Chasing Arrows

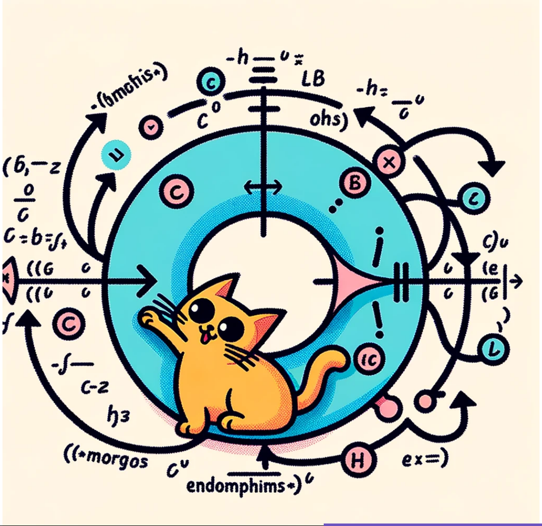
in Categories Containing Functors and Monads

Uli Fahrenberg Jim Newton

EPITA

Scala.IO February 2024

Cat chasing arrows

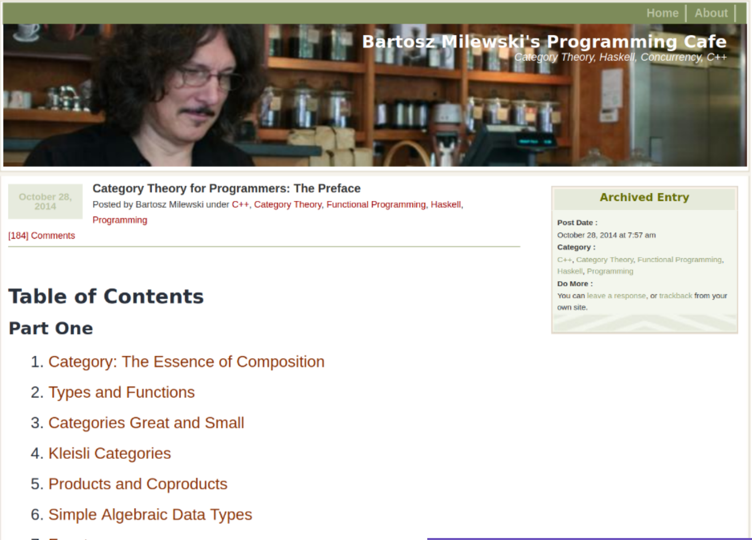


EPITA & CT4P

- EPITA: École Pour l'Informatique et les Techniques Avancées: private engineering school specialized in software engineering
- end of 3rd year / 1st year of engineering education:
  - students know some Assembler, C, C++, Java, Python, OCaml, Lisp, Haskell
  - also some algebra, analysis, linear algebra
  - time to choose an elective course
  - Jim & Uli: why not Scala, functional programming, & category theory
  - AKA cats through the back door
- Category Theory 4 Programmers:
  - 1. What are cats 2. Types & functions 3. Kleisli composition 4. Monads
- heavily inspired by Bartosz Milewski, but getting to the point much faster
- Here: Part 4: Monads

Bartosz Milewski: Category Theory for Programmers

Bartosz Milewski: Category Theory for Programmers


Home | About

Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

October 28, 2014

Category Theory for Programmers: The Preface

Posted by Bartosz Milewski under C++, Category Theory, Functional Programming, Haskell, Programming

[184] Comments

Table of Contents

Part One

Archived Entry

Post Date :
October 28, 2014 at 7:57 am

Category :
C++, Category Theory, Functional Programming, Haskell, Programming

Do More :
You can leave a response, or trackback from your own site.

Jim & Uli

Jim Newton

- University studies in mathematics and electrical engineering
- PhD from Sorbonne in CS
- 35 years as Lisp programmer in IC design industry
- Interested in type systems, functional programming, automata theory
- Scala user for $\approx$ 6 years
- Prof at EPITA since 2015

Uli Fahrenberg

- University studies in mathematics and computer science
- PhD in mathematics
- Worked at Aalborg University (DK), Rennes University, École polytechnique
- Interested in category theory, algebraic topology, automata theory, concurrency theory, verification
- Prof at EPITA since 2021

Function Composition

Function composition

$$\begin{tikzcd} & \texttt{Int} \arrow[dr, "g"] & \\ \texttt{Double} \arrow[ur, "f"] & & \texttt{String} \end{tikzcd}$$

```scala
def f(x:Double):Int = {
  x.round.toInt
}

def g(n:Int):String = {
  s"[$n]"
}
```

- two composable functions

Function composition

$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] & \\
\texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String}
\end{tikzcd}
$$

```scala
def f(x:Double):Int = {
  x.round.toInt
}

def g(n:Int):String = {
  s"[$n]"
}
```

```scala
def g_after_f(x:Double):String = {
  g(f(x))
}
```

- two composable functions
- ... and their composition

Function composition

$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] & \\
\texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String}
\end{tikzcd}
$$

```scala
def f(x:Double):Int = {
  x.round.toInt
}

def g(n:Int):String = {
  s"[$n]"
}
```

```scala
def after[X,Y,Z](g:Y=>Z,f:X=>Y):X=>Z = {
  x:X => g(f(x))
}

val g_after_f: Double=>String = after(g,f)
```

- two composable functions
- . . . and their composition
- . . . generalized

Function composition

$$\begin{tikzcd} & \texttt{Int} \arrow[dr, "g"] & \\ \texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String} \end{tikzcd}$$

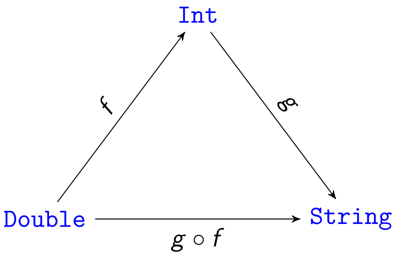
- *Always test your code*

```
f(3.2)
// 3

g(3)
// "[3]"

g(f(3.2))
// "[3]"

g_after_f(3.2)
// "[3]"

after(g,f)(3.2)
// "[3]"
```

Confusing notation

Attention: confusing notation. Which function is applied first? right-to-left *vs.* inner-to-outer.

