Higher-Order Functions In Scala

Concepts applicable to most programming languages

Functional Programming Guild - 20 April 2022

Introduction: Most Common Higher-Order Functions

That appear in Java, Scala, Typescript, Node, Python…

- Filter, forAll
- Map, flatMap
- ForEach (Scala Collection version)
- Fold, foldLeft, foldRight
- Reduce, reduceLeft, reduceLeftOption
- Scan, scanLeft, scanRight

Introduction: What Are Functions Anyway?

Basic syntax and concepts

```
// function definition:
def functionName(paramName1: Type1, paramName1: Type2): ReturnType =  {
  <implementation>
}

def successorWithParam(x: Int): Int =
  x + 1

def successorLambda: Int => Int =
  x => x + 1

// function invocation:
val two1 = successorWithParam(1)
val two2 = successorLambda(1)
```

Introduction: What Are Functions Anyway?

Basic syntax and concepts

"param1" and "param2" are called "formal" parameters

```
// function definition:
def functionName(param1: Int, param2: String): String =  {
 "Param1=" + param1.toString + ", Param2=" + param2
}

def successorWithParam(x: Int): Int =
  x + 1

def successorLambda: Int => Int =
  x => x + 1

// function invocation:
val two1 = successorWithParam(1)
val two2 = successorLambda(1)
```

The "1"s here are the "real parameters

Introduction: What Are Functions Anyway?

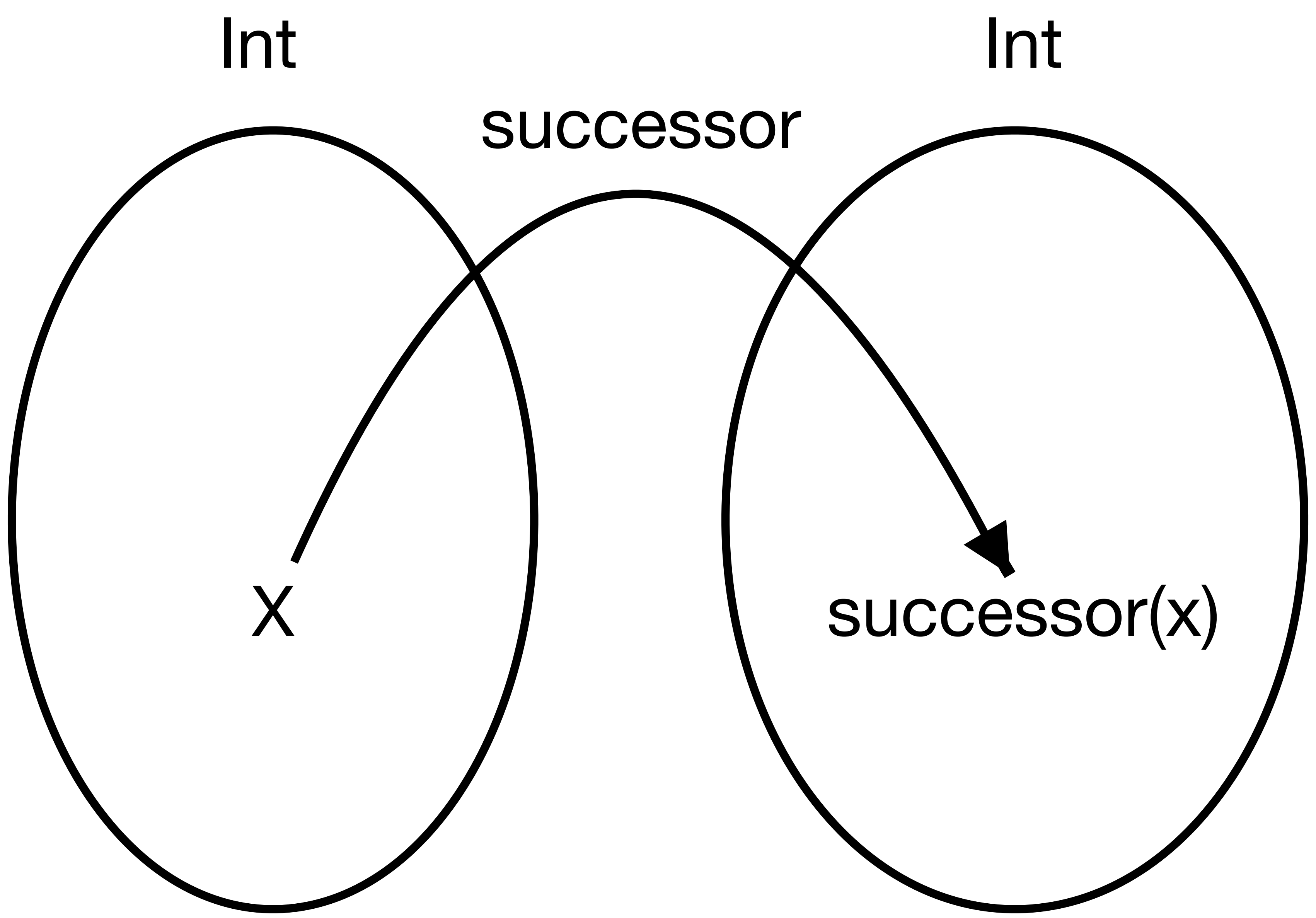
Basic syntax and concepts

```scala
// function definition:
def functionName(param1: Int, param2: String): String =  {
 "Param1=" + param1.toString + ", Param2=" + param2
}

def successorWithParam(x: Int): Int =
  x + 1

def successor: Int => Int =
  x => x + 1

// function invocation:
val two1 = successorWithParam(1)
val two2 = successor(1)
```



