From Functor Composition to Monad Transformers

https://github.com/hermannhueck/monad-transformers

Abstract

In a List[Option[A]] or Future[Option[A]] we want to access the value of type A conveniently without nested mapping and flatMapping.

We can avoid nested mapping with composed Functors and composed Applicatives.

Functors compose, Applicatives compose, but Monads do not!

What can we do?

Monad transformers to the rescue!

After going through the composition of Functors and Applicatives I show how 2 Monads are bolted together with a Monad transformer and how to use this construct. I demonstrate this with the Option transformer OptionT and will end up with best practices.

Agenda

1. Nested Monads in for comprehensions
2. Functors compose.
3. Aside: Creating List[A => B] and List[Option[A => B]]
4. Applicatives compose.
5. Do Monads compose?
6. Monad Transformers - Example OptionT
7. Generic Monad Processing
8. Stacking Monads and Transformers
9. Making the Types Fit
10. Monad Transformers in Cats
11. Best Practices
12. OptionT - Implementation
13. Resources

1. Nested Monads in for comprehensions

Imports for (all of) Cats

```scala
import cats._, cats.data._, cats.implicits._
// import mycats._, transform._, Functor.ops._, Monad.ops._
```

Imports for my own implementation of

Functor, Applicative, Monad and OptionT

```scala
// import cats._, cats.data._, cats.implicits._
import mycats._, transform._, Functor.ops._, Monad.ops._
```

The same client code works with Cats and with my implementation.
Only the imports need to be changed!

For comprehension for List[Option[Int]]

```
val loi1: List[Option[Int]] = List(Some(1), None, Some(2), Some(3))
val loi2: List[Option[Int]] = List(Some(1), None, Some(2), Some(3))

val result1: List[Option[Int]] =
  for {
    oi1: Option[Int] <- loi1
    if oi1.isDefined
    oi2: Option[Int] <- loi2
    if oi2.isDefined
  } yield Option(oi1.get * oi2.get)

// result1: List[Option[Int]] = List(Some(1), Some(2),
//   Some(3), Some(2), Some(4), Some(6), Some(3), Some(6), Some(9))
```

Compiler translates for comprehension to

flatMap, map (and withFilter)

```
val result2 =
loi1
  .filter(_.isDefined)
  .flatMap { oi1 =>
    loi2
      .filter(_.isDefined)
      .map { oi2 =>
        Option(oi1.get * oi2.get)
      }
  }

// result2: List[Option[Int]] = List(Some(1), Some(2),
//   Some(3), Some(2), Some(4), Some(6), Some(3), Some(6), Some(9))
```

A nested for comprehension

```
val result3: List[Option[Int]] =
  for {
    oi1: Option[Int] <- loi1
    oi2: Option[Int] <- loi2
  } yield for {
    i1: Int <- oi1
    i2: Int <- oi2
  } yield i1 * i2

// result3: List[Option[Int]] = List(Some(1), None,
//   Some(2), Some(3), None, None, None, None, Some(2),
//   None, Some(4), Some(6), Some(3), None, Some(6), Some(9))
```

Wanted:

a for comprehension to process Ints nested in 2 contexts

… something like this one:

```
// This code does not compile!
val result4: List[Option[Int]] =
  for {
    i1: Int <- loi1
    i2: Int <- loi2
  } yield i1 * i2
```

… or like this one:

```
// How can we create such a ListOption?
val result4: ListOption[Int] =
  for {
    i1: Int <- loi1
    i2: Int <- loi2
  } yield i1 * i2
```

To be usable in a for comprehension our "ListOption" must provide "map", "flatMap" (and "withFilter") operations. Hence it must be a Functor (for map) as well a Monad (for flatMap).

Every Monad is also a Functor.

But lets first look at Functors. (It's the simpler part.)

