Functional Programming Patterns

(for the pragmatic programmer)

~

@raulraja CTO @47deg

Acknowledgment

- Scalaz
- Rapture : Jon Pretty
- Miles Sabin : Shapeless
- Rúnar Bjarnason : Compositional Application Architecture With Reasonably Priced Monads
- Noel Markham : A purely functional approach to building large applications
- Jan Christopher Vogt : Tmaps

Functions are first class citizens in FP

Architecture

I want my main app services to strive for

- Composability
- Dependency Injection
- Interpretation
- 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 composed and sequenced in for comprehensions

```
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

```
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")
```

Composability : Kleisli

What about when the args are not of the same type?

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

Composability : Kleisli

By using `Kleisli` we just achieved

- **Composability**
- **Dependency Injection**
- Interpretation
- Fault Tolerance

Interpretation : Free Monads

What is a Free Monad?

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

Interpretation : Free Monads

```scala
sealed trait Op[A]

case class Ask[A](a: () ⇒ A) extends Op[A]

case class Async[A](a: () ⇒ A) extends Op[A]

case class Tell(a: () ⇒ Unit) extends Op[Unit]
```

Interpretation : Free Monads

What can you achieve with a custom **ADT** and **Free Monads**?

```
def ask[A](a: ⇒ A): OpMonad[A] = Free.liftFC(Ask(() ⇒ a))

def async[A](a: ⇒ A): OpMonad[A] = Free.liftFC(Async(() ⇒ a))

def tell(a: ⇒ Unit): OpMonad[Unit] = Free.liftFC(Tell(() ⇒ a))
```

Interpretation : Free Monads

Functors and Monads for Free

(No need to manually implement map, flatMap, etc...)

```
type OpMonad[A] = Free.FreeC[Op, A]

implicit val MonadOp: Monad[OpMonad] =
    Free.freeMonad[({type λ[α] = Coyoneda[Op, α]})#λ]
```

Interpretation : Free Monads

At this point a program like this is nothing but
Data
describing the sequence of execution but **FREE**
of it's runtime interpretation.

```
val program = for {
  a <- ask(1)
  b <- async(2)
  _ <- tell(println("log something"))
} yield a + b
```

Interpretation : Free Monads

We isolate interpretations
via Natural transformations AKA `Interpreters`.

In other words with map over
the outer type constructor `Op`

```
object ProdInterpreter extends (Op ~> Task) {
    def apply[A](op: Op[A]) = op match {
        case Ask(a) ⇒ Task(a())

        case Async(a) ⇒ Task.fork(Task.delay(a()))

        case Tell(a) ⇒ Task.delay(a())

    }
}
```

Interpretation : Free Monads

We can have different interpreters for our production / test / experimental code.

```
object TestInterpreter extends (Op ~> Id.Id) {
    def apply[A](op: Op[A]) = op match {
        case Ask(a) ⇒ a()

        case Async(a) ⇒ a()

        case Tell(a) ⇒ a()

    }
}
```

Requirements

- **Composability**
- **Dependency Injection**
- **Interpretation**
- Fault Tolerance

Fault Tolerance

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

```
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
- Coproducts
- 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(_)))
```

Requirements

We have all the pieces we need

Let's put them together!

- **Composability**
- **Dependency Injection**
- **Interpretation**
- **Fault Tolerance**

Solving the Puzzle

How do we assemble a type that is:

Kleisli + Custom ADT + Result

```
for {
  a <- Kleisli((x : String) ⇒ ask(Result.catching[NumberFormatException](x.toInt)))
  b <- Kleisli((x : String) ⇒ ask(Result.catching[IllegalArgumentException](x.toInt)))
} yield a + b
```

We want `a` and `b` to be seen as `Int` but this won't compile
because there are 3 nested monads

Solving the Puzzle : Monad Transformers

`Monad Transformers` to the rescue!

```
type ServiceDef[D, A, B <: Exception] =
    ResultT[({type λ[α] = ReaderT[OpMonad, D, α]})#λ, A, B]
```

Solving the Puzzle : Services

Two services with different dependencies

```
case class Converter() {
  def convert(x: String): Int = x.toInt
}

case class Adder() {
  def add(x: Int): Int = x + 1
}

case class Config(converter: Converter, adder: Adder)

val system = Config(Converter(), Adder())
```

Solving the Puzzle : Services

Two services with different dependencies

```
def service1(x : String) = Service { converter: Converter ⇒
    ask(Result.catching[NumberFormatException](converter.convert(x)))
}

def service2 = Service { adder: Adder ⇒
    ask(Result.catching[IllegalArgumentException](adder.add(22) + " added "))
}
```

Solving the Puzzle : Services

Two services with different dependencies

```
val composed = for {
    a <- service1("1").liftD[Config]
    b <- service2.liftD[Config]
} yield a + b

composed.exec(system)(TestInterpreter)
composed.exec(system)(ProdInterpreter)
```

Conclusion

- Composability : Kleisli
- Dependency Injection : Kleisli
- Interpretation : Free monads
- Fault Tolerance : Dependently typed checked exceptions

Thanks!

@raulraja
@47deg
http://github.com/47deg/func-architecture