The SCAlable LAnguage

The Scala Language:
Object Oriented Functional Programming

Gabor Szokoli

Overview

- Introduction
- Everything is an Object
- Inheritance and mixins
- For comprehensions
- Generic Types: Covariance, contravariance type bounds.
- Java interoperability

Fast facts about Scala

- Pure Object Oriented
  - Every value is an object
  - Ultimate supertype: Any
  - Ultimate subtype: All
- Functional
  - Functions are values, thus objects
- Simple core language, rich library extensions
  - Plus all Java libraries via JVM bytecode compatibility

It's just like Java

```
def sort(xs: Array[int]): unit = {
   def swap(i: int, j: int): unit = {
      val t = xs(i); xs(i) = xs(j); xs(j) = t;
   }
   def sort1(l: int, r: int): unit = {
      val pivot = xs((l + r) / 2); var i = l, j = r;
      while (i <= j) {
         while (xs(i) < pivot) { i = i + 1 }
         while (xs(j) > pivot) { j = j -1 }
         if (xs(i) <= xs(j)) {swap(i, j); i = i + 1; j
           = j -1;}
      }
      if (l < j) sort1(l, j);
      if (j < r) sort1(i, r);
   }
   sort1(0, xs.length -1);
}
```

It's just like SML

```
def sort(xs: List[int]): List[int] =
   if (xs.length <= 1) xs
   else {
      val pivot = xs(xs.length / 2);
      sort(xs.filter(x => x < pivot))
         ::: xs.filter(x => x == pivot)
         ::: sort(xs.filter(x => x > pivot))
   }
```

Everything is an Object

- Even Java value types
- Even functions
- Even case patterns

- It is just hidden behind strange syntax:

`foo.bar(value)` shortened: `foo bar value`

`foo.bar:(value)` is `value bar: foo`

The functional example again:

```
def sort(xs: List[int]): List[int] =
   if (xs.length.<=(1)) xs
   else {
      val pivot = xs(xs.length./(2));
      sort(xs.filter(x => x.<(pivot)))
         ::: xs.filter(x => x.==(pivot))
         ::: sort(xs.filter(x => x.>(pivot)))
   }
```

Functions are Objects, too

```
xs.filter(x => x < pivot)
```

Is equivalent to:

```
val compare = new Function1[boolean,int] {
     def apply(x: int,): int = x < pivot
   }
xs.filter(compare)
```

Where Function1 is a library interface:

```
trait Function1[-a,+b] {
   def apply(x: a): b
}
```

Patterns are case class constructors

Consider a reimplementation of the List class for ints:

```
trait IntList;
case class Nil extends IntList;
case class Cons(head: int, tail: IntList) extends
IntList;

val primes:IntList = Cons(13,Cons(7,Cons(11,Nil())))

def sum(l: IntList): int = l match {
   case Nil => 0
   case Cons(head, tail) => head + sum(tail)
}
```

Inheritance and Mixins

Java has single “full” inheritance and multiple interface-inheritance.
Scala has single “full” inheritance and allows multiple mixins:

```
class Thing(x:int,y:int)extends AnyRef{
   override def toString()=“(”+x+”:”+y+”)”
}
class Point extends Thing{
   def distance(that: Point)=...
}
class PositionedIntList extends IntList with Thing
```

For comprehensions

- Scala has the traditional functional container interface:

```
persons.filter(p=>p.age>20).map(p=>p.name)
```

- But also provides an interesting wrapper syntax for it:

```
for (val p<-persons; p.age>20) yield p.name
```

- It gets more interesting on more complicated examples:

```
for (val i <-List.range(1, n)
      val j <-List.range(1, i);
      isPrime(i+j)) yield Pair(i, j)
```

- Relies only on filter, map, flatMap support in the containers:

```
range(1, n).flatMap(i => range(1, i).filter(j
  => isPrime(i+j)).map(j => (i, j)))
```

Generic Types

Abstracting the IntList example over element type:

```
trait GenList[T];
case class Nil[T] extends GenList[T];
case class Cons[T](head: T, tail: GenList[T]) extends
GenList[t];

val primes:GenList[int]=Cons(13,Cons(7,Cons(11,Nil())))
val objects:GenList[Any]=Cons(13,Cons("seven"Nil()))
```

But this does not work yet:

```
val lists:GenList[GenList[Any]]=Cons(primes,Nil())
 type mismatch;
 found   : GenList[scala.Int]
 required: GenList[scala.Any]
val lists:GenList[GenList[Any]]=Cons(primes,Nil());
```

Co-variant subtyping

GenList[S] should be a subtype of GenList[T] if S is a subtype of T:

```
trait GenList[+T];
case class Nil[T] extends GenList[T];
case class Cons[T](head: T, tail: GenList[T]) extends
GenList[t];
val primes:GenList[int]=Cons(13,Cons(7,Cons(11,Nil())));
val objects:GenList[Any]=Cons(13,Cons("seven"Nil()));
val lists:GenList[GenList[Any]]=
                              Cons(primes,Cons(objects,Nil()));
```

Co-variance II

Now we can use a singleton Nil[All] instead of new, typed instances everywhere if we want to:

```
trait GenList[+T];
object Nil extends GenList[All];
class Cons[T](head: T, tail: GenList[T]) extends
GenList[t];
```

Contravariance

When is a function a subtype of another?
f1:(A1=>R1) is a subtype of f2(A2=>R2) if and only if A1 is a supertype of A1, and R1 is a subtype of R2:

```
abstract class BinaryFunct[-A,+R] {
   def apply(arg:A):R
}
class isOrdered extends BinaryFunct[Pair[Ord,Ord],
bool]{
   def apply(arg){arg._1 =< arg._2}
}

val intPairFunctions: GenList[BinaryFunc[Pair
[int,int],Any]] = Cons(isOrdered,Nil);
```

Type Bounds

Can we write generic functions for non-covariant containers?

Consider the signature of the Array based sort function:

```
sort(xs: Array[Ord]): unit
```

The comparison function is defined in the trait Ord, so all subclasses of Ord are valid inputs for the sorting function. But an array of such elements is not in any subtyping relationship with with Array[Ord]!

We need to say “Array of subtype of Ord”:

```
sort(xs: Array[T :< Ord]): unit
```

Type lower bound can be expressed similarly with `:>`

Java compatibilty

- From java bytecode scala can:
  - instaciate objects, call methods, throw/catch exceptions
  - extend classes, implement interfaces
- Java classes can be mixins when available in source code or decompiled.
- Java locking and concurency model supported, but normaly wrapped.
- Execution uses unmodified JVM bootstrapped with the scala base class library.