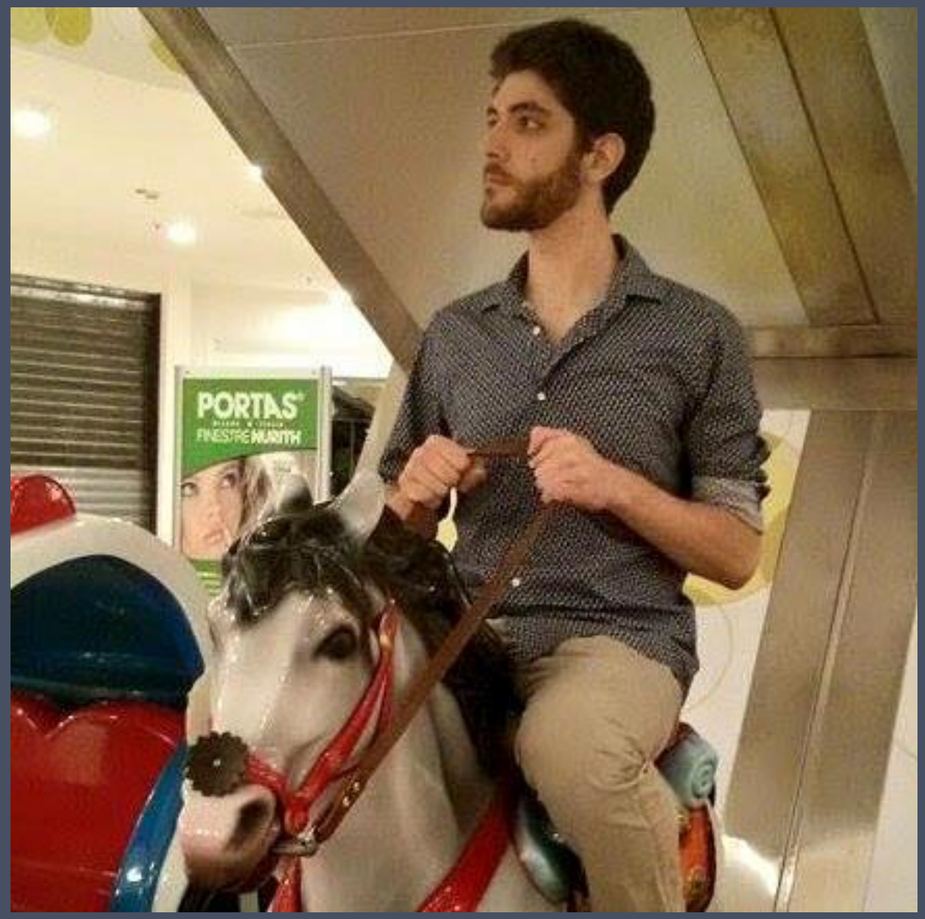
