@AndrzejWasowski
Andrzej Wąsowski

@zhoulaifu
Zhoulai Fu

Advanced Programming

1. Introduction to Functional Programing

IT UNIVERSITY OF COPENHAGEN

SOFTWARE
QUALITY
RESEARCH

- Motivation & Course Goals
- Functional Programming Primer + Scala Intro
- Traits
- Course Organization
- Algebraic Data Types
- Fold functions
- Primary Constructors
- In the next episode ...

AGENDA

Apache Spark

A Motivating Example

- Open-source cluster-computing framework aimed at **Big Data processing**
- Compute queries on large amount of data in **distributed storage**
- **Simple interface**: like local data structures
- **Powerful semantics**: distribution and parallelization
- Originated in 2009 at UC Berkeley, run by **Apache Foundation**
- **600 devs from 200** companies contribute to spark
- **Implemented in Scala**, interfaced to Java, Python, and R
- Key reason of popularity of Scala for Big Data
- Much faster than Hadoop's MapReduce due to heavy use of in-memory operations
- In **high demand**: some of you will land a Spark/Scala job before we finish :)
- **Some Users**: NBCUniversal, Netflix, Uber, Capital One, Baidu, Salesforce.com, ...

Count Word Frequency in a File

```
1 object StorageApp {
2
3   def main(args: Array[String]) {
4
5     val conf = new SparkConf()
6       .setAppName ("SimpleApp")
7       .setMaster ("local[6]") // use 6 cores
8     val sc = new SparkContext (conf)
9     val lines = sc
10      .textFile("/home/wasowski/opt/spark/README.md", 2)
11      .cache
12
13    val wordCounts = lines
14      .flatMap (line => line.split(" ") )
15      .map (word => (word, 1))
16      .reduceByKey (_ + _)
17
18    println (wordCounts.collect.map (_.toString).mkString)
19    sc.stop()
20  }
21 }
```

- Resilient distributed dataset (**RDD**)
- `lines` is an RDD of strings
- **Distributed** fault-tolerant processing
- L14 split each line into words, merge into RDD of words
- L15 RDD of words to RDD of pairs
- L16 merge pairs with same word, summing counters (map-reduce)
- We use **collection operations**
- Transformations (flatMap,map) build **representation of computation**.
- Transformations are **lazy**.
- Actions (reduceByKey) are **eager**: execute (**force**) representations
- **Pure** program, no vars&side effects
- `cache` only works if you have **referential transparency**

© Andrzej Wąsowski, IT University of Copenhagen

Things go wrong with side-effects

Side effects are officially banned in this course!

```
1 var counter = 0
2 var rdd = sc.parallelize(data)
3 // Wrong: Don't do this!!
4 rdd.foreach(x => counter += x)
```

- Line 2: we parallelize a computation
- Line 4: we sum values from an RDD incrementing a counter
- This cannot be done in a distributed way!!!
- Each node gets a **closure** containing the counter, the closure is sent to nodes.
- Each node increments a different copy of the counter!
- This is why we use functions like map and reduceByKey instead of variables

© Andrzej Wąsowski, IT University of Copenhagen

Count Word Frequency in a Real-Time Data Stream

```scala
1 object StreamingApp {
2
3   def main (args: Array[String]) {
4     val sparkConf = new SparkConf()
5       .setAppName("StreamingApp")
6       .setMaster ("local[6]")
7     // Sample every second
8     val ssc = new StreamingContext (sparkConf, Seconds(1))
9     val lines = ssc.socketTextStream("localhost", 9999,
10            StorageLevel.MEMORY_AND_DISK_SER)
11
12    val wordCounts = lines
13      .flatMap (line => line.split(" "))
14      .map(word => (word, 1))
15      .reduceByKey (_ + _)
16
17    wordCounts.print // Nothing gets printed!
18    ssc.start        // The computation starts here
19    ssc.awaitTermination
20  }
21 }
```

- L12-L15: identical **algorithm**
- Because the DStream **interface** is the same as RDD's
- RDDs and Streams are **monads**
- We will understanding this style of API really deeply
- `reduceByKey` needs a commutative associative operator (a **monoid**)
- + is a monoid on integers
- L12-L17 builds a **representation of computation**
- L18: streaming starts, before this **nothing happens**
- L17 printing every 1s until killed
- L17 behaves as if in a **loop**!