- Mathematical notation:
  - Function: $g \circ f$ or $x \mapsto g(f(x))$
  - Application $(g \circ f)(x)$ or $g(f(x))$
  - Abuse: $\mathrm{map}(f)$, $\mathrm{flatMap}(f)$, $\mathrm{map}(f)(x)$, $\mathrm{flatMap}(f)(x)$
- Scala functional notation:
  - `g(f(x))` or `after(g,f)(x)`
- Scala OO notation:
  - `x.f().g()`, `x.map(f).map(g)`, `x.flatMap(f).flatMap(g)`



Introducing `Option`

$$\begin{tikzcd} & \texttt{Int} \arrow[dr, "g"] & \\ \texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String} \end{tikzcd}$$

$$\begin{tikzcd} & \texttt{Option[Int]} \arrow[dr, "g_2"] & \\ \texttt{Option[Double]} \arrow[ur, "f_2"] \arrow[rr, "g_2 \circ f_2"'] & & \texttt{Option[String]} \end{tikzcd}$$

Introducing Option

```scala
def f2(mx:Option[Double]):Option[Int] = {
  mx match {
    case Some(x) => Some(x.round.toInt)
    case None => None
  }
}

def g2(mn:Option[Int]):Option[String] = {
  mn match {
    case Some(n) => Some(s"[$n]")
    case None => None
  }
}
```

- naive implementation

$$
\begin{tikzcd}
 & \texttt{Option[Int]} \arrow[dr, "g_2"] & \\
\texttt{Option[Double]} \arrow[ur, "f_2"] \arrow[rr, "g_2 \circ f_2"'] & & \texttt{Option[String]}
\end{tikzcd}
$$

Introducing Option

```
def f2(mx:Option[Double]):Option[Int] = {
  mx match {
    case Some(x) => Some(f(x))
    case None => None
  }
}

def g2(mn:Option[Int]):Option[String] = {
  mn match {
    case Some(n) => Some(g(n))
    case None => None
  }
}
```

- naive implementation

$$\begin{tikzcd} & \texttt{Option[Int]} \arrow[dr, "g_2"] & \\ \texttt{Option[Double]} \arrow[ur, "f_2"] \arrow[rr, "g_2 \circ f_2"'] & & \texttt{Option[String]} \end{tikzcd}$$

Introducing Option

```
f2(Some(3.2))
// Some(3)

g2(Some(3))
// Some("[3]")

after(g2,f2)(3.2)
// Some("[3]")
```

$$\begin{tikzcd} & \texttt{Option[Int]} \arrow[dr, "g_2"] & \\ \texttt{Option[Double]} \arrow[ur, "f_2"] \arrow[rr, "g_2 \circ f_2"'] & & \texttt{Option[String]} \end{tikzcd}$$

- *testing testing*
- no need to implement composition!

Introducing Option

```scala
def f2(x:Option[Double]): Option[Int] = {
  x.map(f)
}

def g2(x:Option[Int]): Option[String] = {
  x.map(g)
}
```

$$
\begin{tikzcd}
 & \texttt{Option[Int]} \arrow[dr, "g_2"] & \\
\texttt{Option[Double]} \arrow[ur, "f_2"] \arrow[rr, "g_2 \circ f_2"'] & & \texttt{Option[String]}
\end{tikzcd}
$$

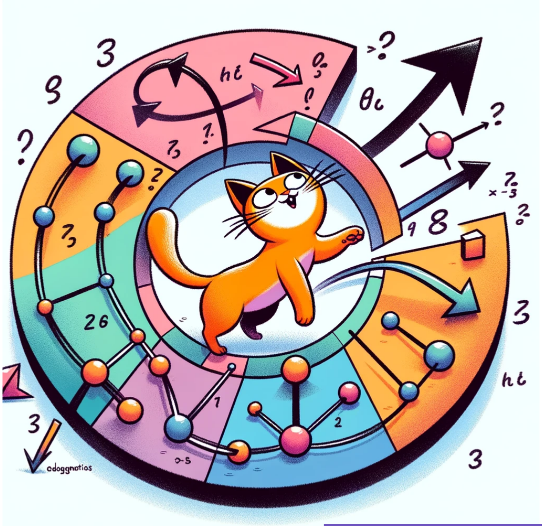
- not so naive
- Option is a functor!

Categories and Functors

What is a functor?

But first, what is a category?

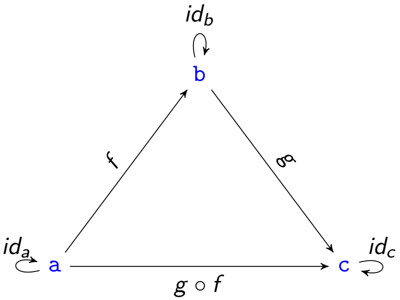
Another cat



What is a category?

A (small) *category* is a *set* of objects, together with a set of distinguishable morphisms between objects.

- with identity morphisms
  - $a \xrightarrow{\text{id}} a$
- and identifies composition (chaining) of morphisms
  - If $a \to b$ and $b \to c$ then $a \to c$.
  - If $a \xrightarrow{f} b$ and $b \xrightarrow{g}$ then $a \xrightarrow{g \circ f} c$.
- Composition is associative.



In Scala?

The set of Scala types is a *category*, and each function between types is a morphism.

$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] & \\
\texttt{Double} \arrow[ur, "f"] & & \texttt{String}
\end{tikzcd}
$$

In Scala?

The set of Scala types is a *category*, and each function between types is a morphism.

$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] & \\
\texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String}
\end{tikzcd}
$$

In Scala?

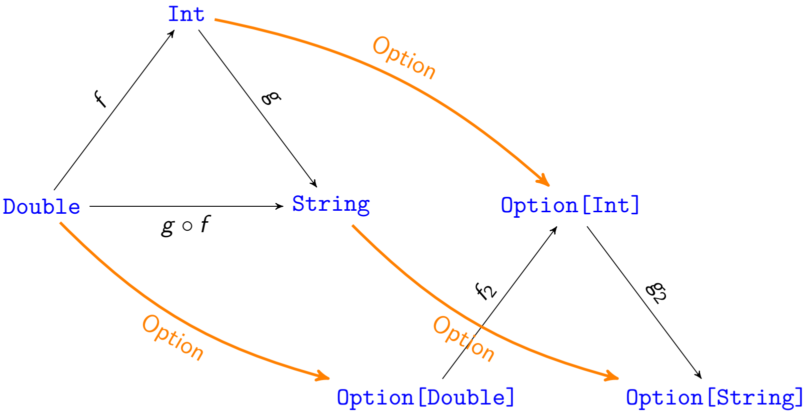
The set of Scala types is a *category*, and each function between types is a morphism.