2. Functors compose.

Recap: What is a Functor? (1/2)

```
// The gist of typeclass Functor

trait Functor[F[_]] {

  def  map[A, B](fa: F[A])(f: A => B): F[B]

}
```

Recap: What is a Functor? (2/2)

```scala
// Simplified typeclass Functor
trait Functor[F[_]] {

  // --- intrinsic abstract functions

  def  map[A, B](fa: F[A])(f: A => B): F[B]


  // --- concrete functions implemented in terms of map

  def fmap[A, B](fa: F[A])(f: A => B): F[B] =
    map(fa)(f)     // alias for map

  def compose[G[_]: Functor]: Functor[Lambda[X => F[G[X]]]] = ???

  // ... other functions ...
}
```

Mapping a List[Int] with a Functor[List]

```scala
val li = List(1, 2, 3)

val lSquared1 = li.map(x => x * x) // invokes List.map
// lSquared1: List[Int] = List(1, 4, 9)

val lSquared2 = li.fmap(x => x * x) // invokes Functor[List].fmap
// lSquared2: List[Int] = List(1, 4, 9)

val lSquared3 = Functor[List].map(li)(x => x * x) // invokes Functor[List].map
// lSquared3: List[Int] = List(1, 4, 9)
```

Mapping a List[Option[Int]] with Functor[List] and Functor[Option]

```
val loi = List(Some(1), None, Some(2), Some(3))

val loSquared1 = loi.map(oi => oi.map(x => x * x))
// loSquared1: List[Option[Int]] = List(Some(1), None, Some(4), Some(9))

val loSquared2 = Functor[List].map(loi)(oi => Functor[Option].map(oi)(x => x * x))
// loSquared2: List[Option[Int]] = List(Some(1), None, Some(4), Some(9))

val loSquared3 = (Functor[List] compose Functor[Option]).map(loi)(x => x * x)
// loSquared3: List[Option[Int]] = List(Some(1), None, Some(4), Some(9))

val cf = Functor[List] compose Functor[Option]
val loSquared4 = cf.map(loi)(x => x * x)
// loSquared4: List[Option[Int]] = List(Some(1), None, Some(4), Some(9))
```

Pimping List[Option[A]]

```
implicit class PimpedListOptionA[A](list: List[Option[A]]) {
  val composedFunctor = Functor[List] compose Functor[Option]
  def map[B](f: A => B): List[Option[B]] = composedFunctor.map(list)(f)
  def fmap[B](f: A => B): List[Option[B]] = composedFunctor.map(list)(f)
}

val loSquared5 = loi.fmap(x => x * x)
// loSquared5: List[Option[Int]] = List(Some(1), None, Some(4), Some(9))

// Attention!!! fmap works but map still doesn't work!!!

val loSquared6 = loi.map(x => x * x)
<console>:25: error: value * is not a member of Option[Int]
       val loSquared6 = loi.map(x => x * x)

val loSquared7 = for { x <- loi } yield x * x
<console>:22: error: value * is not a member of Option[Int]
       val loSquared7 = for { x <- loi } yield x * x
```

3. Aside:

Creating List[A => B]

and List[Option[A => B]]

Creating a List[Int => Int]

```
val lf1_1 = List((x:Int) => x * 1, (x:Int) => x * 2, (x:Int) => x * 3)
// lf1_1: List[Int => Int] = List($$Lambda$..., $$Lambda$..., $$Lambda$...)

val lf1_2 = List((_:Int) * 1, (_:Int) * 2, (_:Int) * 3)
// lf1_2: List[Int => Int] = List($$Lambda$..., $$Lambda$..., $$Lambda$...)
```

Creating a List[Int => Int] from List[Int]

```
val lf1_3 = List(1, 2, 3).map(x => (y:Int) => y * x)
// lf1_3: List[Int => Int] = List($$Lambda$..., $$Lambda$..., $$Lambda$...)

val lf1_4 = List(1, 2, 3).map(x => (_:Int) * x)
// lf1_4: List[Int => Int] = List($$Lambda$..., $$Lambda$..., $$Lambda$...)
```

Creating a List[Option[Int => Int]]

```
val lf1 = lf1_4
// lf1: List[Int => Int] = List($$Lambda$..., $$Lambda$..., $$Lambda$...)

val lof1 = lf1 map Option.apply
// lof1: List[Option[Int => Int]] =
//   List(Some($$Lambda$...), Some($$Lambda$...), Some($$Lambda$...))
```

4. Applicatives compose.

Recap: What is an Applicative? (1/2)

```scala
// The gist of typeclass Applicative

trait Applicative[F[_]] extends Functor[F] {

  def pure[A](a: A): F[A]

  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
}
```

