Introducing Typelevel Scala into an OO Environment

Marcus Henry, Jr.

@dreadedsoftware

You don’t need a Degree to define flatMap

- Understanding Category Theory is not a prerequisite for coding!
- Using the libraries is much simpler than understanding them
- C++ devs take operators more easily than Java devs

Immutability as Default

- Don’t mutate state outside of initializing Function
  - All Function Inputs are Immutable
  - All Function Outputs are Immutable
  - Vars and collection.mutable are local and temporary
  - Function returns are placed into a val

```
def color(str: String): String = {
  str match{
    case "One Fish" => "Red Fish"
    case "Two Fish" => "Blue Fish"
  }
}
```

```
def bad(): mutable.Buffer[String] = {
  val fish = mutable.Buffer[String]()
  var one = "One Fish"
  var two = "Two Fish"

  fish.append(one)
  fish.append(two)

  one = color(one)
  two = color(two)

  fish.append(one)
  fish.append(two)

  fish
}
```

```
def worse(fish: mutable.Buffer[String]): Unit = {
  var one = "One Fish"
  var two = "Two Fish"

  fish.append(one)
  fish.append(two)

  one = color(one)
  two = color(two)

  fish.append(one)
  fish.append(two)
}
```

```
def good(): List[String] = {
  val fish = mutable.Buffer[String]()
  val one = "One Fish"
  val two = "Two Fish"

  fish.append(one)
  fish.append(two)

  fish.append(color(one))
  fish.append(color(two))

  fish.toList
}
```

```
def better(): List[String] = {
  val one = "One Fish"
  val two = "Two Fish"

  List(
      one,
      two,
      color(one),
      color(two)
  )
}
```

Combinators Are Awesome

- Functions produce new state; they don’t destroy old state
- Methods on Structures are Functions
  - Mutable members harm potential sister threads
  - Mutable members confuse data flow (think JSON on the wire)
- Combinators decouple usage from data definition
- Helps replace Loops, Null, Throw
- Handle Bad State at the call site not up the call stack

```
class BadFish(
    private var m_name: String,
    private var m_color: String
){
  def this() = this(null, null)

  def getName(): String = m_name
  def getColor(): String = m_name
  def setName(name: String){
    m_name = name
  }
  def setColor(color: String){
    m_color = color
  }

  def isValid(): Boolean = try{
    check()
    true
  }catch{
    case _: IllegalArgumentException => false
  }

  def check(): Unit = {
    check(m_name, m_color)
  }
  def check(newName: String, newColor: String){
    if(!color(newName).equals(newColor))
      throw new IllegalArgumentException(
          "Fish color and name do not match"
      )
  }
}
```

```
class Fish(val fishName: String){
  val fishColor: String = color(fishName)
  def spawnFish(f: String => String): Fish = {
    new Fish(f(fishName))
  }
}
```

Case Classes & Traits

- Automatic encapsulation
- Immutable by Default
- Data control flow can be fully defined
  - Multiple success Case Classes
  - Single Failure Case Class
  - No need for null or throw on bad input

```
sealed trait Fish{
  val name: String
  val color: String
}
```

```
object NotFish extends Fish{
  override final val name: String = "Ahab"
  override final val color: String = "White Whale"
}
```

```
case object OneFish extends Fish{
  override final val name: String = "One Fish"
  override final val color: String = "Red Fish"
}
case object TwoFish extends Fish{
  override final val name: String = "Two Fish"
  override final val color: String = "Blue Fish"
}
```

```
sealed trait One{self: Fish =>
  override final val name: String = "One Fish"
}
sealed trait Two{self: Fish =>
  override final val name: String = "Two Fish"
}
sealed trait Red{self: Fish =>
  override final val color: String = "Red Fish"
}
sealed trait Blue{self: Fish =>
  override final val color: String = "Blue Fish"
}
object OneFish extends Fish with One with Red
object TwoFish extends Fish with Two with Blue
```

```
sealed trait One extends Fish{
  override final val name: String = "One Fish"
}
sealed trait Two extends Fish{
  override final val name: String = "Two Fish"
}
sealed trait Red extends Fish{
  override final val color: String = "Red Fish"
}
sealed trait Blue extends Fish{
  override final val color: String = "Blue Fish"
}
object OneFish extends Fish with One with Red
object TwoFish extends Fish with Two with Blue
```