$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] & \\
\texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String}
\end{tikzcd}
$$

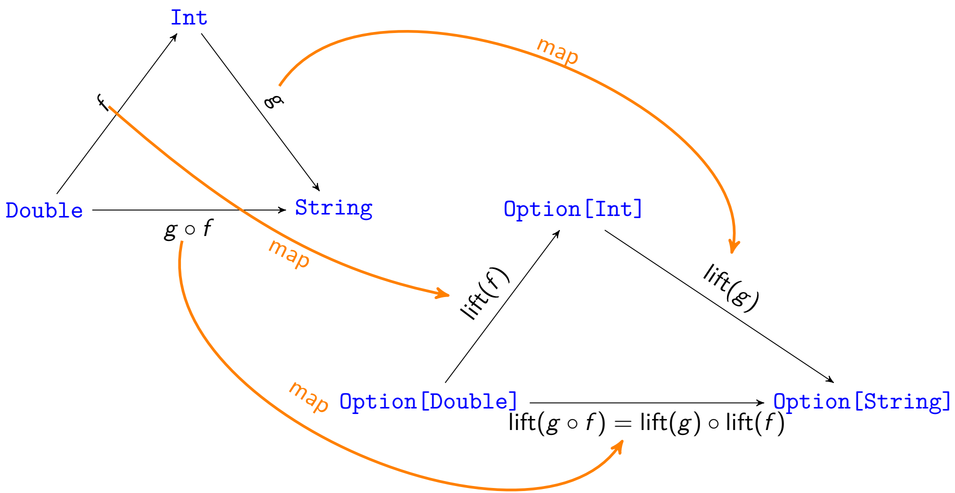
$$\texttt{x=>g(f(x))} = \texttt{x=>after(g,f)(x)}$$

Now, what is a functor?

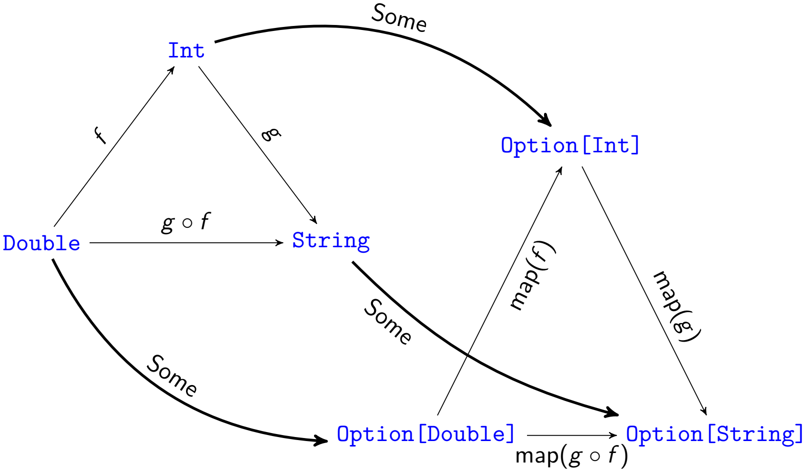
Option maps types.



`map` lifts functions.



`Some` maps objects.



What is a Functor?

Mathematical point of view:

A *Functor*, F,

. . . is a structure-preserving map between categories

- `Double` $\xrightarrow{\text{to}}$ `F[Double]`
- `Int` $\xrightarrow{\text{to}}$ `F[Int]`

lifts functions to functions:

$f \xrightarrow{\text{to}} \text{lift}(f)$

`Double=>Int` $\xrightarrow{\text{lift}}$ `F[Double]=>F[Int]`

Law: $\text{lift}(g \circ f) = \text{lift}(g) \circ \text{lift}(f)$

Law: $\text{lift}(\text{id}) = \text{id}$

Scala point of view:

A *Functor*, `Option`,

. . . is a parameterized type

- `Double` $\xrightarrow{\text{to}}$ `Option[Double]`
- `Int` $\xrightarrow{\text{to}}$ `Option[Int]`

which *correctly* implements `map`.

$f \xrightarrow{\text{lift}} (x \mapsto x.map(f))$

`Double=>Int` $\xrightarrow{\text{lift}}$ `Option[Double]=>Option[Int]`

Law: `x.map(after(g,f)) = x.map(f).map(g)`

Abbreviate as single *functor* mapping

$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] & \\
\texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & \texttt{String}
\end{tikzcd}
\quad \xrightarrow{F} \quad
\begin{tikzcd}
 & F\,[\texttt{Int}] \arrow[dr, "F(g)"] & \\
F\,[\texttt{Double}] \arrow[ur, "F(f)"] \arrow[rr, "F(g \circ f) = F(g) \circ F(f)"'] & & F\,[\texttt{String}]
\end{tikzcd}
$$

Examples of Functors

Example List functor

```
class List[T] {
  ...
  def map[S](f:T=>S):List[S] = {
    this match {
      h::t => f(h) :: t.map(f)  // List cons
      Nil => Nil
}}}
```

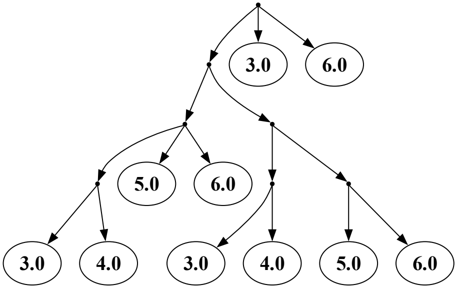
```
data = List(1.1, 2.3, 4.5)

data.map(f)             // List(1, 2, 4)
data.map(f).map(g)      // List("[1]", "[2]", "[4]")
data.map(after(g,f)) // List("[1]", "[2]", "[4]")
```

Example Tree functor

```scala
sealed abstract class Tree[A] {
  def map[B](f:A=>B): Tree[B]
}

case class Node[A](branches: List[Tree[A]]) extends Tree[A] {
  def map[B](f:A=>B): Node[B] =
    Node(branches.map(t => t.map(f)))
}

case class Leaf[A](data: A) extends Tree[A] {
  def map[B](f: A => B): Leaf[B] = Leaf(f(data))
}
```