Recap: What is an Applicative? (2/2)

```scala
// Simplified typeclass Applicative
trait Applicative[F[_]] extends Functor[F] {

  // --- intrinsic abstract Applicative methods
  def pure[A](a: A): F[A]
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]

  // --- method implementations in terms of pure and ap

  def ap2[A, B, Z](ff: F[(A, B) => Z])(fa: F[A], fb: F[B]): F[Z] = ???

  override def map[A, B](fa: F[A])(f: A => B): F[B] = ap(pure(f))(fa)

  def map2[A, B, Z](fa: F[A], fb: F[B])(f: (A, B) ⇒ Z): F[Z] = ???

  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)] = ???

  def tuple2[A, B](fa: F[A], fb: F[B]): F[(A, B)] = product(fa, fb)

  def compose[G[_]: Applicative]: Applicative[Lambda[X => F[G[X]]]] = ???

  // ... other functions ...
}
```

Applying a List[Int => Int] to a List[Int] with Applicative

```
val liResult = Applicative[List].ap(lf1)(li)
// liResult: List[Int] = List(1, 2, 3, 2, 4, 6, 3, 6, 9)
```

Applying a List[Option[Int => Int]]

to a List[Option[Int]] with Applicative

```
val ca = Applicative[List] compose Applicative[Option]
val loiResult = ca.ap(lof1)(loi)
// loiResult: List[Option[Int]] = List(Some(1), None, Some(2), Some(3),
//   Some(2), None, Some(4), Some(6), Some(3), None, Some(6), Some(9))
```

5. Do Monads compose?

Recap: What is a Monad? (1/2)

```scala
// The gist of typeclass Monad

trait Monad[F[_]] extends Applicative[F] {

  def pure[A](a: A): F[A]

  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

Recap: What is a Monad? (2/2)

```scala
// Simplified typeclass Monad
trait Monad[F[_]] extends Applicative[F] {

  // --- intrinsic abstract functions
  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]

  // --- concrete functions implemented in terms of flatMap and pure

  override def map[A, B](fa: F[A])(f: A => B): F[B] = flatMap(fa)(a => pure(f(a)))

  def fmap[A, B](fa: F[A])(f: A => B): F[B] = map(fa)(f)  // alias for map

  def flatten[A](ffa: F[F[A]]): F[A] = flatMap(ffa)(identity)

  override def ap[A, B](ff: F[A => B])(fa: F[A]): F[B] = ???

  // ... other functions ...
}
```

Do Monads compose?

```
val cf = Functor[List] compose Functor[Option]
// cf: cats.Functor[[α]List[Option[α]]] = cats.Functor$$anon$1@386ed52a

val ca = Applicative[List] compose Applicative[Option]
// ca: cats.Applicative[[α]List[Option[α]]] = cats.Alternative$$anon$1@69102316

val cm = Monad[List] compose Monad[Option]
// cm: cats.Applicative[[α]List[Option[α]]] = cats.Alternative$$anon$1@48b5e9dd
```

Trying to compose 2 generic Monads

We cannot implement flatMap without knowing the higher kinded type of the inner Monad.

```
// Hypothetical composition
def composeFWithGImpossible[F[_]: Monad, G[_]: Monad] = {

  type Composed[A] = F[G[A]]

  new Monad[Composed] {

    def pure[A](a: A): Composed[A] = Monad[F].pure(Monad[G].pure(a))

    def flatMap[A, B](fa: Composed[A])(f: A => Composed[B]): Composed[B] = ???
    // !!! Problem! How do we write flatMap? Impossible to implement
    // !!! generically without knowledge of the type of the inner Monad!
  }
}
```

Trying to compose 1 generic and 1 concrete Monad

Knowing the higher kinded type of the inner Monad (Option) we can implement flatMap.

```
def composeFWithOption[F[_]: Monad] = {

  type Composed[A] = F[Option[A]]

  new Monad[Composed] {

    def pure[A](a: A): Composed[A] =  Monad[F].pure(Monad[Option].pure(a))

