TWO FUNCTIONAL PATTERNS @ EQUENS

WHO AM I

- Jos Dirksen
  - Currently doing Devops, Scala stuff
  - At Equeris, lean startup within Equens
- Email me at: jos.dirksen@gmail.com
- I write at: http://www.smartjava.org
- Twitter: josdirksen

I'll try to avoid showing Scalaz internals as much as possible

AGENDA

- *The Free monad*:
  - Seperate structure from behavior
  - Allows you to reason about programs
- *Typeclasses*:
  - Implement adhoc polymorphism

FREE MONADS

WHAT IS A FREE MONAD

"A free monad ***satisfies all the Monad laws****, but does* ***not*** *do any collapsing (i.e., computation). It just builds up a nested series of contexts. The user who creates such a free monadic value is responsible for doing something with those nested contexts, so that the* ***meaning*** *of such a composition can be deferred until* ***after*** *the monadic value has been created."*

WHAT IS A FREE MONAD

Create a structure in the form of a for comprehension

```
val prg = for {
    user <- retrieveUser(userId)
    updated <- updateUser(user)
} yield updated
```

Interpret this program separately

```
object interpreter extends (Operation ~> Id.Id) { .. }

val response = prg.foldMap(interpreter)
```

OBIVOUSLY, THERE IS A LOT MORE TO FREE

- I'll skip the inner details, and focus on usage
- Good introduction: http://www.functionalvilnius.lt/meetups/meetups/2015-04-29-functional-vilnius-03/freemonads.pdf

FREE DSLS FOR YOUR SOFTWARE!

WHAT IS NEEDED TO CREATE A DSL

- **Abstract Syntax Tree**: Structure of your code
- **program**: a 'for' comprehension with Free Monads
- **interpreter**: Runs the program

DSL INGREDIENTS: AST

- Defines what functionality is provided by your DSL
- Implemented as a set of case classes

```
sealed trait GitService[A]
  case class GetProfile() extends GitService[Profile]
  case class GetProjects() extends GitService[List[Project]]
  case class GetRepositories(project: Project) extends GitService[List[Re
  case class GetProject(projectName: String) extends GitService[Project]
```

Just structure, no behavior!

IN OUR CASE

```
sealed trait DAGService[A]

  final case class Store(dag: DAG) extends DAGService[NewWorkflowResponse
  final case class HasResults[T](res: Seq[T], dag: DAG) extends DAGServic
  final case class CheckName(dag: DAG) extends DAGService[Boolean]
  final case class Compile(graph: Json) extends DAGService[GraphMap]
  final case class Serialize(dag: DAG, compiled: GraphMap) extends DAGSer
  final case class Validate(nodes: GraphMap) extends DAGService[Boolean]
  final case class GetExistingVersion(dag: DAG) extends DAGService[Seq[DA
  final case class IsAvailable(dag: DAG) extends DAGService[Boolean]
  final case class IsVersionAvailableForEnvironment(dvo: DAGVersionOption
  final case class GetVersions(dvo: DAGVersionsOptions) extends DAGServic
  final case class GetVersion(dvo: DAGVersionOptions) extends DAGService[
  final case class GetLatestVersions(ldvo: LatestDAGVersionsOptions) exte
  final case class GetDefaultVersions(ddvo: DefaultDAGVersionsOptions) ex
  final case class GetDefaultDAG(ddo: DefaultDAGOptions) extends DAGServi
  final case class GetDAGs(gdo: GetDAGOptions) extends DAGService[List[DA
  final case class UpdateStatus(uso: UpdateStatusOptions) extends DAGServ
```

DSL INGREDIENTS: FREE MONADS

```
// Using Scalaz we can automatically lift our AST to a Free monad
def getProfile() =
       Free.liftF(GetProfile())
def getProject(projectName: String) =
       Free.liftF(GetProject(projectName))
def getProjects() =
       Free.liftF(GetProjects())
def getRepositories(project: Project) =
       Free.liftF(GetRepositories(project))
```

If you're adventurous you could also do this using an implicit

```
type -~>[F[_], G[_]] = Inject[F, G]
object LiftImplicit {
  implicit def lift[F[_], G[_], A](fa: F[A])
      (implicit I: F -~> G): Free[G, A]
        = Free liftF I.inj(fa)
}
```

DSL INGREDIENTS: THE PROGRAM

```scala
// The monad we're working with.
type GitServiceOp[A] = Free[GitService, A]

def findRepositories(projectName: String) = {
  for {
    project <- getProject(projectName)
    repositories <- getRepositories(project)
  } yield repositories
}
```

Just a basic 'for' comprehension, which still does nothing.