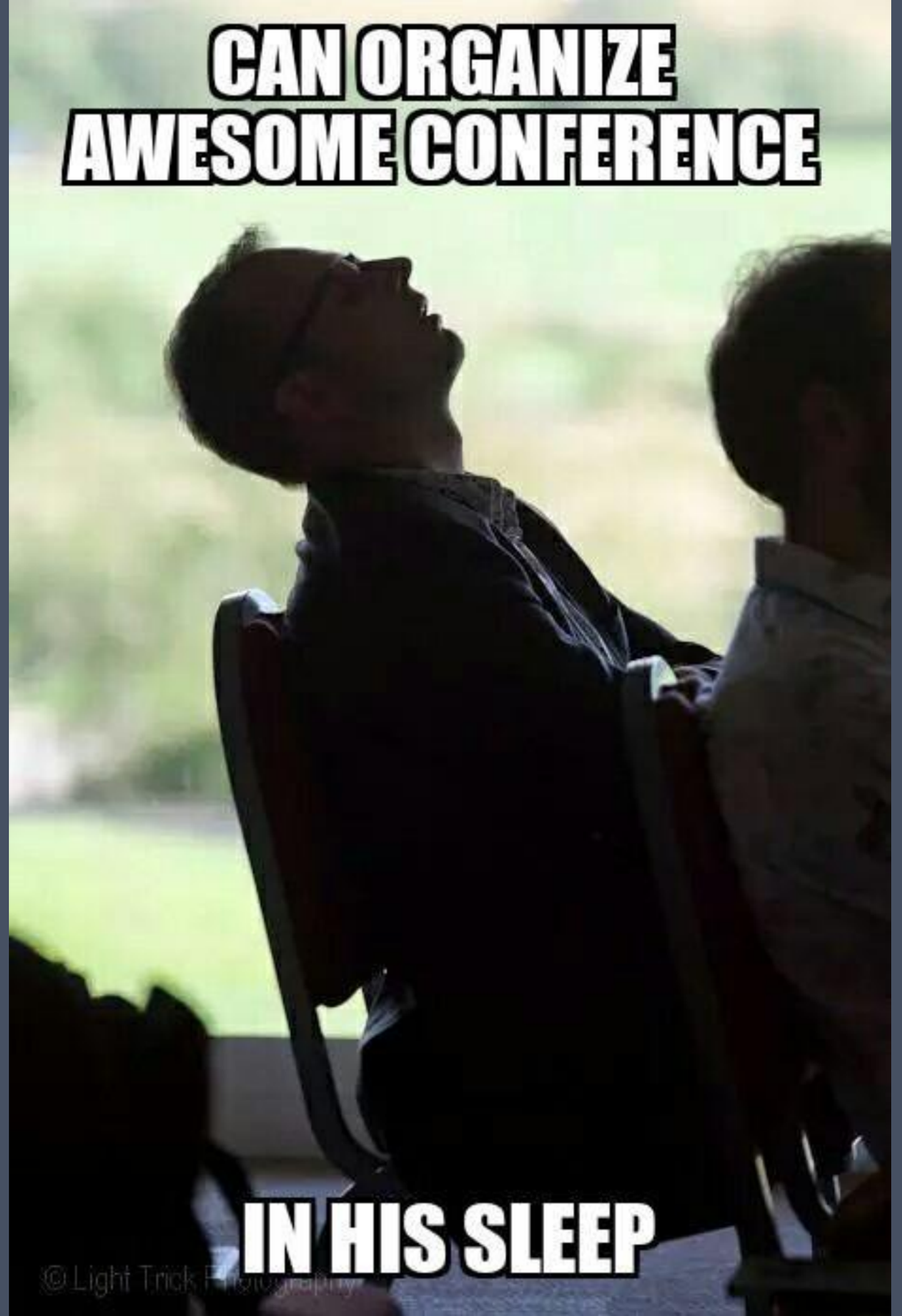