    def flatMap[A, B](fOptA: Composed[A])(f: A => Composed[B]): Composed[B] =
      Monad[F].flatMap(fOptA) {
        case None => Monad[F].pure(Option.empty[B])
        case Some(a) => f(a)
      }
  }
}
```

Do Monads compose?

- Functors compose.
- Applicatives compose.
- **Monads do <u>NOT</u> compose!**

Do Monads compose?

- Functors compose.
- Applicatives compose.
- **Monads do <u>NOT</u> compose!**

Solution: Monad Transformers

Do Monads compose?

- Functors compose.
- Applicatives compose.
- **Monads do NOT compose!**

Solution: Monad Transformers

Monad Transformers are some kind of composition mechanism for Monads.

Do Monads compose?

- Functors compose.
- Applicatives compose.
- **Monads do <u>NOT</u> compose!**

Solution: Monad Transformers

Monad Transformers are some kind of composition mechanism for Monads.

The inner Monad must be concrete, e.g. Option or Either.
The outer Monad is left abstract. F[_]: Monad

6. Monad Transformers - Example OptionT

What is OptionT?

- OptionT is a generic case class which encapsulates an F[Option[A]].
- F is a place holder for any Monad: List, Future, Either[A, ?], Id etc.
- A Monad instance for OptionT (implementing pure and flatMap) must be provided implicitly in order to allow flatMapping over OptionT.

```
final case class OptionT[F[_]: Monad, A](value: F[Option[A]]) {
  // ...
}

object OptionT {

  implicit def monad[F[_]]: Monad[F]: Monad[OptionT[F, ?]] =
    new Monad[OptionT[F, ?]] {
      override def pure[A](a: A) = ???
      override def flatMap[A, B](fa)(f) = ???
    }
}
```

Creating an OptionT[List, Int]

to encapsulate a List[Option[Int]]

```
val loi = List(Some(1), None, Some(2), Some(3))
// loi: List[Option[Int]] = List(Some(1), None, Some(2), Some(3))

val otli = OptionT[List, Int](loi)
// otli: cats.data.OptionT[List,Int] = OptionT(List(Some(1), None, Some(2), Some(3

otli.value // same as: loi
// res6: List[Option[Int]] = List(Some(1), None, Some(2), Some(3))
```

FlatMapping an OptionT[List, Int]

flatMap takes a function of type: A => OptionT[F, B]
In our case the type is: Int => OptionT[List, String]

```
def fillListWith(x: Int): List[Option[String]] = List.fill(x)(Option(x.toString))

val otliFlatMapped = otli.flatMap(x => OptionT[List, String](fillListWith(x)))
// otliFlatMapped: cats.data.OptionT[List,String] =
//   OptionT(List(Some(1), None, Some(2), Some(2), Some(3), Some(3), Some(3)))

otliFlatMapped.value
// res7: List[Option[String]] =
//   List(Some(1), None, Some(2), Some(2), Some(3), Some(3), Some(3))
```

Convenience function flatMapF

flatMapF takes a function of type: A => F[Option[B]]
In our case the type is: Int => List[Option[String]]

```
def fillListWith(x: Int): List[Option[String]] = List.fill(x)(Option(x.toString))

val otliFlatMappedF = otli.flatMapF(x => fillListWith(x))
// otliFlatMappedF: cats.data.OptionT[List,String] =
//   OptionT(List(Some(1), None, Some(2), Some(2), Some(3), Some(3), Some(3)))

otliFlatMappedF.value
// res8: List[Option[String]] =
//   List(Some(1), None, Some(2), Some(2), Some(3), Some(3), Some(3))
```

Mapping an OptionT[List, Int]

```
val otliMapped = Monad[OptionT[List, ?]].map(otli) { _.toString + "!" }
// otliMapped: cats.data.OptionT[[+A]List[A],String] =
//   OptionT(List(Some(1!), None, Some(2!), Some(3!)))

otliMapped.value
// res9: List[Option[String]] = List(Some(1!), None, Some(2!), Some(3!))
```

OptionT.{isDefined, isEmpty, getOrElse, fold}

```
otli.isDefined
// res10: List[Boolean] = List(true, false, true, true)

otli.isEmpty
// res11: List[Boolean] = List(false, true, false, false)

otli.getOrElse(42)
// res12: List[Int] = List(1, 42, 2, 3)

otli.fold(42.0)(_.toDouble)
// res13: List[Int] = List(1.0, 42.0, 2.0, 3.0)
```

For comprehension with OptionT[List, Int] encapsulating List[Option[Int]]

```
val result4: OptionT[List, Int] =
  for {
    x <- otli
    y <- otli
  } yield x * y
// result4: cats.data.OptionT[List,Int] =
//   OptionT(List(Some(1), None, Some(2), Some(3), None, Some(2),
//       None, Some(4), Some(6), Some(3), None, Some(6), Some(9)))

result4.value
// res13: List[Option[Int]] =
//   List(Some(1), None, Some(2), Some(3), None, Some(2),
//       None, Some(4), Some(6), Some(3), None, Some(6), Some(9))
```

7. Generic Monad Processing

Using processIntMonads with OptionT

OptionT is a Monad which encapsulates two other Monads.

Hence an OptionT can be passed to a function accepting any Monad F[Int].