Objects are not Coroutines

- Typically Java breaks all three of the previous ideas
- The usual pattern goes something like
  - Initialize
  - Perform some operation
  - Mutate
  - Perform Operation
  - Mutate
  - Etc…
- The Habit needs to be
  - Define
  - Apply Combinator

```scala
import scala.collection.mutable
class BadSchool(){
  private var name: String = null
  private var depth: Depth = null
  private var location: Location = null
  private var fish: mutable.Buffer[Fish] = null

  def setName(newName: String): Unit = {
    name = newName
  }
  def getName(): String = name

  def setDepth(newDepth: Depth): Unit = {
    depth = newDepth
  }
  def getDepth(): Depth = depth

  def setLocation(newLocation: Location): Unit = {
    location = newLocation
  }
  def getLocation(): Location = location

  def setFish(newFish: mutable.Buffer[Fish]): Unit = {
    fish = newFish
  }
  def removeFish(aFish: Fish): Unit = {
    fish -= aFish
  }
  def addFish(aFish: Fish): Unit = {
    fish += aFish
  }
  def getFish(): mutable.Buffer[Fish] = fish

  override def toString(): String= {
    s"School(\n\t$name,\n\t$depth,\n\t$location,\n\t$fish)"
  }
}
```

```scala
def asCoroutine(): Unit = {
  val coroutine = new BadSchool()
  val (name, depth, location, fish) = someInit()
  coroutine.setName(name)
  coroutine.setDepth(depth)
  coroutine.setLocation(location)
  coroutine.setFish(fish)
  convertToJsonAndPutOnTheWire(coroutine)

  var newFish: Fish = null
  for(i <- (0 to 10)){
    newFish = nextFish(coroutine)
    coroutine.addFish(newFish)
    convertToJsonAndPutOnTheWire(coroutine)
  }
}
type InitType = (String, Depth, Location, mutable.Buffer[Fish])
def someInit(): InitType = {
  ("blah", Deep, North, mutable.Buffer(OneFish))
}
def convertToJsonAndPutOnTheWire(school: BadSchool): Unit = {
  println(school)
}
def nextFish(school: BadSchool): Fish = {
  def fish(one: Float): Fish = {
    if(one < scala.util.Random.nextFloat()){
      OneFish
    }else TwoFish
  }
  school.getFish().last match{
    case OneFish => fish(.3f)
    case TwoFish => fish(.7f)
    case _ => fish(.5f)
  }
}
```

```
import scala.collection.immutable
case class School(
    name: String,
    depth: Depth,
    location: Location,
    fish: immutable.Queue[Fish])
def aBetterWay(): Unit = {
  @annotation.tailrec
  def perform(qty: Int, acc: List[School]): List[School] = {
    if(qty > 0 && acc.nonEmpty){
      val head :: tail = acc
      val currentFish = head.fish.last
      val next = nextFish(currentFish)
      val result = head.copy(fish = head.fish.enqueue(next))
      perform(qty - 1, result :: acc)
    }else acc
  }

  val school = School(
      "Bikini Bottom",
      Deep,
      South,
      immutable.Queue(OneFish))
  val result = perform(10, List(school))
  result.foreach(convertToJsonAndPutOnTheWire)
}

def nextFish(current: Fish): Fish = {
  def fish(one: Float): Fish = {
    if(one < scala.util.Random.nextFloat()){
      OneFish
    }else TwoFish
  }
  current match{
    case OneFish => fish(.3f)
    case TwoFish => fish(.7f)
    case _ => fish(.5f)
  }
}
def convertToJsonAndPutOnTheWire(school: School): Unit = {
  println(school)
}
```