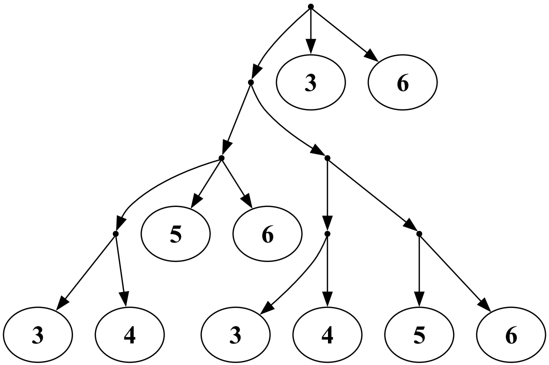
Example Tree functor

```
val l3 = Leaf(3.0)
val l4 = Leaf(4.0)
val l5 = Leaf(5.0)
val l6 = Leaf(6.0)
val doubleTree = Node(List(
  Node(List(Node(List(Node(List(l3, l4)),
                          l5, l6)),
                     Node(List(Node(List(l3,
                                         l4),
                               Node(List(l5,
                                         l6
      ))))))),
  l3,
  l6))
```



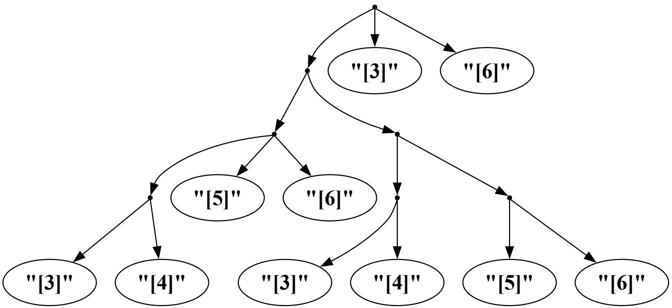
Example `Tree` functor

```
intTree = doubleTree.map(f)
```



Example Tree functor

```
stringTree = intTree.map(g)
stringTree = doubleTree.map(f).map(g) = doubleTree.map(after(g,f))
```

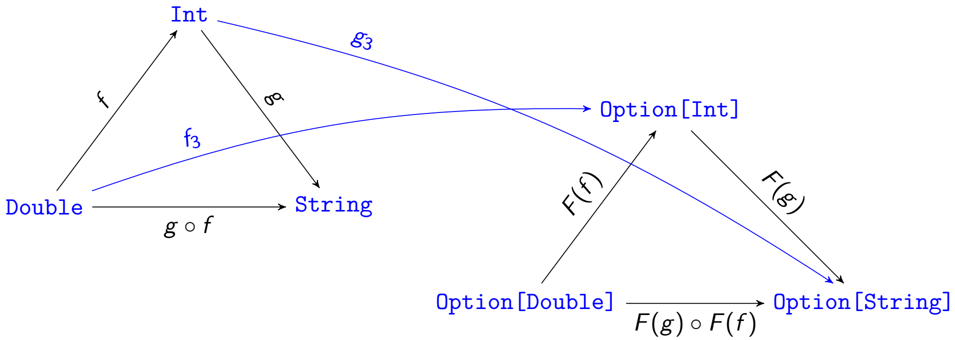


Kleisli Composition and Monads

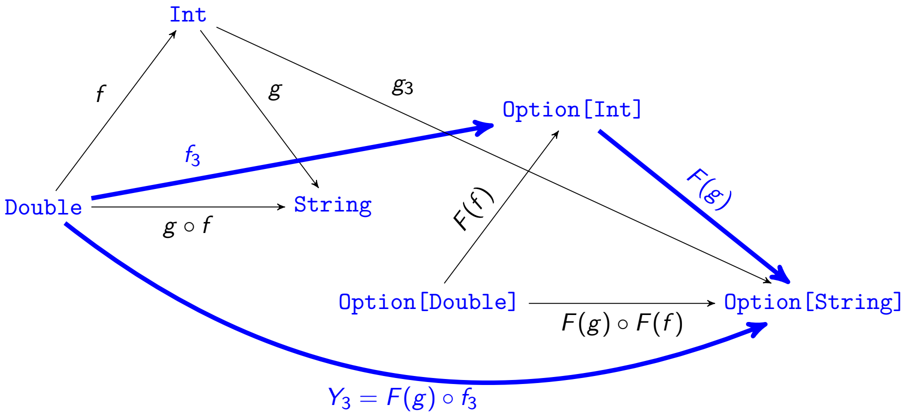
We know what a *functor* is.

What is a *monad*?

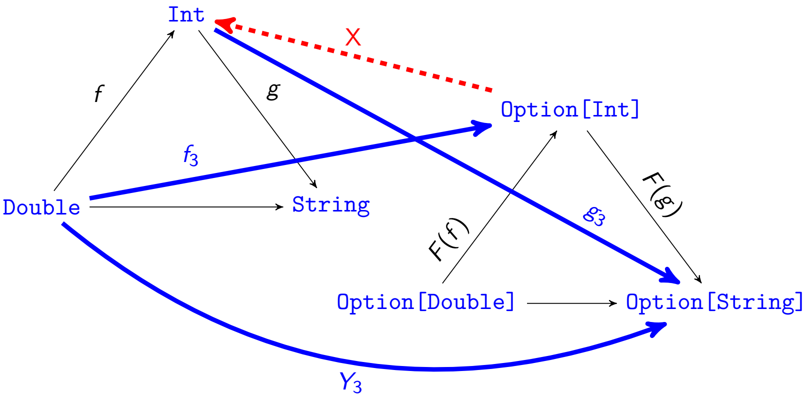
$$
\begin{tikzcd}
 & \texttt{Int} \arrow[dr, "g"] \arrow[rrrrdd, bend left, "g_3"] & & & & \\
\texttt{Double} \arrow[ur, "f"] \arrow[rr, "g \circ f"'] \arrow[rrrr, bend left, "f_3"] & & \texttt{String} & & \texttt{Option[Int]} \arrow[dr, "F(g)"] & \\
 & & & \texttt{Option[Double]} \arrow[ur, "F(f)"] \arrow[rr, "F(g) \circ F(f)"'] & & \texttt{Option[String]}
\end{tikzcd}
$$




```scala
def f3(x:Double):Option[Int] = Some(x.round.toInt)

def g3(n:Int):Option[String] = Some(s"[$n]")

f3(3.4) // Some(3)
g3(3)   // Some("[3]")
```

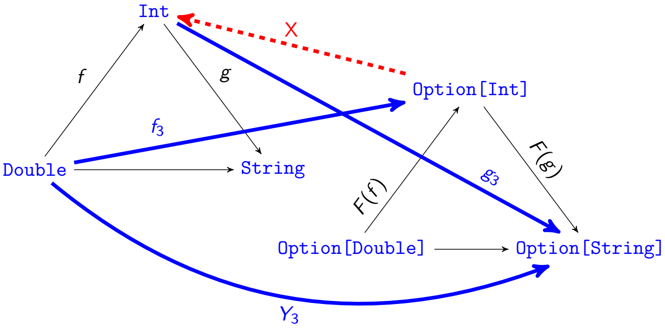


This works *only because* we have defined `f3` as a wrapper around `f`.