```scala
def processIntMonads[F[_]: Monad](monad1: F[Int], monad2: F[Int]): F[Int] =
  for {
    x <- monad1
    y <- monad2
  } yield x * y

val result5 = processIntMonads(otli, otli)
// result5: cats.data.OptionT[List,Int] = OptionT(List(
//    Some(1), None, Some(2), Some(3), None, Some(2),
//    None, Some(4), Some(6), Some(3), None, Some(6), Some(9)))

result5.value
// res14: List[Option[Int]] = List(
//    Some(1), None, Some(2), Some(3), None, Some(2),
//    None, Some(4), Some(6), Some(3), None, Some(6), Some(9))
```

Using processIntMonads with other Monads

- List
- Vector
- Option
- Future

```
processIntMonads(List(1, 2, 3), List(10, 20, 30))
// res16: List[Int] = List(10, 20, 30, 20, 40, 60, 30, 60, 90)

processIntMonads(Vector(1, 2, 3), Vector(10, 20, 30))
// res17: scala.collection.immutable.Vector[Int] = Vector(10, 20, 30, 20, 40, 60,

processIntMonads(Option(5), Option(5))
// res18: Option[Int] = Some(25)

val fi = processIntMonads(Future(5), Future(5))
// fi: scala.concurrent.Future[Int] = Future(<not completed>)
Await.ready(fi, 1.second)
fi
// res20: scala.concurrent.Future[Int] = Future(Success(25))
```

Using processIntMonads with other OptionT-Monads

- OptionT[Vector, Int]
- OptionT[Option, Int]
- OptionT[Future, Int]

```
val otvi = OptionT[Vector, Int](Vector(Option(3), Option(5)))
processIntMonads(otvi, otvi).value
// res21: Vector[Option[Int]] = Vector(Some(9), Some(15), Some(15), Some(25))

val otoi = OptionT[Option, Int](Option(Option(5)))
processIntMonads(otoi, otoi).value
// res22: Option[Option[Int]] = Some(Some(25))

val otfi = processIntMonads(OptionT(Future(Option(5))), OptionT(Future(Option(5))))
Await.ready(otfi.value, 1.second)
otfi.value
// res23: scala.concurrent.Future[Option[Int]] = Future(Success(Some(25)))
```

8. Stacking Monads and Transformers

Stacking 3 Monads with 2 transformers

A database query should be async -> use a Future[???]
Accessing the DB may give you an error -> Future[Either[String, ???]]
The searched entity may not be there -> Future[Either[String, Option[???]]]

With EitherT and OptionT we can stack the 3 Monads together.

```
def compute[A]: A => Future[Either[String, Option[A]]] =
  input => Future(Right(Some(input)))

def stackMonads[A]: A => OptionT[EitherT[Future, String, ?], A] =
  input => OptionT(EitherT(compute(input)))

val stackedResult: OptionT[EitherT[Future, String, ?], Int] =
  for {
    a <- stackMonads(10)
    b <- stackMonads(32)
  } yield a + b

val future: Future[Either[String, Option[Int]]] = stackedResult.value.value

val result: Either[String, Option[Int]] = Await.result(future, 3.seconds)

println(result.right.get) // 42
```

9. Making the Types Fit

When types don't fit ...

```scala
type Nickname = String
type Name = String
type Age = Int

final case class User(name: Name, age: Age, nickname: Nickname)

// return types of these functions are incompatible
def getUser(nickname: Nickname): Future[Option[User]] = ???
def getAge(user: User): Future[Age] = ???
def getName(user: User): Option[Name] = ???
```

… make them fit

… the classical way

```
def getNameAge(nickname: Nickname): Future[Option[(Name, Age)]] =
  (for {
    user <- OptionT(getUser(nickname))
    age <- OptionT(getAge(user).map(Option(_)))
    name <- OptionT(Future(getName(user)))
  } yield (name, age)).value
```

... make them fit

... the classical way

```
def getNameAge(nickname: Nickname): Future[Option[(Name, Age)]] =
  (for {
    user <- OptionT(getUser(nickname))
    age <- OptionT(getAge(user).map(Option(_)))
    name <- OptionT(Future(getName(user)))
  } yield (name, age)).value
```

... or with OptionT.liftF and OptionT.fromOption.

