Introduction to Functional Programming in Scala

Juris Krikis
Evolution Gaming

Juris Krikis

- At Evolution Gaming since 2014
- Head of Scala Department, Riga
- Writing code since 1997
  - Lately Scala
  - Java & JavaScript before

Setting expectations...

- This is an introductory talk
  - Feedback about the first meet-up showed such interest
- If you are an experienced Functional Programmer in Scala, you may not find much new in it
  - You are very welcome to do a speech on a more advanced topic on the next meet-up
  - Such as the Monad Transformers lecture after this one

What is Functional Programming?

- Different definitions exist
  - Treat computation as evaluation of mathematical functions
  - Avoid changing state and mutable data

Also a “programming paradigm”

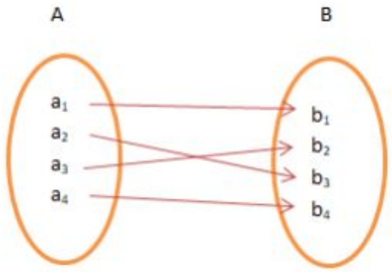
- Not only learning the syntax
- “Patterns”, “culture”, programming style and terminology to learn
- Practices not part of “narrowly defined FP”
- Usually aimed at increasing maintainability and type safety
- Present in some languages, less common in others
- Taking a Scala-centric view today

Why Functional Programming?

- We should write code which obviously has no defects …
  - … instead of code without obvious defects
- Remove complexity from code
  - Functions simpler to understand since they don't depend on state
- Easier to…
  - … understand
  - … maintain
  - … test
  - … refactor
  - … combine / compose
  - … debug
  - … parallelise

Functions

- In programming - a sequence of instructions, packaged as a unit
  - aka procedure, routine, subroutine, callable unit, etc.
- Pure functions
  - No side effects
    - Mutation of variables or by-ref arguments
    - IO
  - Return value depends only on the parameters and its internal logic
    - Always the same if parameters the same
    - Easy to reason about, less to test
- Impure functions
  - Mutate state
  - Do IO



Referential Transparency

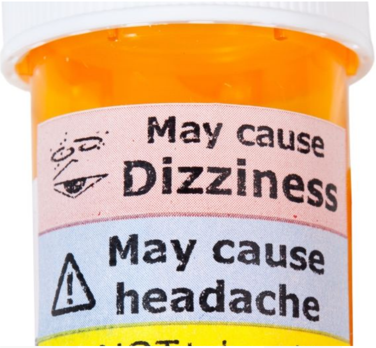
- Expression can be replaced with its resulting value without changing the behaviour of the program
  - Such an expression is pure
  - The expression value is the same for the same inputs
  - Its evaluation has no side effects

```
def squarePure(x: Int): Int = x * x // squarePure(4) can be replaced by 16 everywhere
```

```
def squareImpure(x: Int): Int = {
 println(s"Squaring $x")
 x * x // Replacing squareImpure(4) with 16 will change program functionality
}
```

Side Effects

- A function *f* is pure if *f(x)* is referentially transparent, given a referentially transparent *x*
  - Other functions are side-effecting
- Methods which do side effects often return *Unit* in Scala / *void* in Java
  - (… or an IO Monad)
  - As they should - doing side effects in functions which don’t do so can be misleading
- Examples
  - Mutate function arguments
  - Mutate a variable
  - Input/Output - Console, Network, etc.



Impure function - artificial example

```
var found: Point = _
var bestDistance: Double = _

def closest(x: Point, points: Set[Point]): Point = {
 bestDistance = 10000000d

 points foreach { point =>
   var thisDistance = point.distanceTo(x)
   if (thisDistance < bestDistance) {
     found = point
     bestDistance = thisDistance
   }
 }

 found
}
```

- What issues do you see?

- Null safety
- Magical numbers
- Magical number may be too low
- “found” isn’t properly initialised
- Thread unsafe

Rewritten function

```
def closest(x: Point, points: Set[Point]): Point = points.maxBy(x.distanceTo)
```

- Isn’t it an unfair comparison?
  - *maxBy* does all the work now
  - Internally, it does something similar as we did in imperative code
- This is indeed so… but it’s the result that matters
  - It's more readable and shorter
- *maxBy* is written & tested once, used in many places
  - Takes different functions as a parameter
- Is it actually pure though?