Challenge: construct X so that $Y_3 = \overbrace{g_3 \bullet f_3}^{\text{Kleisli composition}} = \overbrace{g_3 \circ X \circ f_3}^{\text{normal composition}}$

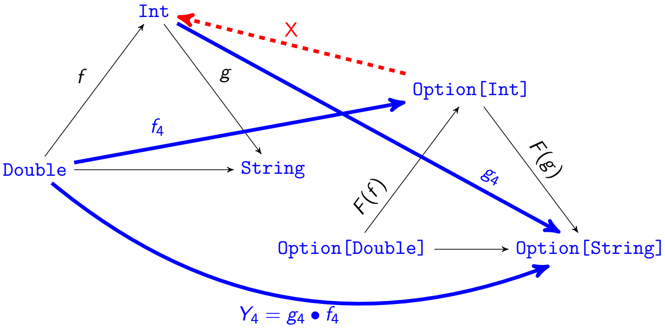
Dubious definition of X




```scala
def X(mx:Option[Int]):Int = {
  mx match {
    case Some(x) => x
    case None => 0
  }
}

def f3(x:Double):Option[Int] =
    Some(x.round.toInt)

def g3(n:Int):Option[String] =
    Some(s"[$n]")
```

This works accidentally, because `f3` never returns `None`.

Failure of X




```
def f4(x:Double):Option[Int] =
  if ( abs(x) <= MAX_INT)
    // Some(f(x))
    Some(x.round.toInt)
  else
    None

def g4(n:Int):Option[String] =
  if ( n >= 0)
    // Some(g(n))
    Some(s"[$n]")
  else
    None
```

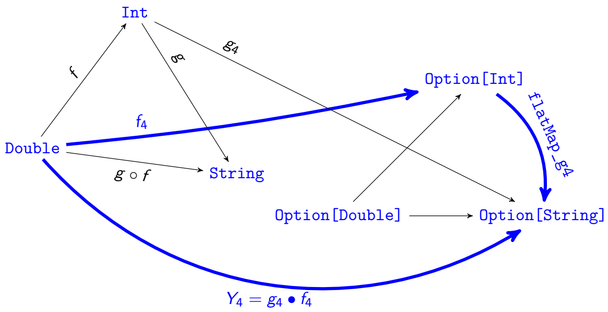
$Y_4 \neq g_4 \circ X \circ f_4$. Composition fails. *E.g.* $Y_4(10.0 \times \text{MAX\_INT}) =$ "[0]", expecting None.

Kleisli Composition

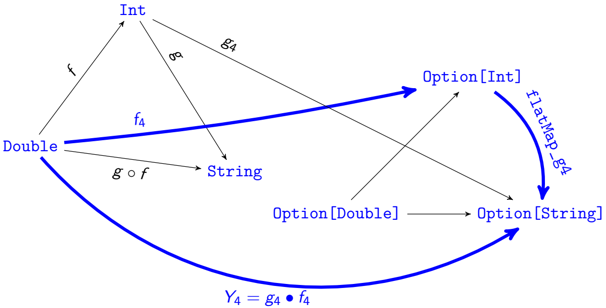
If we have an object of type `Option[Int]`, then we either have `Some(n)` or `None`.

- In the case that we have `Some(n)` we compute `Some(g3(n))`,
- otherwise we *compute* `None`.

```
def flatMap_g4(oi:Option[Int]):Option[String] = {
  oi match {
    case Some(i) => g4(i)
    case None => None
  }
}
```




```scala
def flatMap_g4(oi:Option[Int]
  ):Option[String] = {
    oi match {
      case Some(i) => g4(i)
      case None => None
    }
}
```

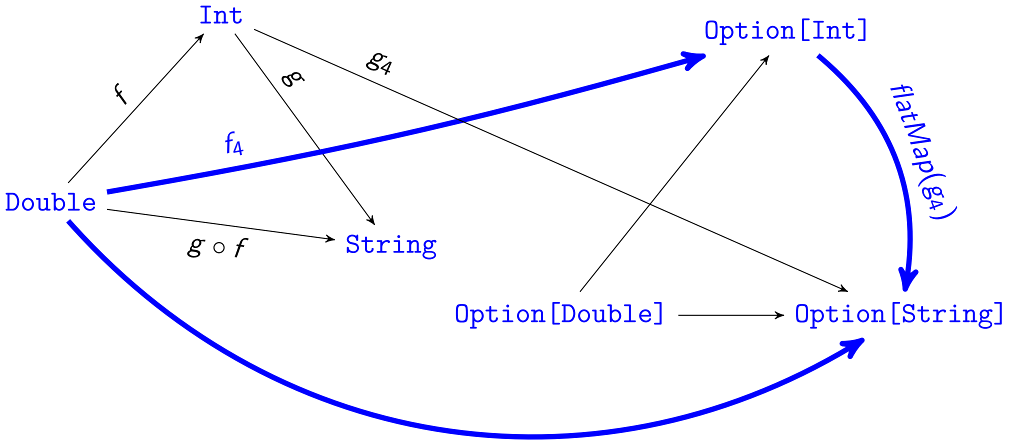



```scala
def flatMap_g4(oi:Option[Int]
  ):Option[String] = {
    oi match {
      case Some(i) => g4(i)
      case None => None
    }
}
```

We can avoid hard-coding g4 into flatMap.

