Effective Scala

Recommendations for a Pleasant Scala Experience

Inspired by:

- Effective ML -- Presentation by Yaron Minsky
- Effective Scala -- Style Guide by Marius Eriksen, Twitter Inc.
- Effective Java -- Book by Joshua Bloch
- Boundaries -- Presentation by Gary Bernhardt

Agenda

- Why Scala?
- (Type) Safety
- Functional Core
- Imperative Shell
- Recommended Practices

Why Scala?

Why Scala?

- JVM and Java Libraries
- Sophisticated Type System
- Allows Functional Programming Techniques
- Concise and Approachable

JVM and Java Libraries

- JVM Safety
  - Sandbox untrusted code from the OS
- Runtime Constraints
- Battle-tested Java Libraries
  - Especially important for security

Sophisticated Type System

- Statically Typed with Type Inference
- Algebraic Data Types
- Type Classes
- Generics
- Higher-Order Functions

Functional Programming Techniques

- Easy to Reason About
- Immutable Data Structures
- Strong Library Support
- Scales Well
- Inherent Safety Gains

Concise and Approachable

- Reduced Boilerplate
- Familiar Syntax for many devs
- "Fusion Approach" of OOP & FP
- Acceptable Learning Curve

Using Scala Effectively

(Type) Safety

Type Safety

- Use the most specific types available
  - *Cats*: `NonEmptyList[A]`
  - *Refined*: `Int Refined Positive`
- When not available, create your own
  - Algebraic Data Types -- `case classes` are cheap
  - `extends AnyVal`, Phantom Types, etc.
- Fighting the compiler? Is there a better approach?

Algebraic Data Types (Primer)

Introduction to Algebraic Data Types in Scala

What do we mean by "algebraic?"

How many values of type `Nothing`*?*

How many values of type `Unit`*?*

How many values of type `Boolean`*?*

How many values of type `Byte`*?*

How many values of type `String`*?*

What do we mean by "algebraic?"

How many values of type `Nothing`*?* → *0*

How many values of type `Unit`*?* → *1*

How many values of type `Boolean`*?* → *2*

How many values of type `Byte`*?* → *256*

How many values of type `String`*?* → *many*

What do we mean by "algebraic?"

How many values of type `(Byte, Boolean)`*?*

How many values of type `(Byte, Unit)`*?*

How many values of type `(Byte, Byte)`*?*

How many values of type `(Byte, Boolean, Boolean)`*?*

How many values of type `(Boolean, String, Nothing)`*?*

What do we mean by "algebraic?"

How many of type `(Byte, Boolean)`*?* → *2 × 256 = 512*

How many of type `(Byte, Unit)`*?* → *256 × 1 = 256*

How many of type `(Byte, Byte)`*?* → *256 × 256 = 65536*

How many of type `(Byte, Boolean, Boolean)`*?* → *256 × 2 × 2 = 1024*

How many of type `(Boolean, String, Nothing)`*?* → *2 × many × 0 = 0*

What do we mean by "algebraic?"

How many of type `(Byte, Boolean)`*?* → *2 × 256 = 512*

How many of type `(Byte, Unit)`*?* → *256 × 1 = 256*

How many of type `(Byte, Byte)`*?* → *256 × 256 = 65536*

How many of type `(Byte, Boolean, Boolean)`*?* → *256 × 2 × 2 = 1024*

How many of type `(Boolean, String, Nothing)`*?* → *2 × many × 0 = 0*

Product types! This *and* That

Product types

Tuples!

```
type Person = (String, Int)
```

Classes!

```
case class ScalaPerson(name: String, age: Int)

class JavaPerson {
  final String name;
  final Int age;
}
```

What do we mean by "algebraic?"

How many values of type `Byte` *or* `Boolean`*?*

How many values of type `Boolean` *or* `Unit`*?*

How many values of type `(Byte, Boolean)` *or* `Boolean`*?*

How many values of type `Boolean` *or* `(String, Nothing)`*?*

What do we mean by "algebraic?"

How many of type `Byte` *or* `Boolean`*?* → *2 + 256 = 258*

How many of type `Boolean` *or* `Unit`*?* → *2 + 1 = 3*

How many of type `(Byte, Boolean)` *or* `Boolean`*?* → *(256 × 2) + 2 = 514*

How many of type `Boolean` *or* `(String, Nothing)`*?* → *2 + (many × 0) = 2*

What do we mean by "algebraic?"