Introduction: Partial Application Of Functions

Technique know as “Currying”

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

def curriedSum(x: Int)(y: Int): Int =    x + y

def sumX: Int => Int => Int =            x => y => x + y

def successorVersion2: Int => Int =      sumX(1)

def successorVersion3: Int => Int =      curriedSum(1)
```

Introduction: Lists and Pattern Matching

Scala syntax for defining recursion over lists

```scala
def sumOfElements(list: List[Int]): Int = list match {
  case Nil => 0
  case x :: xs => x + sumOfElements(xs)
}

def lengthOfList(list: List[Int]): Int = list match {
  case Nil => 0
  case _ :: xs => 1 + lengthOfList(xs)
}

val len = lengthOfList(List(2,2,3,4,3,2)) // len: 6
val sum = sumOfElements(List(1,2,3,4,5)) // sum: 15
```

Introduction: Higher-Order Functions

Taking functions as parameters

```
def doubleSalary(x: Int): Int = x * 2

def processList(x: List[Int], f: Int => Int): List[Int] = x match {
  case Nil => Nil
  case x::xs => f(x) :: processList(xs, f)
}


val salaries = List(20000, 70000, 40000)
val newSalaries = processList(salaries, doubleSalary)// List(40000, 140000, 80000)
```

Introduction: Higher-Order Functions

Returning functions as values

```scala
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"

def getURL = urlBuilder(ssl = true, domainName)

val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query)
```

// "https://www.example.com/users?id=1": String

Introduction: Blocks Of Code

The last expression is the result value

```
def doubleApplication(x: Int)(y: Int)(f: Int => Int):
(Int, Int) = (f(x), f(y))
```

Introduction: Blocks Of Code

Lambdas are also code and can be passed as parameters !!

```
def doubleApplication(x: Int)(y: Int)(f: Int => Int):
(Int, Int) = (f(x), f(y))


val fancy = doubleApplication(1)(2)(num => num * 4)
```

Introduction: Blocks Of Code

Lamdas are also code !!

