CSCI 334:
Principles of Programming Languages

Lecture 20: Scala II

Instructor: Dan Barowy

Williams

Announcements

HW9

Type Upper Bounds

Box should contain a specific Fruit (not Fruit in general)

```
trait Fruit

trait Box[F <: Fruit] {
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}
```

Type Upper Bounds

Box should contain a specific Fruit (not Fruit in general)

```
trait Fruit

trait Box[F <: Fruit] {
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}


class Apple extends Fruit

class AppleBox(apple: Apple) extends Box[Apple] {
  def fruit = apple
}
```

Type Upper Bounds

```
class Orange extends Fruit

val o = new Orange
val abox = new AppleBox(o)

<console>:13: error: type mismatch;
 found   : Orange
 required: Apple
       val abox = new AppleBox(o)
                               ^
```

Good!

Covariance

Apple <: Fruit, so we can do this:

```
val a = new Apple
val f: Fruit = a

scala> val f: Fruit = a
f: Fruit = Apple@4e61a863
```

But we can't do this. Why not?!

```
val abox = new AppleBox(a)

scala> val box: Box[Fruit] = abox
<console>:14: error: type mismatch;
 found   : AppleBox
 required: Box[Fruit]
Note: Apple <: Fruit (and AppleBox <: Box[Apple]), but trait Box is
invariant in type F.
You may wish to define F as +F instead. (SLS 4.5)
       val box: Box[Fruit] = abox
                             ^
```

Covariance

What we want:

```
F <: Fruit

Box[F] <: Box[Fruit]
```

This is not true in Scala by default

(but the fix is simple)

Covariance

```
trait Box[+F <: Fruit] {
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}


class AppleBox(a: Apple) extends Box[Apple] {
  def fruit = a
}
```

Now it works:

```
scala> val abox = new AppleBox(new Apple)
abox: AppleBox = AppleBox@38d895e8

scala> val box: Box[Fruit] = abox
box: Box[Fruit] = AppleBox@38d895e8
```

Type Constructors

What is a type constructor anyway?

Basically: a function that produces new objects.

We get them "for free" when we define classes.

```
scala> class Apple extends Fruit
defined class Apple

scala> new Apple
res1: Apple = Apple@77c41838
```

Or when we explicitly provide definitions for them.

```
class AppleBox(a: Apple) extends Box[Apple] {
  …
}

scala> val abox = new AppleBox(new Apple)
abox: AppleBox = AppleBox@38d895e8
```

Type Constructors

E.g., for the AppleBox class:

```
class AppleBox(a: Apple) extends Box[Apple] {
  …
}
```

The type of the constructor is:

```
Apple -> AppleBox
```

Type Constructor Polymorphism

We already know that generic functions are useful:

```
def chooseFruit[F <: Fruit](pair: (F,F)) = pair._1

scala> chooseFruit((new Apple, new Apple))
res2: Apple = Apple@55e073c8
```

What about generic constructors?

Type Constructor Polymorphism

Let's build a Truck that carries Fruit boxes.

```
scala> class Truck(boxes: List[Box])
<console>:12: error: trait Box takes type parameters
       class Truck(boxes: List[Box])
                               ^
```

What parameter should we put here? What if we instead write:

```
class Truck[B <: Box[Fruit]](boxes: List[B]) {
  def honk = "HONK!"
}
```

Type Constructor Polymorphism

Seems to work…

```
scala> val abox = new AppleBox(new Apple)
abox: AppleBox = AppleBox@325f9758

scala> val obox = new OrangeBox(new Orange)
obox: OrangeBox = OrangeBox@16f453c9

scala> val t = new Truck(List(abox, obox))
t: Truck[Box[Fruit]] = Truck@15804891
```

But wait… Truck now takes type parameters. Do we really care what kind of Box the Truck carries?

```
scala> def honker(t: Truck[Box]) = t.honk
<console>:15: error: trait Box takes type parameters
       def honker(t: Truck[Box]) = t.honk
```

Type Constructor Polymorphism: Kinds

Instead, we need to say that we don't care about the type of Fruit:

```
import scala.language.higherKinds

class Truck[Box[_ <: Fruit]](boxes: List[Box[_]]) {
  def honk = "HONK!"
}

def honker(t: Truck[Box]) = t.honk


scala> def honker(t: Truck[Box]) = t.honk
honker: (t: Truck[Box])String

scala> honker(t)
res4: String = HONK!
```

Existential Types

But actually… we could go even further. Isn't there really just one kind of Truck? They all carry boxes.

```
class Truck(boxes: List[Box[_]]) {
  def honk = "HONK!"
}

def honker(t: Truck) = t.honk


scala> val t = new Truck(List(abox, obox))
t: Truck = Truck@4b186d43

scala> honker(t)
res5: String = HONK!
```

One Weird Type Trick

We used generics when creating AppleBox before. We could have used a type variable instead.

```
trait Box {
  type F <: Fruit
  def fruit: F
  def contains(aFruit: Fruit) = fruit == aFruit
}

class AppleBox(a: Apple) extends Box {
  type F = Apple
  def fruit = a
}
```

It plays nice without covariance annotations because we never had to specify a generic parameter to `box`.

```
scala> val box: Box = new AppleBox(new Apple)
box: Box = AppleBox@611c3eae
```

Implicit Conversions

Implicit conversions are common in many languages.

Here's a simple demonstration in Ruby:

```
def foo(i)
  i / 2.0
end

a = 1
b = foo(a)

puts a.class // prints "Fixnum"
puts b.class // prints "Float"
```

Implicit Conversions

Scala gives you precise control of implicit conversions.

Suppose I want to be able to write the following:

```
scala> 1.repeat(10)
```

and get

```
res4: List[Int] = List(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

How would I make this happen?

Implicit Conversions

You have to put methods in a classes… somewhere.

```
class BetterInt(i: Int) {
  def repeat(n: Int): List[Int] = List.fill(n)(i)
}
```

But this isn't quite what we want:

```
scala> val b = new BetterInt(1)
b: BetterInt = BetterInt@335896bd

scala> b.repeat(10)
res4: List[Int] = List(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

Implicit Conversions

Implicit conversions tells Scala that it's OK to silently convert `Int` to `BetterInt`.

As usual, we have to enable the feature first:

```
scala> import scala.language.implicitConversions
import scala.language.implicitConversions
```

Define the conversion:

```
scala> implicit def Int2BetterInt(i: Int) = new BetterInt(i)
Int2BetterInt: (i: Int)BetterInt
```

Now we can do what we want:

```
scala> 1.repeat(10)
res5: List[Int] = List(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

Pointer Exercises from HW8 3/4