© Andrzej Wąsowski, IT University of Copenhagen

Introduction to Scala I

- A rich modern OO programming language with a functional part; eager by default, statically typed
- Compiles to JVM, compatible with Java on byte code level
- Designed by prof. Martin Odersky at EPFL in Lausanne
- First official release in 2004
- **This is not a course about Scala**, we use Scala to learn concepts that apply to other languages (F#, Haskell, Ocaml, Java, Python, Ruby, etc)
- Today's goals: (1) [recall] basics of functional programming and (2) teach Scala syntax and concepts
- Easy for those that have seen functional programming and Scala (focus primarily on harder exercises and on mapping your knowledge from other languages to Scala)
- Hard for those that are new to functional programming and Scala. Focus on the easiest exercises first, and really work hard the first 5-6 weeks.

Basics of Scala

A singleton class and its only instance

object creates a name space; used to build modules. Access the namespace with navigation: `MyModule.abs(42)`

```
1 object MyModule {
2
3   def abs(n: Int): Int = if (n < 0) -n else n
4
5   private def formatAbs(x: Int) =
6     s"The absolute value of $x is ${abs (x)}"
7
8   val magic :Int = 42
9   var result :Option[Int] = None
10
11  def main(args: Array[String]): Unit = {
12    assert (magic - 84 == magic.-(84))
13    println (formatAbs (magic-100))
14  }
15 }
```

def Defines a function (l.3)

A body **expression** (statements secondary in Scala)

Use braces if more expressions needed.

A named **value** declaration (final, immutable). Use this a lot.

A **variable** declaration. Avoid if possible.

Instantiation of a generic type

`None` is a singleton "constructor". Construct case classes without **new**

Operators are functions, can be overloaded:
minus is `Int.-(Int) :Int`
Unary methods can be used infix: `MyModule abs -42` legal

Every value is an object

Line 6 shows an **interpolated** character string

The example adapted from [Chiusano, Bjarnason 2015]

Traits: Rich or Fat Interfaces

Scala idiom: decompose classes into traits

```
// A class with a final property 'name' and
// a constructor. You can still add
// more members like in Java in braces.
abstract class Animal (val name :String)

// concrete methods
trait HasLegs {
  def run  () :String = "after you!"
  def jump () :String = "hop!"
}
// abstract method
trait Audible { def makeNoise () :String }
// field
trait Registered { var id :Int = 0 }

// multiple traits mixed in
class Frog (name:String) extends
  Animal(name) with HasLegs with Audible {
  def makeNoise () :String = "croak!"
} // Frog concrete, so provide makeNoise
```

```
// Mix directly into an object
val f = new Frog ("Kaj") with
          Registered
// f: Frog with Registered =
//              $anon$1@88f0bea
f.id = 42
println ( s"My name is ${f.name}")
println ( "I'm running " + f.run )
println( "I'm saying " + f.makeNoise)

}
```

	class	abstr. class	trait
mult. inheritance	−	−	+
data	+	+	+
concr. methods	+	+	+
abstr. methods	−	+	+
constr. params.	+	+	−

[Horstmann 2012, chpt. 10], [Odersky et al. 2014, chpt. 12] have more info than [Chiusano, Bjarnason 2014]

Pure Functions

Def. Referentially transparent expression (e)

Expression e is RT iff replacing e by its value in programs does not change their semantics

(Java) append an element to a list

`a.add(5)` // non RT

value `void`; substitution is pointless; the meaning is in the references reachable from `a` (change over time for the same `a`)

(Scala) append to an immutable list

`val b =Cons(5,a)` // RT

The value is a list `b`, identical to `a`, modulo the added head element

Def. Pure function (f)

Iff every expression $f(x)$ is referentially transparent for all referentially transparent expressions x. Otherwise **impure** or **effectful**.

In practice: **A function is pure if it does not have side effects** (writes/reads variables, files or other streams, modifies data structures in place, sets object fields, throws exceptions, halts with errors, draws on screen)

Pure code shows dependencies in interface, good for mocking, testable

Referential Transparency Poll

Which of the following computations are referentially transparent?

1. `a = a + 42`
2. `a == b + 42`
3. `a[x] == 42`
4. `println("42")`
5. `throw DivideByZero()`
6. `f(f(x))` if f is pure
7. `z = z + f(f(x))` if f is pure

© Andrzej Wąsowski, IT University of Copenhagen

Loops and Recursion

An imperative factorial

```
def factorial (n :Int) :Int = {
  var result = 1
  for (i <- 2 to n)
    result *= i
  return result
}
```

