Advanced Models and Programs

The Scala language

Peter Sestoft

Wed 2011-02-09*

Agenda

• Object-oriented programming in Scala
  – Classes
  – Singletons (object)
  – Traits
• Compiling and running Scala programs
• Functional programming in Scala
  – Type List[T], higher-order and anonymous functions
  – Case classes and pattern matching
  – The Option[T] type
  – For-expressions (comprehensions à la Linq)
• Type system
  – Generic types
  – Co- and contra-variance
  – Type members
Scala object-oriented programming

- Scala is designed to
  - work with the Java platform
  - be somewhat easy to pick up if you know Java
  - be much more powerful and concise
- Scala has classes like Java and C#
- And abstract classes
- But no interfaces
- Instead, traits = partial classes

- You will also need a Java implementation

Java and Scala

```java
class PrintOptions {
    public static void main(String[] args) {
        for (String arg : args)
            if (arg.startsWith("-"))
                System.out.println(" " + arg.substring(1));
    }
}
```

```scala
object PrintOptions {
    def main(args : Array[String]) = {
        for (arg <- args)
            if (arg.startsWith("-"))
                System.out.println(" " + arg.substring(1));
    }
}
```

Singleton class, no statics
Declaration syntax
Array[T] is generic type
for expression
Use Java class libraries
**Compiling and running Scala**

- Use **scalac** to compile *.scala files
- Use **scala** to run the object class file
  - uses **java** runtime with Scala’s libraries

```
sestoft@mac$ scalac Example.scala
sestoft@mac$ scala PrintOptions -help -verbose do it help verbose
```

![Diagram of the compilation process]

**Interactive Scala**

- Scala also has an interactive top-level
  - Like Scheme, F#, most functional languages

```
sestoft@mac $ scala
Welcome to Scala version 2.8.1.final (Java HotSpot(TM))

scala> def fac(n:Int):Int = if (n==0) 1 else n*fac(n-1)
fac: (n: Int)Int

scala> fac(10)
res0: Int = 3628800

scala> def fac(n : Int):BigInt = if (n==0) 1 else fac(n-1)*n
fac: (n: Int)BigInt

scala> fac(100)
res1: BigInt = 9332621544394415268169923388526670049071596
82643816214685929638952175999993229915608941463976156518286
253697920827223758251185210916864000000000000000000000000
```

![Diagram of the interactive Scala environment]
**Much lighter syntax**

- All declarations start with keyword (no `int` x)
- `void` and `()` and `{}` can often be left out
- All values are objects and have methods
  - So `2.to(10)` is a legal expression
- All operators are methods
  - So `x+y` same as `x.+(y)`
- Method calls can be written infix
  - So `2.to(10)` can be written `2 to 10`

```scala
for (x <- 2 to 10)
    println(x)
```

**Uniform type system (like C#)**

![Type System Diagram](image)
Singletons (object declaration)

- Scala has no static fields and methods
- An object is a singleton (instance of a) class

```scala
object PrintOptions {
  def main(args : Array[String]) = {
    ...
  }
}
```

- Can create an Application as a singleton

```scala
object ListForSum extends Application {
  val xs = List(2,3,5,7,11,13)
  var sum = 0
  for (x <- xs)
    sum += x
  println(sum)
}
```

Classes

```scala
abstract class Person(name : String) {
  def Print
  val Name = name
}

class Student(name : String, programme : String) extends Person(name) {
  def Print {
    println(name + " studies " + programme)
  }
}
```

```scala
object ClassHierarchy extends Application {
  val p : Person = new Student("Ole", "SDT");
  p.Print
  println(p.Name)
}
```
Anonymous subclass and instance

```scala
val p = new Student("Kasper", "SDT") {
  override def Print {
    super.Print
    println("and does much else")
  }
}
scala> p.Print
Kasper studies SDT and does much else
```

- Same as Java's anonymous inner classes:

```java
pause.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    canvas.run(false);
  }
});
```

Traits: fragments of classes

- Can have fields and methods, no instances

```scala
trait Counter {
  private var count = 0
  def Increment { count += 1 }
  def Count = count
}
```

- Allows mixin: multiple “base classes”

```scala
class NumberedPerson(name : String)
  extends Person(name) with Counter {
    def Print {
      Increment
      println(name + " has been printed " + Count + " times")
    }
  }
val q1 : Person = new NumberedPerson("Hans")
val q2 : Person = new NumberedPerson("Laila")
q1.Print; q1.Print;
q2.Print; q2.Print; q2.Print
```
Generic class List\[T\]

- A list
  - has form Nil, the empty list, or
  - has form x::xr, first element is x, rest is xr