Exceptions in pure functions

```scala
def closest(x: Point, points: Set[Point]): Point = points.maxBy(x.distanceTo)
```

- *maxBy* throws an exception if the points are empty!
- Exceptions thrown aren't values being returned, this is not pure FP
- Let us change the return value

```scala
def closest(x: Point, points: Set[Point]): Option[Point] =
  if (points.isEmpty) None else Some(points.maxBy(x.distanceTo))
```

But we don’t need Scala for this...

```
public Optional<Point> closest(Point x, Set<Point> points) {
 return points.stream().max(
   Comparator.comparingDouble(o -> o.distanceTo(x))
 );
}
```

- Indeed, we don’t!
- All mainstream programming languages now support FP features
  - This is really great!
- Some languages support more features / more conveniently

First-class functions

- Pass functions as arguments
- Return functions as return values
- Assign functions to values / variables
- Named vs. anonymous functions

```scala
val list: List[Double] = List(0.1, 0.2)
val f: Double => Double = x => x * 0.5
list.map(f) // List(0.05, 0.1)
list.map(x => x * 0.5)
list.map(_ * 0.5)
```

Higher-order functions

- Takes one or more functions as arguments

  … and / or …

- Returns a function as its result

```
List(1, 2, 3).map(_ * 2)                       // List(2, 4, 6)
List(1, 2, 3).flatMap(x => List(x, x * 2))  // List(1, 2, 2, 4, 3, 6)
List(1, 2, 3).filter(_ < 2)                    // List(1)

def createMultiplyWith(x: Int): Int => Int = _ * x
val m2 = createMultiplyWith(2)
List(1, 2, 3).map(m2)                          // List(2, 4, 6)
```

FP-style Scala

- Important practices
  - Not part of FP by a strict definition
  - Practiced in conjunction with FP and thus part of “FP-style Scala”
- Catch defects in compile time instead of using tests or runtime
  - The compiler helps you think less and worry less
  - This allows you to be braver about refactoring

Let the compiler help you

- Avoid *Any / AnyRef*
- Avoid *asInstanceOf / isInstanceOf*
- Avoid *null*
- Universal equality isn't type safe
  - Cats Eq
- Type aliases aren't type safe
  - Value classes
  - Shapeless tagged types

```scala
println("test" == 4)


type UserId = String
type AccountId = String

val a: UserId = "test"
val b: AccountId = a
```

Immutability

- Scala has good support for immutable data types
  - *scala.collection.immutable*
  - Case classes
- Prefer immutable values to mutable variables
- Safer in multi-threaded contexts
- Performance is rarely an actual concern
  - Test before deciding to use mutable data
- Modern generational GCs handle object churn well

Encoding Data

- Algebraic Data Types (ADTs)
  - Type formed by combining other types
  - Most data can be - and should be - encoded as ADTs
  - Sum type (disjoint unions)
  - Product type (usually contains fields)
- Implemented in Scala as sealed trait with case classes/objects
  - Case classes are (by default) immutable
  - Exhaustiveness checking when pattern matching on sealed traits!

ADT example

```scala
sealed trait Tree[+T]
case object Empty extends Tree[Nothing]
case class Leaf[T](value: T) extends Tree[T]
case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]

val example: Tree[Int] = Node(
 Node(
   Empty,
   Leaf(2),
 ),
 Node(
   Leaf(4),
   Empty,
 )
)
```

The “don't do it this way”

```scala
def sum(tree: Tree[Int]): Int = {
 if (tree.isInstanceOf[Empty.type]) {
   0
//} else if (tree.isInstanceOf[Leaf[Int]]) { // we forgot to write this part
//  tree.asInstanceOf[Leaf[Int]].value
 } else if (tree.isInstanceOf[Node[Int]]) {
   val x = tree.asInstanceOf[Node[Int]]
   sum(x.left) + sum(x.right)
 } else {
   sys.error(s"Unknown type $tree")
 }
}

sum(example)  // Exception in thread "main" java.lang.RuntimeException: Unknown type Leaf(2)
```

- Runtime exceptions make us very sad

Exhaustiveness checking

```
def sum(tree: Tree[Int]): Int = tree match {
 case Empty => 0
// case Leaf(x) => x // we forgot to write this part
 case Node(left, right) => sum(left) + sum(right)
}

Compile Warning:(53, 37) match may not be exhaustive.
It would fail on the following input: Leaf(_)
    def sum(tree: Tree[Int]): Int = tree match {
```

- The compiler tells us - in compile time - what we missed!
- This is nice, but this just works with *Tree[Int]*, not other *Tree*-s
  - Hold that thought...