Loops compute with effects;
cannot be used in pure code

Tail recursive, pure factorial

```
def factorial (n :Int) = {
  def f (n :Int, r :Int) :Int =
    if (n<=1) r
    else f (n-1, n*r)
  f (n,1)
}
```

call in tail position

Call tails are automatically compiled to loops with O(1) space overhead

A pure recursive factorial

```
def factorial (n :Int) :Int =
  if (n<=1) 1
  else n * factorial (n-1)
```

call not in tail position

Example execution

`factorial(5)`

$\rightsquigarrow 5 * (\texttt{factorial(4)})$

$\rightsquigarrow 5 * (4 * (\texttt{factorial(3)}))$

$\rightsquigarrow 5 * (4 * (3 * (\texttt{factorial(2)})))$

$\rightsquigarrow 5 * (4 * (3 * (2 * (\texttt{factorial(1)}))))$

$\rightsquigarrow 5 * (4 * (3 * (2 * 1)))$

$\rightsquigarrow 5 * (4 * (3 * 2))$

$\rightsquigarrow 5 * (4 * 6)$

$\rightsquigarrow 5 * 24$

$\rightsquigarrow 120$

Uses O(n) stack space;
Technically exponential
(for this example)!

Def. Call in **tail position**

The caller immediately returns the value of the call

Course Organization

- Our **website** is LearnIT + a git-repo
- **Reading:** read prescribed book chapters and papers **before class**.
  Without reading you may not be able to solve exercises.
- **Lectures:** ca. 13 weeks.
  Summarize the main points, but may skip details needed in exercises
- **Exercises (Homeworks):** ca. 13 weeks, same days as lectures
  implement small well defined tasks on the topic of the day's lecture
- **ADPRO Cafe:** Mondays 14:00-16:00 in room 2A20 (an emergency meeting: please help me!) (no new stuff)
- 2–3 person groups (no single person groups)
- Communicate **in class** on Thursday, and daily on the **LearnIT forum**.
  Andrzej is very good on handling LearnIT, and very bad in handling email. He gets needlessly annoyed with email. Use email only for sensitive matters.
- **Exam:** written, need to pass 7 homeworks to be admitted (10 homeworks are graded)

Algebraic Data Types (ADTs)

Def. Algebraic Data Type

A type generated by one or several constructors, each of which may contain zero or more arguments.

Sets generated by constructors are **summed**, each constructor is a **product** of its arguments; thus **algebraic**.

Example: immutable lists

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head :A, tail :List[A]) extends List[A]
```

sealed: extensible in the same file only

`Nothing`: **subtype of any type**

Example: operations on lists

```
object List {
  def sum(ints :List[Int]) :Int =
    ints match { case Nil => 0
                 case Cons(x,xs) => x + sum(xs) }
  def apply[A](as :A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

companion object of `List[+A]`

pattern matching uses case class constructors

overload function application for the object

variadic function

© Andrzej Wąsowski, IT University of Copenhagen

Poll: How is your recursion?

```scala
def f (a :List[Int]) :Int = a match {
  case Nil => 0
  case h::t => h + f(t)
}
```

What is the result of `f (List(42,-1,1,-1,1,-1)` ?

© Andrzej Wąsowski, IT University of Copenhagen

Function Values

- In functional programing **functions are values**
- Functions can be **passed to other functions**, composed, etc.
- Functions operating on function values are **higher order** (HOFs)

```
def map (a :List[Int]) (f :Int => Int) :List[Int] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

A functional (pure) example

```
val mixed = List(-1, 2, -3, 4)
map (mixed) (abs _)
```

```
map (mixed) ((factorial _) compose (abs _))
```

see method `factorial` as a function value

alternatively type it explicitly:

`(abs :Int => Int)`

An imperative (impure) example

```
val mixed = Array (-1, 2, -3, 4)
for (i <- 0 until mixed.length)
  mixed(i) = abs (mixed(i))
```

```
val mixed1 = Array (-1, 2, -3, 4)
for (i <- 0 until mixed1.length)
  mixed1(i) = factorial(abs(mixed1(i)))
```

Anonymous Functions

Literals

```
val l =List(1, -2, 3)
val a =Array(-1, 2, -3)
```

Function Literals (Anonymous Functions)

We need the same for functions

```
val negative =(x :Int) =>x < 0
negative (-42) ⇝ true
```

Use to create functions in place:

```
l.filter ((x :Int) =>x < 0) ⇝ ?
a.filter ((x :Int) =>x > 0) ⇝ ?
```

Alternative concise syntax

```
(abs _) ⇝ (x :Int) =>MyModule.abs x
```

Scala distinguishes functions and methods.

We used this syntax before to turn a method into a function (like above).

Currying and partial application

```
val add2 =(x :Int, y :Int) =>x + y
val add =(x :Int) =>(y :Int) =>x + y
```

a curried function

What is the type of `add`? What is the value of `add (2) (3) ⇝ ?`

Curried functions can be partially applied: `val incr =add (1)`

a partial application

Type of `incr`? Value of `incr (7) ⇝ ?`

Methods can also be curried: `def add (x:Int) (y:Int) :Int =x + y`

Parametric Polymorphism

Monomorphic functions operate on fixed types:

A monomorphic map in Scala

```
def map (a :List[Int]) (f :Int => Int) :List[Int] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

There is nothing specific here regarding Int.

