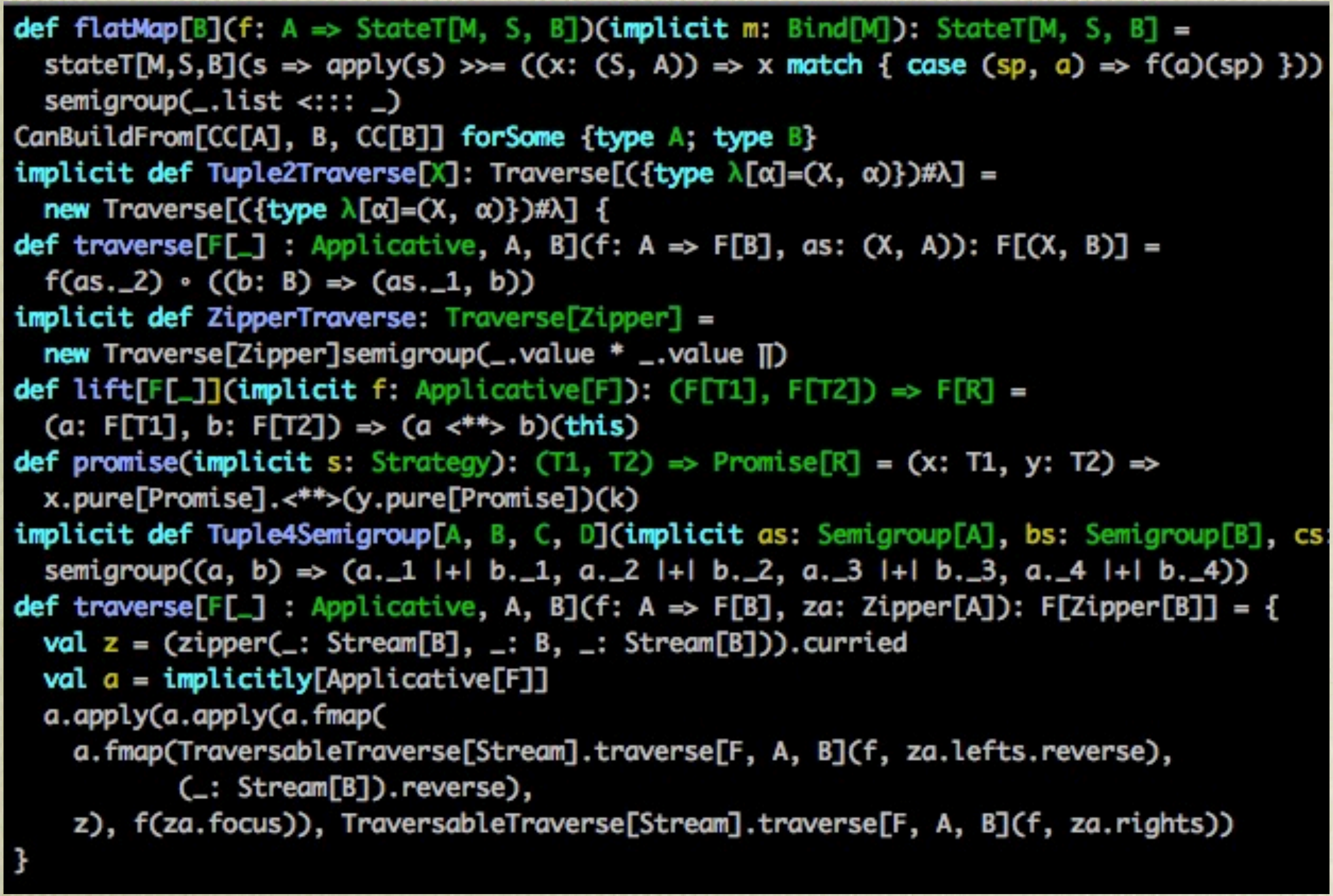
functors, monads, and you

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] {
  override 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 = //..
}

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

typeclass review III:

type relationships

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


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


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

def myeq[A](lhs: A, rhs: A)(ev: Equal[A]) =
  lhs equals rhs
```

functional programming

monoids:

things you can add

```
//NOT SCALA SYNTAX

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

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

Int is a monoid:

oops, Int *has a* monoid

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

  def zero: Int = 0
}

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

What else is a monoid?

- `String`
- `List[A]`
- Any semigroups that aren’t monoids?

functors:

things that can map

```
//NOT SCALA SYNTAX

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

monads

monoids:

things that can flatMap

```
//NOT SCALA SYNTAX

trait Monad[A] extends Functor[A] {

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

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