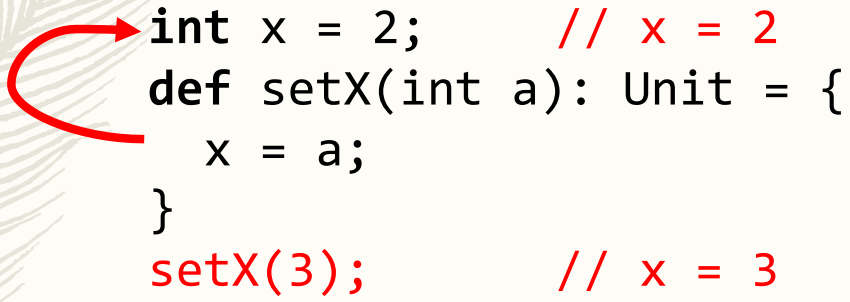
Side effects?

an unexpected or unplanned result of a situation or event

--- Longman Dictionary of Contemporary English

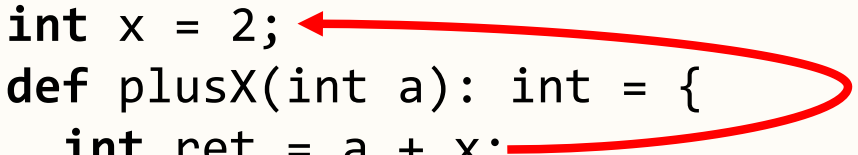
In imperative programming

- We imperatively change the states in a program by statements
  - There are “states” during program execution



In imperative programming (cont.)

- We do something based on the states in a program
  - Calling a function with the same arguments might get different results
    - *affected by the states*



```
int x = 2;
def plusX(int a): int = {
  int ret = a + x;
  ret;
}
int y = plusX(3);      // y = 5
x = 4;
y = plusX(3);          // y = 7
```

Using OO with imperative style

- Usually we let objects hold states
  - Do something in functions (methods) based on the states
    - *E.g. set width/height and get width/height*
    - *Setters change the states out of themselves*
    - *What getters return depends on external states*

```
class Rectangle {
  var width: Int = 30
  var height: Int = 20
  def getWidth(): Int = width
  def setWidth(w: Int): Unit = width = w
  def getHeight(): Int = height
  def setHeight(h: Int): Unit = height = h
}
```

So… what are side-effects?

- If a function changes the states out of itself
  - we say it has side effects
  - Since what it does other things besides returning a result
  - Why it is a “side effect”? The purpose of setter is setting the state!
    - Well, the purpose of calling a function should be getting the return value

A simple example: buyCoffee

- Suppose we want to implement handling purchases at a coffee shop
  - In Scala, with side effects
    - *buyCoffee returns a cup of coffee, and then the states of cc is changed!*

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    cup
  }
}
```

After executing buyCoffee, the states of cc will be changed!

Reference

- *Paul Chiusano and Rúnar Bjarnason. Functional Programming in Scala, 2014.*

Side effects mean…

- If a function has side effects, it means
  - It does something in addition to returning a result
    - *Change the value of a variable globally*
    - *Modify the given arguments*
    - *Throw an exception*
    - *Get user input or print to the screen*

Side effects mean… (cont.)

- In other words, if a function change the program's state
  - Impure, has side effects
- It might not be safe if a function change something beyond its scope
  - Make code hard to reason about
    - *Calling a function might get different result every time*
      - Which functions make the change?
      - When the change happens?

Wait…is it possible?

- How could we write a program without any side effect??
  - Since we need I/O operations
    - *Print string on the screen*
    - *Get input from keyboard or mouse*
    - *Get timestamp*
  - Indeed it is quite hard to eliminate all side effects, but we can limit its scope
    - Let it happen in only a few functions

Why functional?

This eliminates a major source of bugs, and also makes the order of execution irrelevant — since no side-effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control.

--- Why Functional Programming Matters. John Hughes, 1990.

Why functional? (cont.)

Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa — that is, programs are "referentially transparent". This freedom helps make functional programs more tractable mathematically than their conventional counterparts.

--- Why Functional Programming Matters. John Hughes, 1990.

Referentially transparent?

- Code that do not mutate or depend on the external world
  - You can use it in any context
  - Always returns the same value for the same argument

For example, a function intToString

- It takes an Int and returns a String
  - *We may use such notations for its type*

```
Int -> String
```

or

```
Int => String
```

  - Denote the type after the colon following the function name