How many of type `Byte` *or* `Boolean`*?* → *2 + 256 = 258*

How many of type `Boolean` *or* `Unit`*?* → *2 + 1 = 3*

How many of type `(Byte, Boolean)` *or* `Boolean`*?* → *(256 × 2) + 2 = 514*

How many of type `Boolean` *or* `(String, Nothing)`*?* → *2 + (many × 0) = 2*

Sum types! This *or* That

Option

```
val maybeByte: Option[Byte] = Some(0x07)
```

Either

```
val test: Either[String, Byte] = Left("Could not read byte")
```

Make Illegal States Unrepresentable

– Yaron Minsky

Make Illegal States Unrepresentable

```
case class LibraryBook(isbn: Int, atLibrary: Option[String], dueDate: Option[Long], checkedOutBy: Option[String])

def checkOut(book: LibraryBook, cardHolder: String): LibraryBook = {
  LibraryBook(book.isbn, None, Some(System.currentTimeMillis()), Some(cardHolder))
}
```

Make Illegal States Unrepresentable

```
val book1 = LibraryBook(123, Some("Multnomah County"), None, None)
// book1: LibraryBook = LibraryBook(
//   isbn = 123,
//   atLibrary = Some(value = "Multnomah County"),
//   dueDate = None,
//   checkedOutBy = None
// )
val checkedOut = checkOut(book1, "Alice")
// checkedOut: LibraryBook = LibraryBook(
//   isbn = 123,
//   atLibrary = None,
//   dueDate = Some(value = 1601611920961L),
//   checkedOutBy = Some(value = "Alice")
// )
```

Make Illegal States Unrepresentable

```
def remind(cardHolder: String, isbn: Int): String = {
  s"Hey $cardHolder! Give us back $isbn!"
}

def sendRemindersStub(books: List[LibraryBook]): List[String] = ???
```

collect

```
val books = List(checkedOut)
// books: List[LibraryBook] = List(
//   LibraryBook(
//     isbn = 123,
//     atLibrary = None,
//     dueDate = Some(value = 1601611920961L),
//     checkedOutBy = Some(value = "Alice")
//   )
// )
def sendReminders1(books: List[LibraryBook]): List[String] = {
  books.collect { case LibraryBook(isbn, _, Some(date), Some(person))
    if date < System.currentTimeMillis() => remind(person, isbn)
  }
}
sendReminders1(books)
// res0: List[String] = List("Hey Alice! Give us back 123!")
```

Invalid Data?

```
// checkout book with no person?
val invalid = LibraryBook(321, None, Some(System.currentTimeMillis()), None)
// invalid: LibraryBook = LibraryBook(
//   isbn = 321,
//   atLibrary = None,
//   dueDate = Some(value = 1601611920963L),
//   checkedOutBy = None
// )
```

Invalid Data?

```
val mixed1 = List(checkedOut, invalid)
// mixed1: List[LibraryBook] = List(
//   LibraryBook(
//     isbn = 123,
//     atLibrary = None,
//     dueDate = Some(value = 1601611920961L),
//     checkedOutBy = Some(value = "Alice")
//   ),
//   LibraryBook(
//     isbn = 321,
//     atLibrary = None,
//     dueDate = Some(value = 1601611920963L),
//     checkedOutBy = None
//   )
// )
sendReminders1(mixed1)  // silent failure!
// res1: List[String] = List("Hey Alice! Give us back 123!")
```

Better Types

```
case class AtLibraryBook(isbn: Int, atLibrary: String)
case class CheckedOutBook(isbn: Int, dueDate: Long, checkedOutBy: String)

def checkOut(book: AtLibraryBook, cardHolder: String): CheckedOutBook = {
  CheckedOutBook(book.isbn, System.currentTimeMillis(), cardHolder)
}
```

Better Types

```
val book2 = AtLibraryBook(345, "Multnomah County")
// book2: AtLibraryBook = AtLibraryBook(
//   isbn = 345,
//   atLibrary = "Multnomah County"
// )
val checkedOut2 = checkOut(book2, "Bob")
// checkedOut2: CheckedOutBook = CheckedOutBook(
//   isbn = 345,
//   dueDate = 1601611920966L,
//   checkedOutBy = "Bob"
// )
def sendReminders2(books: List[CheckedOutBook]): List[String] = {
  books.map(b => remind(b.checkedOutBy, b.isbn))
}
```