```
def getNameAge(nickname: Nickname): Future[Option[(Name, Age)]] =
  (for {
    user <- OptionT(getUser(nickname))
    age <- OptionT.liftF(getAge(user))
    name <- OptionT.fromOption(getName(user))
  } yield (name, age)).value
```

Compare the solution without OptionT

```scala
def getNameAge(nickname: Nickname): Future[Option[(Name, Age)]] =
  for {
    maybeUser <- getUser(nickname)
    if maybeUser.isDefined
    user = maybeUser.get
    maybeAge <- getAge(user).map(Option(_))
    maybeName <- Future(getName(user))
  } yield for {
    name <- maybeName
    age <- maybeAge
  } yield (name, age)
```

10. Monad Transformers in Cats

- cats.data.OptionT for Option
- cats.data.EitherT for Either
- cats.data.IdT for Id
- cats.data.WriterT for Writer
- cats.data.ReaderT for Reader (ReaderT is an alias for Kleisli)
- cats.data.StateT for State

11. Best Practices

Don't expose Monad transformers to your API!

Don't expose Monad transformers to your API!

- This would make your API harder to understand.

Don't expose Monad transformers to your API!

- This would make your API harder to understand.
- The user of your API may not know what a Monad transformer is.

Don't expose Monad transformers to your API!

- This would make your API harder to understand.
- The user of your API may not know what a Monad transformer is.
- Just call transformer.value before you expose it.

Don't expose Monad transformers to your API!

- This would make your API harder to understand.
- The user of your API may not know what a Monad transformer is.
- Just call transformer.value before you expose it.
- Thus you expose List[Option[A]] instead of OptionT[List, A].

Don't expose Monad transformers to your API!

- This would make your API harder to understand.
- The user of your API may not know what a Monad transformer is.
- Just call transformer.value before you expose it.
- Thus you expose List[Option[A]] instead of OptionT[List, A].
- Or you expose Future[Option[A]] instead of OptionT[Future, A].
- See example

Don't stack too many Monads!

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.
- Does it really slow down your app? Maybe IO is the bottleneck? Benchmark if necessary!

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.
- Does it really slow down your app? Maybe IO is the bottleneck? Benchmark if necessary!

  Consider other approaches:

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.
- Does it really slow down your app? Maybe IO is the bottleneck? Benchmark if necessary!

  Consider other approaches:

- Free Monads

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.
- Does it really slow down your app? Maybe IO is the bottleneck? Benchmark if necessary!

  Consider other approaches:

- Free Monads
- Tagless Final

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.
- Does it really slow down your app? Maybe IO is the bottleneck? Benchmark if necessary!

  Consider other approaches:

- Free Monads
- Tagless Final
- Or: Go back to nested for comprehensions.

Don't stack too many Monads!

- A Monad transformer is just another Monad (wrapping 2 Monads).
- You can wrap 3 Monads with 2 transformers (= 5 Monads).
- You can wrap 4 Monads with 3 transformers (= 7 Monads), etc.
- Too many stacked transformers do not make your code clearer.
- Too many stacked transformers may degrade performance.
- Structures consume heap.
- Does it really slow down your app? Maybe IO is the bottleneck? Benchmark if necessary!

  Consider other approaches:

- Free Monads
- Tagless Final
- Or: Go back to nested for comprehensions.

Follow the "Principle of least power"!

12. OptionT - Implementation

ListOption - implementation