Type Classes

- Pattern originating from Haskell
- Extend existing code with new functionality
  - Don’t alter original library source code
- The word *class* doesn’t really mean what *class* means in Java or Scala

Type Class Components

- Type class
  - Interface we want to implement
- Instances for particular types
  - Implementations for types we want to extend
- Interface methods we expose to users
  - Accept instances of the type class as implicit parameters

Type Class Example

```scala
type Json = String // to simplify the example, don't do this for real
case class Person(name: String, age: Int)
```

```scala
trait ConvertableToJson {
  def toJson: Json
}
```

```scala
def write(x: Any): Json = x match {
  case x: Person => writePerson(x)
  case x: Int => writeInt(x)
  case x: Option[Person] => writeOptionPerson(x)
  case x: Option[Int] => writeOptionInt(x)
  case x => sys.error(s"Don't know how to write $x")
}
```

Type Class Example

```scala
// type class interface
trait JsonWriter[A] { def write(value: A): Json }

// interface method to expose to the users
def toJson[A](value: A)(implicit w: JsonWriter[A]): Json =
  w.write(value)

// type class instance for Int
implicit val intWriter: JsonWriter[Int] = _.toString

// using it
toJson(4) // 4
```

Type Class Example

```
// type class instance for String
implicit val stringWriter: JsonWriter[String] = x => s"'$x'"
toJson("something") // 'something'

// type class instance for Person
implicit val personWriter: JsonWriter[Person] = x =>
    s"{name: ${ toJson(x.name) }, age: ${ toJson(x.age) }}"

toJson(Person("James", 25)) // {name: 'James', age: 25}

toJson(47f) // What about Float-s?
```

```
Compile Error:(60, 7) could not find implicit value for parameter w:
JsonWriter[Float]
```

Extending to Option

```scala
implicit def intOptionWriter: JsonWriter[Option[Int]] = x => x match {
  case None    => "null"
  case Some(x) => toJson(x)
}

implicit def personOptionWriter: JsonWriter[Option[Person]] = x => x match {
  case None    => "null"
  case Some(x) => toJson(x)
}

toJson(Person("James", 25).some) // {name: 'James', age: 25}
toJson(none[Person]) // null

toJson(none[Int]) // null
toJson(4.some) // 4
```

Recursive Implicit Resolution

```
implicit def optionWriter[T : JsonWriter]: JsonWriter[Option[T]] = x =>
  x match {
   case None    => "null"
   case Some(x) => toJson(x)
  }


toJson(Person("James", 25).some) // {name: 'James', age: 25}
toJson(none[Person]) // null

toJson(none[Int]) // null
toJson(4.some) // 4
```

Let's return to the tree

❖ We wanted to extend our code to work with various “summables”

```scala
def sum(tree: Tree[Int]): Int = tree match {
 case Empty => 0
 case Leaf(x) => x
 case Node(left, right) => sum(left) + sum(right)
}
```

A wild *Monoid* appears!

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

/* We can “sum” all Tree[T]-s which have a Monoid type class instance for T */
def sum[T](tree: Tree[T])(implicit monoid: Monoid[T]): T = tree match {
 case Empty => monoid.empty
 case Leaf(x) => x
 case Node(left, right) => monoid.combine(sum(left), sum(right))
}
```

- We can now sum any *Tree[T]* for which *Monoid[T]* is defined

Recap - Type Classes

- Allow adding new functionality to a type without changing existing code
- Retain type safety
  - Compiler tells you if a type class resolution fails

Monads

- Really important
  - Monadic programming style is a design pattern
- Lots of ways how they are explained
  - Often easier to explain by example than by a formal definition

API with *Option*-s