Monocle & Argonaut

- Makes it easy to produce & traverse compositions of Case Classes
- Every product of sufficient user base has a persistent settings store
  - Complicated “readLine” based settings become one-liners
  - JSON settings are web (Javascript) friendly

```
import scala.language.higherKinds
case class Color(r: Byte, g: Byte, b: Byte)
case class FishTank(liters: Int, color: Color, fish: List[Fish])

val (tankLiters, tankColor, tankFish) = {
  val gen = GenLens[FishTank]
  (gen(_.liters), gen(_.color), gen(_.fish))
}
val (colorR, colorG, colorB) = {
  val gen = GenLens[Color]
  (gen(_.r), gen(_.g), gen(_.b))
}
val (tankColorR, tankColorG, tankColorB) = (
    tankColor.composeLens(colorR),
    tankColor.composeLens(colorG),
    tankColor.composeLens(colorB)
)

implicit def codecTank: CodecJson[FishTank] =
  casecodec3(
      FishTank.apply, FishTank.unapply
  )("liters", "color", "fish")
implicit def codecColor: CodecJson[Color] =
  casecodec3(
      Color.apply, Color.unapply
  )("r", "g", "b")
implicit def codecFish: CodecJson[Fish] =
  CodecJson(
    (f: Fish) =>
      ("name" := f.name) ->:
      ("color" := f.color) ->:
      jEmptyObject,
    (c: HCursor) => for{
      name <- (c --\ "name").as[String]
      color <- (c --\ "color").as[String]
    }yield{(name, color) match{
      case ("One Fish", "Red Fish") => OneFish
      case ("Two Fish", "Blue Fish") => TwoFish
      case _ => NotFish
    }}
  )
```

```
object settings{
  private val settings: mutable.Map[String, FishTank] =
    mutable.Map()

  def apply(key: String): Option[FishTank] = settings.get(key)
  def update(key: String, byte: Byte): Unit = {
    settings(key) = settings.get(key) match{
      case Some(tank) =>
        tankColor.modify { _ => Color(byte, byte, byte) }(tank)
      case None =>
        FishTank(0, Color(byte, byte, byte), Nil)
    }
  }
  def update(key: String, size: Int): Unit = {
    settings(key) = settings.get(key) match{
      case Some(tank) =>
        tankLiters.modify(_ => size)(tank)
      case _ =>
        FishTank(size, Color(0,0,0), Nil)
    }
  }
  def update(key: String, fish:List[Fish]): Unit = {
    settings(key) = settings.get(key) match{
      case Some(tank) =>
        tankFish.modify(_ => fish)(tank)
      case _ =>
        FishTank(1, Color(0,0,0), fish)
    }
  }

  def persist(): Unit = {
    val jsonRaw = settings.toList.asJson
    val json = jsonRaw.nospaces
    putOnWire(json)
    writeToDisk(json)
  }
  def recall(): Unit = {
    val str = getFromDisk()
    val opt = str.decodeOption[List[(String, FishTank)]]
    opt.foreach{list =>
      settings ++= list.toMap
    }
  }
}
```

```
object asyncSettings{
  private sealed trait Message
  private case class Get(key: String)
      extends Message
  private case class SetGrey(key: String, hue: Byte)
      extends Message

  private class Perform extends Actor{
    override val receive: Receive = step(Map())
    def step(map: Map[String, FishTank]): Receive = {
        case Get(key) => sender ! map(key)
        case SetGrey(key, value) =>
          val newTank: FishTank = ???
          val newMap = map + (key -> newTank)
          context.become(step(newMap))
    }
    override def preStart(): Unit = ???//recall
    override def postStop(): Unit = ???//persist
  }

  val actor: ActorRef = actorSystem.actorOf{
    Props(new Perform())
  }
  def apply(key: String): Future[FishTank] =
    (actor ? Get(key)).collect{
      case Some(t: FishTank) => t
    }
  def update(key: String, hue: Byte) =
    actor ! SetGrey(key, hue)
}
```