Invalid Data?

```
val mixed2 = List(book2, checkedOut2)  // bad state
// mixed2: List[Product with Object with Serializable] = List(
//   AtLibraryBook(isbn = 345, atLibrary = "Multnomah County"),
//   CheckedOutBook(isbn = 345, dueDate = 1601611920966L, checkedOutBy = "Bob")
// )

sendReminders2(mixed2)  // won't compile!
// error: type mismatch;
//  found   : List[Product with java.io.Serializable]
//  required: List[repl.MdocSession.App.CheckedOutBook]
// sendReminders2(mixed2)  // won't compile!
//                ^^^^^^
```

Functional core,
imperative shell

– Gary Bernhardt

Functional Core

Functional Programming Concepts

- Immutability
- Higher Order Functions / Higher Kinded types
- Total vs Partial Functions
- Referential Transparency

We reason about our programs by substitution.
– Rob Norris

Referential Transparency

Are these the same program?

```
// program 1
val a = compute(5)
(a, a)

// program 2
(compute(5), compute(5))
```

Referential Transparency

Are these the same program?

```
// program 1
val a = compute(5)
(a, a)

// program 2
(compute(5), compute(5))
```

It depends...

Referential Transparency

Are these the same program?

```
// program 1
val a = compute(5)
(a, a)

// program 2
(compute(5), compute(5))
```

For functional programming, the answer is always YES.

Referential Transparency

- Every expression is either referentially transparent, or
- it is a **side-effect**.
- This is a **syntactic property of programs**

Referential Transparency

- Functions must be:
  - Deterministic
  - Total
  - Pure

Referential Transparency

By counterexample: *Determinism*

```
import java.security.SecureRandom

val rand = new SecureRandom
// rand: SecureRandom = NativePRNG
rand.nextInt(100)
// res3: Int = 3
rand.nextInt(100)
// res4: Int = 35
```

Referential Transparency

By counterexample: *Totality*

```
def divide(num: Int, denom: Int): Int = num / denom
```

```
divide(15, 0)
// java.lang.ArithmeticException: / by zero
//  at repl.MdocSession$App.divide(effective-scala.md:125)
//  at repl.MdocSession$App$$anonfun$15.apply$mcI$sp(effective-scala.md:132)
//  at repl.MdocSession$App$$anonfun$15.apply(effective-scala.md:132)
//  at repl.MdocSession$App$$anonfun$15.apply(effective-scala.md:132)
```

Referential Transparency

By counterexample: *Pure*

```
def reportedIncrement(x: Int): Int = {
  println(s"Was $x, is now ${x + 1}")
  x + 1
}
val a = reportedIncrement(5)
// Was 5, is now 6
// a: Int = 6
(a, a)
// res5: (Int, Int) = (6, 6)
(reportedIncrement(5), reportedIncrement(5))
// Was 5, is now 6
// Was 5, is now 6
// res6: (Int, Int) = (6, 6)
```

Referential Transparency

- Functions must be:
  - Deterministic
  - Total
  - Pure
- How do we do anything useful?

Referential Transparency

- Functions must be:
  - Deterministic
  - Total
  - Pure
- How do we do anything useful?
  - *Effects*

Effect vs Side-Effect

- Effects are **good**
- Side-effects are **bugs**

Partial Function

```
divide(15, 0)
// java.lang.ArithmeticException: / by zero
//  at repl.MdocSession$App.divide(effective-scala.md:125)
//  at repl.MdocSession$App$$anonfun$22.apply$mcI$sp(effective-scala.md:158)
//  at repl.MdocSession$App$$anonfun$22.apply(effective-scala.md:158)
//  at repl.MdocSession$App$$anonfun$22.apply(effective-scala.md:158)
```

Total Function

```
import scala.util._

def safeDivide(num: Int, denom: Int): Try[Int] = {
  Try(num / denom)
}
```

Total Function

```
safeDivide(15, 0)
// res7: Try[Int] = Failure(
//   exception = java.lang.ArithmeticException: / by zero
// )
```