```
intToString: Int -> String
```

or

```
intToString:: Int -> String
```

intToString should…

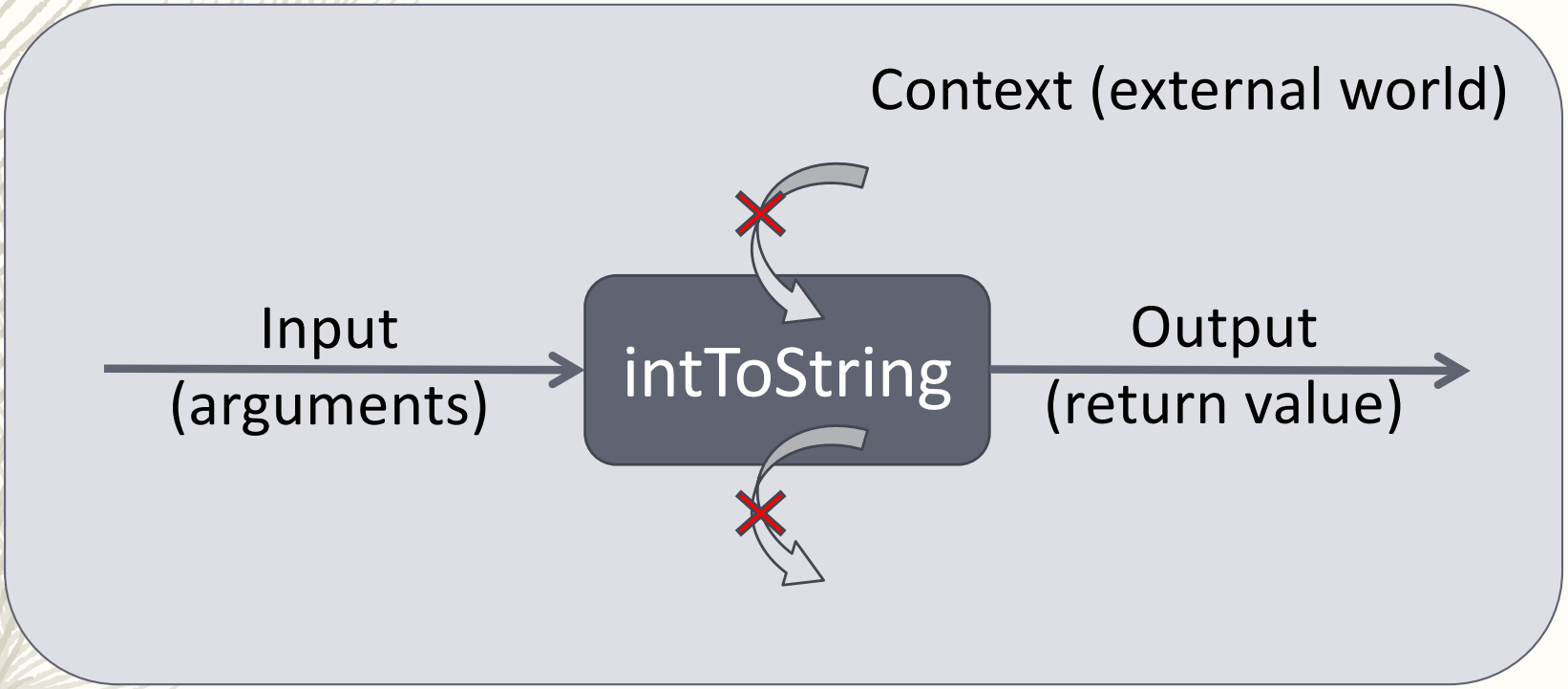
- Should only convert the given integer to a string, nothing else
- If it change anything else in the program
  - It is surprising, and therefore unsafe
  - Hard to know which function did a change and why it did!
  - It should not mutate anything in the external world!

intToString should not…

- Should not do the conversion based on the environment
- If the result rely on the states of the program
  - It is also surprising and unsafe
  - The result might be different every time, and the order of execution matters
- It should not depend on the external world!

i.e. not context-dependent

- Not affected by
- Cause no side effects



In other words, purity

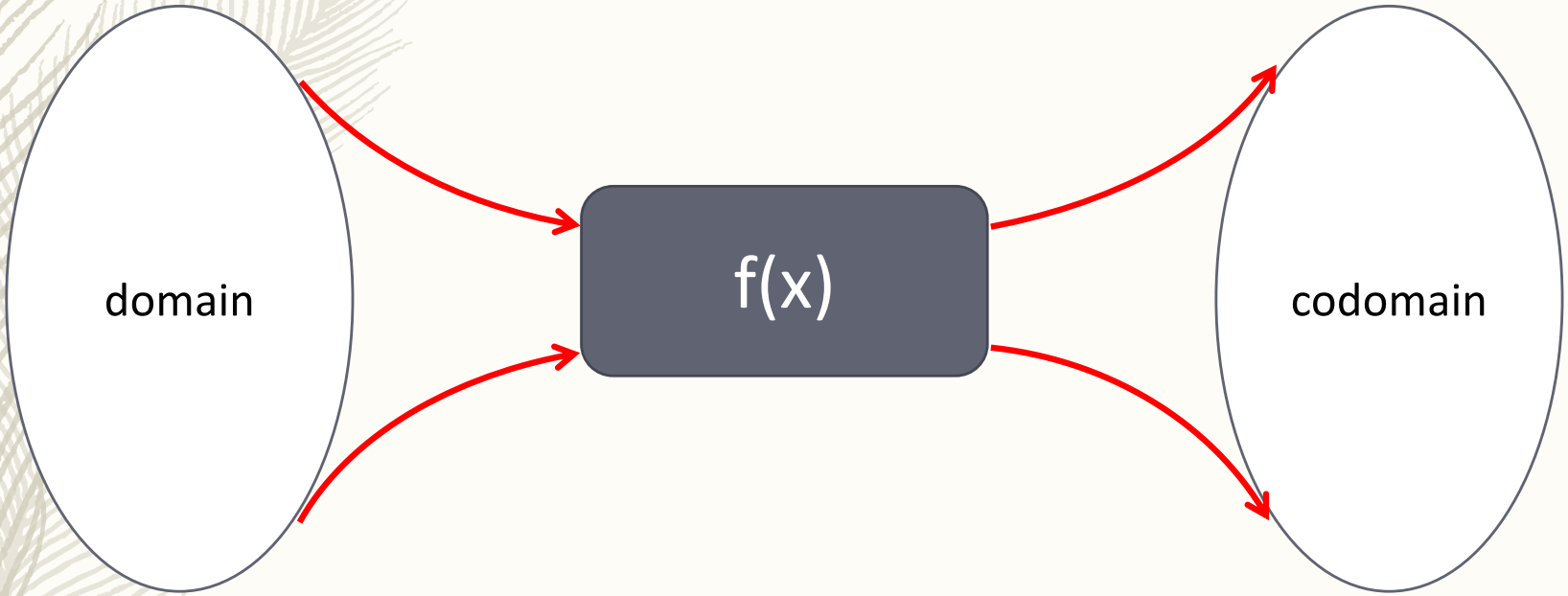
- Remember mathematical functions
  - Mapping between two sets: domain to codomain
- Given an element from its domain, it yields an element from its codomain based on some formula
  - Nothing else

→ Purity

Mathematical functions

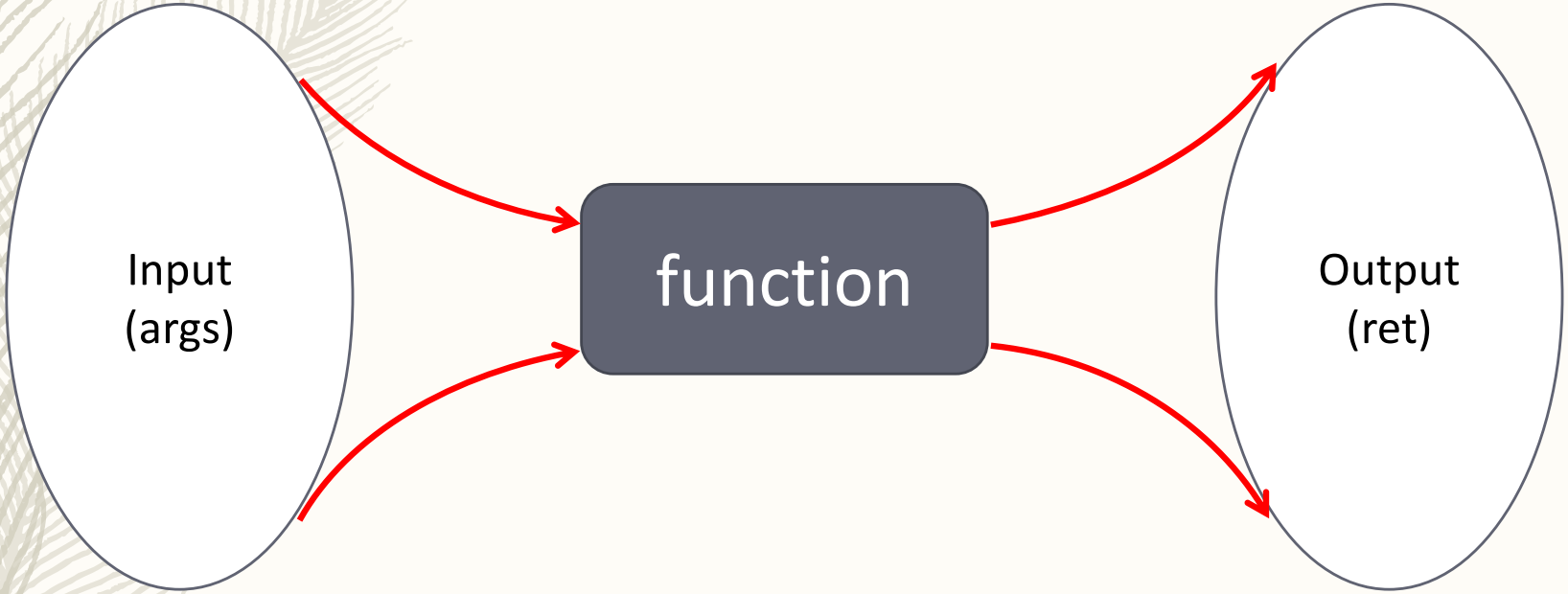
y = f(x)

mapping values from domain to codomain



Pure functions

- Resemble mathematical functions
  - The output depends entirely on input arguments
  - Cause no side effects



Substitution model

- Allow replacing any function call with its return value

- Since it is always evaluated to the same value
  - at any time
  - if we give the same arguments

So, what is functional programming?

- A programming paradigm that emphasizes functions and avoids state mutation
  - Declarative and expressive
- Quite different from imperative programming
  - Usually you need to spend more time on thinking how to write rather than writing

The opposite of OOP?

- NO!
- Indeed in most cases the term “OOP” implies the one used in an imperative style
  - Heavily rely on state mutation and use explicit control flow
- It is still possible to use objects with FP
  - Actually many functional programming languages such as OCaml and Scala support some form of OOP

Recently FP is getting popular

- It makes code more concise and strict
  - To reduce bugs
- Several imperative OO languages also give certain FP support
  - To write “FP style” code
    - *There are some books teaching how to write Java or C# code with FP flavor*

The languages used in this slides

- Haskell is a pure FP language
  - Named after the mathematician Haskell Brooks Curry
  - Here we use to explain several FP concepts
    - *If you are interested in FP, you should check it*
- Scala: object-oriented meets functional
  - Supports multiple paradigms
    - *Includes functional and imperative*
  - Here we use for some examples since it is easy to compare with Java
    - *Compatible with Java programs*

Let's start from buyCoffee example

- Suppose we want to implement handling purchases at a coffee shop
  - In Scala, with side effects
  - buyCoffee returns a cup of coffee, and charge is a side effect

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    cup
  }
}
```

But, how to test?

- We don't want to actually charge the card!
  - Actually CreditCard should not have any knowledge about charge
- Change our design by adding Payment object
  - Then we can implement a mock of Payment to test

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payment): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

How to reuse?

- Suppose someone want to buy 12 cups of coffee
  - Call it 12 times in a loop?
  - Implement another buy12Coffee function?

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payment): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

A functional implementation

- Eliminate the side effect in buyCoffee
  - Returns a pair: a cup of coffee and a Payment
  - Payment will be handled elsewhere

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Payment) = {
    val cup = new Coffee()
    (cup, new Payment(cc, cup.price))
  }
}
```

Such Payments can be combined

- Then it is possible to combine Payments
  - By creating a new Payment, if they use the same card
  - And the charge operation can be done once

