A Tour of Functional Typeclasses

An introduction to FP & typeclasses ilustrating the power of coding to abstractions and Tagless Final Architectures

@raulraja
@47deg
Interactive Presentation

What is Functional Programming

In computer science, functional programming is a programming paradigm.
A style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

-- Wikipedia

Common traits of Functional Programming

- Higher-order functions
- Immutable data
- Referential transparency
- Lazy evaluation
- Recursion
- Abstractions

When a functions takes another function as argument or returns a function as return type:

```scala
def transform[B](list : List[Int])(transformation : Int => B) =
  list map transformation

transform(List(1, 2, 4))(x => x * 10)
```

Inmutable data

Once a value is instantiated it can't be mutated in place.
How can we change it's content then?

```scala
case class Conference(name : String)
```

When a computation returns the same value each time is invoked

Transparent :

```scala
def pureAdd(x : Int, y : Int) = x + y
```

Opaque :

```scala
var x = 0
def impureAdd(y : Int) = x += y; x
```

When a computation is evaluated only if needed

```scala
import scala.util.Try

def boom = throw new RuntimeException

def strictEval(f : Any) = Try(f)

def lazyEval(f : => Any) = Try(f)
```

Recursion is favored over iteration

```
def reduceIterative(list : List[Int]) : Int = {
  var acc = 0
  for (i <- list) acc = acc + i
  acc
}
```

Recursion is favored over iteration

```scala
def reduceRecursive(list : List[Int], acc : Int = 0) : Int =
  list match {
    case Nil => acc
    case head :: tail => reduceRecursive(tail, head + acc)
  }
```

Abstractions

Each significant piece of functionality in a program should be implemented in just one place in the source code.

-- Benjamin C. Pierce in Types and Programming Languages (2002)

What is a Typeclass

A typeclass is an interface/protocol that provides a behavior for a given data type.

This is also known as **Ad-hoc Polymorphism**

We will learn typeclasses by example...

[ ] **Monoid** : Combine values of the same type
[ ] **Functor** : Transform values inside contexts

Monoid

A `Monoid` expresses the ability of a value of a type to `combine` itself with other values of the same type in addition it provides an `empty` value.

```
import simulacrum._

@typeclass trait Monoid[A] {
  def combine(x : A, y : A) : A
  def empty : A
}
```

Monoid

```scala
implicit val IntAddMonoid = new Monoid[Int] {
  def combine(x : Int, y : Int) : Int = ???
  def empty = ???
}
```

Monoid

```scala
implicit val IntAddMonoid = new Monoid[Int] {
  def combine(x : Int, y : Int) : Int = x + y
  def empty = 0
}
```

Monoid

```scala
implicit val StringConcatMonoid = new Monoid[String] {
  def combine(x : String, y : String) : String = x + y
  def empty = ""
}
```

Monoid

```scala
implicit def ListConcatMonoid[A] = new Monoid[List[A]] {
  def combine(x : List[A], y : List[A]) : List[A] = x ++ y
  def empty = Nil
}
```

We can code to abstractions instead of coding to concrete
types.

```scala
def uberCombine[A : Monoid](x : A, y : A) : A =
  Monoid[A].combine(x, y)

uberCombine(10, 10)
```

[x] **Monoid** : Combine values of the same type
[ ] **Functor** : Transform values inside contexts

Functor

A `Functor` expresses the ability of a container to transform its content given a function

```
@typeclass trait Functor[F[_]] {
  def map[A, B](fa : F[A])(f : A => B) : F[B]
}
```

Most containers transformations can be expressed as Functors.

```scala
implicit def ListFunctor = new Functor[List] {
  def map[A, B](fa : List[A])(f : A => B) = fa map f
}
```

Functor

Most containers transformations can be expressed as Functors.

```
implicit def OptionFunctor = new Functor[Option] {
  def map[A, B](fa : Option[A])(f : A => B) = fa map f
}
```

Most containers transformations can be expressed as Functors.

```scala
import scala.concurrent.{Future, Await}
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

implicit def FutureFunctor = new Functor[Future] {
  def map[A, B](fa : Future[A])(f : A => B) = fa map f
}
```

Functor

We can code to abstractions instead of coding to
concrete
types.

```
def uberMap[F[_] : Functor, A, B](fa : F[A])(f : A => B) : F[B] =
  Functor[F].map(fa)(f)

uberMap(List(1, 2, 3))(x => x * 2)
```

[x] **Monoid** : Combine values of the same type
[x] **Functor** : Transform values inside contexts

Can we combine multiple abstractions & behaviors?

What are our application layers?

```
├── algebras
│   ├── datasource
│   │   └── NlpDataSource.scala
│   ├── services
│   │   ├── Config.scala
│   │   └── TagService.scala
│   ├── ui
│   │   └── Presentation.scala
│   └── usecases
│       └── FetchTagsUseCase.scala
├── app
│   └── main.scala
└── runtime
    ├── datasource
    │   └── TextRazorNlpDataSource.scala
    ├── runtime.scala
    ├── services
    │   └── SystemEnvConfig.scala
    └── ui
        └── ConsolePresentation.scala
```

What are our application layers?

Presentation

```scala
import simulacrum._
@typeclass trait Presentation[F[_]] {
  def onUserRequestedTags(text: String): F[Unit]
}
```

What are our application layers?