WE USE THIS TO CREATE MORE ADVANCED FUNCTIONS

```
for {
  session <- adwordsSession(clientCustomerId)
  placementPerformances <- readPlacementsPerformanceApi(clientCustome
  adgroups <- readAdgroupsApi(campaignId, session)
  newAdgroups <- filterNewAdgroups(adgroups, placementPerformances)
  addResult <- createPlacementsForAdgroups(placementPerformances.map
} yield addResult

for {
  session <- adwordsSession(clientCustomerId)
  adgroups <- readAdgroupsApi(campaignId, session)
  notTargetedUrls <- loadDbUrlsWithStatus(clientId, NotTargeted)
  result <- createPlacementsForAdgroups(notTargetedUrls, adgroups, se
  _ <- updateDbUrlsWithStatus(clientId, notTargetedUrls, NotTargeted
} yield result
```

DSL INGREDIENTS: THE INTERPRETER

```scala
object NoOpInterpreter extends (GitService ~> Id.Id) {

  override def apply[A](fa: GitService[A]): Id.Id[A] = fa match {
    case GetProfile() =>
      println("GetProfile called") ; Profile("", "", "")

    case GetProjects() =>
      println("GetProjects called") ; List.empty[Project]

    case GetRepositories(project) =>
      println("GetRepositores called") ; List.empty[Repository]

    case GetProject(projectName) =>
      println("GetProjects called") ; Project("", "")
  }
}
```

Adds functionality to the AST

AND RUN THE PROGRAM

```
// Run with the sample interpreter, returns Id monad
val uninterpreted = findRepositories("repo1")
val response = uninterpreted.foldMap(NoOpInterpreter)
```

You can run with any interpreter which provides a `NaturalTransformation` (the `~>` symbol) from `GitService` to another type, which is usually a box type of some kind (e.g `Future`)

DEMO...

1. Show our AST, Domain, Program and interpreter
2. Show how to compose programs
3. AOP, we don't need no stinking AOP!
4. Interpreter which results in `Future[A]`
5. Real world interpreter

LOOKS NICE, BUT WHY?

- Seperation of concerns
- Centralize the complex stuff
- Easy to reason about the program
- Allows you to easilt organize your thoughts
- Start with the syntax, later look at the semantics

"One thing I really like is how a lot of scary stuff gets concentrated in one place: the interpreter. If there's a bug that seems to go outside the semantic realm of my program, it must be in my interpreter. If I use anything too coarse for specification (IO being the extreme), I need to audit a lot more code to see where my types haven't adequately constrained the realm of consideration my program is concerned with. Start with a runtime capable of exactly nothing and you have no bugs. Now add capabilities back in one at a time, and you end up confining the danger zone to the interpreter, which itself is free of the nuance of your program logic. It's liberating in a way similar to how types themselves make you more sure of your pure function logic." - Someone at Stackoverflow

ADVANCED TOPICS

LIMIT AST SIZE

```
sealed trait DBService[A]
case class Execute[A](action: DBAction[A]) extends DBService[A]

trait DBAction[A]
case class StartTransaction() extends DBAction[Transaction]
case class StopTransaction(trans: Transaction)
                              extends DBAction[Unit]
case class Rollback() extends DBAction[Unit]
case class Commit() extends DBAction[Unit]
case class ExecuteQuery(query: String)
                              extends DBAction[Try[List[Result]]]
```

There is no real need to create a separate case class for each action

DEMO

COMBINING INTERPRETERS

- Uses `Inject` to create a `Coproduct`

```
object CoProductUtils {

  def liftI[F[_], G[_], A](fa: F[A])(implicit I: Inject[F, G])
                    : Free[G, A] = Free.liftF(I.inj(fa))

  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)
      }
    }
}
```