GABRIELE PETRONELLA
MONAD TRANSFORMERS
JUST. WHY?

ME. HI!



STUFF I DO


CAN ORGANIZE
AWESOME CONFERENCE
IN HIS SLEEP

A QUESTION

Mapping Scala Futures

2

Let's say I have a `Future[Seq[Int]]` which I want to convert to `Future[Seq[String]]`. Currently I'm doing it like this:

```
val futureSeqString = futureSeqInt.map( x => x.map(_.toString()))
```

This works but the nested map seems a bit awkward. The equivalent conversion for `Future[Option[Int]]` is slightly better but it still doesn't feel like I'm doing it the best way:

```
val futureOptionString = futureOptionInt.map {
  case Some(x) => x.toString();
  case _ => None;
}
```

Is there a better way of dealing with this?

scala

share edit close flag

edited May 8 at 16:26

Peter Neyens

5,154 ● 4 ● 24

asked May 8 at 11:56

d80tb7

35 ● 4

Option has a map method, so you could (I think should) do the conversion in the same way you've done it for Seq. – Steve Waldman May 8 at 12:32

add a comment

start a bounty

THE PROBLEM

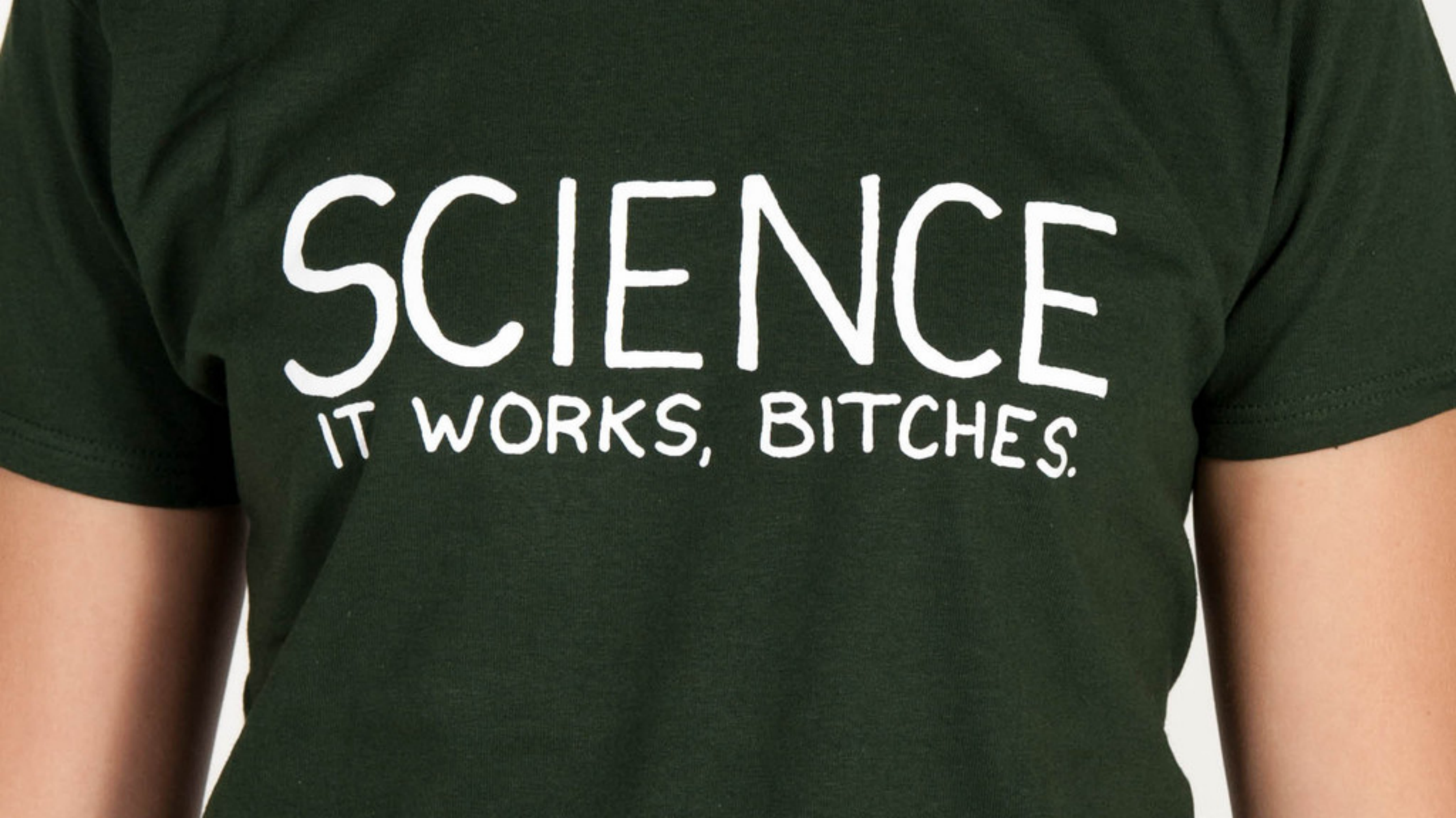
```
val x: Future[List[Int]] = ???

futureList.map(list => list.map(f))
               ^                 ^
               |_________________|
```

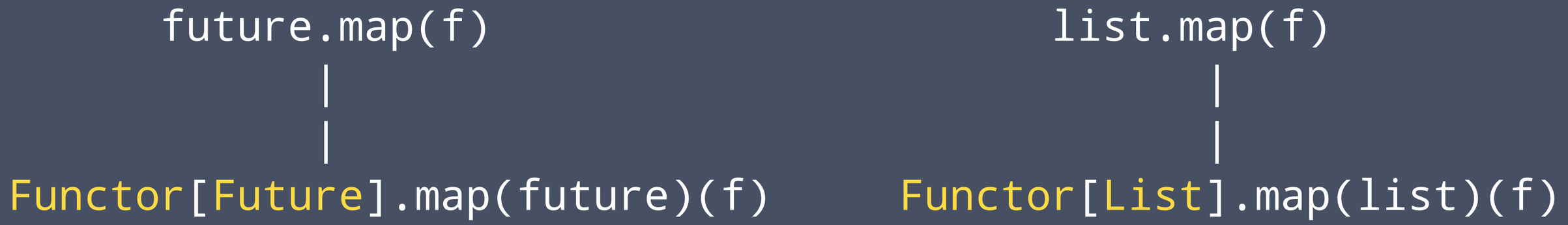
2 MAPS 1 FUNCTION

CAN WE DO BETTER?