- A list of integers, type List[Int]:
  
  ```scala
  List(1,2,3)
  1 :: 2 :: 3 :: Nil
  ```

- A list of Strings, type List[String]:
  
  ```scala
  List("foo", "bar")
  ```

- A list of pairs, type List[(String, Int)]
  
  ```scala
  List(("Peter", 1962), ("Lone", 1960))
  ```

Functional programming

- Supported just as well as object-oriented
  - Three ways to print the elements of a list
    
    ```scala
    for (x <- xs)
    println(x)
    xs.foreach(println)
    xs foreach println
    ```

- Anonymous functions; two ways to sum
  
  ```scala
  var sum = 0
  for (x <- xs)
  sum += x
  ```

  ```scala
  var sum = 0
  xs foreach (x => sum += x)
  ```

  Like C#
List functions, pattern matching

- Compute the sum of a list of integers
  ```scala
  def sum(xs : List[Int]) : Int =
  xs match {
    case Nil   => 0
    case x::xr => x + sum(xr)
  }
  ```

- A generic list function
  ```scala
  def repeat[T](x : T, n : Int) : List[T] = {
    if (n==0) Nil
    else x :: repeat(x, n-1)
  }
  ```

Fold and foreach on lists

- Computing a list sum using a fold function
  ```scala
  def sum(xs : List[Int]) =
  xs.foldLeft(0)((res,x)=>res+x)
  ```

- Same, expressed more compactly:
  ```scala
  def sum(xs : List[Int]) =
  xs.foldLeft(0)(_+_
  ```

- Actually, `foreach` is also a function:
  ```scala
  def foreach[T](xs : List[T], act : T=>Unit) : Unit =
  xs match {
    case Nil   => { }
    case x::xr => { act(x); foreach(xr, act) }
  }
  ```
Enumeration types

- A type to represent binary operators
  
  \[ + \ - \ * \ / \ == \ != \ < \ <= \ > \ >= \]

```scala
object Operator extends Enumeration {
  val Add, Sub, Mul, Div,
    Eq, Ne, Lt, Le, Gt, Ge = Value
}
```

Case classes and pattern matching

- Case class objects allow pattern matching
- Good for representing tree data structures
- Example: An Expression is
  - a variable with a name, or
  - a constant integer
  - or a binary operator applied to two expressions

```scala
abstract class Expression

case class Variable(name : String) extends Expression

case class Constant(value : Int) extends Expression

case class BinOp(op : Operator.Value,
                  e1 : Expression, e2 : Expression)
          extends Expression
```
Representation of expressions

• An expression is a tree

No parentheses

7 + 9 * 10
7 + (9 * 10)

• Representing it with case class objects:

```
BinOp(Operator.Add,
     Constant(7),
     BinOp(Operator.Mul,
           Constant(9),
           Constant(10)))
```

Evaluation of expressions

```
def eval(expr : Expression) : Int =
  expr match {
    case Variable(name)  => throw new Exception("ak")
    case Constant(value) => value
    case BinOp(op, e1, e2) => {
      val i1 = eval(e1)
      val i2 = eval(e2)
      op match {
        case Operator.Add => i1+i2
        case Operator.Sub => i1-i2
        case Operator.Mul => i1*i2
        case Operator.Div => i1/i2
        ...
      }
    }
  }
```

eval(BinOp(Operator.Add, Constant(42), Constant(27)))
The built-in Option[T] case class

- Values `None` and `Some(x)`
- Same purpose as C# nullable value types
  ```scala
def sqrt(x : Double) : Option[Double] =
  if (x<0) None else Some(Math.sqrt(x))
  ```
- Use pattern matching to distinguish them
  ```scala
def mul3(x : Option[Double]) =
  x match {
    case None    => None
    case Some(v) => Some(3*v)
  }
  ```

Scala for-expressions

- Just like C#/Linq:
  ```scala
  for (x <- primes; if x*x < 100) yield 3*x
  from x in primes where x*x < 100 select 3*x
  ```
- No groupby, orderby
- No built-in aggregates (sum, min, max, ...)