```
class Payment(cc: CreditCard, amount: Double) {

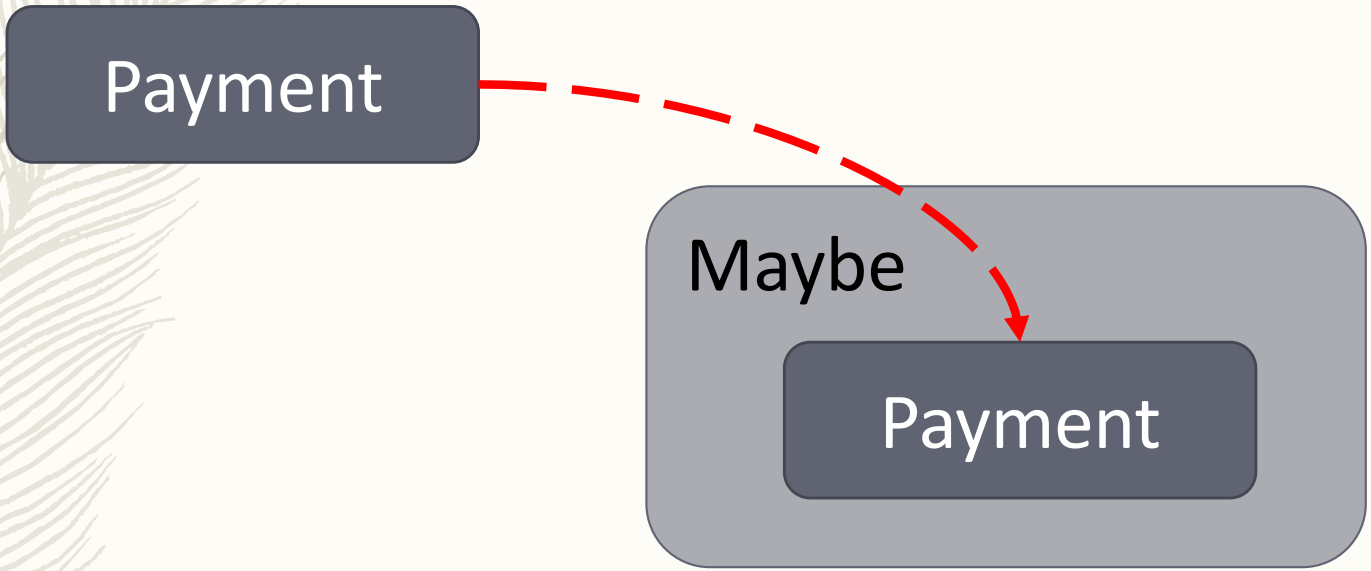
  def combine(another: Payment): Payment = {
    if (cc == another.cc)
      new Payment(cc, amount + another.amount)
    else
      throw new Exception("cards in them must be the same")
  }
}
```

Return a Maybe

- You might notice that our combine function might throw an exception
  - It is “dishonest”
  - It doesn’t always return a Payment as it declares
- Instead of returning a Payment
  - It may return a Maybe
    - *The Maybe might contain a Payment or nothing*

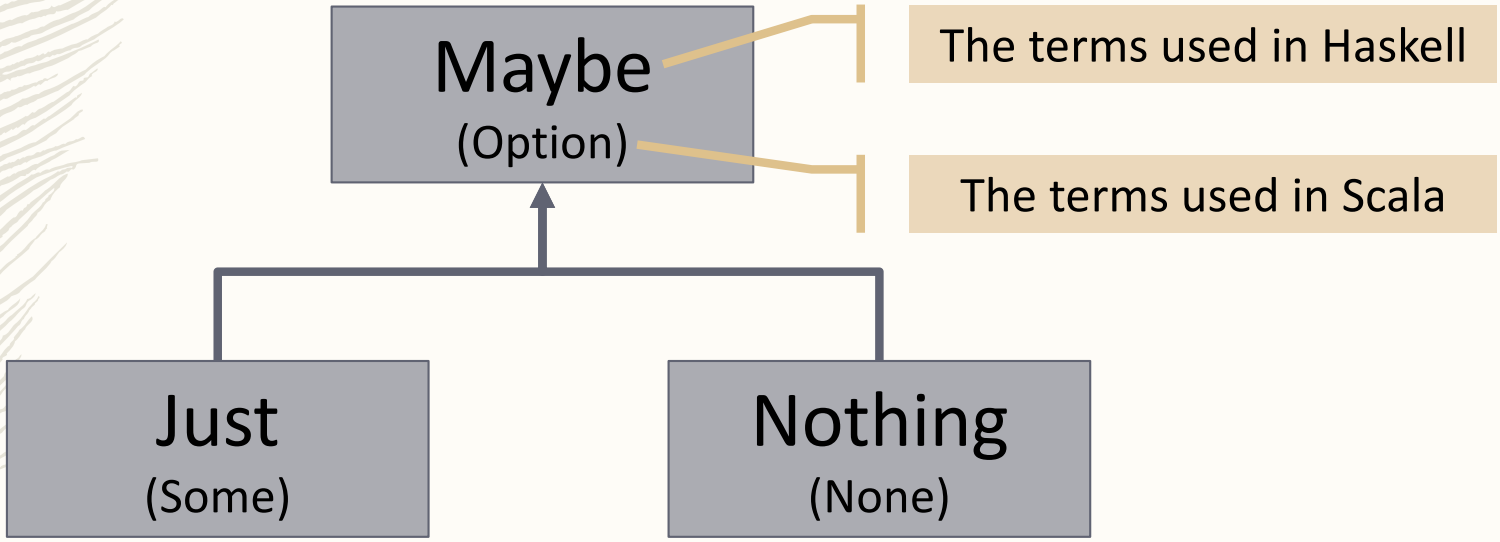
A “boxed” result

- Maybe is some sort of “box”, “container”
  - Might or might not contain a result



There are two kinds of Maybe

- Maybe<T> = Just(T) | Nothing
- In OO languages we can use subclassing
  - For normal cases, returns Just with the result
  - For the exception case, returns Nothing



An example of using Maybe

- Suppose we have a function to calculate the mean of the values in a sequence
- With exceptions

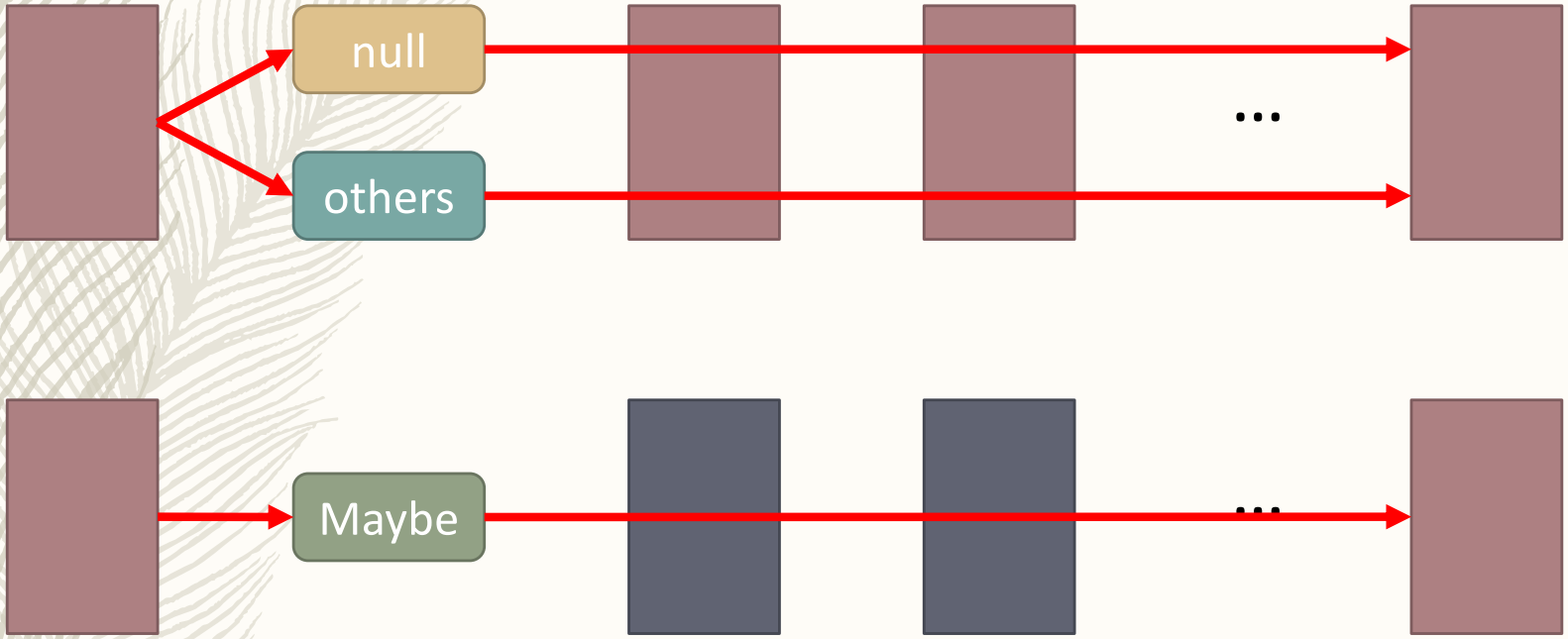
```
def mean(xs: Seq[Double]): Double =
  if (xs.isEmpty)
    throw new ArithmeticException("mean of empty list!")
  else xs.sum / xs.length
```

- Without exceptions