INDENT!

```
futureList.map { list =>
  list.map(f)
}
```


SCIENCE
IT WORKS, BITCHES.

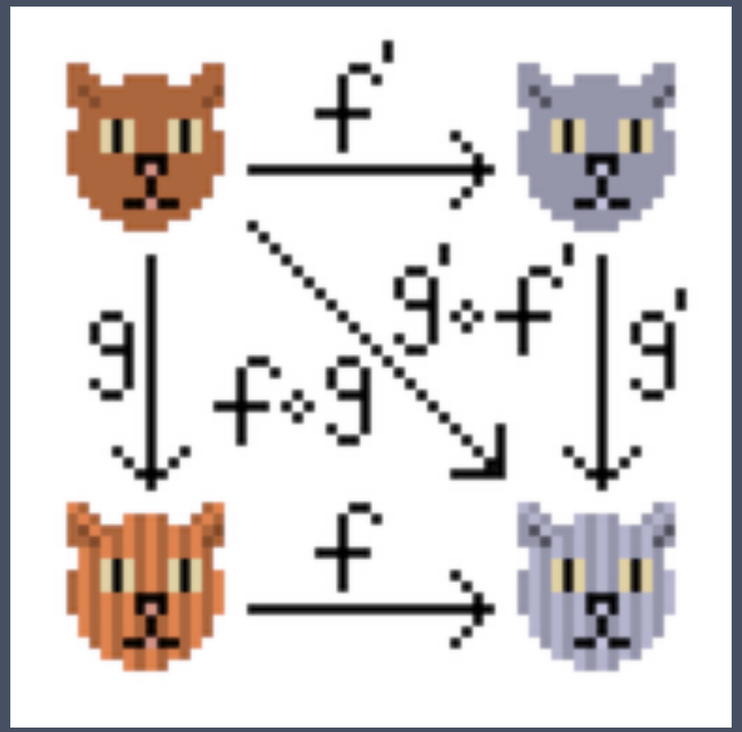

future.map(f)
Functor[Future].map(future)(f)
list.map(f)
Functor[List].map(list)(f)

```
futureList.map(f) // not really
        |
        |
Functor[Future[List]].map(futureList)(f)
```

IN PRACTICE

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import cats._; import std.future._; import std.list._

// create a `Functor[Future[List]]`
val futureListF = Functor[Future].compose(Functor[List])

val data: Future[List[Int]] = Future(List(1, 2, 3))

// only one map!
futureListF.map(data)(_ + 1) // Future(List(2, 3, 4))
```

CATS



HTTPS://GITHUB.COM/TYPELEVEL/CATS

ABOUT FLATTENING

```
List(1, 2, 3).map(_ + 1) // List(2, 3, 4)

List(1, 2, 3).map(n => List.fill(n)(n))
// List(List(1), List(2, 2), List(3, 3, 3))

List(1, 2, 3).map(n => List.fill(n)(n)).flatten
// List(1, 2, 2, 3, 3, 3)
```

FLATMAP

```
flatten ∘ map = flatMap
```

SO

```
List(1, 2, 3).map(n => List.fill(n)(n)).flatten
```

==

```
List(1, 2, 3).flatMap(n => List.fill(n)(n))
```

IN OTHER WORDS

WHEN LIFE GIVES YOU `F[F[A]]`
YOU PROBABLY WANTED `flatMap`

E.G.

