

Elements of Programming Languages

Lecture 11: Object-oriented functional programming

James Cheney

University of Edinburgh

October 28, 2024

Overview

- We've now covered:
 - basics of functional programming (with semantics)
 - basics of modular and OO programming (via Scala examples)
- Today, finish discussion of “programming in the large”:
 - some more advanced OO constructs
 - and how they co-exist with/support functional programming in Scala
 - *list comprehensions* as an extended example

Advanced constructs

- So far, we've covered the “basic” OOP model (circa Java 1.0), plus some Scala-isms
- Modern languages extend this model in several ways
- We can define a structure (class/object/trait) inside another:
 - As a member of the enclosing class (tied to a specific instance)
 - or as a static member (shared across all instances)
 - As a local definition inside a method
 - As an anonymous local definition
- Java (since 1.5) and Scala support “generics” (*parameterized types* as well as polymorphic functions)
- Some languages also support *mixins* (e.g. Scala traits)

Motivating inner class example

- A nested/inner class has access to the private/protected members of the containing class
- So, we can use nested classes to expose an interface associated with a specific object:

```
class List<A> {  
    private A head;  
    private List<A> tail;  
    class ListIterator<A> implements Iterator<A> {  
        ... (can access head, tail)  
    }  
}
```

Classes/objects as members

- In Scala, classes and objects (and traits) can be nested arbitrarily

```
class A { object B { val x = 1 } }
```

```
scala> val a = new A
```

```
object C {class D { val x = 1 } }
```

```
scala> val d = new C.D
```

```
class E { class F { val x = 1 } }
```

```
scala> val e = new E
```

```
scala> val f = new e.F
```

Local classes

- A *local class* (Java terminology) is a class that is defined inside a method

```
def foo(): Int = {  
  val z = 1  
  class X { val x = z + 1}  
  return (new X).x  
}  
scala> foo()  
res0: Int = 2
```

Anonymous classes/objects

- Given an interface or parent class, we can define an anonymous instance without giving it an explicit name
- In Java, called an *anonymous local class*
- In Scala, looks like this:

```
abstract class Foo { def foo() : Int }  
val foo1 = new Foo { def foo() = 42 }
```

- We can also give a *local name* to the instance (useful since this may be shadowed)

```
val foo2 = new Foo { self =>  
  val x = 42  
  def foo() = self.x  
}
```

Parameterized types

- As mentioned earlier, types can take *parameters*
- For example, `List[A]` has a type parameter `A`
- This is related to (but different from) polymorphism
 - A polymorphic function (like `map`) has a type that is parameterized by a given type.
 - A parameterized type (like `List[_]`) is a type *constructor*: for every type `T`, it constructs a type `List[T]`.

Defining parameterized types

- In Scala, there are basically three ways to define parameterized types:
 - In a type abbreviation (NB: multiple parameters)

```
type Pair[A,B] = (A,B)
```

- in a (abstract) class definition

```
abstract class List[A]  
case class Cons[A](head: A, tail: List[A])  
  extends List[A]
```

- in a trait definition

```
trait Stack[A] { ...  
}
```

Using parameterized types inside a structure

- The type parameters of a structure are implicitly available to all components of the structure.
- Thus, in the `List[A]` class, `map`, `flatMap`, `filter` are declared as follows:

```
abstract class List[A] {  
  ...  
  def map[B](f: A => B): List[B]  
  def filter(p: A => Boolean): List[A]  
  def flatMap[B](f: A => List[B]): List[B]  
    // applies f to each element of this,  
    // and concatenates results  
}
```

Parameterized types and subtyping

- By default, a type parameter is *invariant*
 - That is, neither covariant nor contravariant
- To indicate that a type parameter is *covariant*, we can prefix it with +

```
abstract class List[+A] // see tutorial 6
```

- To indicate that a type parameter is *contravariant*, we can prefix it with -

```
trait Fun[-A,+B] // see next few slides...
```

- Scala **checks** to make sure these variance annotations make sense!

Type bounds

- Type parameters can be given *subtyping bounds*
- For example, in an interface (that is, trait or abstract class) I:

```
type T <: C
```

says that abstract type member T is constrained to be a subtype of C.

- This is checked for any module implementing I
- Similarly, type parameters to function definitions, or class/trait definitions, can be bounded:

```
fun f[A <: C](...) = ...  
class D[A <: C] { ... }
```

- Upper bounds A >: U are also possible...

Traits as mixins

- So far we have used Scala's `trait` keyword for “interfaces” (which can include type members, unlike Java)
- However, traits are considerably more powerful:
 - Traits can contain fields
 - Traits can provide (“default”) method implementations
- This means traits provide a powerful form of modularity: *mixin composition*
 - Idea: a trait can specify extra fields and methods providing a “behavior”
 - Multiple traits can be “mixed in”; most recent definition “wins” (avoiding some problems of multiple inheritance)
- Java 8's support for “default” methods in interfaces also allows a form of mixin composition.

