Programming in Scala

Lecture Four

Angelo Corsaro

3 December 2018

Chief Technology Officer, ADLINK Technology Inc.

Table of contents

Implicits

Scala Implicits

Scala implicit definitions provide a general mechanism to control how the compiler resolves types or argument mismatches.

In other terms, it control the set of transformation available to the compiler to *fix* a compile time error.

Example

Suppose we have a function that sums the elements of a lists and we want to also use it to sum arrays.

If we just pass an array the compiler will raise a type error as he does not know how to convert an Array[Int] into a List[Int].

```
def sum(xs: List[Int]): Int = xs match {
        case List() => 0
        case y::ys => y + sum(ys)
}

sum(Array[Int](1,3,4,5,6)) // Compiler error!
```

Implicit Functions

To fix this error and allow this function operate on Arrays as well as on List we can define an implicit conversion as shown below:

```
import scala.language.implicitConversions
object Converter {
        implicit def array2List(as: Array[Int])= as.toList
        implicit def strintToInt(s: String) = s.toInt
}

import Converter._

sum(Array(1,3,4,5,6)) // Compiler inserts the conversion array2List
sum(Array[Int](1,"3",4,"5",6)) // Compiler inserts the conversions array2List and stringToInt
```

Implicit Classes

Implicit classes are commonly used to emulate *open classes* in Scala.

In other terms, one can define implicit transformation to extend or enrich the protocol supported bu given type.

Example

```
object Implicits {

  implicit class RichRunnable[T](runner:  => T) extends Runnable {
    def run() { runner() }
  }

}

val thread = new Thread {
        print("Hello!")
}
```

Implicit Parameters

Implicit parameters can be used to let the compiler fill in missing parameters in curried functions.

Example

```
object Logger {
  def log(msg: String)(implicit prompt: String) {
    println(prompt + msg)
  }
}

implicit val defaultPrompt = ">>> "

Logger.log("Testing logger")
```

Implicit Resolution Rules

The compiler looks for potential conversions available in the current scope.

Thus, conversions needs to be imported in order for the compiler to apply them.

Mathematical Induction and Recursive Definitions

Proof

A proof is essentially a convincing argument that something is true

The mechanics of a proof typically requires to derive some statements from:

- Assumptions or Hypothesis
- Statements that already have been derived, and
- other generally accepted facts

Where the term *derive* means to derive using general principle of logical reasoning.

Usually what we are trying to prove inolves a statement of the form $p \to q$.

Direct Proof

A **direct prrof** assumes that a statement p is true and, using this asumption, shows that q is true.

Direc Proof Example

Prove that for any intergers a and b, if a and b are odd, then ab is odd.

Proof.
The first known fact that we will exploit is that an integer n is odd if and only if there exit an integer k such that $n = 2k + 1$. As such let's assume that exist x and y integers such that $a = 2x + 1$ and $b = 2y + 1$. From this we can write:

$$ab = (2x + 1)(2y + 1) = 4xy + 2x + 2y + 1 = 2(2xy + x + y) + 1$$

Say $z = 2xy + x + y$ we have:

$$ab = 2z + 1$$

Thus ab is odd. $\square$

Constructive Proof

The previos is an example of a *constructive proof*. In other terms, we prove statements of the form "There exist z such that... " by constructing a specific z that works.

Proof by Counter-Example

Suppose we have a statement $\forall x : P(x)$ that we want to proof false, in this case we can build a proof by *counter-example* by simply constructing an x for which the statement $P(x)$ is false.

Proof by Contraposive

The alternative to a *direct proof* is an *indirect proof*.

The simplest form of an *indirect proof* is a *proof by contrapositive*, using the logical equivalence of $p \to q$ and $\neg q \to \neg p$

Proof by Contraddiction

In its most general form, proving a statement *p* by contraddiction means showing that if it is not true, some contraddiction results.

Formally this means showing that $\neg p \rightarrow \mathit{false}$ is true. It follows from the *contrapositive* statement that $\mathit{true} \rightarrow p$, which is equivalent to p.

Proof by Contraddiction Example

One of the most famous example of proof by contraddiciton is that proof known to ancient Greeks that $\sqrt{2}$ is irrational.

Prove it as an exercise.

Principle of Mathematical Induction

Suppose $P(n)$ is a statement involving and integner n. That to prove that $P(n)$ is true for every $n \geq n_0$, it is sufficient to show that:

1. $P(n_0)$ is *true*
2. For any $k \geq n_0$, if $P(k)$ is *true* then $P(k+1)$ is *true*

A *proof by induction* is an application of this principle. The two parts of this proof are called *basic step* and the *inductive steps*.

Proof by Induction – Example 1

Prove that:

$$1 + 2 + 3 + ... + n = \frac{n * (n + 1)}{2}$$

Proof by Induction – Example 2

Prove that:

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

defines a function that computes the sum of the elements of a list.

Toward the Principle of Strong Induction

Suppose we wanted to prove the following proposition $P(n)$ for every $n \in \mathbb{N}$:

$$\begin{cases} \text{n is prime,} & \text{or} \\ \text{n is the product of two or more primes} \end{cases} \tag{1}$$

Building this proof would require that as part of the inductive step we should be able to assume not only that $P(k)$ is true, but that $P(i)$ is true for all $i \leq k$.

This assumption is stronger than what we assume on the principle of induction.

Principle of Strong Induction

Suppose that $P(n)$ is a statement involving an integer n. Then to prove that $P(n)$ is true for every $n \geq n_0$, it is sufficient to show:

1. $P(n_0)$ is *true*
2. For any $K \geq n_0$, if $P(n)$ is true for every n satisfying $n_0 \leq nleqk$, then $P(k+1)$ is *true*.

The principle of *strong induction* is also referred as the principle of *complete* induction.

Something worth knowing is that the princple of strong induction is equivalent to the principle of induction and that neither of them can be proved nor disproved using standard properties of the natural numbers. Thus we must adopt them as axioms!

Homeworks

Reading Assignment

Read the following chapters:

- Chapter 21
- Chapter 32