```
val f: Future[Future[Int]] = Future(42).map(x => Future(24))

val g: Future[Int] = Future(42).flatMap(x => Future(24))
```

A LESS CONTRIVED EXAMPLE

```
def getUser(name: String): Future[User]
def areFriends(a: User, b: User): Boolean

val f: Future[Boolean] =
  getUser("Gabriele").flatMap(
    gab => getUser("Giovanni").map(
      gio => areFriends(gab, gio)
    )
  )
```

LET'S COMPREHEND THIS

```
val f: Future[Boolean] =
  for {
    gab <- getUser("Gabriele")
    gio <- getUser("Giovanni")
  } yield areFriends(gab, gio)
```

LESSONS

MONADS ALLOW SEQUENTIAL EXECUTION

MONADS CAN SQUASH `F[F[A]]` INTO `F[A]`

QUESTIONS?

WAIT A SECOND...

WHAT ABOUT F[G[X]]

BACK TO THE REAL WORLD

```
def getUser(name: String): Future[User] // <- really?
def areFriends(a: User, b: User): Boolean
```

BACK TO THE REAL WORLD

```
def getUser(name: String): Future[Option[User]] // better
def areFriends(a: User, b: User): Boolean
```

UH. OH...

```
val f: Future[Boolean] =
  for {
    gab <- getUser("Gabriele")
    gio <- getUser("Giovanni")
  } yield areFriends(gab, gio) // 😰 FAIL
```

EVENTUALLY

```
val f: Future[Option[Boolean]] =
  for {
    gab <- getUser("Gabriele")
    gio <- getUser("Giovanni")
  } yield areFriends(gab.get, gio.get) // 😱
```


GREAT SUCCESS!

DO YOU EVEN YIELD. BRO?

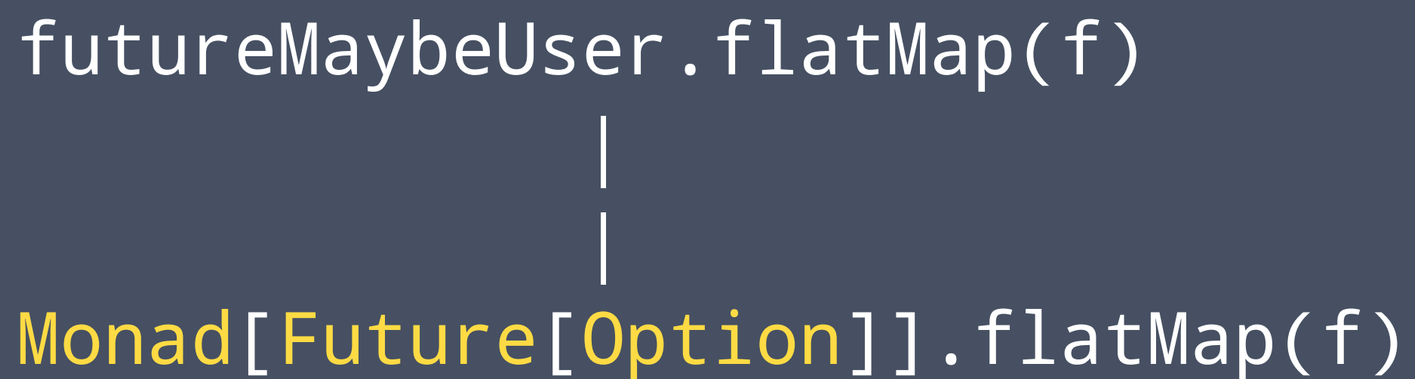
```
val f: Future[Option[Boolean]] =
  for {
    maybeGab <- getUser("Gabriele")
    maybeGio <- getUser("Giovanni")
  } yield for {
    gab <- maybeGab
    gio <- maybeGio
  } yield areFriends(gab, gio) // 😳
```

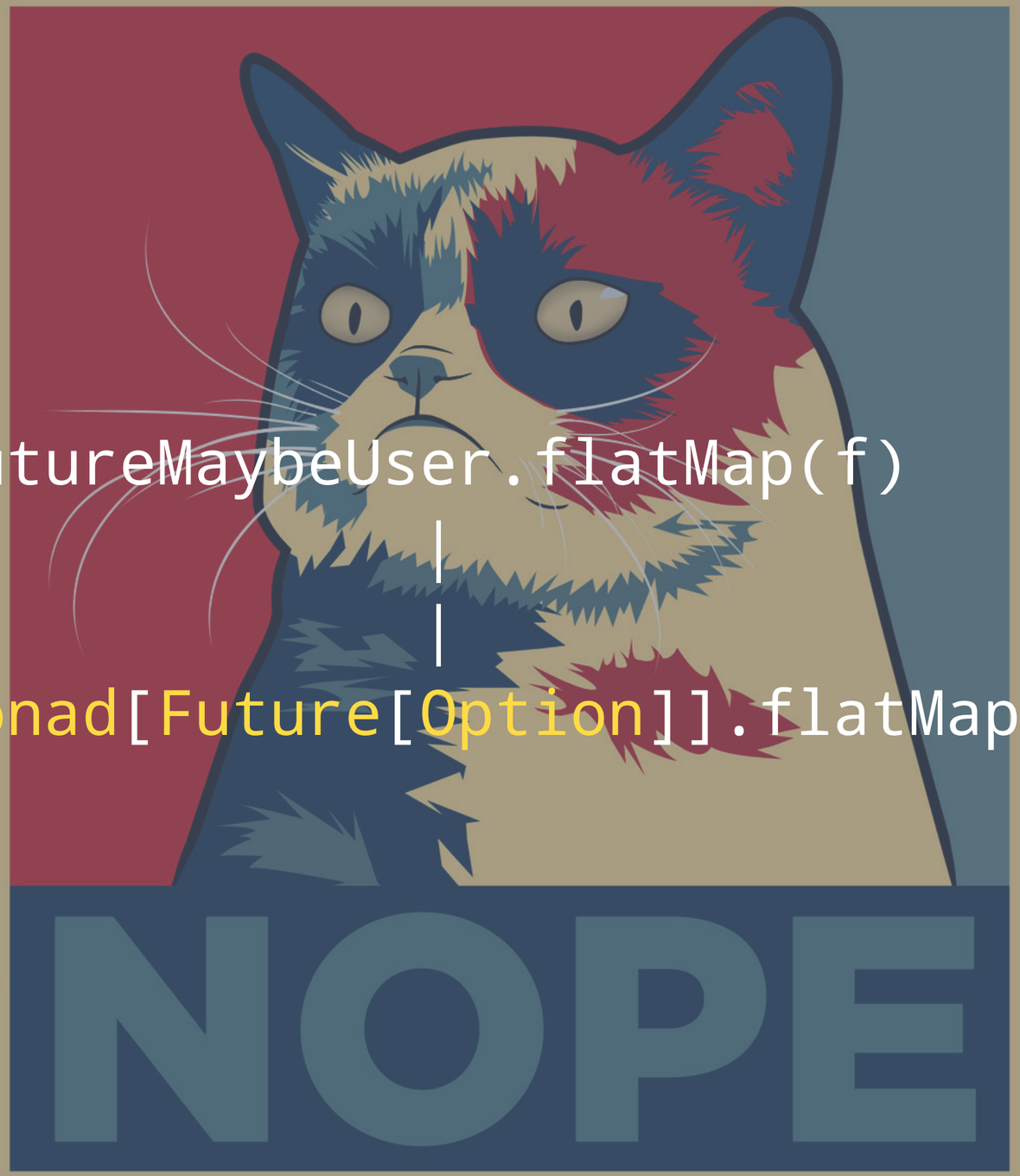
DO YOU EVEN MATCH. BRO?