Type Classes

- Type Classes decouple functionality from data
- Far more powerful than subclassing
- Application components can expose simple Case Classes and leave usage rules open
- Implicit arguments ensure dependencies exist at compile time

```
trait Adder[Type]{
  def add(other: Type): Type
}
trait Chainer[Arg, Type[Arg]]{
  def chain[Res](f: Arg => Type[Res]): Type[Res]
}

case class Team[Type](members: List[Type])
  extends Adder[Team[Type]]
  with Chainer[Type, Team]{
  override def add(other: Team[Type]): Team[Type] = {
    Team(members ++ other.members)
  }
  override def chain[Res](
      f: Type => Team[Res]): Team[Res] = {
    val list = members.flatMap(member => f(member).members)
    Team(list)
  }
}

case class TeamStructured[Type](members: List[Type])
  extends Adder[TeamStructured[Type]]
  with Chainer[Type, TeamStructured]{
  override def add(
      other: TeamStructured[Type]): TeamStructured[Type] = {
    val (lead1, indi1) = members.splitAt(2)
    val (lead2, indi2) = other.members.splitAt(2)
    TeamStructured(lead1 ++ lead2 ++ indi1 ++ indi2)
  }
  override def chain[Res](
      f: Type => TeamStructured[Res]): TeamStructured[Res] = {
    val (leaders, individuals) = members.map{member =>
      val mems = f(member).members
      mems.splitAt(2)
    }.unzip
    TeamStructured(
        leaders.flatMap {x=>x} ++
        individuals.flatMap{x=>x})
  }
}
```

```
trait Adder[Type[_]]{
  def add[Item](
      left: Type[Item], right: Type[Item]): Type[Item]
}
trait Chainer[Type[_]]{
  def chain[Item, Res](
      arg: Type[Item], f: Item => Type[Res]): Type[Res]
}

case class Team[Type](members: List[Type])
```

```
object structured{
  implicit def adder: Adder[Team] = new Adder[Team]{
    override def add[Item](
        left: Team[Item], right: Team[Item]): Team[Item] = {
      val (lead1, indi1) = left.members.splitAt(2)
      val (lead2, indi2) = right.members.splitAt(2)
      Team(lead1 ++ lead2 ++ indi1 ++ indi2)
    }
  }

  implicit def chainer: Chainer[Team] = new Chainer[Team]{
    override def chain[Item, Res](
        arg: Team[Item], f: Item => Team[Res]): Team[Res] = {
      val (leaders, individuals) = arg.members.map{member =>
        val mems = f(member).members
        mems.splitAt(2)
      }.unzip
      Team(
          leaders.flatMap {x=>x} ++
          individuals.flatMap{x=>x})
    }
  }
}
```

```
object unstructured{
  implicit def adder: Adder[Team] = new Adder[Team]{
    override def add[Item](
        left: Team[Item], right: Team[Item]): Team[Item] = {
      Team(left.members ++ right.members)
    }
  }

  implicit def chainer: Chainer[Team] = new Chainer[Team]{
    override def chain[Item, Res](
        arg: Team[Item], f: Item => Team[Res]): Team[Res] = {
      val list = arg.members.flatMap(
          member => f(member).members)
      Team(list)
    }
  }
}
```

Cats

- Its not as complicated as it seems!!!
- Not Morphism; Function
- Not Monoid; Additive or Multiplicative
- Not Monad; Has map/flatMap
  - think java8 Optional & Stream
  - No real Cpp analog; possible with template<template …>
- Coaching math is not important; coaching usage is
- Coupled Scala monad support is far less powerful than Type Class Monads with Implicits

```
case class Team[Type](members: List[Type])
```

```
object unstructured{
  implicit def adder[Arg]: Monoid[Team[Arg]] =
    new Monoid[Team[Arg]]{
      override def empty: Team[Arg] = Team(Nil)
      override def combine(
          left: Team[Arg], right: Team[Arg]): Team[Arg] = {
        val newMembers = left.members ++ right.members
        Team(newMembers)
      }
    }
  implicit def chainer: Monad[Team] = new Monad[Team]{
    override def flatMap[Arg, Ret](
        team: Team[Arg])(f: Arg => Team[Ret]): Team[Ret] = {
      val newMembers = team.members.flatMap(f(_).members)
      Team(newMembers)
    }
  }
}
```

```
object structured{
  implicit def adder[Arg]: Monoid[Team[Arg]] =
    new Monoid[Team[Arg]]{
      override def empty: Team[Arg] = Team(Nil)
      override def combine(
          left: Team[Arg], right: Team[Arg]): Team[Arg] = {
        val (lead1, indi1) = left.members.splitAt(2)
        val (lead2, indi2) = right.members.splitAt(2)
        val newMembers = lead1 ++ lead2 ++ indi1 ++ indi2
        Team(newMembers)
      }
    }
  implicit def chainer: Monad[Team] = new Monad[Team]{
    override def flatMap[Arg, Ret](
        team: Team[Arg])(f: Arg => Team[Ret]): Team[Ret] = {
      val (leaders, individuals) = team.members.map{member =>
        val mems = f(member).members
        mems.splitAt(2)
      }.unzip
      Team(
          leaders.flatMap {x=>x} ++
          individuals.flatMap{x=>x})
    }
  }
}
```

In Conclusion

- ✓ You don’t need a degree to define flatMap
- ✓ Default to Immutability
- ✓ Combinators over loops, null and throw
- ✓ Case Classes for auto-encapsulation
- ✓ Objects are not coroutines
- ✓ Monocle & Argonaut for settings and JSON
- ✓ Type Classes over Subclasses for functionality
- ✓ Cats for Combinable Structures & Chainable Operations