```scala
def findUserId(x: Name): Option[UUID] = ???
def findUser(x: UUID): Option[User] = ???
def findAccounts(x: User): Option[List[Account]] = ???

def findAccountsByName(x: String): Option[List[Account]] =
 val userId = findUserId(x)
 userId match {
   case None  => None
   case Some(x) =>
     val user = findUser(x)
     user match {
       case None => None
       case Some(x) =>
         findAccounts(x)
     }
 }
```

for statement

```scala
def flatMap[B](f: A => Option[B]): Option[B] =
   if (isEmpty) None else f(this.get) // in Option

def findAccountsByName(x: String): Option[List[Account]] =
 findUserId(x) flatMap { userId =>
   findUser(userId) flatMap { user =>
     findAccounts(user)
   }
 }
```

```scala
def findAccountsByName(x: String): Option[List[Account]] =
  for {
    userId   <- findUserId(x)
    user     <- findUser(userId)
    accounts <- findAccounts(user)
  } yield accounts
```

API with *Either-s*

```scala
def findUserId(x: Name): Either[Error, UUID] = ???
def findUser(x: UUID): Either[Error, User] = ???
def findAccounts(x: User): Either[Error, List[Account]] = ???
```

```scala
def findAccountsByName(x: String): Either[Error,
List[Account]] =
 val userId = findUserId(x)
 userId match {
   case Left(x)  => Left(x)
   case Right(x) =>
     val user = findUser(x)
     user match {
       case Left(x)  => Left(x)
       case Right(x) =>
         findAccounts(x)
     }
 }
```

for with *Either*

```
def flatMap[A1 >: A, B1](f: B => Either[A1, B1]): Either[A1, B1] = this match {
  case Right(b) => f(b)
  case _        => this.asInstanceOf[Either[A1, B1]]
} // in Either

def findAccountsByName(x: String): Either[Error, List[Account]] =
  for {
    userId   <- findUserId(x)
    user     <- findUser(userId)
    accounts <- findAccounts(user)
  } yield accounts
```

API with IO

```scala
def findUserId(x: String): IO[UUID] = ???
def findUser(x: UUID): IO[User] = ???
def findAccounts(x: User): IO[List[Account]] = ???

def findAccountsByName(x: String): IO[List[Account]] =
 for {
   userId   <- findUserId(x)
   user     <- findUser(userId)
   accounts <- findAccounts(user)
 } yield accounts
```

Abstract over any Monad

```scala
class AccountService[F[_] : Monad] {
 import cats.implicits._

 def findUserId(x: String): F[UUID] = ???
 def findUser(x: UUID): F[User] = ???
 def findAccounts(x: User): F[List[Account]] = ???

