A Tour of Functional Typeclasses

An introduction to FP & typeclasses ilustrating
the power of coding to abstractions

@raulraja
@47deg
Interactive
Presentation

Acknowledgment

- ☞ Cats
- ☞ Typeclassopedia
- ☞ FP
- ☞ Abstractions
- ☞ Simulacrum

Overview

Typeclasses & Data Structures

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

Higher Order Functions

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

```
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 instantitated it can't be mutated in place.
How can we change it's content then?

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

Referential Transparency

When a computation returns the same value
each time is invoked

Transparent :

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

Opaque :

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

Lazy Evaluation

When a computation is evaluated
only if needed

```
import scala.util.Try

def boom = throw new RuntimeException

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

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

Recursion

Recursion is favored over iteration

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

Recursion

Recursion is favored over iteration

```
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...

Typeclasses

[ ] **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] {
    @op("|+|") def combine(x : A, y : A) : A
    def empty : A
}
```

Monoid

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

Monoid

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

Monoid

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

Monoid

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

Monoid

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

```
import Monoid.ops._

def uberCombine[A : Monoid](x : A, y : A) : A =
    x |+| y

uberCombine(10, 10)
```

Typeclasses

[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]
}
```

Functor

Most containers transformations can be expressed as Functors.

```
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
}
```

Functor

Most containers transformations can be expressed as Functors.

```
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)
```

Typeclasses

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

Typeclasses

Can we combine multiple
abstractions & behaviors?

Typeclasses

Yes we can! Let's do a real world example

```
import cats.data.Xor
import io.circe._, io.circe.generic.auto._, io.circe.parser._
import scala.io.Source

case class CodeInfo(total_count : Int)

def searchGithub(query : String) : Int = {
    println("Searching github in " + Thread.currentThread.getName)
    val json = Source.fromURL(s"https://api.github.com/search/code?q=$query").mkString
    val codeInfo = decode[CodeInfo](json)
    codeInfo.map(_.total_count).valueOr(error => 0)
}

def sample = searchGithub("null+in:file+user:pedrovgs")
```

Typeclasses

```
import cats.{Foldable, Applicative, Traverse}
import cats.syntax.traverse._
import cats.std.all._

def reduceOps[F[_] : Applicative : Functor, A : Monoid](ops : List[F[A]]) : F[A] = {
    val op : F[List[A]] = ops.sequence
    val reduced : F[A] = Functor[F].map(op) { list =>
        Foldable[List].foldLeft(list, Monoid[A].empty) { (acc, a) =>
            Monoid[A].combine(acc, a)
        }
    }
    reduced
}
```

Typeclasses

```
def reduceOps[
    G[_] : Traverse : Foldable,
    F[_] : Applicative : Functor,
    A : cats.Monoid]
    (ops : G[F[A]]) : F[A] =
        Functor[F].map(ops.sequence)(Foldable[G].fold(_))
```

Typeclasses

```
val searches = List("raulraja", "dialelo", "pedrovgs") map {
    user => s"null+in:file+user:$user"
}

def op1 =
    reduceOps(searches map { query => Future(searchGithub(query)) })

def op2 =
    reduceOps(
        Option("Software") ::
        Option("Craftsmanship") ::
        Option("Pamplona-Iruñea") :: Nil)
```

Recap

Don't settle for a programming language
that does not support FP.

Recap

Higher Kinded Types matter!

What's next?

Scala Exercises!

Questions? & Thanks!

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