From Functor Composition to Monad Transformers

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

Agenda

1. Nested Monads in for comprehensions
2. Functors compose.
3. Aside: Creating List[Int => Int] and List[Option[Int => Int]]
4. Applicatives compose.
5. Do Monads compose?
6. Monad Transformers - Example OptionT
7. Best Practices
8. Resources

1. Nested Monads in for comprehensions

Imports for Cats

```scala
import cats._, cats.data._, cats.implicits._
```

For comprehension for List[Option[Int]]

```scala
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

```scala
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

```scala
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:

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

2. Functors compose.

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]

```scala
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]]

```scala
implicit class PimpedListOptionA[A](list: List[Option[A]]) {
  // functor type: Functor[Lambda[X => List[Option[X]]]]
  val functor = Functor[List] compose Functor[Option]
  def map[B](f: A => B): List[Option[B]] = functor.map(list)(f)
  def fmap[B](f: A => B): List[Option[B]] = functor.map(list)(f)
}

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

3. Aside: Creating List[Int => Int] and List[Option[Int => Int]]

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$...), S
```

4. Applicatives compose.

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

```scala
val loiResult = (Applicative[List] compose Applicative[Option]).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?

Do Monads compose?

```
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
```

Do Monads compose?

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

The solution: Monad Transformers

6. Monad Transformers - Example OptionT

What is OptionT?

- OptionT is a generic case class which encapsulates an F[Option[A]].
- F is the generic type constructor of another monad like List, Future, Either, Id.
- A Monad instance for OptionT (implementing pure and flatMap) must be provided in implicit scope in order to allow flatMapping over OptionT.

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

object OptionT {

  implicit def monad[F[_]](implicit F: Monad[F]): Monad[OptionT[F, ?]] =

    new Monad[OptionT[F, ?]] {

      override def pure[A](a: A): OptionT[F, A] = ???

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

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]]

```
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}

```
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)
```

For comprehension with List[Option[Int]]

```scala
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. Best Practices

Don't stack too many transformers!

- A monad transformer is just another monad (wrapping 2 monads).
- You can wrap the transformer into another transformer (2 Transformers = 3 monads)
- This procedure creates a transformer stack.
- Too many stacked transformers do not make your code more understandable.
- Too many stacked transformers can degrade performance.

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].

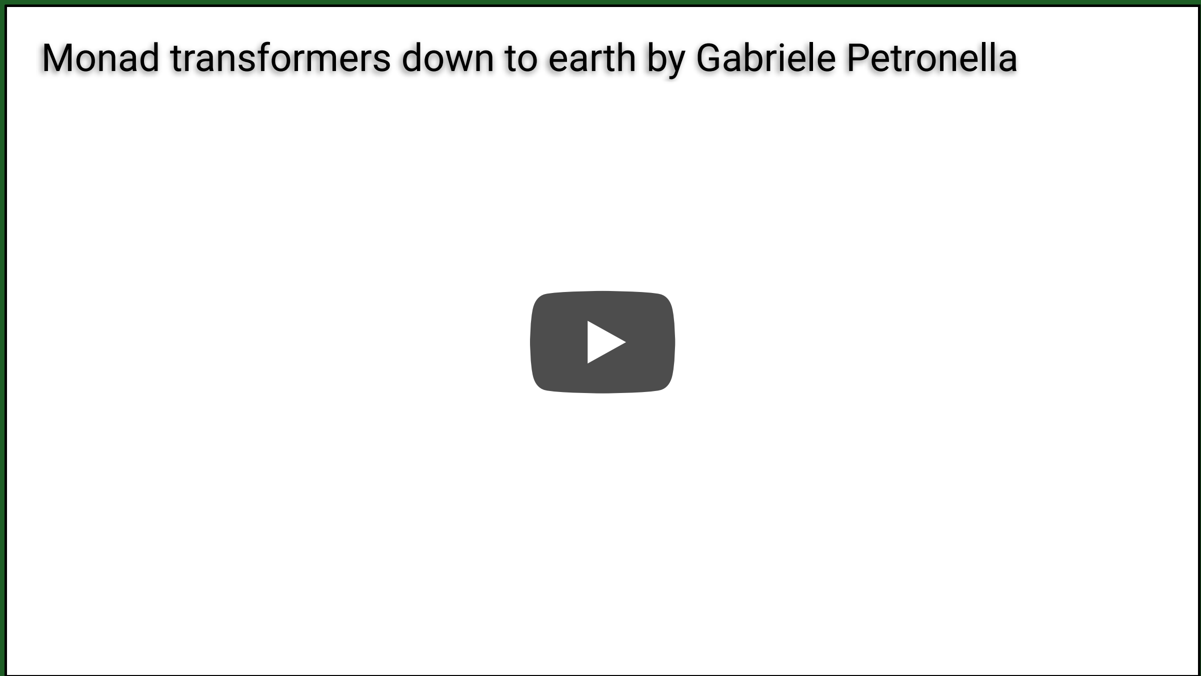
8. Resources

Resources 1/3

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

Resources 2/3

"Monad transformers down to earth"
Talk by Gabriele Petronella at Scala Days Copenhagen 2017



Resources 3/3

"FSiS Part 7 - OptionT transformer"
Live coding video tutorial by Michael Pilquist, 2015

Thanks for Listening

Q & A