```
def mean(xs: Seq[Double]): Maybe[Double] =
  if (xs.isEmpty) Nothing
  else Just(xs.sum / xs.length)
```

Another example of using Maybe

- Avoid special handling for null pointer
  - null can be wrapped in Maybe



Problems with the null pointer

- The null pointer was invented in 1965
  - by Tony Hoare (C. A. R. Hoare)
- What he said 44 years later:
  - I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Reference

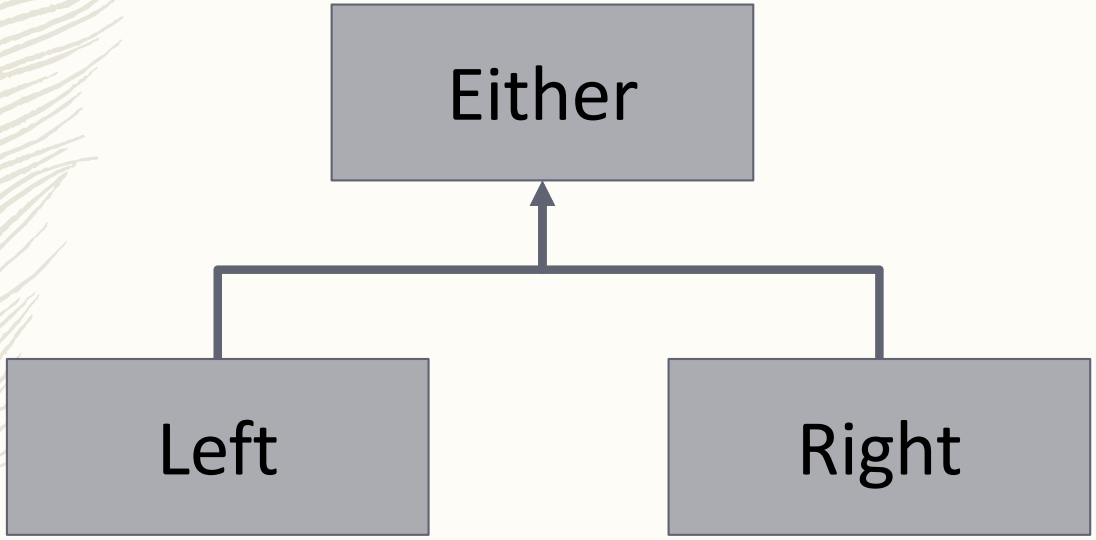
- *Tony Hoare. Null References: The Billion Dollar Mistake, 2009.*

With Maybe type

- We can handle errors
  - without null pointer
  - without exceptions
- However, Maybe cannot provide information for Nothing
  - i.e. the reason for errors
  - E.g. the string in an Exception

The Either type

- Either<L, R> = Left<L> | Right<R>
- Two kinds of Either
  - Left: something wrong
  - Right: all right



Using exceptions, Maybe, or Either

- Now we can compare the three versions of mean
  - With exceptions

```
def mean(xs: Seq[Double]): Double =
  if (xs.isEmpty)
    throw new ArithmeticException("mean of empty list!")
  else xs.sum / xs.length
```

  - With Maybe

```
def mean(xs: Seq[Double]): Maybe[Double] =
  if (xs.isEmpty) Nothing
  else Just(xs.sum / xs.length)
```

  - With Either

```
def mean(xs: Seq[Double]): Either[String, Double] =
  if (xs.isEmpty) Left("mean of empty list!")
  else Right(xs.sum / xs.length)
```

Lists and Tuples in Haskell

- Lists are homogenous
  - To store elements of the same type

```
[1, 2, 3]
```

```
["Haskell", "Curry"]
```

- Tuples are heterogenous
  - Contain a combination of several types

```
("Haskell", 1)
```

  - A list of tuples

```
[("Haskell", 1), ("Curry", 2)]
```

Use :type in GHCi

- In GHCi you can check the type of an expression

```
> :type [1, 2, 3]
[1, 2, 3] :: Num t => [t]
> :type ["Haskell", "Curry"]
["Haskell", "Curry"] :: [[Char]]

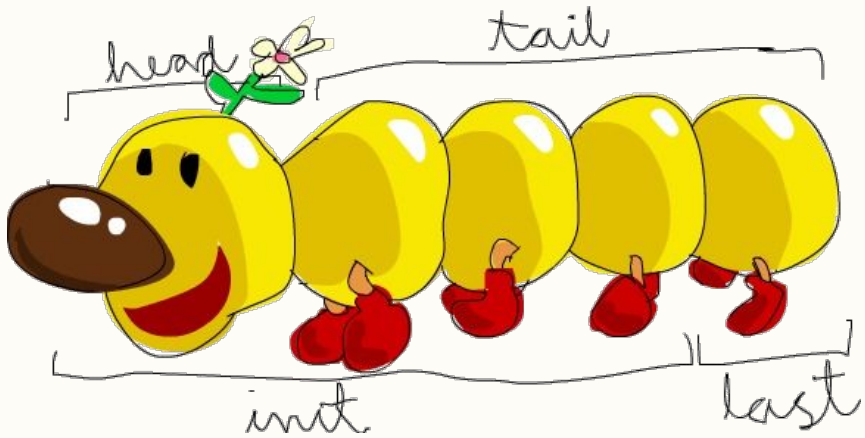

> :type ("Haskell", 1)
("Haskell", 1) :: Num t => ([Char], t)

> :type [("Haskell", 1), ("Curry", 2)]
[("Haskell", 1), ("Curry", 2)] :: Num t => [([Char], t)]
```

Some useful functions for lists

- head, tail, init, and last
  - Here I borrow a cute but easy-to-understand figure from a website*

```
> head [1, 2, 3]
1
> tail [1, 2, 3]
[2,3]
> init [1, 2, 3]
[1,2]
> last [1, 2, 3]
3
```



Reference

* *Starting Out - Learn You a Haskell for Great Good!*
http://learnyouahaskell.com/starting-out#an-intro-to-lists

Some useful functions for lists (cont.)

– length: give you the length of a list

```
> length [2, 4, 6, 0, 1]
5
> length ["Who", "Am", "I"]
3

> :type length
length :: Foldable t => t a -> Int
```

it works for number lists

but also works for string lists?!

– reverse: reverse the list

```
> :type reverse
reverse :: [a] -> [a]

> reverse [2, 4, 6, 0, 1]
[1, 0, 6, 4, 2]
> reverse ["Who", "Am", "I"]
["I", "Am", "Who"]
```

reverse accepts any type of lists!

it accepts any a that has type t!
t is a Foldable

A type class, we will talk it later

Some useful functions for lists (cont.)

– take: take a number of elements in a list

```
> take 3 [2, 4, 6, 0, 1]
[2,4,6]
> take 1 ["Who", "Am", "I"]
["Who"]
```

– takeWhile: take elements from a list while the predicate holds

```
> takeWhile (<5) [2, 4, 6, 0, 1]
[2,4]
```

Recursion looks much natural

- Can simply write functions with themselves
- For example, a function returning i-th Fibonacci number
  - ```
    fib :: Integer -> Integer
    fib 0 = 0
    fib 1 = 1
    fib n = fib (n-1) + fib (n-2)
    ```

A way to define a function in Haskell: pattern match

  - If you want to try it in GHCi within one line

```
> let { fib 0 = 0 ; fib 1 = 1 ; fib n = fib (n-1) + fib (n-2) }
> fib 3
2
```
```

Higher-order functions

- HOFs are functions that
  - Take other functions as arguments
  - Return functions
- Functions are first-class values
  - In languages that don't support first-class value, they are usually represented by classes

The map function is higher-order

- The idea of map function is essential in FP
  - It takes a function and a list
  - Applies that function to each element in the list
    - *An example in Haskell:*

```
> map head ["bat", "cat", "dog"]
"bcd"
```

```
head :: [a] -> a
```

    - *Example 2:*

```
> map (take 2) ["bat", "cat", "dog"]
["ba", "ca", "do"]
```

```
(take 2) :: [a] -> [a]
```

The fold function

– Folding a list using foldl

```
> foldl (+) 0 [1, 2, 3, 4]
10
```

```
foldl (+) 0 [1, 2, 3, 4]
foldl (+) 1 [2, 3, 4]
foldl (+) 3 [3, 4]
foldl (+) 6 [4]
foldl (+) 10 []
```

The fold function (cont.)

- Useful to transform the list to a result
  - By apply the given function

- For example,
  calculate the sum of the numbers,
  check if anyone in the list says True,
  combine the result of applying a function to all

Infinite lists?!

- You may try the follows in GHCi

```
> [1..5]
[1,2,3,4,5]
> [1..]
[1,2,3,4,5,.....................
```

it never stops!

- However…

```
> take 2 [1..]
[1,2]
> takeWhile (<5) [1..]
[1,2,3,4]
```

Lazy is a good thing ;)

- Why infinite lists can be “evaluated”?
  - → Lazy
  - Do not evaluate until it is necessary
    - Easy to describe abstractly
    - Code looks better
      - *For example, take even integers that are larger or equal than 1 but are smaller than 10*

```
> takeWhile (<10) (filter even [1..])
[2,4,6,8]
```

Lazy is…

"If you know that can be done tomorrow, don't do it today."

Look it inside

- How the type of such a HOF looks like?

```
> :type takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
```

  - takeWhile takes
    - *a function that takes a and returns Bool*
    - *and a list*
    - *then returns a list*

Look it inside (cont.)

- So the predicate we used in takeWhile, (<10), is also a function!

```
> :type (<10)
(<10) :: (Ord a, Num a) => a -> Bool
```

- (<10) takes
  - *a, which is an Ord and a Num*
  - *then returns a Bool*

Uh? So what is “(<)”?

- (<) is a function as well

```
> :type (<)
(<) :: (Ord a) => a -> a -> Bool
```

- (<) takes
  - *a, which is an Ord*
  - *and another a*
  - *then returns a Bool*

Wait… (<) and (<10)?

