Functional Programming Principles in Scala

Putting the Fun in Functional Programming

Christian Simon

illucIT Software GmbH

Outline

- Functional Programming Principles
  - Functions as First-Class objects
  - Higher Order Functions
  - Side-Effects
  - Pure Functions
  - Immutability
  - State
- Function Operations
  - Partially Applied Functions
  - Currying
  - Functional Composition
- Monads
  - Monad Operations
  - For-Comprehensions
  - Collections
  - Option, Either, Try
  - Future
- Functional Patterns
  - Fold
  - Decomposition (TBD)
  - Extractor Objects (TBD)
  - Recursion (TBD)
  - Tail-Recursion (TBD)
  - Magnet Pattern (TBD)

Functional Programming

Principles of the Functional Programming Paradigm

Functions as First-Class Objects

- Classic OOP
  - State is managed in Objects
  - Methods change and/or create objects
  - Polymorphism

- Functional programming
  - Functions as variables, arguments, return values
  - Functions are stateless and deterministic
  - Higher-level functions allow algorithms with variable implementations
  - Composition of functions to create complex logic
  - Side-Effects are limited in function executions
  - Making heavy used of immutable data structures

Functions as First-Class Objects (2)

Example (E1):

```
// Function as Type
type Fun = String => Boolean

// Function as value
val print: String => Unit = println

// Function as parameter
def printValues(values: Seq[Int], printNumber: Int => Unit): Unit =
  values.foreach(printNumber)
```

Higher Order Function

"Functions that work on functions, not on values"

Example (E2):

```
// General Function composition
def compose[A, B, C](fun: A => B, nextFun: B => C): A => C = {
  a => nextFun(fun(a))
}

// Apply given function to generic collection of values
def forEach[A](values: TraversableOnce[A], consume: A => Unit): Unit =
  values.foreach(consume)
```

Side Effects

"Every action that consumes or changes the state outside of the function, except arguments and return value of the function"

Examples:

- Print to STDOUT or Log-File => Side Effect!
- Read a File => Side Effect!
- Change non-local Variable => Side Effect!
- Create Random Number => Side Effect!
- Read current system time => Side Effect!
- Modify mutable value of parameter => Side Effect!

Side Effects (2)

- Problems:
  - Non-deterministic
  - External dependencies that are not clear

  ⇨ **Hard to test**

  - Race conditions

  ⇨ **Hard to parallelize**

- Mitigations:
  - Instead of provoking side effects, return information about actions to take
  - Encapsulate in Monads

Pure Functions

Totally deterministic by its input values, no side effects

<u>**“Referential Transparency”:**</u>

Each occurrence of the function can be replaced by the deterministic value of the function application for the given input without altering the program logic

Pure Functions (2)

Example (E3):

```
var lastCheckedMillis: Long = _

def nonPureDateMillis(): Long = {
  // Side Effect: Read global state
  val now = LocalDateTime.now()
  // Side Effect: Throw Exception
  if (now.getDayOfWeek == DayOfWeek.MONDAY) throw new IllegalStateException("Mondays not allowed")
  val millis = now.toInstant(ZoneOffset.UTC).toEpochMilli
  // Side Effect: Change global state
  lastCheckedMillis = millis
  millis
}

def pureDateMillis(now: LocalDateTime): Try[Long] = {
  now.getDayOfWeek match {
    case DayOfWeek.MONDAY => Failure(new IllegalStateException("Mondays not allowed"))
    case _ => Success(now.toInstant(ZoneOffset.UTC).toEpochMilli)
  }
}
```

Immutablility

- Immutable data structure are inherently useful for FP:
  - Cannot be changed when given as function parameter
  - No concurrency issues
  - Can be cached after calculation, because it cannot change any more

- Only if all properties are immutable, an object can be immutable
  - E.g. immutable List of mutable StringBuffer is still mutable

- Rather use mutable reference (var) of immutable data structure (e.g. immutable.List) than immutable reference (val) of mutable data structure (e.g. mutable.List)
  - Values can leave the scope without the risk of being modified on the outside
  - No extra Data Transfer Object needed

State

- Most non-trivial applications need state
- Problem: Managing State is difficult, modifying state from many positions can lead to all kinds of bugs, hard to verify
- Idea:
  - Isolate state management to a small part of the program
  - Use pure function to perform complex calculations, providing required values from the state as input parameters and return mutated state (copy) as function result
  - Pure functions build network/flow for calculation

Function Operations

Partially Applied Functions

- Function with multiple arguments, reduced to function with some of the arguments already filled in

- Function with multiple argument blocks (= higher order function) applied for some of the parameter blocks, creating function with less degrees of freedom

- Example (E4):

```
def sum(x: Int, y: Int): Int = x + y

def sum5(y: Int): Int = sum(5, y)

val sum99: Int => Int = sum(99, _)
```

Currying

- Converting a function with multiple arguments to a function with multiple argument blocks
- E.g.

```
(String, String, Int) => Boolean
```

converted to

```
String => String => Int => Boolean
```

- Easy to partially apply

Currying (2)

Example (E5):

```scala
// Function which takes 1 argument Request and return function Response => Unit
type LogRequestFuncCurried = Request => Response => Unit

val logWithTimeStamp: LogRequestFuncCurried = req => {
  val begin = LocalDateTime.now()
  res => {
    val end = LocalDateTime.now()
    println("Response took " + SECONDS.between(begin, end) + " seconds")
  }
}
```