```scala
final case class ListOption[A](value: List[Option[A]]) {

  def map[B](f: A => B): ListOption[B] =
    ListOption(value.map(_ map f))
  def flatMap[B](f: A => ListOption[B]): ListOption[B] =
    ListOption(
      value.flatMap {
        case None => List(Option.empty[B])
        case Some(a) => f(a).value
      }
    )
  def flatMapF[B](f: A => List[Option[B]]): ListOption[B] =
    flatMap(a => ListOption(f(a)))

  def isDefined: List[Boolean] = value.map(_.isDefined)
  def isEmpty: List[Boolean] = value.map(_.isEmpty)
  def getOrElse(default: => A): List[A] = value.map(_.getOrElse(default))
}
object ListOption {
  implicit def monad: Monad[ListOption] = new Monad[ListOption] {
    override def pure[A](a: A): ListOption[A] = ListOption(List(Option(a)))
    override def flatMap[A, B](fa: ListOption[A])
                    (f: A => ListOption[B]): ListOption[B] = fa flatMap f
  }
}
```

Abstracting over List gives you OptionT.

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]
- Save an instance of Monad F in a value: val F = Monad[F]

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]
- Save an instance of Monad F in a value: val F = Monad[F]
- Replace every occurence of List with type constructor F.

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]
- Save an instance of Monad F in a value: val F = Monad[F]
- Replace every occurence of List with type constructor F.
- Replace every ListOption[A] with ListOption[F, A].

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]
- Save an instance of Monad F in a value: val F = Monad[F]
- Replace every occurence of List with type constructor F.
- Replace every ListOption[A] with ListOption[F, A].
- Replace invocations of map/flatMap on List by invocations on Monad F.

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]
- Save an instance of Monad F in a value: val F = Monad[F]
- Replace every occurence of List with type constructor F.
- Replace every ListOption[A] with ListOption[F, A].
- Replace invocations of map/flatMap on List by invocations on Monad F.
- Replace invocations of List.apply by invocations of F.pure.

Abstracting over List gives you OptionT.

- Parameterize ListOption with type constructor: ListOption[F[_], A]
- F must be a Monad! Use bounded context: ListOption[F[_]: Monad, A]
- Save an instance of Monad F in a value: val F = Monad[F]
- Replace every occurence of List with type constructor F.
- Replace every ListOption[A] with ListOption[F, A].
- Replace invocations of map/flatMap on List by invocations on Monad F.
- Replace invocations of List.apply by invocations of F.pure.
- Rename ListOption to OptionT.

OptionT - implementation

```
final case class OptionT[F[_]: Monad, A](value: F[Option[A]]) {
  val F = Monad[F]

  def map[B](f: A => B): OptionT[F, B] =
    OptionT(F.map(value)(_ map f))
  def flatMap[B](f: A => OptionT[F, B]): OptionT[F, B] =
    OptionT(
      F.flatMap(value) {
        case None => F.pure(Option.empty[B])
        case Some(a) => f(a).value
      }
    )
  def flatMapF[B](f: A => F[Option[B]]): OptionT[F, B] =
    flatMap(a => OptionT(f(a)))

  def isDefined: F[Boolean] = F.map(value)(_.isDefined)
  def isEmpty: F[Boolean] = F.map(value)(_.isEmpty)
  def getOrElse(default: => A): F[A] = F.map(value)(_.getOrElse(default))
}
object OptionT {
  implicit def monad[F[_]: Monad]: Monad[OptionT[F, ?]] =
    new Monad[OptionT[F, ?]] {
      override def pure[A](a: A): OptionT[F, A] =
        OptionT(Monad[F].pure(Option(a)))
      override def flatMap[A, B](fa: OptionT[F, A])
                      (f: A => OptionT[F, B]): OptionT[F, B] = fa flatMap f
    }
}
```

13. Resources

Resources (1/2)

- Code and Slides of this Talk:
  https://github.com/hermannhueck/monad-transformers

- Cats documentation:
  https://typelevel.org/cats/datatypes/optiont.html
  https://typelevel.org/cats/datatypes/eithert.html

- "Scala with Cats"
  Book by Noel Welsh and Dave Gurnell
  https://underscore.io/books/scala-with-cats/

- "Herding Cats, day 10: Stacking Future and Either"
  Blogpost by eed3si9n (Eugene Yokota)
  http://eed3si9n.com/herding-cats/stacking-future-and-either.html

Resources (2/2)

- "Monad transformers down to earth"
  Talk by Gabriele Petronella at Scala Days Copenhagen 2017
  https://www.youtube.com/watch?v=jd5e71nFEZM

- "FSiS Part 7 - OptionT transformer"
  Live coding video tutorial by Michael Pilquist, 2015
  https://www.youtube.com/watch?v=ZNUTMabdgzo

Thanks for Listening

Q & A

https://github.com/hermannhueck/monad-transformers