```
> :type (<10)
(<10) :: (Ord a, Num a) => a -> Bool
> :type (<)
(<) :: (Ord a) => a -> a -> Bool
```

- You might want to try this

```
> :type (<) 10
(<) 10 :: (Ord a, Num a) => a -> Bool
```

- The function (<) takes two a and returns a Bool
  However, it can also be regarded as
  - taking one a and returns a function that takes one a and returns a Bool
  - `a -> a -> Bool` can be written as `a -> (a -> Bool)`

Currying

- Named after mathematician Haskell Curry
- The process of transforming an n-ary function that takes t1, t2, t3, …, tn
  into a unary function that takes t1
  and yield a new function that takes t2, and so on
- In other words, an n-ary function with signature:

```
(T1, T2, T3, …, Tn) -> R
```

- When curried, it has signature:

```
T1 -> T2 -> T3 -> … -> Tn -> R
```

A simple example of currying

- For example, a function f that does “x plus y”
  - It takes two integers and returns an integer

```
f :: Int -> Int -> Int
```

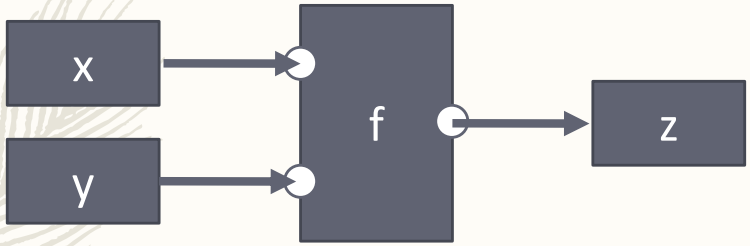
- If we give it a “x”, it returns a function f’
  - f’ takes an integer and returns an integer

```
f :: Int -> (Int -> Int)
```

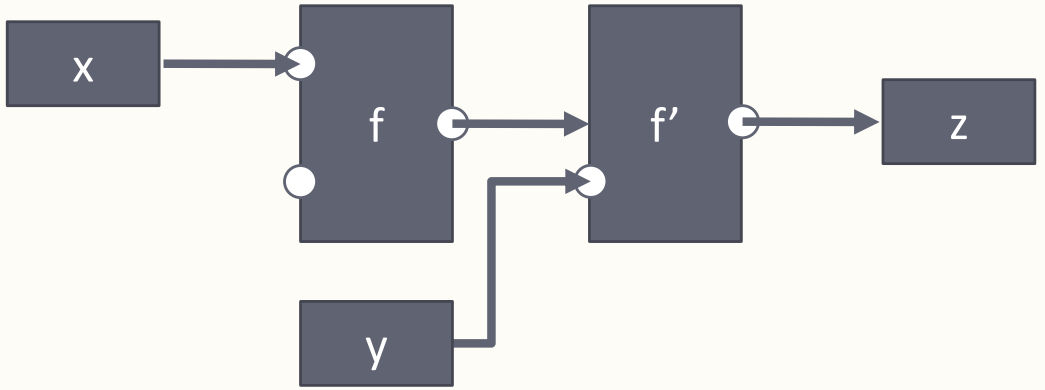
```
f’ :: Int -> Int
```

A simple example of currying (cont.)

- For example, a function f that does “x plus y”
  - It takes two integers and returns an integer

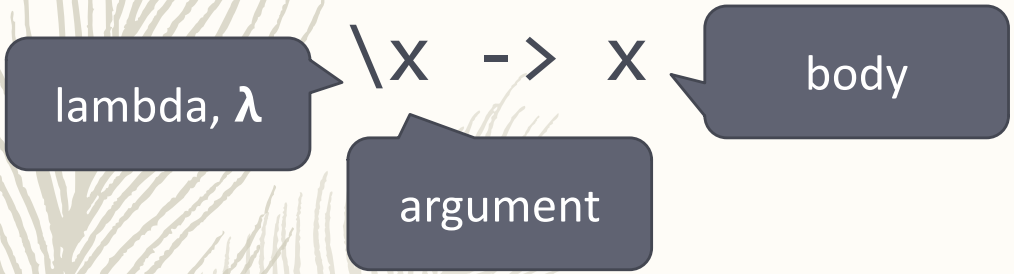


- If we give it a “x”, it returns a function f’
  - f’ takes an integer and returns an integer



Lambda functions

– A function without a name; anonymous function




```
> (\x -> x) 1
1
> (\x -> x * x) 2
4

> (\x y -> x * y) 3 4
12
> (\x -> (\y -> x * y)) 3 4
12

> :type ((\x y -> x * y) 3)
((\x y -> x * y) 3) :: Num a => a -> a
> :type (\x -> (\y -> x * y)) 3
(\x -> (\y -> x * y)) 3 :: Num a => a -> a
```

Function composition

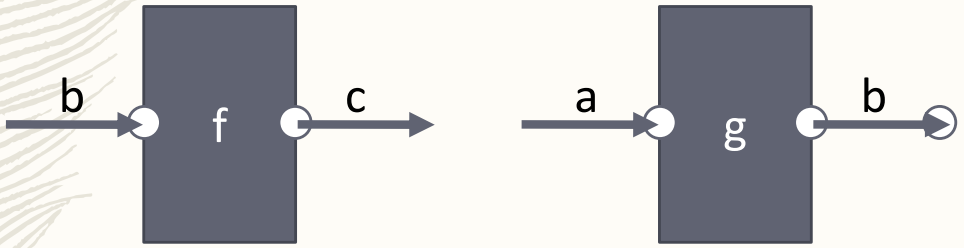
– In mathematics,

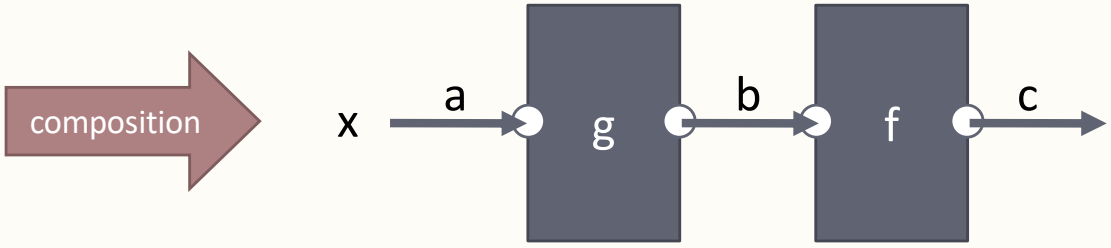
$(f \circ g)(x) = f(g(x))$

– In Haskell,

(f . g) x is equivalent to f (g x)

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





Function composition examples

```
> ((+2) . (*3)) 4
14
```

4 * 3 + 2

```
> f = negate . abs
> map f [-2, 4, -6, 0, -1]
[-2,-4,-6,0,-1]
```

abs → [2, 4, 6, 0, 1]
negate → [-2, -4, -6, 0, -1]

```
> map (sum . tail) [[1..3], [3..5], [5..7]]
[5, 9, 13]
```

tail → [[2, 3], [4, 5], [6, 7]]
sum → [2+3, 4+5, 6+7]

Algebraic data type

- Remember three parts in mathematical algebra
  - Objects: numbers such as 1, 2, 3, or x, y, z, etc.
  - Operations: addition, multiplication, etc.
  - Laws: the ones algebra obeys

    e.g. 0 + x = x

    1 * x = x

Algebraic data type (cont.)

- In functional programming, algebraic data type is the type formed by other types, for example
  - data Bool = True | False
  - data Maybe a = Just a | Nothing
  - data Either a b = Left a | Right b
- The above examples are enumeration types
  - Others include pattern matching and recursive

Type constructor and type parameter

- Type constructors take types as parameters to produce new types
  - For example,
    data Maybe a = Just a | Nothing
  - `Maybe`
    is a type constructor, and
    `Maybe Int`
    is a concrete type
  - a is type parameter

```
> :type Just "hello"
Just "hello" :: Maybe [Char]
> :type Nothing
Nothing :: Maybe a
```

Type variables

- You might notice the following thing in our examples

```
> :type head
head :: [a] -> a
```

  - a is a type variable
  - a can be any type;
    in other words, can't exactly be a type
- Functions that have type variables are called polymorphic functions
  - like generics in other languages, but looks much simpler

Type classes

- You might also notice such a “type of type”

```
> :type foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

Class constraint

  - Foldable defines data that can be folded to a summary

- Strictly speaking, type class is not “type of type”
  - A kind of interface that defines some behavior
  - Similar to the interfaces in other languages

Examples of type classes

- Eq defines equality and inequality

```
class Eq a where
minimal complete definition: (==) | (/=)
```

  - As an instance (type) of Eq (type class), that type must complete the definition of either `(==)` or `(/=)`
  - its instance include Bool, Float, Int, etc.