Type Magic

DEMO

THE TRAMPOLINE PATTERN

- Avoid stack overflow exceptions, with a `Trampoline`

```
type Trampoline[+A] = Free[Function0, A]
```

- Trampoline can convert any program into a stackless one

```
def dumbCopyChar(char: String, n: Integer): String = {
  if (n == 1) return char
  return dumbCopyChar(char, n -1) + char
}
```

- Rúnar Bjarnason: https://skillsmatter.com/skillscasts/3244-stackless-scala-free-monads

AS A STACKLESS PROGRAM

```
def trampolineCopyChar(char: String, n: Integer): Trampoline[String] =
  // shortcircuit when n = 1, to avoid an extra letter.
  if (n == 1) Trampoline.done(char)
  else {
    // Suspend should contain the call to the recursive function
    Trampoline.suspend(trampolineCopyChar(char, n - 1)).flatMap { p =>
      // you can use flatmap to do something with the result,
      // before returning
      Trampoline.done(p + char)
    }
  }
}
```

- Let's show this

FREE ~> HOLY GRAIL

WHAT WORKED FOR US

- Easily testable code and components
- Creating domain and AST beforehand
- Side effects are all in one place

NOT ALL UNICORNS AND RAINBOWS

- We're learning and improving
- Our ASTs and Domain model can be improved
  - Implementation details in `programs` (e.g. `getSession`)
  - Not all side effect only in `interpreters`
- More difficult to understand (when first looking)

AND I PROMISED TO TELL SOMETHING ABOUT THE TYPE CLASS PATTERN....

TYPECLASSES CAN PROVIDE AD HOC POLYMORPHISM

"In programming languages, ad hoc polymorphism is a kind of polymorphism in which polymorphic functions can be applied to arguments of **different types**, because a polymorphic function can denote a number of distinct and potentially **heterogeneous implementations** depending on the type of argument(s) to which it is applied.", *Wikipedia*

FLEXIBLE EXTENSION MODEL

- Also known as *Pimp my library* pattern
- Add functionality to existing classes, without their source or recompilation
- Different implementation per context
- Avoid the need for implementing traits or interfaces for your own objects

TYPECLASS

"The basic idea is that with a typeclass, you provide evidence that a class satisfies a specific interface."

EXAMPLE

```scala
// define the interface
trait CanSayWhatAmI[A] {
  def whatAmI(x: A): String
}

// provide implementations for the interface
implicit object MyObjectCanSayWhatAmI extends CanSayWhatAmI[String] {
  def whatAmI(x: String) = s"I'm a String, with value: $x"
}

def functionWhatAreYou[A : CanSayWhatAmI](x: A) = {
  println(implicitly[CanSayWhatAmI[A]].whatAmI(x))
}
```

~And a simple demo~

AND THAT'S ALL!

- But we can do more..
- Implicit conversion allows **`pimping`**:

```
implicit class CanFooOps[A:CanSayWhatAmI](x: A) {
  def whatAmI = implicitly[CanSayWhatAmI[A]].whatAmI(x)
}

"Hello".whatAmI
```

~And another simple demo~

WHERE DO WE USE IT?

- `CanBeJson`: Checks for a conversion to JSON
- `Cachable`: Makes sure the object can be stored in a distributed cache.
- `Measurable`: Provides set of functions needed to calculate a specific metric.

```
/**
  * A type is a Measureable metric when it provides a number of functions
  * the metric.
  */
trait MeasurableMetric[T <: RolledUpMetric] {

  /**
   * Get the uniquely identifying name of the metric we're working with.
   */
  def getMetricName(): String

...
```

THANK YOU!

MORE INFORMATION:

- http://www.slideshare.net/kenbot/running-free-with-the-monads
- http://www.slideshare.net/DavidHoyt/drinking-the-free-koolaid
- http://underscore.io/blog/posts/2015/04/14/free-monads-are-simple.html

And for an alternative approach using Cats

- https://github.com/underscoreio/scalax15-interpreters