Continuing

Yinghui Wang, Mehrdad Alemzadeh

CAS 706

November 4, 2010
Introduction

OOP

FP
  example

Mixin Classes Extending A

Pattern Matching
  So what does pattern matching do?

Types
  Parameterized Types
  Abstract Types
Introduction

- The design of Scala started in 2001 at the EPFL by Martin Odersky
- stands for "Scalable language"
  - desirable feature of a program or algorithm
  - Aspects of scalability
- Multi-paradigm Language
  allow programmers to use the best tool for a job
Scala is a pure OO language
- extended by subclassing and multiple inheritance
- runs on the standard Java and .NET platforms
- interoperates seamlessly with all Java libraries
- "Scala goes further than all other well-known languages in fusing object oriented and functional programming." (Martin Odersky)
- Main goals
also supports functional programming

- anonymous function, Higher-order functions, currying, Pattern matching, Tail call

- languages = no side effects

- FP can include:
  - garbage collection, Abstract types, functions as first-class values, lazy evaluation
Examples I

- **def** functionName(arg1: Type1, arg2: Type2): ReturnType =
  functionDefinition

  - *scala* > **def** timesTwo(n: Int): Int = n * 2
    timesTwo: (Int)Int
  - *scala* > timesTwo(10)
    res0: Int = 20

- **Higher-Order** Functions

  *scala* > **def** applyFn(fn: Int => Int, arg: Int) = fn(arg)
  applyFn: ((Int) => Int,Int)Int
  applyFn(timesTwo, 10)
  res2: Int = 20
Examples II

- Anonymous functions:
  \[(\text{arg1}: \text{Type1}, \text{arg2}: \text{Type2}) \rightarrow \text{functionDefinition}\]
  scala> (n: Int) => n * 3
  res4: (Int) => Int = <\text{function}> \\
  And used like so:
  scala> applyFn((n: Int) => n * 3, 10)
  res5: Int = 30
  scala> applyFn(* 3, 10)
  res7: Int = 30
import scala.io._

def toInt(in: String): Option[Int] = try {
  Some(Integer.parseInt(in.trim))
} catch {
  case e: NumberFormatException => None
}
Mixin Classes Extending A

trait RichIterator extends A {
  def foreach(f: T => Unit) {
    while (hasNext) f(next)
  }
}

class StringIterator(s: String) extends A {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = {
    val ch = s.charAt i;
    i += 1;
    ch
  }
}
object StringliteratorTest {
  def main(args: Array[String]) {
    class Iter extends Stringliterator(args(0)) with Richliterator
    val iter = new Iter
    Iter foreach println }
}
Pattern Matching

▶ a first-match policy.

▶ **case class** `Person(firstName: String, lastName: String);`

```
val People = List(
    Person("Jane", "Smith"),
    Person("John", "Doe"),
    Person("Jane", "Eyre"));
for(Person("Jane", last) |- people)yield "Ms. " + last;
```

t-match policy.

▶ Results "Ms. Smith", "Ms. Eyre"
So what does pattern matching do?

- Sort of like a switch statement in Java. You match what are essentially the creation forms of objects.
  
  - case Nil => ...
  
  - case x :: xs => ...

- Patterns actually nest, just like expressions nest, so you can have very deep patterns. Generally the idea is that a pattern looks just like an expression.

- So why do you need pattern matching?
Scala is a statically-typed language

- comprehensive, complete, and consistent
- Scala’s parameterized types are similar to Java and C# generics and C++ templates
- a declaration like class List[+A] means that List is parameterized by a single type, represented by A. The + is called a variance annotation.
Sometimes, a Parameterized type like list is called a type constructor, because it is used to create specific types. For example, List is the type constructor for List[String] and List[Int], which are different types. In fact, it is more accurate to say that all traits and classes are type constructors. Those without type parameters are effectively zero-argument, parameterized types.
Scala also supports abstract types, which are common in functional languages. They overlap somewhat with Parameterized types. Parameterized types are the most natural fit for parameterized container types like List and Option.

- Consider the declaration of Some from the standard library.
  ```scala
case final class Some[+A](val x : A) { ... }
```

- Abstract types
  ```scala
case final class Some(val x : ???) { type A ... }
```
If a type will have constructor arguments declared using a "placeholder" type that has not yet been defined, then parameterized types are the only good solution (short of using Any or AnyRef).

You can use abstract types as method arguments and return values within a function.
Resources I

First step
http://www.artima.com/scala/scalazine/articles/steps.html

Pattern matching
http://www.artima.com/scala/scalazine/articles/pattern_matching.html

Type classes
http://lambda-the-ultimate.org/taxonomy/term/32

Wiki scala
http://en.wikipedia.org/wiki/Scala_(programming_language)
Resources II

Types

http://programming−scala.labs.oreilly.com/ch12.html

http://www.scala−lang.org/

http://www.cs.caltech.edu/ mvanier/hacking/ rants/scalable_computer_programming_languages.html