```
def doubleApplication(x: Int)(y: Int)(f: Int => Int):
(Int, Int) = (f(x), f(y))

def f1: Int => Int = num => num * 4

val fancy = doubleApplication(1)(2)(f1)
```

Introduction: Blocks Of Code

Lambdas are also code !!

```
def doubleApplication(x: Int)(y: Int)(f: Int => Int):
(Int, Int) = (f(x), f(y))

val fancy = doubleApplication(1)(2){
    num => num * 4
}
```

Introduction: Blocks Of Code

The last expression is the result value

```
def doubleApplication(x: Int)(y: Int)(f: Int => Int):
(Int, Int) = (f(x), f(y))

val fancy = doubleApplication(1)(2) { num =>
  val res = num * 4
  println(s"Calculating result of f($num) = $res")
  res
}
```

The **scala.collection.immutable.List[A]** Class

The most common data structure on Scala

- Optimised chained List data structure
- Easy creation: **List(1,2,3)** or **List(“a”, “b”, “c”)** or **List(Person(“Alex”))**
- Inductively defined and constructed, good for pattern matching & recursion:
    - Case empty: **Nil**
    - Case not empty: **(x::xs)**
- Can also be used as queues or stacks data structures
- Concat operation expensive

Map

The most common function to transform collections

```
def map[B](f: A => B): List[B]
```

Builds a new list by applying a function to all elements of this list.

FlatMap

Maybe the most important function in FP

```
def flatMap[B](f: A => List[B]): List[B]
```

Converts this list of traversable collections into a list formed by the elements of these traversable collections.

- Also named as “bind” in Haskell, it is useful to sequence computations with side effects.
- In combination with Map, can be used to define for-comprehensions (which simplify code)

```
// This for comprehension
for {
  bound <- list
  out   <- f(bound)
} yield out

// is translated by the Scala compiler as ...
list.flatMap { bound =>
  f(bound).map { out =>
    out
  }
}
```

Iterating Collections: forEach

Used when it is necessary to execute a side-effect on each element

```
def foreach[U](f: (A) => U): Unit
```

Apply f to each element for its side effects Note: [U] parameter needed to help scalac's type inference.

Example:

```
List("Hello", "World", "of", "FP").foreach { elem =>
     print(s" Elem: $elem")
}
```

Filtering Collections: filter, filterNot, dropWhile

Discard or keep elements in a collection, based on a predicate function

```scala
def filter(p: (A) => Boolean): List[A]
// Selects all elements of this list which satisfy a predicate.

def filterNot(p: (A) => Boolean): List[A]
// Selects all elements of this list which do not satisfy a predicate.

def dropWhile(p: (A) => Boolean): List[A]
// Drops longest prefix of elements that satisfy a predicate.
```

Example:

```scala
List("Hello", "World", "of", "FP").filter(elem => elem.toLowerCase().contains("f")) // List("of", "FP")
```

Working With Boolean Predicates

Predicates: Functions from elements type to Boolean

```
def forall(p: (A) => Boolean): Boolean
// Tests whether a predicate holds for all elements of this list.

def exists(p: (A) => Boolean): Boolean
// Tests whether a predicate holds for at least one element of this list.

def find(p: (A) => Boolean): Option[A]
// Finds the first element of the list satisfying a predicate, if any.

def findLast(p: (A) => Boolean): Option[A]
// Finds the last element of the list satisfying a predicate, if any.

def count(p: (A) => Boolean): Int
// Counts the number of elements in the list which satisfy a predicate.
```

Grouping Elements

Partition lists and grouping elements by a certain property

```
def partition(p: (A) => Boolean): (List[A], List[A])
// A pair of, first, all elements that satisfy predicate p and, second, all elements that do
not.

def partitionMap[A1, A2](f: (A) => Either[A1, A2]): (List[A1], List[A2])
/* Applies a function f to each element of the list and returns a pair of lists:
   the first one made of those values returned by f that were wrapped in scala.util.Left,
   and the second one made of those wrapped in scala.util.Right.
 */

def groupBy[K](f: (A) => K): Map[K, List[A]]
// Partitions this list into a map of lists according to some discriminator function.

