scalaz, functors, monads, and you

Yuvi Masory - @ymasory
Combinatory Solutions Inc
Philly Area Scala Enthusiasts - June 12, 2012
yuvimasory.com/talks

meet your foe

```
def flatMap[B](f: A => StateT[M, S, B])(implicit m: Bind[M]): StateT[M, S, B] =
  stateT[M,S,B](s => apply(s) >>= ((x: (S, A)) => x match { case (sp, a) => f(a)(sp) }))
  semigroup(_.list <::: _)
CanBuildFrom[CC[A], B, CC[B]] forSome {type A; type B}
implicit def Tuple2Traverse[X]: Traverse[({type λ[α]=(X, α)})#λ] =
  new Traverse[({type λ[α]=(X, α)})#λ] {
def traverse[F[_] : Applicative, A, B](f: A => F[B], as: (X, A)): F[(X, B)] =
  f(as._2) ∘ ((b: B) => (as._1, b))
implicit def ZipperTraverse: Traverse[Zipper] =
  new Traverse[Zipper]semigroup(_.value * _.value ∏)
def lift[F[_]](implicit f: Applicative[F]): (F[T1], F[T2]) => F[R] =
  (a: F[T1], b: F[T2]) => (a <**> b)(this)
def promise(implicit s: Strategy): (T1, T2) => Promise[R] = (x: T1, y: T2) =>
  x.pure[Promise].<**>(y.pure[Promise])(k)
implicit def Tuple4Semigroup[A, B, C, D](implicit as: Semigroup[A], bs: Semigroup[B], cs:
  semigroup((a, b) => (a._1 |+| b._1, a._2 |+| b._2, a._3 |+| b._3, a._4 |+| b._4))
def traverse[F[_] : Applicative, A, B](f: A => F[B], za: Zipper[A]): F[Zipper[B]] = {
  val z = (zipper(_: Stream[B], _: B, _: Stream[B])).curried
  val a = implicitly[Applicative[F]]
  a.apply(a.apply(a.fmap(
    a.fmap(TraversableTraverse[Stream].traverse[F, A, B](f, za.lefts.reverse),
           (_: Stream[B]).reverse),
    z), f(za.focus)), TraversableTraverse[Stream].traverse[F, A, B](f, za.rights))
}
```

scalaz 7

- scalaz (*scala-zed*, or *scala-zee*, depending on how cool you are) is a library bringing purely functional typeclasses and libraries to Scala. No compiler plugins, no language support.

- scalaz 7 (to be released), drops unicode, increases modularity and discoverability.

scalaz isn't *all* scary ...

```
"1".toInt
"foo".toInt //uh oh

val iOpt: Option[Int] = "foo".parseInt
iOpt err "not an int!" //compare with get
println(iOpt.isDefined ? "parsed" | "nada")
"i love o-o".println
```

scalaz fixes Java legacies

aka == considered harmful

```
val curUser: Option[RegisteredUser]
val admin: RegisteredUser
if (curUser == admin) {
  //fail, never entered
}
```

=== considered awesome

```
val curUser: Option[RegisteredUser]
val admin: RegisteredUser

implicit def userEqual = equalA[User]
```

```
if (curUser === admin) {
  //DOESN'T COMPILE
}
```

typeclasses

- ad-hoc polymorphism
- you should all be experts ... Erik Osheim (Sep 20, 2011, PHASE, http://plastic-idolatry.com/typcls)
- what, you missed it? here's a 3-slide review.

typeclass review I

`Equal` via subtyping

```
trait Equal[A] {
  // A => A => Boolean
  def ===(rhs: A): Boolean
}

case class Person(
  name: String,
  numCars: Int
) extends Equal[Person] {
  def ===(rhs: A): Boolean = //...
}

Person("yuvi", 0) === Person("colleen", 1)
```

typeclass review II

`Equal` via typeclass pattern

```
trait Equal[A] {
  // A => A => Boolean
  def equals(lhs: A, rhs: A): Boolean
}

case class Person(name: String, numCars: Int)
def PersonHasEqual: Equal[Person] = new Person {
  def equals(lhs: Person, rhs: Person): Boolean = //..
}

PersonHasEqual.equals(
  Person("yuvi", 0),
  Person("colleen", 1)
)
```

typeclass review III

type relationships

```
// A “is a” Equal
def myeq[A <: Equal](lhs: A, rhs: A)
```

```
// A “can be a” Equal
def myeq[A <% Equal](lhs: A, rhs: A)
```

```
// A “has a” Equal
def myeq[A:Equal](lhs: A, rhs: A)
def myeq[A](lhs: A, rhs: A)(ev: Equal[A])
```

functional programming

monoids

things you can add

```
//NOT SCALA SYNTAX

trait Semigroup[A] {
  def |+| : A => A => A
}

trait Monoid[A] extends Semigroup[A] {
  val zero : A
}
```

`Int` is a monoid

oops, `Int` *has a* monoid

```
def IntHasMonoid extends Monoid[Int] {
  def |+| (lhs: Int, rhs: Int): Int =
    lhs + rhs

  val zero: Int = 0
}

1 |+| 1 === 2
```

what else is a monoid?

