Functional Programming Patterns v3

(for the pragmatic programmer)

~

@raulraja CTO @47deg

Acknowledgment

- Cats : Functional programming in Scala
- Rúnar Bjarnason : Compositional Application Architecture With Reasonably Priced Monads
- Noel Markham : A purely functional approach to building large applications
- Wouter Swierstra : FUNCTIONAL PEARL Data types a la carte
- Free Applicative Functors : Paolo Caprioti
- Rapture : Jon Pretty

All meaningful architectural patterns can be achieved with pure FP

When I build an app I want it to be

- Free of Interpretation
- Support Parallel Computation
- Composable pieces
- Dependency Injection / IOC
- Fault Tolerance

When I build an app I want it to be

- Free of Interpretation : **Free Monads**
- Support Parallel Computation : **Free Applicatives**
- Composable pieces : **Coproducts**
- Dependency Injection / IOC : **Implicits & Kleisli**
- Fault tolerant : **Dependently typed checked exceptions**

Interpretation : Free Monads

What is a Free Monad?

-- A monad on a custom algebra that can be run through an Interpreter

Interpretation : Free Monads

What is an Application?

-- A collection of algebras and the Coproduct resulting from their interaction

Interpretation : Free Monads

Let's build an app that reads a Cat, validates some input and stores it

Interpretation : Free Monads

Our first Algebra models our program interaction with the end user

```scala
sealed trait Interact[A]

case class Ask(prompt: String) extends Interact[String]

case class Tell(msg: String) extends Interact[Unit]
```

Interpretation : Free Monads

Our second Algebra is about persistence and data validation

```scala
sealed trait DataOp[A]

case class AddCat(a: String) extends DataOp[Unit]

case class ValidateCatName(a: String) extends DataOp[Boolean]

case class GetAllCats() extends DataOp[List[String]]
```

Interpretation : Free Monads

An application is the Coproduct of its algebras

```scala
import cats.data.Coproduct

type CatsApp[A] = Coproduct[DataOp, Interact, A]
```

Interpretation : Free Monads

We can now lift different algebras to our App monad via smart constructors and compose them

```scala
import cats.free.{Inject, Free}

class Interacts[F[_]](implicit I: Inject[Interact, F]) {
  def tell(msg: String): Free[F, Unit] = Free.inject[Interact, F](Tell(msg))
  def ask(prompt: String): Free[F, String] = Free.inject[Interact, F](Ask(prompt))
}
object Interacts {
  implicit def instance[F[_]](implicit I: Inject[Interact, F]): Interacts[F] = new Interacts[F]
}
```

Interpretation : Free Monads

We can now lift different algebras to our App monad via smart constructors and compose them

```
class DataSource[F[_]](implicit I: Inject[DataOp, F]) {
  def addCat(a: String): Free[F, Unit] = Free.inject[DataOp, F](AddCat(a))
  def validateCatName(a: String): Free[F, Boolean] = Free.inject[DataOp, F](ValidateCatName(a))
  def getAllCats: Free[F, List[String]] = Free.inject[DataOp, F](GetAllCats())
}
object DataSource {
  implicit def dataSource[F[_]](implicit I: Inject[DataOp, F]): DataSource[F] = new DataSource[F]
}
```

Interpretation : Free Monads

At this point a program is nothing but **Data** describing the sequence of execution but **FREE** of its runtime interpretation.

```scala
def program(implicit I : Interacts[CatsApp], D : DataSource[CatsApp]) = {

  import I._, D._

  for {
    cat <- ask("What's the kitty's name")
    valid <- validateCatName(cat)
    _ <- if (valid) addCat(cat) else tell(s"Invalid cat name '$cat'")
    cats <- getAllCats
    _ <- tell(cats.toString)
  } yield ()
}
```

Interpretation : Free Monads

We isolate interpretations via Natural transformations AKA **`Interpreters`**.
In other words with map over the outer type constructor of our Algebras.

```scala
import cats.~>
import scalaz.concurrent.Task

object ConsoleCatsInterpreter extends (Interact ~> Task) {
  def apply[A](i: Interact[A]) = i match {
    case Ask(prompt) =>
      Task.delay {
        println(prompt)
        //scala.io.StdIn.readLine()
        "Tom"
      }
    case Tell(msg) =>
      Task.delay(println(msg))
  }
}
```

Interpretation : Free Monads

We isolate interpretations via Natural transformations AKA **`Interpreters`**.
In other words with map over the outer type constructor of our Algebras.