```
val f: Future[Option[Boolean]] =
  for {
    maybeGab <- getUser("Gabriele")
    maybeGio <- getUser("Giovanni")
  } yield (maybeGab, maybeGio) match {
    case (Some(gab), Some(gio)) => Some(areFriends(gab, gio))
    case _ => None
  } // 😵
```

```
futureUser.flatMap(f)
          |
          |
Monad[Future].flatMap(futureUser)(f)
```

```
maybeUser.flatMap(f)
          |
          |
Monad[Option].flatMap(maybeUser)f)
```


futureMaybeUser.flatMap(f)
|
|
Monad[Future[Option]].flatMap(f)




```
futureMaybeUser.flatMap(f)
                |
                |
Monad[Future[Option]].flatMap(f)
```

MONADS DO NOT COMPOSE

HTTP://BLOG.TMORRIS.NET/POSTS/MONADS-DO-NOT-COMPOSE/


MBC2 HD
컴백하다는?

WHAT'S THE IMPOSSIBLE PART?

```
// trivial
def compose[F[_]: Functor, G[_]: Functor]: Functor[F[G[_]]] = ✅

// impossible
def compose[M[_]: Monad, N[_]: Monad]: Monad[M[N[_]]] = 🔥


// (not valid scala, but you get the idea)
```

MONADS DO NOT COMPOSE

GENERICALLY

BUT YOU CAN COMPOSE THEM SPECIFICALLY

```scala
case class FutOpt[A](value: Future[Option[A]])
```

```scala
new Monad[FutOpt] {

  def pure[A](a: => A): FutOpt[A] = FutOpt(a.pure[Option].pure[Future])

  def map[A, B](fa: FutOpt[A])(f: A => B): FutOpt[B] =
    FutOpt(fa.value.map(optA => optA.map(f)))

  def flatMap[A, B](fa: FutOpt[A])(f: A => FutOpt[B]): FutOpt[B] =
    FutOpt(fa.value.flatMap(opt => opt match {
      case Some(a) => f(a).value
      case None => (None: Option[B]).pure[Future]
    }))

}
```

AND USE

```
val f: FutOpt[Boolean] =
  for {
    gab <- FutOpt(getUser("Gabriele"))
    gio <- FutOpt(getUser("Giovanni"))
  } yield areFriends(gab, gio) // 🎉

val g: Future[Option[Boolean]] = f.value
```

WHAT IF

```
def getUsers(query: String): List[Option[User]]
```

```scala
case class ListOpt[A](value: List[Option[A]])
```

```scala
new Monad[ListOpt] {

  def pure[A](a: => A): ListOpt[A] = ListOpt(a.pure[Option].pure[List])

  def map[A, B](fa: ListOpt[A])(f: A => B): ListOpt[B] =
    ListOpt(fa.value.map(optA => optA.map(f)))

  def flatMap[A, B](fa: ListOpt[A])(f: A => ListOpt[B]): ListOpt[B] =
    ListOpt(fa.value.flatMap(opt => opt match {
      case Some(a) => f(a).value
      case None => (None: Option[B]).pure[List]
    }))

}
```

```scala
new Monad[FutOpt] {

  def pure[A](a: => A): FutOpt[A] = FutOpt(a.pure[Option].pure[Future])

  def map[A, B](fa: FutOpt[A])(f: A => B): FutOpt[B] =
    FutOpt(fa.value.map(optA => optA.map(f)))

  def flatMap[A, B](fa: FutOpt[A])(f: A => FutOpt[B]): FutOpt[B] =
    FutOpt(fa.value.flatMap(opt => opt match {
      case Some(a) => f(a).value
      case None => (None: Option[B]).pure[Future]
    }))

}
```

MEET OptionT

OptionT[[F[_], A]

```
val f: OptionT[Future, Boolean] =
  for {
    gab <- OptionT(getUser("Gabriele"))
    gio <- OptionT(getUser("Giovanni"))
  } yield areFriends(gab, gio) // 🎉

val g: Future[Option[Boolean]] = f.value
```

ANOTHER EXAMPLE

```
def getUser(id: String): Future[Option[Int]] = ???
def getAge(user: User): Future[Int] = ???
def getNickname(user: User): Option[String] = ???

val lameNickname: Future[Option[String]] = ???
// e.g. Success(Some("gabro27"))
```

I KNOW THE TRICK!

```
val lameNickname: Future[Option[String]] =
  for {
    user <- OptionT(getUser("123"))
    age  <- OptionT(getAge(user))  // sorry, nope
    name <- OptionT(getName(user)) // sorry, neither
  } yield s"$name$age"
```

DO YOU EVEN LIFT. BRO?

```
val lameNickname: Future[Option[String]] =
  for {
    user <- OptionT(getUser("123"))
    age  <- OptionT.liftF(getAge(user))
    name <- OptionT.fromOption(getName(user))
  } yield s"$name$age"
```

EXAMPLE: UPDATING A USER

- CHECK USER EXISTS
- CHECK IT CAN BE UPDATED
- UPDATE IT

THE NAIVE WAY

```
def checkUserExists(id: String): Future[Option[User]]
def checkCanBeUpdated(u: User): Future[Boolean]
def updateUserOnDb(u: User): Future[User]
```

PROBLEMS

