Functional Programming Patterns v2

(for the pragmatic programmer)

~

@raulraja CTO @47deg

Acknowledgment

- Scalaz : 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
- Rapture : Jon Pretty

Functions are first class citizens in FP Architecture

Most functional patterns are derived from Category Theory

When I build an app I want it to be

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

When I build an app I want it to be

- Free of Interpretation : **Free Monads**
- 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 contact and performs some operations with it

Interpretation : Free Monads

A very simple model

```
case class Contact(
        firstName: String,
        lastName: String,
        phoneNumber: String)
```

Interpretation : Free Monads

Our first Algebra is interaction with a 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

```scala
sealed trait DataOp[A]

case class AddContact(a: Contact) extends DataOp[Unit]

case class GetAllContacts() extends DataOp[List[Contact]]
```

Interpretation : Free Monads

An application is the Coproduct of its algebras

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

Interpretation : Free Monads

Coyoneda can gives functors for free for our Algebras

```
type ACoyo[A] = Coyoneda[AgendaApp,A]
type AFree[A] = Free[ACoyo,A]
```

Interpretation : Free Monads

We can now lift different algebras to our App Coproduct

```
def lift[F[_], G[_], A](fa: F[A])(implicit I: Inject[F, G]): FreeC[G, A] =
    Free.liftFC(I.inj(fa))
```

Interpretation : Free Monads

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

```scala
class Interacts[F[_]](implicit I: Inject[Interact, F]) {

  def tell(msg: String): Free.FreeC[F, Unit] = lift(Tell(msg))

  def ask(prompt: String): Free.FreeC[F, String] = lift(Ask(prompt))

}

object Interacts {

  implicit def interacts[F[_]](implicit I: Inject[Interact, F]): Interacts[F] = new Interacts[F]

}
```

Interpretation : Free Monads

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

```scala
class DataSource[F[_]](implicit I: Inject[DataOp, F]) {

  def addContact(a: Contact): FreeC[F, Unit] = lift[DataOp, F, Unit](AddContact(a))

  def getAllContacts: FreeC[F, List[Contact]] = lift[DataOp, F, List[Contact]](GetAllContacts())

}

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.

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

  import I._, D._

  for {
   firstName <- ask("First Name:")
   lastName <- ask("Last Name:")
   phoneNumber <- ask("Phone Number:")
   _ <- addContact(Contact(firstName, lastName, phoneNumber))
   contacts <- getAllContacts
   _ <- tell(contacts.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
object ConsoleContactReader extends (Interact ~> Id.Id) {
  def apply[A](i: Interact[A]) = i match {
    case Ask(prompt) =>
      println(prompt)
      scala.io.StdIn.readLine()
    case Tell(msg) =>
      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

```
object InMemoryDatasourceInterpreter extends (DataOp ~> Id.Id) {

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

  override def apply[A](fa: DataOp[A]) = fa match {
    case AddContact(a) => memDataSet.append(a); ()
    case GetAllContacts() => memDataSet.toList
  }
}
```

Interpretation : Free Monads

A tree of interpreters may be described to branch accordingly on each algebra
when evaluating a Coproduct

```
def or[F[_], G[_], H[_]](f: F ~> H, g: G ~> H): ({type cp[α] = Coproduct[F, G, α]})#cp ~> H =
  new NaturalTransformation[({type cp[α] = Coproduct[F, G, α]})#cp, H] {
    def apply[A](fa: Coproduct[F, G, A]): H[A] = fa.run match {
      case -\/(ff) => f(ff)
      case \/-(gg) => g(gg)
    }
  }
```

Interpretation : Free Monads

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

```
val interpreters: AgendaApp ~> Id.Id = or(InMemoryDatasourceInterpreter, ConsoleContactReader)

val coyoint: ({type f[x] = Coyoneda[AgendaApp, x]})#f ~> Id.Id = Coyoneda.liftTF(interpreters)
```

Interpretation : Free Monads

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

```
val evaled = program mapSuspension coyoint
```

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.

```scala
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**
- **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

Introducing `rapture.core.Result`

Fault Tolerance : Dependently-typed Acc Exceptions

`Result` is similar to `\/` 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 Monads** : Free of Interpretation
- **Coproducts** : Composable pieces
- **Implicits & Kleisli** : Dependency Injection / IOC
- **Dependently typed checked exceptions** Fault tolerant

What's next?

If you want to sequence or comprehend over unrelated 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.

Attend @larsr_h talk about monad transformers **OptimusPrimeT**

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-v2