- Ord is used for totally ordered data

```
class Eq a => Ord a where
minimal complete definition: compare | (<=)
```

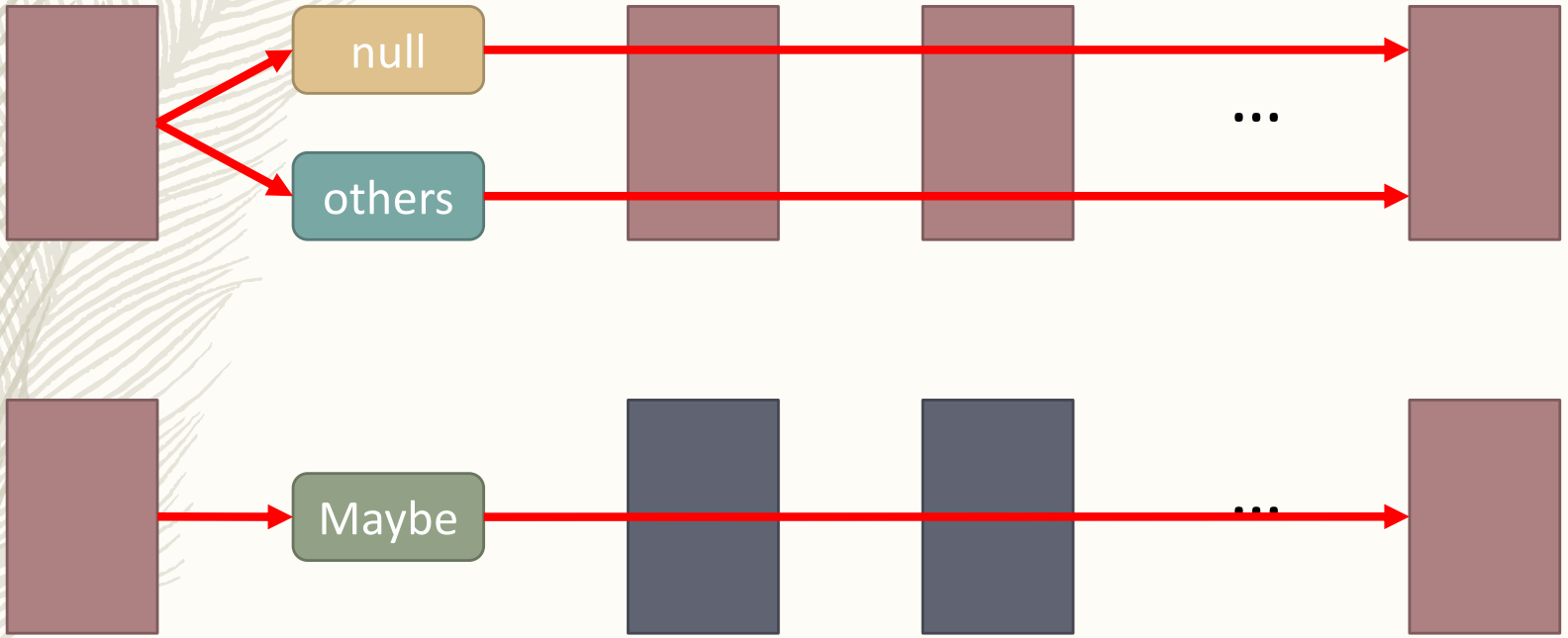
  - An Ord must be also be an Eq!

Type classes (cont.)

- Don't confuse with data types like Maybe
  - Terminology in Haskell
  - keyword `data` for data type (class)
  - keyword `class` for type classes
- We will see an interesting type classes later: Functor

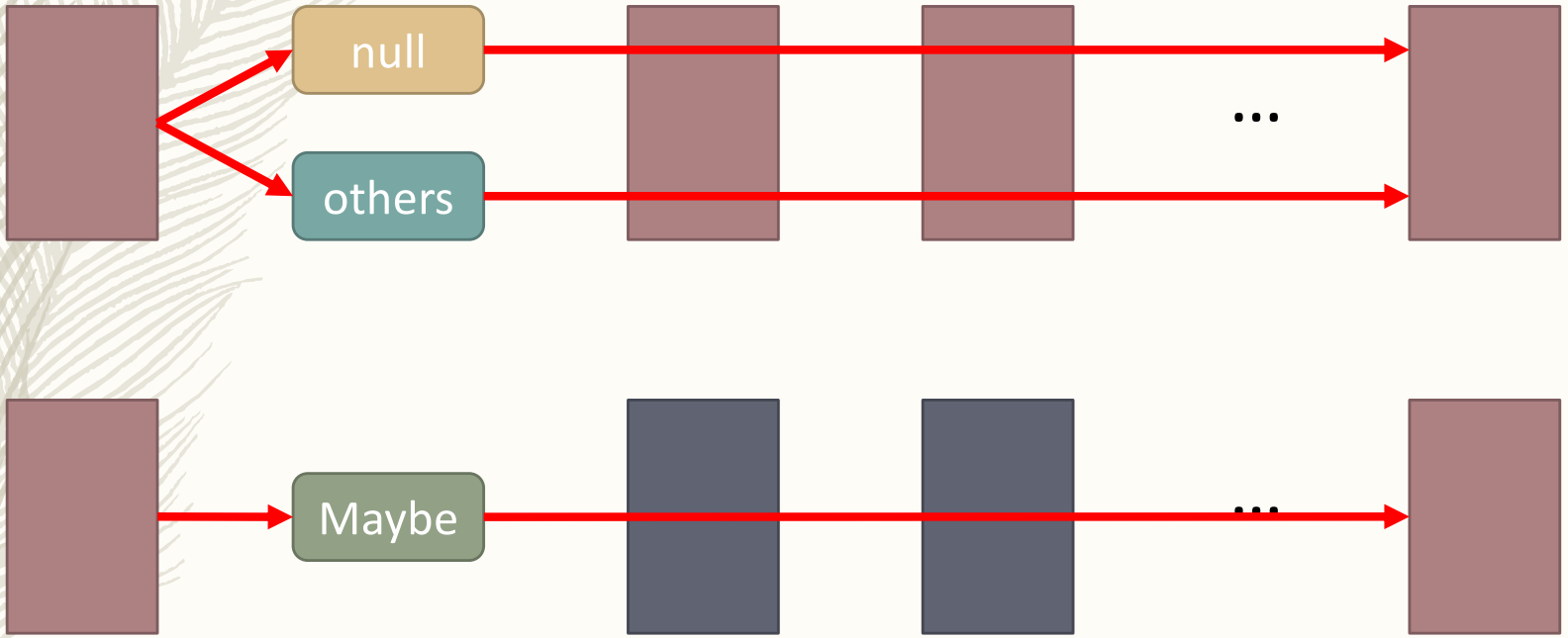
Why we wrap up values?

- Make the handling smooth
  - Avoid side effects in functions during processing
    - *null and exceptions are wrapped up*



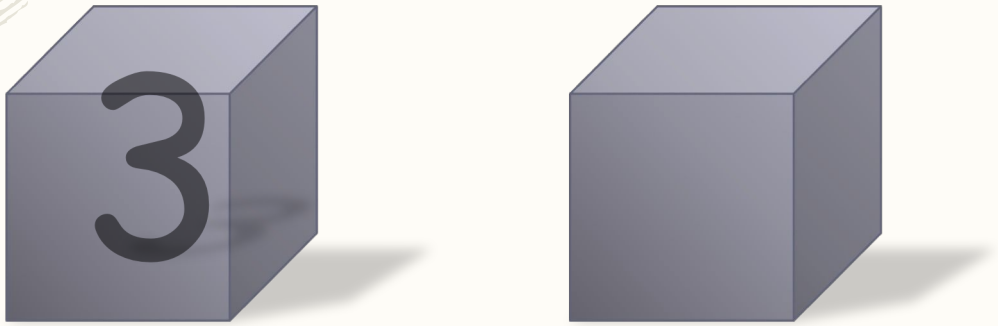
Why we wrap up values? (cont.)

- Get the value in the container only when we really have to do
  - Only a limited number of functions have side effects



How to use such boxed values?

– For example, a Maybe<Int> might be an integer or nothing!



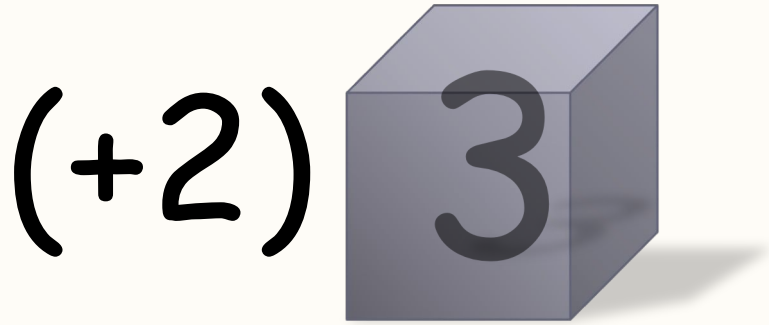
Reference

- *http://learnyouahaskell.com/functors-applicative-functors-and-monoids*
- *http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html*

How to use such boxed values?