 def findAccountsByName(x: String): F[List[Account]] =
   for {
     userId   <- findUserId(x)
     user     <- findUser(userId)
     accounts <- findAccounts(user)
   } yield accounts
```

Monads - conclusion

- A theoretical concept from category theory, with formally defined laws
- A container which helps abstract away our business logic from other "things"
  - Result may or may not exist - Option
  - Result either a success or failure - Either
  - Outside world interaction, possibly async - IO
  - Lots of other monads…
- Really common in Scala

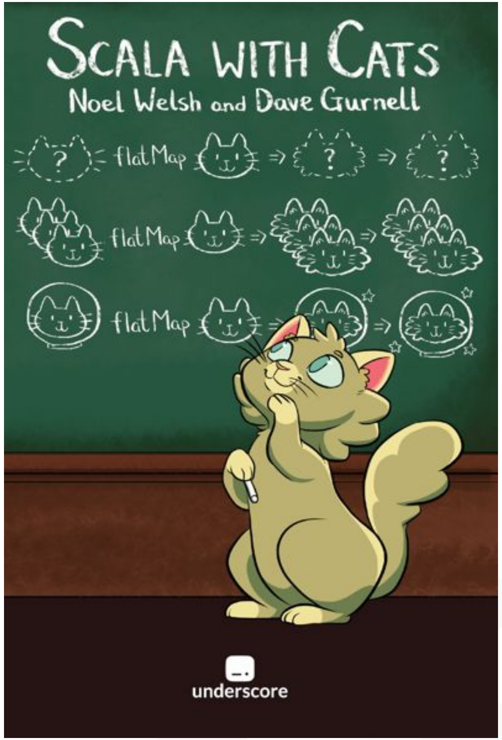
What did we not discuss?

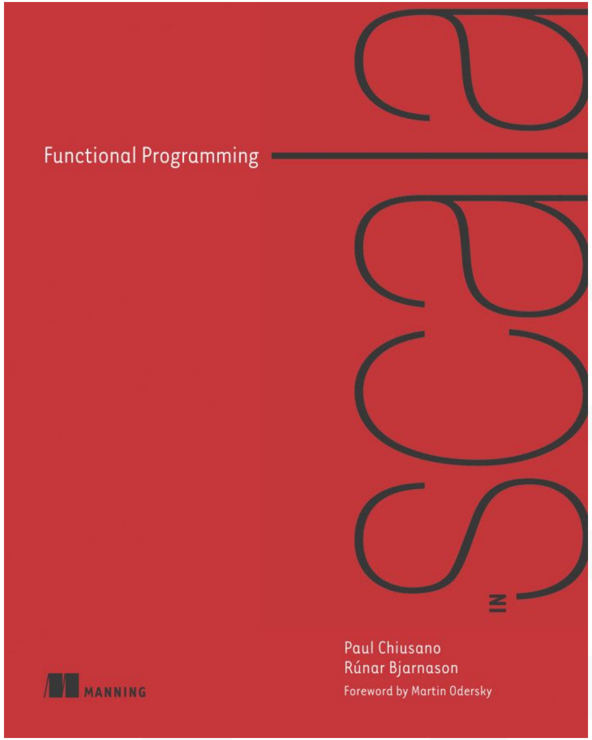
- Doing side effects in functional programming
  - Both synchronous and asynchronous
  - Separation of side effect from the pure part (IO Monad)
- Variances - Covariance, contravariance, invariances
- Type Bounds, Higher Kinded Types, Dependent Types, Lenses
- Other type classes and category theory concepts
  - Functors, Kleislis, Applicatives, etc.
  - Monad Transformers
    - You're in luck, Mikhail will talk about those...

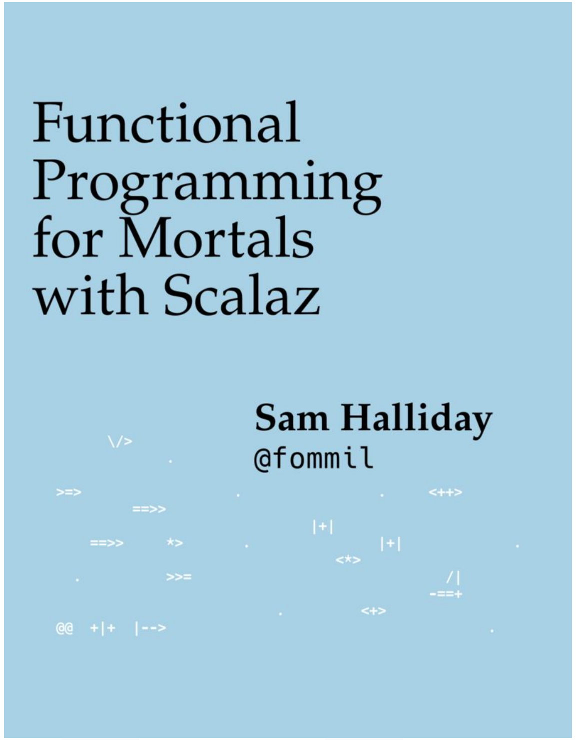
Do we really have to?

- You can write in Scala like a “slightly nicer Java”
- But then you lose out on a lot of the benefits
- And you often don’t understand library code

Further reading...







In conclusion...

- Write code in functional programming style
  - In Scala or in other programming languages
  - When you don't, know why you didn't
- You will be a happier & more productive developer

Thanks for listening!

❖ Suggestions for topics you want to hear on future meetups - tell us!

Q&A

❖ Mikhail will continue with “Monad Transformers - what, when, why”