In other words with map over
the outer type constructor of our Algebras

```scala
import scala.collection.mutable.ListBuffer

object InMemoryDatasourceInterpreter extends (DataOp ~> Task) {
  private[this] val memDataSet = new ListBuffer[String]
  def apply[A](fa: DataOp[A]) = fa match {
    case AddCat(a) => Task.delay(memDataSet.append(a))
    case GetAllCats() => Task.delay(memDataSet.toList)
    case ValidateCatName(name) => Task.now(true)
  }
}
```

Interpretation : Free Monads

Now that we have a way to combine interpreters
we can lift them to the app Coproduct

```
val interpreters: CatsApp ~> Task = InMemoryDatasourceInterpreter or ConsoleCatsInterpreter
```

Interpretation : Free Monads

And we can finally apply our program applying the interpreter to the free monad

```
import Interacts._, DataSource._
import cats.Monad

implicit val taskMonad = new Monad[Task] {
  override def flatMap[A, B](fa: Task[A])(f: (A) => Task[B]): Task[B] = fa flatMap f
  override def pure[A](x: A): Task[A] = Task.now(x)
}

val evaled = program foldMap interpreters
```

Requirements

- **Free of Interpretation**
- Support Parallel Computation
- Composable pieces
- Dependency Injection / IOC
- Fault Tolerance

Interpretation : Free Applicatives

What about parallel computations?

Interpretation : Free Applicatives

Unlike Free Monads which are good for dependent computations
Free Applicatives are good to represent independent computations.

```
sealed abstract class ValidationOp[A]

case class ValidNameSize(size: Int) extends ValidationOp[Boolean]

case object IsAlleyCat extends ValidationOp[Boolean]
```

Interpretation : Free Applicatives

We may use the same smart constructor pattern to lift our Algebras
to the **FreeApplicative** context.

```
import cats.free.FreeApplicative

type Validation[A] = FreeApplicative[ValidationOp, A]

object ValidationOps {

  def validNameSize(size: Int): FreeApplicative[ValidationOp, Boolean] =
    FreeApplicative.lift[ValidationOp, Boolean](ValidNameSize(size))

  def isAlleyCat: FreeApplicative[ValidationOp, Boolean] =
    FreeApplicative.lift[ValidationOp, Boolean](IsAlleyCat)

}
```

Interpretation : Free Applicatives

```scala
import cats.data.Kleisli

type ParValidator[A] = Kleisli[Task, String, A]

implicit val validationInterpreter : ValidationOp ~> ParValidator =
  new (ValidationOp ~> ParValidator) {
    def apply[A](fa: ValidationOp[A]): ParValidator[A] =
      Kleisli { str =>
        fa match {
          case ValidNameSize(size) => Task.unsafeStart(str.length >= size)
          case IsAlleyCat  => Task.unsafeStart(str.toLowerCase.endsWith("unsafe"))
        }
      }
  }
```

Interpretation : Free Applicatives

|@| The Applicative builder helps us composing applicatives

And the resulting **`Validation`** can also be interpreted

```
import cats.implicits._

def InMemoryDatasourceInterpreter(implicit validationInterpreter : ValidationOp ~> ParValidator) =
 new (DataOp ~> Task) {

  private[this] val memDataSet = new ListBuffer[String]

  def apply[A](fa: DataOp[A]) = fa match {
    case AddCat(a) => Task.delay(memDataSet.append(a))
    case GetAllCats() => Task.delay(memDataSet.toList)
    case ValidateCatName(name) =>
      import ValidationOps._, cats._, cats.syntax.all._
      val validation : Validation[Boolean] = (validNameSize(5) |@| isAlleyCat) map { case (l, r) => l && r }
      Task.fork(validation.foldMap(validationInterpreter).run(name))
  }
}
```

Interpretation : Free Applicatives

Our program can now be evaluated again and it's able to validate inputs in a parallel fashion

```
val interpreter: CatsApp ~> Task = InMemoryDatasourceInterpreter or ConsoleCatsInterpreter
val evaled = program.foldMap(interpreter).unsafePerformSyncAttempt
```

Requirements

- **Free of Interpretation**
- **Support Parallel Computation**
- Composable pieces
- Dependency Injection / IOC
- Fault Tolerance

Composability

Composition gives us the power
to easily mix simple functions
to achieve more complex workflows.