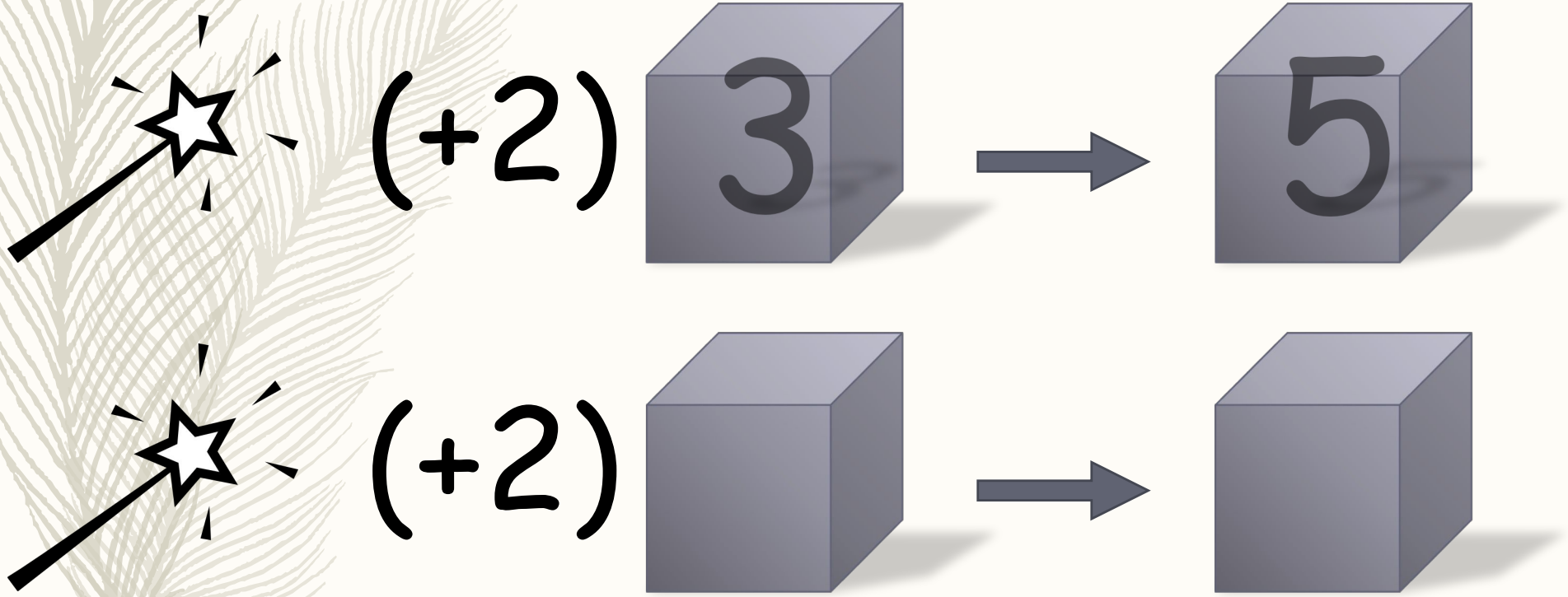
- For a normal value, we can simply apply functions to it

- For a boxed value?? Ouch! Normal functions do not know how to handle it…



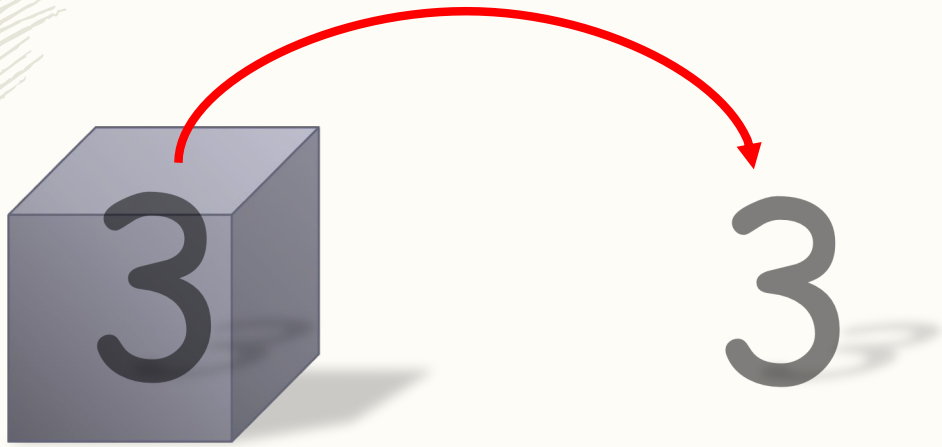
We need such a magic!

– We need a magic that knows how to apply the given function to the given boxed value!



What the magic should do is…

1. Unwrap the boxed value

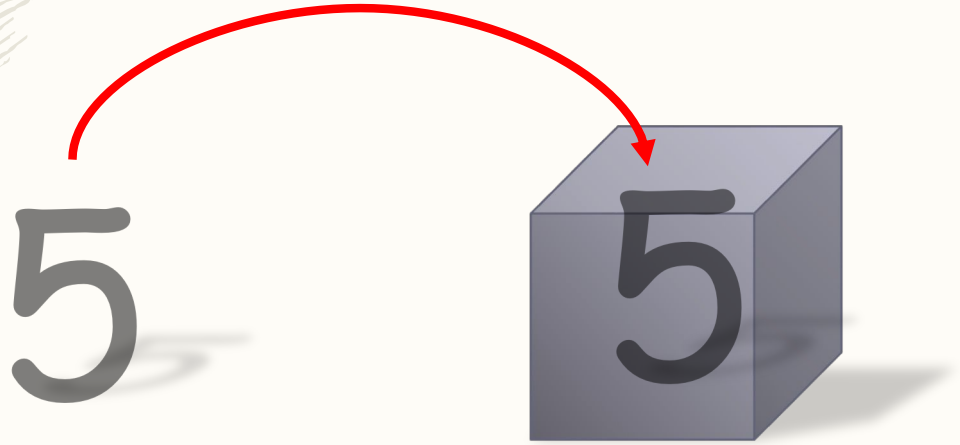


What the magic should do is… (cont.)

2. Give the value to the function

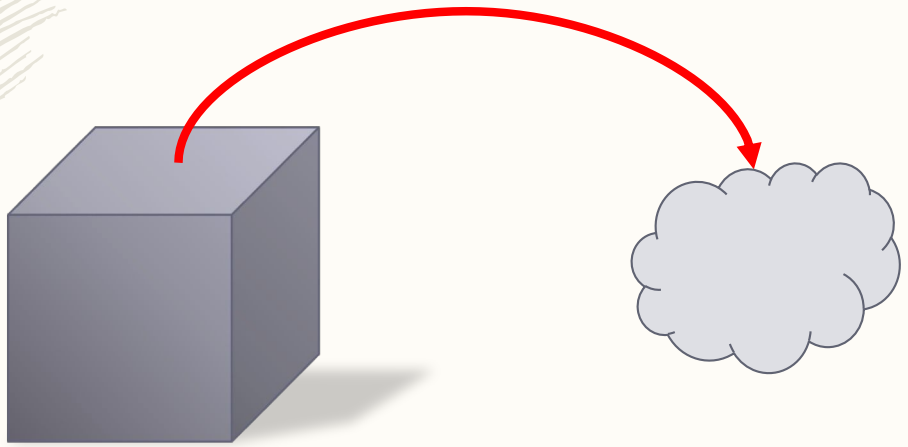
What the magic should do is… (cont.)

3. Wrap up the result in a box



What the magic should do is… (cont.)

1. If nothing is in the box

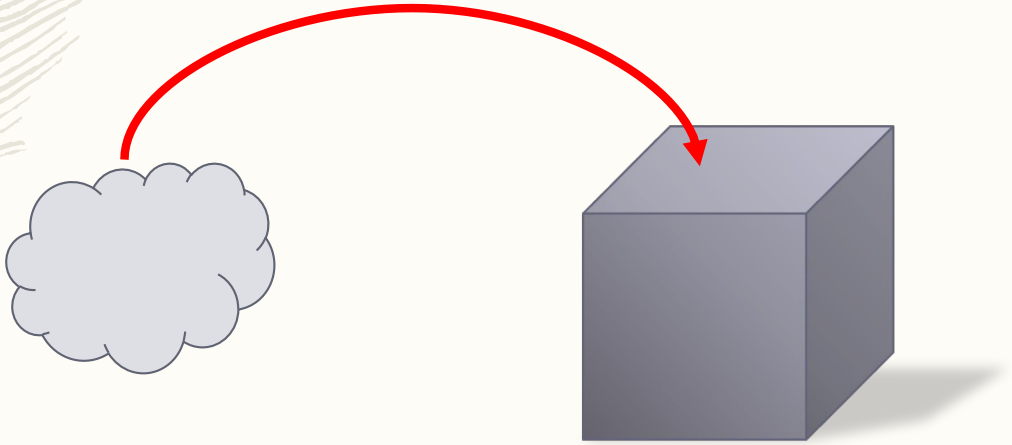


What the magic should do is… (cont.)

2. Do not apply the function

What the magic should do is… (cont.)