def groupMap[K, B](key: (A) => K)(f: (A) => B): Map[K, List[B]]
// Partitions this list into a map of lists according to a discriminator function key.
```

```
Example:
List(“Hello", "World", "of", "FP").groupBy(elem => elem.length)) // HashMap(5 -> List(Hello, World), 2 -> List(of, FP))
```

Reducing collections: The Reduce variants

Applying a generic operator to all the elements in the list

```
def reduce[B >: A](op: (B, B) => B): B
// Reduces the elements of this list using the specified associative binary operator.

def reduceLeft[B >: A](op: (B, A) => B): B
// Applies a binary operator to all elements of this list, going left to right.

def reduceLeftOption[B >: A](op: (B, A) => B): Option[B]
// Optionally applies a binary operator to all elements of this list, going left to right.

def reduceOption[B >: A](op: (B, B) => B): Option[B]
// Reduces the elements of this list, if any, using the specified associative binary operator.

def reduceRight[B >: A](op: (A, B) => B): B
// Applies a binary operator to all elements of this list, going right to left.

def reduceRightOption[B >: A](op: (A, B) => B): Option[B]
// Optionally applies a binary operator to all elements of this list, going right to left.
```

Example: `List("David", "Rubén", "Ignacio").reduce((res, elem) => res + ", " + elem)`

```
List().reduce((res, elem) => res + ", " + elem)
// Exception: java.lang.UnsupportedOperationException: List().reduceLeft
```

Reducing collections: The Fold Function

When you have an initial value

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
// Folds the elements of this list using the specified associative binary
operator.

def foldLeft[B](z: B)(op: (B, A) => B): B
// Applies a binary operator to a start value and all elements of this
list, going left to right.

def foldRight[B](z: B)(op: (A, B) => B): B
// Applies a binary operator to all elements of this list and a start
value, going right to left.
```

Example: `List("Luis", "Adrián", "Ignacio").fold("SNCF Team: ")((res, elem) => res + ", " + elem)`

Processing collections: The Scan Variants

The result is a new collection!

```
def scan[B >: A](z: B)(op: (B, B) => B): List[B]
// Computes a prefix scan of the elements of the collection.

def scanLeft[B](z: B)(op: (B, A) => B): List[B]
// Produces a list containing cumulative results of applying
the operator going left to right, including the initial value.

def scanRight[B](z: B)(op: (A, B) => B): List[B]
// Produces a collection containing cumulative results of
applying the operator going right to left.
```

DSL Languages on Scala

Define your own language !

```scala
/**
 * Execute a block of code, providing a JDBC connection.
 * The connection and all created statements are automatically released.
 *
 * @param block code to execute
 * @return the result of the code block
 */
def withConnection[A](block: Connection => A): A


/**
 * Execute a block of code in the scope of a JDBC transaction.
 * The connection and all created statements are automatically released.
 * The transaction is automatically committed, unless an exception occurs.
 *
 * @param block code to execute
 * @return the result of the code block
 */
def withTransaction[A](block: Connection => A): A

def findByExternalId(externalId: String): Future[Option[User]] = Future {
  db.withConnection { implicit connection =>
    SQL"select * from users where uid = $externalId".as(userParser.singleOpt)
  }
}
```

Conclusions

Higher-Order Functions are a big deal !

- You can pass code to functions to make them more general (and powerful)
- You can define your own syntax for your problem (DSLs)
- You can pass functions without name in parameters (lambdas)
- You can partially apply functions to specialise generic functions (currying)
- Blocks of code can execute any kind of side-effects and the last expression is the result of the block (that is also a function)
- What else can you ask for a JVM language?

Acknowledgements

Please keep the FP Guild alive !!

Thank you all for your attention !!

See you soon !!   :-)

Ignacio