- `String`
- `List[A]`
- `Option[A:Monoid]`
- Any semigroups that aren't monoids?

what did an explicit `Monoid` representation get us?

Just one random example ...

```
def orZero(opt: Option[A:Monoid]): A =
  opt match {
    case Nil     => zero[A]
    case Some(a) => a
  }

orZero(Some(5)) === 5
val nopt: Option[Int] = None
orZero(nopt) === 0
~ nopt === 0
```

functors

things that can map

```
//NOT SCALA SYNTAX

trait Functor[A] {
  def map[A, B] : F[A] => (A => B) => F[B]
}
```

```
def map(opt: Option[A], fun: A => B): Option[B] =
  opt match {
    case Some(a) => Some(fun(a))
    case None    => None
  }
```

monads

monads

things that can `flatMap`

```
//NOT SCALA SYNTAX

trait Monad[A] extends Functor[A] {

  def map[A, B] :
    M[A] => (A => B) => M[B]

  def flatMap[A, B] :
    M[A] => (A => M[B]) => M[B]
}
```

`Option` is a monad

```
def OptionHasMonad extends Monad[Option] {

  //M[A] => (A => M[B]) => M[B]
  def flatMap(
    opt: Option[A], fun: A => Option[B]
  ): Option[B] = opt match {
    case None    => None
    case Some(a) => fun(a)
  }
}
```

aside: stop matching on `Option`, damnit

```
// Option[A] => (A => B) => Option[B]
Some(5) map { _.toString } === Some("5")

// Option[A] => A, passing default
Some(5) getOrElse 13 === 5
None getOrElse 13 === 13

// Option[A] => B, passing default, from scalaz
Some(5) ? "inty" | "none" === "inty"
None ? "inty" | "none" === "none"

// Option[A] => Option[A], passing default
Some(5) orElse Some(13) === Some(5)
None orElse Some(13) === None

// Option[A] => (A => B) => B => B, catamorphism, scalaz
Some(5) cata (_ + 1, 13) === 6
None cata (_ + 1, 13) === 13
```

aside: stop matching on `Option`, damnit

```
// Option[A] => (A => Option[B]) => Option[B]
Some(100) flatMap { safeLog10(_) } === Some(2)

//Option[A] => Boolean, checking for emptiness
Some(5).isDefined === true
Some(5).isEmpty === false
None.isDefined === false
None.isEmpty === true

// Option[A] => Boolean, using a predicate
Some(5) exists { _ > 0 } === true

// Option[A:Monoid] => A, from scalaz
~ Some(5) === 5
~ None    === 0
```

List is a monad

```
def ListHasMonad extends Monad[List[_]] {

  //M[A] => (A => M[B]) => M[B]
  def flatMap(
    as: List[A], fun: A => List[B]
  ): List[B] = as match {
    case Nil     => Nil
    case h :: t  => fun(h) ++ flatMap(t, fun)
  }
}
```

why can't you handle an A?

given `A`, what problems do ...

- `Option[A]`
- `List[A]`

... protect you from?

IO

a monad you may not know

... or want to know

- What can go wrong with i/o computations?
- Why don't i/o functions fit into functional programming well?

referential transparency

```
def now: Long = System.currentTimeMillis

val n1 = now
val n2 = now

n2 - n1
```

IO typeclass

```
trait IO[A] {

  // "don't call until the end of the universe"
  def unsafePerformIO: A
}
```

See, not all monads can be thought of as collections!

```scala
object CatsWithHatsApp {
  def addHat(p: Picture): Picture = ... // pure

  def pmain(args: Vector[String]): IO[Unit] = {
    val Vector(ext) = args
    val glob = "*." + ext
    for {
      files  <- listDir(glob) // List[File] <- IO[List[File]]
      file   <- files         // File <- List[File]
      inPic  <- readPic(file) // File <- IO[File]
      outPic =  addHat(inPic)
      _      <- writeFile(    // () <- IO[Unit]
        new File(file.getName + ".cat"),
        outPic.serialized
      )
    } yield ()
  }

  def main(args: Array[String]) = {
    val program = pmain.(Vector.empty ++ args).except {
      case e => putStrLn("failing with: " + e.toString)
    }
    program.unsafePerformIO // end of the universe!
  }
}
```

ymasory/puritan

https://github.com/ymasory/puritan

README.md

Puritan: Referentially Transparent IO for Scala

> "There are many things in this world that a child must not ask about ... When an uninstructed multitude attempts to see with its eyes, it is exceedingly apt to be deceived."
>
> - *The Scarlet Letter*, Nathaniel Hawthorne

Puritan lets you do IO with Scalaz 7 without compromising on functional purity. It's also a great way to annoy everyone.

Thank you! Questions?