```
def updateUser(u: User): Future[Option[User]] =
  checkUserExists.flatMap { maybeUser =>
    maybeUser.map { user =>
      checkCanBeUpdated(user).flatMap { canBeUpdated =>
        if (canBeUpdated) {
          updateUserOnDb(u)
        } else {
          Future(None)
        }
      }
    }
  }
```

MORE PROBLEMS (SPECIFIC ERRORS)

```
case class MyError(msg: String)
def updateUser(user: User): Future[Xor[MyError, User]] =
  checkUserExists(user.id).flatMap { maybeUser =>
    maybeUser.map { user =>
      checkCanBeUpdated(user).flatMap { canBeUpdated =>
        if (canBeUpdated) {
          updateUserOnDb(u)
        } else {
          Future(MyError("user cannot be updated"))
        }
      }
    }.getOrElse(MyError("user not existing"))
  }
```

MORE TRANSFORMERS

`XorT[F[_], A, B]`

HOW ABOUT

```
case class MyError(msg: String)
type Result[+A] = Xor[MyError, A]
type ResultT[F[_], A] = XorT[F, MyError, A]]
type FutureResult[A] = ResultT[Future, A]
```

BETTER?

```
def checkUserExists(id: String): FutureResult[User] = Future {
  if (id === "123")
    User("123").right
  else
    MyError("sorry, no user").left
}

def checkCanBeUpdated(u: User): FutureResult[User] = ???

def updateUserOnDb(u: User): FutureResult[User] = ???
```

BETTER?

```
def updateUser(user: User): FutureResult[User] = for {
  u <- checkUserExists(user.id)
  _ <- checkCanBeUpdated(u)
  updatedUser <- updateUser(user)
} yield updatedUser
```

PERSONAL TIPS

TIP #1

STACKING MORE THAN TWO MONADS
GETS BAD REALLY QUICKLY

EXAMPLE (FROM DJSPIEWAK/EMM)

```
val effect: OptionT[EitherT[Task, String, ?], String] = for {
  first <- readName.liftM[EitherT[?[_], String, ?]].liftM[OptionT]
  last <- readName.liftM[(EitherT[?[_], String, ?]].liftM[OptionT]

  name <- if ((first.length * last.length) < 20)
    OptionT.some[EitherT[Task, String, ?], String](s"$first $last")
  else
    OptionT.none[EitherT[Task, String, ?], String]

  _ <- (if (name == "Daniel Spiewak")
    EitherT.fromDisjunction[Task](\/.left[String, Unit]("your kind isn't welcome here"))
  else
    EitherT.fromDisjunction[Task](\/.right[String, Unit](()))).liftM[OptionT]

  _ <- log(s"successfully read in $name").liftM[EitherT[?[_], String, ?]].liftM[OptionT]
} yield name
```

TIP #2

KEEP YOUR TRANSFORMERS FOR YOUSELF

```
def publicApiMethod(x: String): OptionT[Future, Int] = 😰
def publicApiMethod(x: String): Future[Option[Int]] = 😊
```

BY THE WAY

```
val x: OptionT[Future, Int] = OptionT(Future(Option(42)))
val y: Future[Option[Int]] = x.value // Future(Option(42))
```

MONAD TRANSFORMERS: TAKEAWAYS

- THEY END WITH `T`
- `F[G[X]]` BECOMES `GT[F[_], X]`
- CAN BE STACKED UNDEFINITELY. BUT GETS AWKWARD
- THEY ARE A TOOL FOR STACKING EFFECTS

WHAT ELSE?

FREE MONADS

- CLEARLY SEPARATE STRUCTURE AND INTERPRETATION
- EFFECTS ARE SEPARATED FROM PROGRAM DEFINITION

HTTPS://GITHUB.COM/TYPELEVEL/CATS/BLOB/MASTER/DOCS/SRC/MAIN/TUT/FREEMONAD.MD

EFF

https://github.com/atnos-org/eff-cats

"Extensible effects are an alternative to monad transformers for computing with effects in a functional way"

Based on Freer monads, more extensible effects by Oleg Kiselyov

EMM

https://github.com/djspiewak/emm

OTHERWISE KNOWN AS "LESS CONFUSING MONAD TRANSFORMERS"

HTTP://SLACK.SCALA-ITALY.IT/

WORK@BUILDO.IO

questionsT

@GABRO27
@BUILDO
@MILANOSCALA
@SCALAITALY