3. And simply put nothing in a box



Actually it is not a magic

– For a container (box) “f”, we have to define such a function  :

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

which takes a function and a boxed value a, and returns a boxed value

Functor and fmap

- The container “f” that has such a function is called functor
  - Such a function is called “fmap” or “map”
  - Functors come from mathematics
    - *Category theory*
- You may simply think of interfaces
  - A kind of interface that defines such a behavior

Functor laws

1. Identity law
   `fmap id = id`
   - i.e. `fmap id F = F`
   - id is the identity function, which simply return its parameter, i.e. `\x -> x`

2. Composition law
   `fmap (f . g) = fmap f . fmap g`
   - i.e. `fmap (f . g) F = fmap f (fmap g F)`
   - Composing two functions and then map the resulting function over a functor should be the same as mapping them to the functor in order

So, if we have a fmap for Maybe

- We can apply any function to Maybe
  - Apply the function if it has a value
  - Do nothing if it is nothing (null)

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

Similarly, for Either

- It represents either a result or an error message
  - Apply the function to the boxed value, and wrap up the result in a new Either
  - Or simply propagate the error by wrapping up the error message

-
```
instance Functor (Either e) where
  fmap f (Left a)  = Left a
  fmap f (Right a) = Right (f a)
```

Recall: the map function for lists

- It takes a function and a list
  - Applies that function to each element in the list
    - *An example in Haskell:*

```
> map head ["bat", "cat", "dog"]
"bcd"
```

```
head :: [a] -> a
```

    - *Example 2:*

```
> map (take 2) ["bat", "cat", "dog"]
["ba", "ca", "do"]
```

```
(take 2) :: [a] -> [a]
```

Lists are functors

- fmap for functors

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

- map for lists

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

- Yes, list is a functor since it follows functor laws
  - In fact, the implementation of its fmap is just map

fmap id = id

```
> fmap id ["bat", "cat", "dog"]
["bat", "cat", "dog"]
```

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

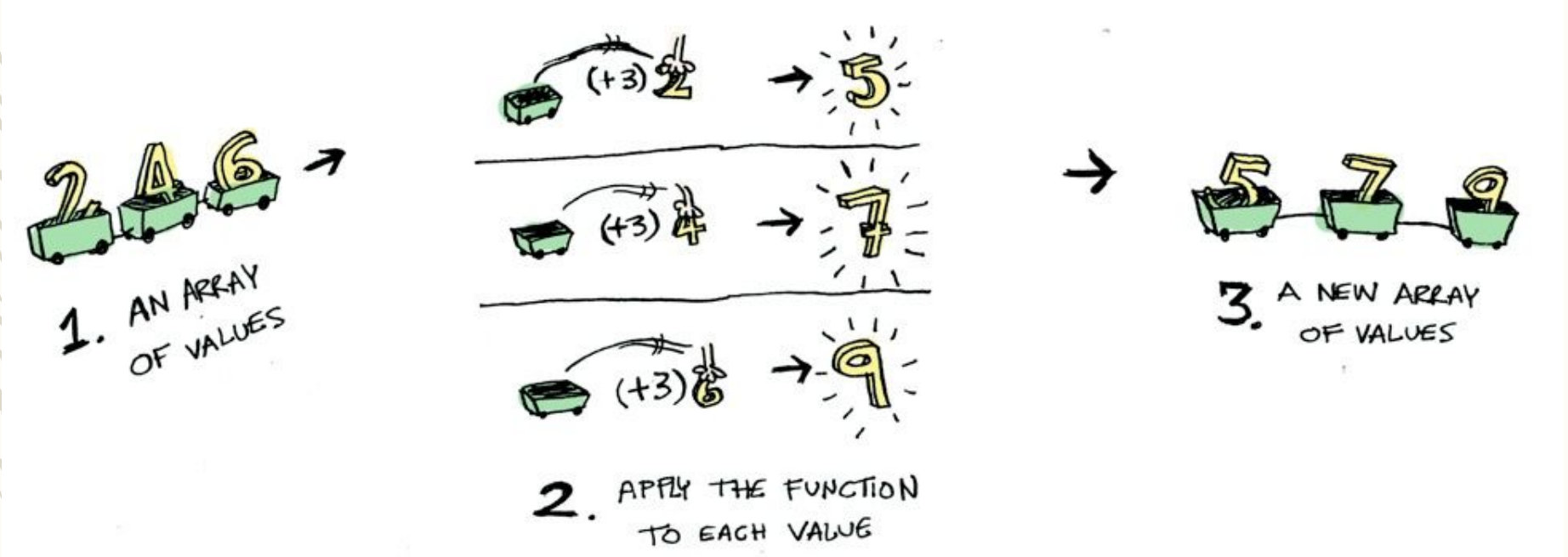
```
> fmap (sum . tail) [[1..3], [3..5], [5..7]]
[5, 9, 13]
> fmap sum (fmap tail [[1..3], [3..5], [5..7]])
[5, 9, 13]
```

the infix version of fmap in Haskell

```
> (sum . tail) <$> [[1..3], [3..5], [5..7]]
[5, 9, 13]
```

Lists are functors (cont.)

```
> fmap (+3) [2, 4, 6]
[5, 7, 9]
```



Reference

- http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

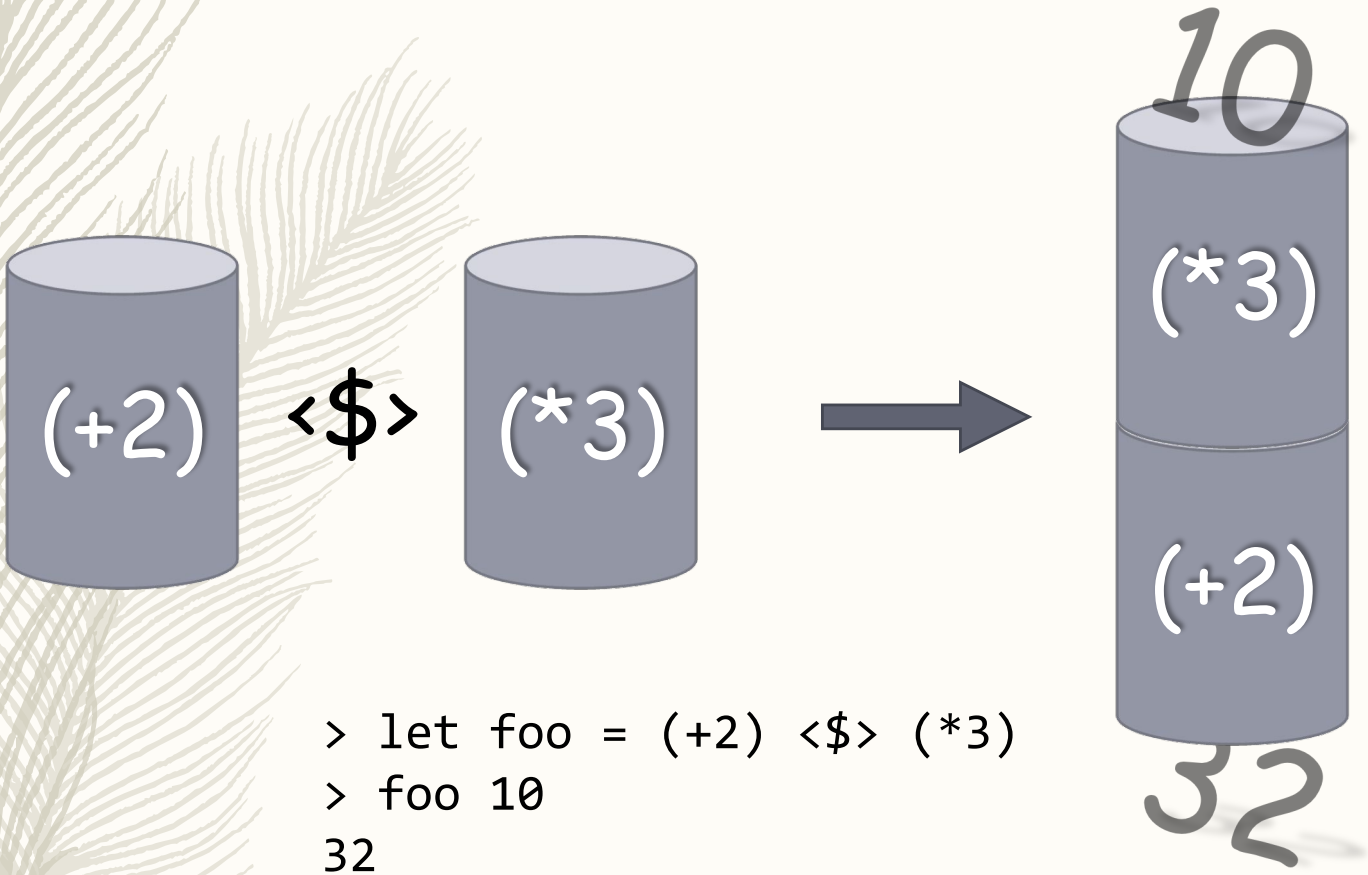
Functions are also functors!

– (*3) is a function


```
> (*3) 10
30
```

Functions are also functors! (cont.)

– Use <$> on two functions?




```
> let foo = (+2) <$> (*3)
> foo 10
32
```

Next lecture

11/11 **Practice 4**

11/18 Interface, Mixin, and Trait