Definition of flatMap

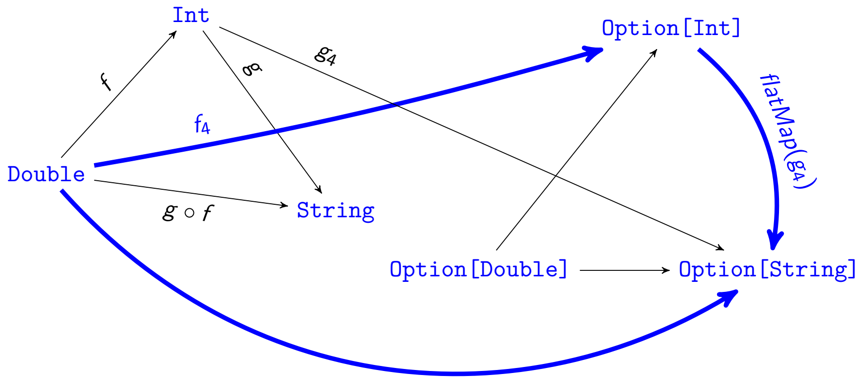
```
class Option[T] {
  ...
  def flatMap[S](f:T=>Option[S]):Option[S] = {
    this match {
      case Some(t) => f(t)
      case None => None
    }
  }
  ...
}

def flatMap_g4[T] = x:Option[T] => x.flatMap(g4)
```



$$Y_4 = g_4 \bullet f_4 = x \mapsto f_4(x).flatMap(g_4)$$

Finally: What is a monad?

A MONAD is a functor with a correctly defined `flatMap` *function which enforces* $g_4 \bullet f_4 = \text{flatMap}(g_4) \circ f_4$.

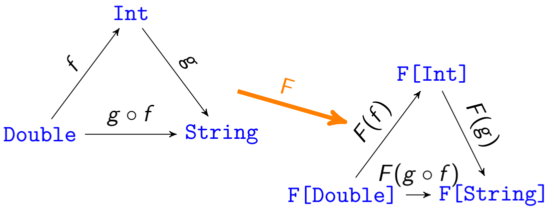


$Y_4 = g_4 \bullet f_4 = x \mapsto f_4(x).flatMap(g_4)$

Flatten

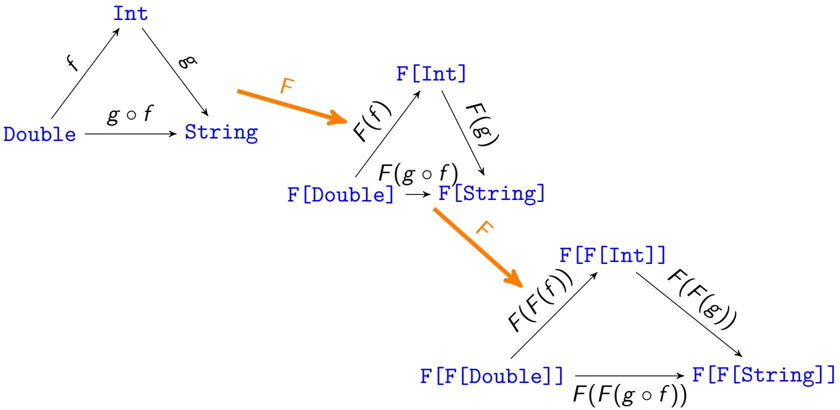
What is *flat* about `flatMap`?

Remember: `Option` is a functor.



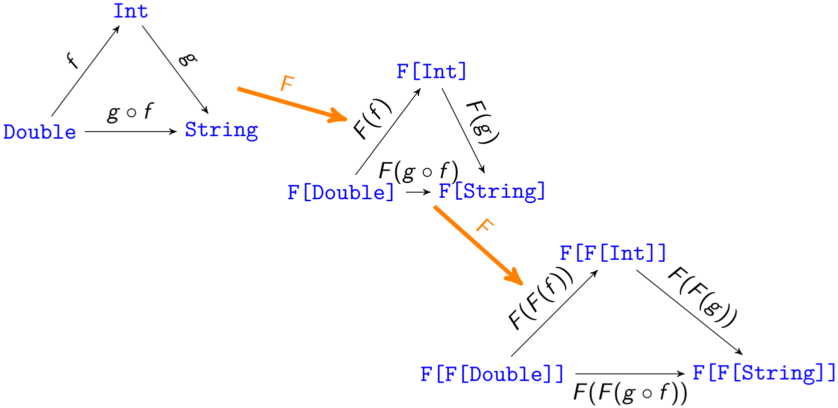
Remember: `Option` is a functor.

We can apply it to any type, including `Option[Double]`, `Option[Int]` etc.

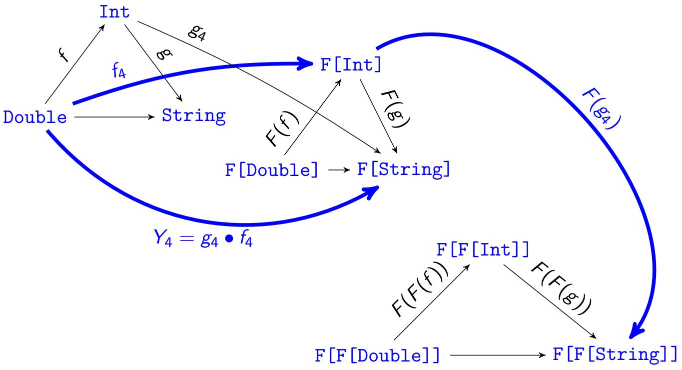


Remember: `Option` is a functor.

We can apply it to any type, including `Option[Double]`, `Option[Int]` etc.



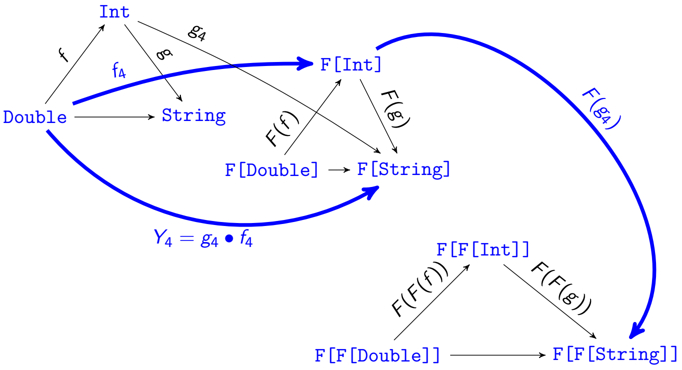
to get `Option[Option[Double]]`, `Option[Option[Int]]` etc.



We can *lift* g_4.