Total Function

```
val denom = 3
// denom: Int = 3
safeDivide(15, denom) match {
  case Success(num) =>
    List.fill(denom)(s"$num for you").mkString(", and ")
  case Failure(err) =>
    err.getMessage
}
// res8: String = "5 for you, and 5 for you, and 5 for you"
```

Functional Error Handling

Representation	When to use
Exception	Avoid
Option	Modeling Absence
Try	Capturing Throwable
Either	Sequential Errors
Validated	Parallel Errors

Error Handling in Scala with FP

Functional Error Handling

~~Try~~, `Either.catchNonFatal` from *Cats*

Representation	When to use
Exception	Avoid
Option	Modeling Absence
Either	Capturing Throwable
Either	Sequential Errors
Validated	Parallel Errors

Error Handling in Scala with FP

Effects

`F[A]`

This is a program in F that computes a value of type A

Imperative Shell

Imperative Shell

"End of the World"

- Http
- Database
- Logging
- Etc.

cats.effect.IO

cats.effect.IO

```scala
import cats.effect.IO

def delayedIncrement(x: Int): IO[Int] = IO {
  println(s"Was $x, is now ${x + 1}")
  x + 1
}
```

cats.effect.IO

```
// program 1
val a = delayedIncrement(5)
// a: IO[Int] = IO$250138122
(a, a)
// res10: (IO[Int], IO[Int]) = (IO$250138122,IO$250138122)

// program 2
(delayedIncrement(5), delayedIncrement(5))
// res11: (IO[Int], IO[Int]) = (IO$276867782,IO$1637561061)
```

vs scala.concurrent.Future

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def futureIncrement(x: Int): Future[Int] = Future {
  println(s"Was $x, is now ${x + 1}")
  x + 1
}
```

vs scala.concurrent.Future

```
// program 1
val b = futureIncrement(5)
// Was 5, is now 6
// b: Future[Int] = Future(Success(6))
(b, b)
// res12: (Future[Int], Future[Int]) = (Future(Success(6)), Future(Success(6)))

// program 2
(futureIncrement(5), futureIncrement(5))
// Was 5, is now 6
// Was 5, is now 6
// res13: (Future[Int], Future[Int]) = (Future(Success(6)), Future(Success(6)))
```

The End of the World

cats.effect.IO

```
val prog = delayedIncrement(5)
// prog: IO[Int] = IO$58937229
prog.unsafeRunSync() // "end of the world"
// Was 5, is now 6
// res14: Int = 6 // "end of the world"

prog.flatMap(delayedIncrement).unsafeRunSync() // "end of the world"
// Was 5, is now 6
// Was 6, is now 7
// res15: Int = 7
```

cats.effect.IO

```
def delayedDecrement(x: Int): IO[Int] = IO {
  println(s"Was $x, is now ${x - 1}")
  x - 1
}

val program = for {
  x <- delayedIncrement(5)
  y <- delayedIncrement(x)
  z <- delayedDecrement(y)
} yield z
// program: IO[Int] = IO$556138842
```

cats.effect.IO

```
program // just a _value_
// res16: IO[Int] = IO$556138842 // just a _value_

program.unsafeRunSync() // "end of the world"
// Was 5, is now 6
// Was 6, is now 7
// Was 7, is now 6
// res17: Int = 6
```

Best Practices

Favor Code Readers Over Code Writers

- Capture invariance in types rather than in the logic surrounding the types
- Make Common Errors Obvious
- Avoid Complex Type Hackery
- Don't Be Puritanical About Purity

Effective ML -- Yaron Minsky

Style Suggestions

- Prefer `List` to `Seq`
- Avoid `return`
- Prefer return type annotations
  - Required for recursion
  - Often helps type inference and compile times
- Prefer Explicit conversions
  - Can be *provided* by `implicits`

References:

- Effective ML -- Yaron Minsky
- Effective Scala -- Marius Eriksen, Twitter Inc.
- Effective Scala: Reloaded! -- Mirco Dotta
- Effective Java -- Joshua Bloch
- Boundaries -- Gary Bernhardt
- Programming with Effects -- Rob Norris
- Introduction to Algebraic Data Types in Scala -- Rob Norris
- Moving Beyond Defensive Coding -- Changlin Li
- Thinking Less with Scala -- Daniel Sivan
- Benefits of IO? -- Reddit Post
- Error Handling in Scala with FP -- Jun Tomioka