Use Case

```scala
case class Tag(value: String)
case class TaggedParagraph(text: String, tags: List[Tag])

@typeclass trait FetchTagsUseCase[F[_]] {
  def fetchTagsInText(text: String): F[TaggedParagraph]
}
```

What are our application layers?

Services

```scala
case class AnalysisRequest(text: String)

@typeclass trait TagService[F[_]] {
  def tag(request: AnalysisRequest): F[List[Tag]]
}
```

What are our application layers?

Data Source

```scala
case class Category(value: String)
case class Entity(value: String)
case class Topic(value: String)
case class AnalysisResponse(
  categories: List[Category],
  entities: List[Entity],
  topics: List[Topic]
)

@typeclass trait NlpDataSource[F[_]] {
  def analyze(text: String): F[AnalysisResponse]
}
```

What are our application layers?

Configuration

```scala
case class NlpApiKey(value: String)

@typeclass trait Config[F[_]] {
  def nlpApiKey: F[NlpApiKey]
}
```

Implementation

Presentation

```scala
import cats._
import cats.Functor
import cats.implicits._

class ConsolePresentation[F[_]: Functor: FetchTagsUseCase] extends Presentation[F] {
  def onUserRequestedTags(text: String): F[Unit] =
    FetchTagsUseCase[F].fetchTagsInText(text).map { paragraph =>
      println(paragraph.tags.mkString(", "))
    }
}
```

Implementation?

Use case

```scala
class DefaultFetchTagsUseCase[F[_]: Functor: TagService] extends FetchTagsUseCase[F] {
  def fetchTagsInText(text: String): F[TaggedParagraph] =
    TagService[F].tag(AnalysisRequest(text)).map { tags =>
      TaggedParagraph(text, tags)
    }
}
```

Implementation?

Tag Service

```
class DefaultTagService[F[_]: Functor: NlpDataSource] extends TagService[F] {
  def tag(request: AnalysisRequest): F[List[Tag]] =
    NlpDataSource[F].analyze(request.text).map { r =>
      (r.categories, r.entities, r.topics).mapN { case (category, entity, topic) =>
        List(Tag(category.value), Tag(entity.value), Tag(topic.value))
      }.flatten.distinct
    }
}
```

Implementation?

Config

```
class SystemEnvConfig[F[_]](implicit AE: ApplicativeError[F, Throwable]) extends Config[F] {
  val key = System.getenv("NLP_API_KEY")
  def nlpApiKey: F[NlpApiKey] =
    if (key == null || key == "")
      AE.raiseError(new IllegalStateException("Missing nlp api key"))
    else
      AE.pure(NlpApiKey(key))
}
```

Implementation?

Data Source

```scala
import collection.JavaConverters._
import scala.concurrent._
import java.util.Arrays
import com.textrazor.TextRazor

object TextRazorClient {
  def client(apiKey: NlpApiKey) : TextRazor = {
    val c = new TextRazor(apiKey.value)
    c.addExtractor("entities")
    c.addExtractor("topics")
    c.setClassifiers(Arrays.asList("textrazor_newscodes"))
    c
  }
}
```

Implementation?

Data Source

```scala
class TextRazorNlpDataSource[F[_]: Config](implicit ME: MonadError[F, Throwable]) extends NlpDataSource[F] {
   def analyze(text: String): F[AnalysisResponse] =
    for {
      apiKey <- Config[F].nlpApiKey
      response <- ME.catchNonFatal {
         val r = blocking {
           TextRazorClient.client(apiKey).analyze(text).getResponse
         }
         AnalysisResponse(
           r.getCategories.asScala.toList.map { c => Category(c.getLabel) },
           r.getEntities.asScala.toList.map { e => Entity(e.getEntityId) },
           r.getTopics.asScala.toList.map { t => Topic(t.getLabel) }
         )
      }
    } yield response
}
```

Implementation?

Runtime Module

```
object runtime {
  implicit def presentation[F[_]: Functor: FetchTagsUseCase]: Presentation[F] =
    new ConsolePresentation
  implicit def useCase[F[_]: Functor: TagService] : FetchTagsUseCase[F] =
    new DefaultFetchTagsUseCase
  implicit def tagService[F[_]: Functor: NlpDataSource] : TagService[F] =
    new DefaultTagService
  implicit def dataSource[F[_]: Config]
    (implicit ME: MonadError[F, Throwable]) : NlpDataSource[F] =
      new TextRazorNlpDataSource
  implicit def config[F[_]](implicit A: ApplicativeError[F, Throwable]): Config[F] =
    new SystemEnvConfig
}
```

Implementation?

Application

```
import runtime._
import scala.util.Try

val text =
    """|
      | And now here is my secret, a very simple secret:
      | It is only with the heart that one can see rightly;
      | what is essential is invisible to the eye.
      | – Antoine de Saint-Exupéry, The Little Prince
      """.stripMargin

Presentation[Try].onUserRequestedTags(text)
```

Pros

- Pure Applications and Libraries
- Testing flexibility
- Controlled Effects at the edge
- Separation of concerns on steroids
- A unified model to create components and compose them
- A unified API to rule all data types when using just type classes.
- Restricted to thoroughly tested lawful declarations with a base on mathematics

Cons

— Requires understanding type classes and higher kinds
— Potential peer and status quo push-back

Questions? & Thanks!

@raulraja
@47deg
http://github.com/47deg/typeclasses-tour
https://speakerdeck.com/raulraja/typeclasses-tour