More for-expression examples

• Example sum

```csharp
sum(for (x <- 1 to 200; if x%5!=0 && x%7!=0) yield 1.0/x)
```

```csharp
(from x in Enumerable.Range(1, 200)
 where x%5!=0 && x%7!=0
 select 1.0/x).Sum()
```

• All pairs (i,j) where i>j and i=1..10

```csharp
for (i <- 1 to 10; j <- 1 to i) yield (i,j)
```

Co-variance and contra-variance

• If generic class C[T] only outputs T’s it may be co-variant in T:

```csharp
class C+[T](x : T) {
    def outputT : T = x
}
```

• If generic class C[T] only inputs T’s it may be contra-variant in T:

```csharp
class C[-T](x : T) {
    def inputT(y : T) { }
}
```

• Much like C#, with "+" for out and "−" for in
Scala co/contra-variance examples

trait Iterable[+A] extends ... {
  def iterator : Iterator[A]
}

trait Iterator[+A] extends ... {
  def hasNext : Boolean
  def next() : A
}

trait MyComparer[-T] {
  def compare(x : T, y : T) : Boolean = ...
}

As for C#
IEnumerable, IEnumerator

Scala's actual Comparator is from Java and is not contravariant

Type members in classes

- May be abstract; may be further-bound

```scala
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}

class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food : SuitableFood) { }
}

class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  override def eat(food : SuitableFood) { }
}
```

Abstract type member
Final-binding
Simple Scala Swing example

- Scala interface to Java Swing

```scala
import scala.swing._

object FirstSwingApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "First Swing App"
    contents = new Button {
      text = "Click me"
    }
  }
}

reactions += {
  case scala.swing.event.ButtonClicked(_) =>
    System.out.println("Button clicked")
}
```

Concurrency, Scala actors

- Using threads and locks is difficult
  - Too little locking: conflicting updates and reads
  - Too much locking: deadlock
- Scala actors = mostly-sequential programs that communicate via messages
- Similar to the Erlang language; similar to Communicating Sequential Processes (1980) but with asynchronous communication
Higher-kinded types

- A type represents many similar values
  - Eg: Int represents 0, 1, 2, ..., -1, -2, ...
  - Eg: List[Int] represents List(2,3), List(13,21), ...
- A kind represents many similar types
  - Eg: * represents Int, String, Double, ...
  - Eg: * -> * represents List[_], Tree[_], ...

- Sounds cool, but why bother?
- Because even generic types lead to some code duplication (and maintenance trouble)

Motivation: filter and remove

 trait Iterable[T] {
   def filter(p: T => Boolean): Iterable[T]
   def remove(p: T => Boolean): Iterable[T]
   = this.filter(x => !p(x))
 }

 trait List[T] extends Iterable[T] {
   def filter(p: T => Boolean): List[T] = ...
   override def remove(p: T => Boolean): List[T]
   = this.filter(x => !p(x))
 }

Examples due to David Christiansen

- Neat enough
- Not neat: silly to have to redefine

- If remove were just inherited, it would have type Iterable[T] not List[T]
- But the result of removing from a list should be a list, not an iterable
Type constructors to the rescue

- A type constructor is a function from types to type. A generic class is a type constructor.
- Idea: Allow type constructors as arguments to types.

```scala
trait Iterable[T, Result[_]] {
  def filter(p: T => Boolean): Result[T]
  def remove(p: T => Boolean): Result[T]
    = this.filter(x => !p(x))
}
trait List[T] extends Iterable[T, List] {
  def filter(p: T => Boolean): List[T] = ...
}
```

- In List[T] the inherited remove has return type Result[T] = List[T] as desired
- No need to redefine remove

The need for kinds

- Some uses of Iterable are meaningless: Iterable[Int, Double]
- Double is a type (*), not type function (*->*)
- We can use kinds to describe legal uses of List, Iterable and so on:
  - Int : *
  - List : * -* *
  - List[Int] : *
  - Result : * -* *
  - Iterable : * -* (* -* *) -* *

- See Moors, Piessens, Odersky: Generics of a higher kind, OOPSLA2008
Other Scala features

- Implicit arguments
- Pattern matching on user-defined types
- Scala Lift (web programming) framework
- Simple build tool sbt
- Developer and language design community
- Limited tail call optimization (Java platform)

Commercial use of Scala

- Twitter is written in Scala
- Eclipse plugins can be written in Scala (eg. Hannes does this)
- Several companies use Scala
- Also in Copenhagen ...
  - Because it works with Java libraries
  - And Scala code is shorter and often much clearer
References

- A Scala tutorial for Java programmers, 2010
- An overview of the Scala programming language, 2006
- Find the above at: http://www.scala-lang.org
- http://www.infoq.com/interviews/functional-langs: A discussion between the inventors of Scala, F# and Erlang

What’s next

- Mon 14 Feb: Abstract syntax, interpretation, lexing, parsing
- Wed 16 Feb: Grammar transformations, compilation, stack machines

- You should:
  - Do Exercise sheet 2
  - Hand in solutions no later than Friday 18 Feb by mail to sestoft@itu.dk