$\mathrm{lift}(g_4) = \mathrm{map}(g_4)$

Composes: $\mathrm{lift}(g_4) \circ f_4$.



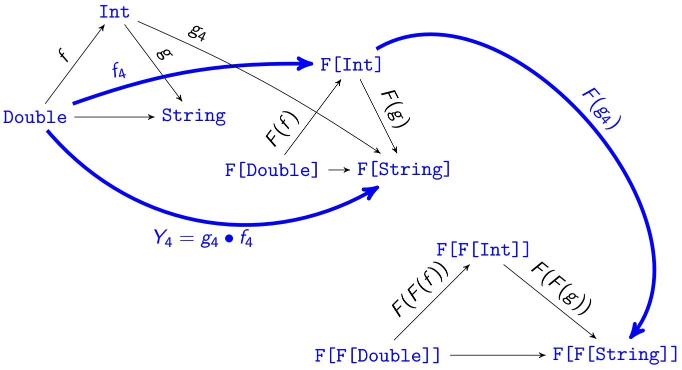
We can *lift* g_4.

$\text{lift}(g_4) = \text{map}(g_4)$

Composes: $\text{lift}(g_4) \circ f_4$.

Sadly:

`Double=>Option[Option[String]]`



We can *lift* g_4.

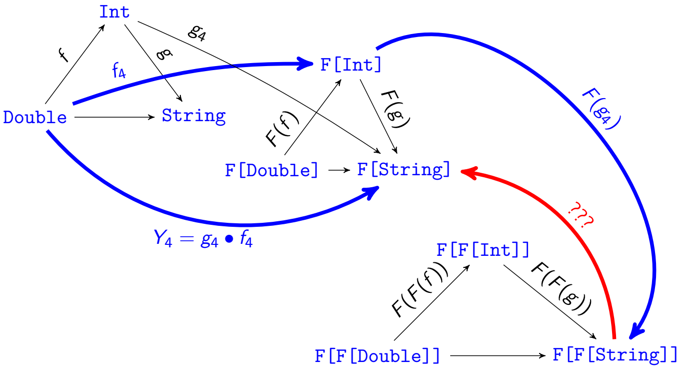
$\text{lift}(g_4) = \text{map}(g_4)$

Composes: $\text{lift}(g_4) \circ f_4$.

Sadly:

`Double=>Option[Option[String]]`

- Goal $Y_4 = g_4 \bullet f_4$: `Double=>Option[String]`

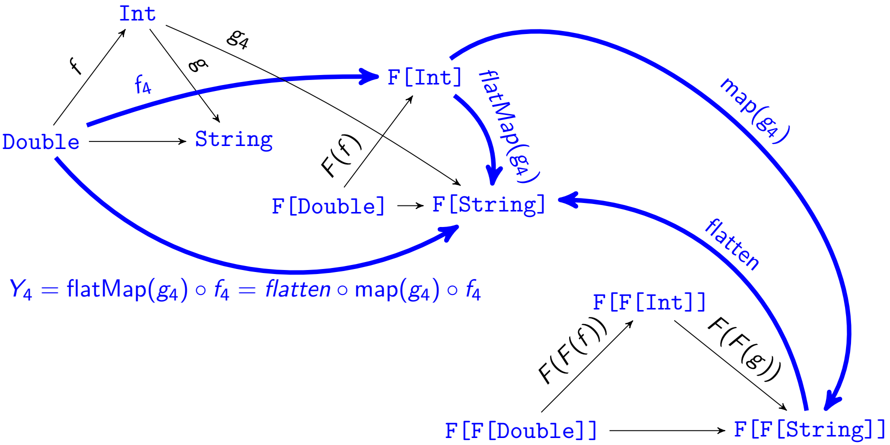


We can *lift* g_4.
$\mathrm{lift}(g_4) = \mathrm{map}(g_4)$
Composes: $\mathrm{lift}(g_4) \circ f_4$.
Sadly:
`Double=>Option[Option[String]]`

- Goal $Y_4 = g_4 \bullet f_4$: `Double=>Option[String]`
- We need ???: `Option[Option[String]] =>Option[String]`
- $g_4 \bullet f_4 =$??? $\circ\ \mathrm{lift}(g_4) \circ f_4$.

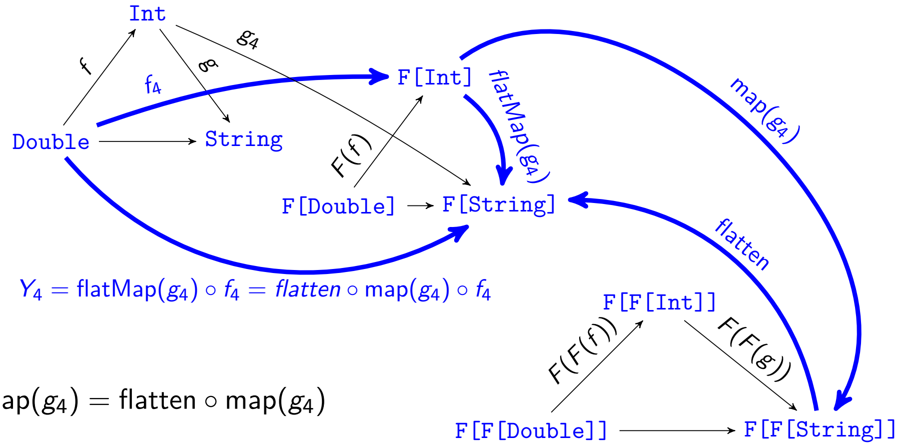
```
def flatten(oos:Option[Option[String]]):Option[String] = {
  oos match {
    case Some(Some(s)) => Some(s)
    case Some(None) => None
    case None => None
  }
}

flatten(Some(Some("[3]"))) // Some("[3]")
flatten(Some(None))        // None
flatten(None)              // None
```

What is flat about `flatMap`?



$Y_4 = \mathrm{flatMap}(g_4) \circ f_4 = \mathit{flatten} \circ \mathrm{map}(g_4) \circ f_4$

What is flat about `flatMap`?