Tastes terrific, and look at that shine!

- Shimmer is a floor wax!

```
trait FloorWax { def clean(f: Floor) { ... } }
```

- No, it's a delicious dessert topping!

```
trait TastyDessertTopping {  
  val calories = 1000  
  def addTo(d: Dessert) { d.addTo(calories) }  
}
```

- In Scala, it can be both:

```
object Shimmer extends FloorWax  
  with TastyDessertTopping { ... }
```

Pay no attention to the man behind the curtain...

- Scala bills itself as a “multi-paradigm” or “object-oriented, functional” language
- How do the “paradigms” actually fit together?
- Some features, such as case classes, are more obviously “object-oriented” versions of “functional” constructs
- Until now, we have pretended pairs, λ -abstractions, etc. are primitives in Scala
- **They are not primitives**; and they need to be implemented in a way compatible with Java/JVM assumptions
 - But how do they really work?

Function types as interfaces

- Suppose we define the following interface:

```
trait Fun[-A,+B] { // A contravariant, B covariant
  def apply(x: A): B
}
```

- This says: an object implementing `Fun[A,B]` has an `apply` method
- Note: This is basically the `Function` trait in the Scala standard library!
 - Scala translates `f(x)` to `f.apply(x)`
 - Also, `{x: T => e}` is essentially syntactic sugar for `new Function {def apply(x:T) = e }`!

Iterators and collections in Java

- Java provides standard interfaces for *iterators* and *collections*

```
interface Iterator<E> {
    boolean hasNext()
    E next()
    ...
}
interface Collection<E> {
    Iterator<E> iterator()
    ...
}
```

- These allow programming over different types of collections in a more abstract way than “indexed for loop”

Iterators and foreach loops

- Since Java 1.5, one can write the following:

```
for(Element x : coll) {  
    ... do stuff with x ...  
}
```

Provided `coll` implements the `Collection<Element>` interface

- This is essentially syntactic sugar for:

```
for(Iterator<Element> i = coll.iterator();  
    i.hasNext(); ) {  
    Element x = i.next();  
    ... do stuff with x ...  
}
```

foreach in Scala

- Scala has a similar for construct (with slightly different syntax)

```
for (x <- coll) { ... do something with x ... }
```

- For example:

```
scala> for (x <- List(1,2,3)) { println(x) }  
1  
2  
3
```

foreach in Scala

- The construct `for (x <- coll) { e }` is syntactic sugar for:

```
coll.foreach{x => ... do something with x ...}
```

if `x: T` and `coll` has method `foreach: (A => ()) => ()`

- Scala expands for loops **before** checking that `coll` actually provides `foreach` of appropriate type
- If not, you get a somewhat mysterious error message...

```
scala> for (x <- 42) {println(x)}  
<console>:11: error: value foreach is not a  
  member of Int
```

Comprehensions: Mapping

- Scala (in common with Haskell, Python, C#, F# and others) supports a rich “comprehension syntax”
- Example:

```
scala> for(x <- List("a","b","c")) yield (x + "z")  
res0: List[Int] = List(az,bz,cz)
```

- This is shorthand for:

```
List("a","b","c").map{x => x + "z"}
```

where `map[B](f: A => B): List[B]` is a method of `List[A]`.

- (In fact, this works for any object implementing such a method.)

Comprehensions: Filtering

- Comprehensions can also include *filters*

```
scala> for(x <- List("a","b","c");  
        if (x != "b")) yield (x + "z")  
res0: List[Int] = List(az,cz)
```

- This is shorthand for:

```
List("a","b","c").filter{x => x != "b"}  
  .map{x => x + "z"}
```

where `filter(f: A => Boolean): List[A]` is a method of `List[A]`.

Comprehensions: Multiple Generators

- Comprehensions can also iterate over several lists

```
scala> for(x <- List("a","b","c");
         y <- List("a","b","c");
         if (x != y)) yield (x + y)
res0: List[Int] = List(ab,ac,ba,bc,ca,cb)
```

- This is shorthand for:

```
List("a","b","c").flatMap{x =>
  List("a","b","c").flatMap{y =>
    if (x != y) List(x + y) else {Nil}}}
```

where `flatMap(f: A => List[B]): List[B]` is a method of `List[A]`.

Summary

- In the last few lectures we've covered
 - Modules and interfaces
 - Objects and classes
 - How they interact with subtyping, type abstraction
 - and how they can be used to implement “functional” features (particularly in Scala)
- This concludes our tour of “programming in the large”
- (though there is much more that could be said)
- Next time:
 - imperative programming