Functional Composition

- Build network of complex functions by combining simple building blocks
- Example (E6):

```scala
case class Person(firstName: String, lastName: String, age: Option[Int])

def loadPersons(): Seq[Person] = ???

def hasAge(predicate: Int => Boolean)(person: Person): Boolean = person.age.exists(predicate)

def legalAge(age: Int): Boolean = age >= 18

def named(firstName: String)(person: Person): Boolean = person.firstName == firstName

val namedCharles: Person => String = named("Charles")

def fullName(person: Person): String = s"${person.firstName} ${person.lastName}"

val nameRegisterForAdultCharles =
  loadPersons()
    .filter(hasAge(legalAge))
    .filter(namedCharles)
    .map(fullName)
    .mkString("Name Register: ", ", ", "")
```

Monads

Monads

- Higher Order types `M[A]` for element type `A`
- Encapsulates values of type `A` with additional semantic
- Can be used to encapsulate side effects (e.g. Try, Future)
- Allows working with values `A`, without having the need to materialize them
- Common subset of transformation function with similar semantic
- Examples:
  - Collections
  - Option
  - Either
  - Try
  - Future

Monad Operations

```scala
trait ExampleMonad[+A] {

  // Transform value of type A to type B
  def map[B](f: A => B): ExampleMonad[B]

  // Transform value of type A to type Monad[B]
  def flatMap[B](f: A => ExampleMonad[B]): ExampleMonad[B]

  // Filter Monad values by predicate
  def filter(p: A => Boolean): ExampleMonad[A]

}
```

For-Comprehensions

- Scala Syntax "`for { … } yield {…}`" syntactic sugar for Monad operations
- Each "`for`" clause is bound together by `flatMap`
- "`yield`" clause is connected with `map`
- "`if`" conditions are applied by `filter`
- Can be used with all typed which implement the said Monad operation (but only between the same type)

For-Comprehensions (2)

- Example (E7)

```scala
case class Person(lastName: String, firstName: Option[String], deceased: Boolean)

def loadPerson(): Option[Person] = ???

def getCreditCardNumber(firstName: String, lastName: String): Option[String] = ???

def creditCardValid(cardNumber: String): Boolean = ???

val creditReport: Option[String] = for {
  person <- loadPerson() if !person.deceased
  firstName <- person.firstName
  fullName = s"$firstName ${person.lastName}"
  creditCardNumber <- getCreditCardNumber(firstName, person.lastName) if creditCardValid(creditCardNumber)
} yield {
  s"Credit report for $fullName: Card number $creditCardNumber valid"
}
```

Try / Either

- Try[A]:
  - Success[A](a: A) if the operation was success
  - Failure(e: Throwable) if the operation raised an Exception
  - Try{} factory can be used to catch NonFatal Exceptions

- Either[A, B]
  - Right[_, B](b: B), indicating that the result is correct ("right")
  - Left[A, _](a: A), indicating that the result was wrong (e.g. value explaining the problem)
  - Functions map, flatMap applies to right side (since Scala 2.12)
  - Generalization of Try, but failure must not be of type Throwable, and is typed by B

Future

- Value is evaluated in an asynchronous process by a Executor
- Chain of functions can be built together in synchronous process, and will be evaluated, when the value is ready
- When completed, contains a Try, so can be either Success or Failure
- To force into synchronous process, use Await.result or Await.ready
- Attach Side-Effect to Future with onComplete

Future (2)

- Example (E8)

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

val future: Future[Int] = Future {
  Range(1, 1000000).sum // Operation that might take some time
}

def loadValueFromDataBase(): Future[Int] = ???

val futurePlus100 = future map { _ + 100 }

val futureFlatMapped: Future[Int] = future flatMap { value =>
  loadValueFromDataBase() map { dbValue =>
    value + dbValue
  }
}

future.onComplete {
  case Success(number) => println(s"Result: $number")
  case Failure(e) => println("No luck this time ;-(")
}

val result = Await.result(futurePlus100, 10.seconds)
```

Functional Patterns

How to apply Functional Programming in daily work

Fold / FoldLeft

- Reduce collection by aggregation function
- Start with neutral element (“zero”)
- Give algorithm to combine aggregate with next element
- Always Tail-recursive

Fold / FoldLeft (2)

- Example (E10):

```scala
def sum(numbers: Seq[Int]): Int = numbers.fold(0)(_ + _)

def product(numbers: Seq[Int]): Int = numbers.fold(1)(_ * _)

def max(numbers: Seq[Int]): Int = numbers.fold(Int.MinValue)((aggregate, next) => if (aggregate < next) next else aggregate)

def mean(numbers: Seq[Double]): Double = {
  val (sum, count) = numbers.foldLeft((0.0, 0)) {
    case ((currentSum, currentCount), nextNumber) => (currentSum + nextNumber, currentCount + 1)
  }
  if (count == 0) Double.NaN else sum / count
}

def combineLines(lines: Seq[String]): String = lines.fold("")(_ + "\n" + _)
```

Thank you for your attention !

github.com/metaxmx

simon@illucit.com

www.illucit.com