$Y_4 = \text{flatMap}(g_4) \circ f_4 = \textit{flatten} \circ \text{map}(g_4) \circ f_4$

$\text{flatMap}(g_4) = \text{flatten} \circ \text{map}(g_4)$

$$\text{flatMap}(g_4) \circ f_4 = \text{flatten} \circ \text{map}(g_4) \circ f_4$$

$$\begin{aligned}
Y_4(3.4) &= (\text{flatMap}(g_4) \circ f_4)\,(3.4) \\
&= f_4(3.4).\text{flatMap}(g_4) \\
&= \mathit{Some}(3).\text{flatMap}(g_4) \\
&= \mathit{Some}(\texttt{"[3]"})
\end{aligned}$$

$$\begin{aligned}
Y_4(3.4) &= (\text{flatten} \circ \text{map}(g_4) \circ f_4)\,(3.4) \\
&= \text{flatten}(\text{map}(g_4)(f_4(3.4))) \\
&= \text{flatten}(f_4(3.4).\mathit{map}(g_4)) \\
&= \text{flatten}(\mathit{Some}(3).\mathit{map}(g_4)) \\
&= \text{flatten}(\mathit{Some}(\mathit{Some}(\texttt{"[3]"}))) & ;\, g_4(3) \to \mathit{Some}(\texttt{"[3]"}) \\
&= \mathit{Some}(\texttt{"[3]"})
\end{aligned}$$

`Option` flattening is lossy

We get `None` if `f4` fails or if `g4` fails, and we cannot distinguish from the final output.

$$
\begin{aligned}
Y_4(3.4 \times 10^{88}) &= \text{flatten}(f_4(3.4 \times 10^{88}).map(g_4)) && \\
&= \text{flatten}(None.map(g_4)) && ;\, f_4(3.4 \times 10^{88}) \Rightarrow \text{ None} \\
&= \text{flatten}(None) && \\
&= None && \\
Y_4(-3.4) &= \text{flatten}(f_4(-3.4).map(g_4)) && \\
&= \text{flatten}(Some(-3).map(g_4)) && \\
&= \text{flatten}(Some(None)) && ;\, g_4(-3) \Rightarrow \text{ None} \\
&= None &&
\end{aligned}
$$

Categorically speaking

A monad is

- an endofunctor $F : \mathcal{C} \to \mathcal{C}$
- together with a unit $\nu : \mathrm{Id} \to F$ `Some`
- and a multiplication $\mu : FF \to F$ `flatten`

Axioms:

$$\begin{tikzcd} F^3 \arrow[r, "F\mu"] \arrow[d, "\mu F"'] & F^2 \arrow[d, "\mu"] \\ F^2 \arrow[r, "\mu"'] & F \end{tikzcd}$$

double flatten in any order

$$\begin{tikzcd} F \arrow[r, "F\nu"] \arrow[d, "\nu F"'] \arrow[dr, equal] & F^2 \arrow[d, "\mu"] \\ F^2 \arrow[r, "\mu"'] & F \end{tikzcd}$$

$Some(None) = None$

Categorically speaking

A monad is

- an endofunctor $F : \mathcal{C} \to \mathcal{C}$
- together with a unit $\nu : \mathrm{Id} \to F$
- and a multiplication $\mu : FF \to F$

`Some`

`flatten`

Axioms:

$$\begin{tikzcd} F^3 \arrow[r, "F\mu"] \arrow[d, "\mu F"'] & F^2 \arrow[d, "\mu"] \\ F^2 \arrow[r, "\mu"'] & F \end{tikzcd} \qquad \begin{tikzcd} F \arrow[r, "F\nu"] \arrow[d, "\nu F"'] \arrow[dr, equal] & F^2 \arrow[d, "\mu"] \\ F^2 \arrow[r, "\mu"'] & F \end{tikzcd}$$

- flatMap $=$ flatten $\circ$ map flatten $=$ flatMap(map(id))

Examples of Monads

The Option Monad

```
class Option[T] {
  def map[S](f:T=>S):Option[S] = {
    this match {
      Some(s) => Some(f(s))
      None => None
    }
  }

  def flatMap[S](f:T=>Option[S]):Option[S] = {
    this match {
      case Some(t) => f(t)
      case None => None
    }
  }
  ...
}
```

The List Monad

```
class List[T] {
  def map[S](f:T=>S):List[S] = {
    this match {
      h::t => f(h) :: t.map(f)  // List cons
      Nil => Nil
    }
  }

  def flatMap[S](f:T=>List[S]):List[S] = {
    this match {
      h::t => f(h) ::: t.flatMap(f) // List append
      Nil => Nil
    }
  }
  ...
}
```

The Xyzzy Monad

```scala
class Xyzzy[T] {
  def map[S](f:T=>S):Xyzzy[S] = {
    ...
  }

  def flatMap[S](f:T=>Xyzzy[S]):Xyzzy[S] = {
    ...
  }
  ...
}
```

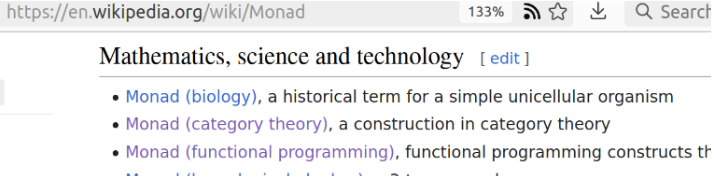
Conclusion

- CT4P: fast introduction to categories & functors
  - 7 hours of lectures, 2 hours exercise class, plus 6 hours of Bartosz Milewski
- in order to be able to talk about Scala, monads, and monads

Conclusion

- CT4P: fast introduction to categories & functors
  - 7 hours of lectures, 2 hours exercise class, plus 6 hours of Bartosz Milewski
- in order to be able to talk about Scala, monads, and monads

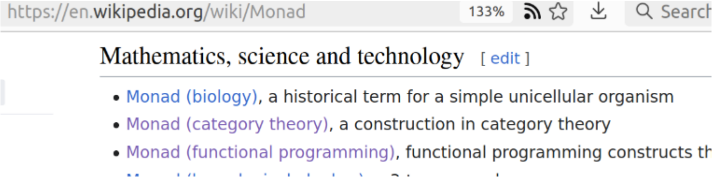
same thing! $\longrightarrow$



Conclusion

- CT4P: fast introduction to categories & functors
  - 7 hours of lectures, 2 hours exercise class, plus 6 hours of Bartosz Milewski
- in order to be able to talk about Scala, monads, and monads

same thing! ⟶



- things we didn't talk about: universal constructions, natural transformations, limits, colimits, adjoints, algebraic data types, etc.
- (Bartosz has monads in Chapter 20; we did them as Chapter 5)