Composability

We can achieve monadic function composition
with **Kleisli Arrows**

```
A ⇒ M[B]
```

In other words a function that
for a given input it returns a type constructor…

```
List[B], Option[B], Either[B], Task[B], Future[B]…
```

Composability

When the type constructor `M[_]` it's a Monad it can be monadically composed

```
val composed = for {
  a <- Kleisli((x : String) ⇒ Option(x.toInt + 1))
  b <- Kleisli((x : String) ⇒ Option(x.toInt * 2))
} yield a + b
```

Composability

The deferred injection of the input parameter enables **Dependency Injection**. This is an alternative to implicits commonly known as DI with the Reader monad.

```
val composed = for {
  a <- Kleisli((x : String) ⇒ Option(x.toInt + 1))
  b <- Kleisli((x : String) ⇒ Option(x.toInt * 2))
} yield a + b

composed.run("1")
```

Requirements

- **Free of Interpretation**
- **Support Parallel Computation**
- **Composable pieces**
- **Dependency Injection / IOC**
- Fault Tolerance

Fault Tolerance

Most containers and patterns generalize to the most common super-type or simply **`Throwable`** loosing type information.

```scala
val f = scala.concurrent.Future.failed(new NumberFormatException)
val t = scala.util.Try(throw new NumberFormatException)
val d = for {
 a <- 1.right[NumberFormatException]
 b <- (new RuntimeException).left[Int]
} yield a + b
```

Fault Tolerance

We don't have to settle for **`Throwable`**!!!

We could use instead...

- Nested disjunctions
- Delimited, Monadic, Dependently-typed, Accumulating Checked Exceptions

Fault Tolerance : Dependently-typed Acc Exceptions

```
import rapture.core._
```

Fault Tolerance : Dependently-typed Acc Exceptions

`Result` is similar to `Xor`, `\/` and `Ior` but has 3 possible outcomes

(Answer, Errata, Unforeseen)

```
val op = for {
  a <- Result.catching[NumberFormatException]("1".toInt)
  b <- Result.errata[Int, IllegalArgumentException](
           new IllegalArgumentException("expected"))
} yield a + b
```

Fault Tolerance : Dependently-typed Acc Exceptions

`Result` uses dependently typed monadic exception accumulation

```
val op = for {
  a <- Result.catching[NumberFormatException]("1".toInt)
  b <- Result.errata[Int, IllegalArgumentException](
           new IllegalArgumentException("expected"))
} yield a + b
```

Fault Tolerance : Dependently-typed Acc Exceptions

You may recover by `resolving` errors to an `Answer`.

```
op resolve (
    each[IllegalArgumentException](_ ⇒ 0),
    each[NumberFormatException](_ ⇒ 0),
    each[IndexOutOfBoundsException](_ ⇒ 0))
```

Fault Tolerance : Dependently-typed Acc Exceptions

Or `reconcile` exceptions into a new custom one.

```
case class MyCustomException(e : Exception) extends Exception(e.getMessage)

op reconcile (
    each[IllegalArgumentException](MyCustomException(_)),
    each[NumberFormatException](MyCustomException(_)),
    each[IndexOutOfBoundsException](MyCustomException(_)))
```

Recap

- Free of Interpretation : **Free Monads**
- Support Parallel Computation : **Free Applicatives**
- Composable pieces : **Coproducts**
- Dependency Injection / IOC : **Implicits & Kleisli**
- Fault tolerant : **Dependently typed checked exceptions**

What's next?

If you want to compute with unrelated nested monads you need Transformers.

Transformers are supermonads that help you flatten through nested monads such as
Future[Option] or Kleisli[Task[Disjuntion]] binding to the most inner value.

http://www.47deg.com/blog/fp-for-the-average-joe-part-2-scalaz-monad-transformers

Questions? & Thanks!

@raulraja
@47deg
http://github.com/47deg/func-architecture-v3
https://speakerdeck.com/raulraja/functional-programming-patterns-v3