A polymorphic map in Scala

```
def map[A,B] (a :List[A]) (f :A => B) :List[B] =
  a match { case Nil      => Nil
            case h::tail => f(h)::map (tail) (f) }
```

An example of use (type parameters are inferred):

```
map[Int,String] (mixed_list) { _.toString } compose
  (factorial _) compose (abs _))
```

- A **polymorphic** function operates on values of (m)any types (some restriction possible in Scala)
- A polymorphic type defines a parameterized family of types
- Don't confuse with OO-polymorphism roughly equal to "dynamic method dispatch" (dependent on the inheritance hierarchy)

HOFs in Scala Standard Library

Methods of class `List[A]`, operate on `this` list, type `A` is bound in the class

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

Translates `this` list of `A`s into a list of `B`s using `f` to convert the values

`filter(p: A =>Boolean): List[A]`

Compute a sublist of `this` by selecting the elements satisfying the predicate `p`

`flatMap[B](f: A =>List[B]): List[B]` *type slightly simplified

Builds a new list by applying `f` to elements of `this`, concatenating results.

`take(n: Int): List[A]`

Selects first n elements.

`takeWhile(p: A =>Boolean): List[A]`

Takes longest prefix of elements that satisfy a predicate.

`forall(p: A =>Boolean): Boolean`

Tests whether a predicate holds for all elements of this sequence.

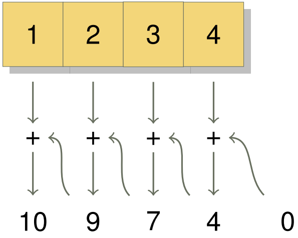
`exists(p: A =>Boolean): Boolean`

Tests whether a predicate holds for some of the elements of this sequence.

More at `http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List`

[Right]Folding: Functional Loops

Compute a sum of list's elements



What characterizes similar computations?

- An **input list** `l = List(1,2,3,4)`
- An **initial value** `z = 0`
- A **binary operation** `f :Int => Int = _ + _`
- An **iteration algorithm** (folding)

```
def foldRight[A,B] (f : (A,B) => B) (z :B) (l :List[A]) :B =
  l match {
    case Cons(x,xs) => f(x, foldRight (f) (z) (xs))
    case Nil => z
  }
val l1 = List (1,2,3,4,5,6)
val sum     = foldRight[Int,Int] (_+_) (0) (l1)
val product = foldRight[Int,Int] (_*_) (1) (l1)
def map[A,B] (f :A=>B) (l: List[A])=
  foldRight[A,List[B]] ((x, z) => Cons(f(x),z)) (Nil) (l)
```

Many HOFs can be implemented as special cases of folding

The Primary Constructor

```scala
class Person (val name: String, val age: Int) {
  println ("Just constructed a person")
  def description = s"$name is $age years old"
}
```

```java
class Person {
  private String name;
  private int age;
  public String name() { return name; }
  public int age() { return age; }

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
    System.out.println("Just constructed a person");
  }

  public String description ()
  { return name + "is " + age + " years old"; }
}
```

- Parameters become fields
- 'val' parameters become values, 'var' become variables
- If no parameter list, primary constructor takes none
- Constructor initializes fields and executes top-level statements of the class
- Like for all functions, parameters can take default values, reducing the need for overloading
- Note: primary constructors are used with case classes
- Known from F# and C# as well

[Horstmann 2012, Chpt. 5.7] explains the primary constructors in Scala

Scala: Summary

- **Basics** (objects, modules, functions, expressions, values, variables, operator overloading, infix methods, interpolated strings.)
- **Pure functions** (referential transparency, side effects)
- **Loops and recursion** (tail recursion)
- **Functions as values** (higher-order functions)
- **Parametric polymorphism** (monomorphic functions, dynamic and static dispatch)
- **Standard HOFs** in Scala's library
- **Anonymous functions** (currying, partial function application)
- **Traits** (fat interfaces, multiple inheritance, mixins)
- **Algebraic Data Types** (pattern matching, case classes)
- **Folding**
- **Primary constructors** (default parameter values)

In the next episode ...

- Variance of type parameters
- Basics of functional design: exceptions vs values, partial functions, the Option data type, exception oriented API of Option, for comprehensions, Either
- Experience your first computation in a Monad (but we will not call it so yet)
- The reading should be relatively easy, so you should really try it!