The Scala Type System

Part 3

Existential Types, Structural Types, Refinement Types, Self Types, Infix Type Notation

Agenda

1. Existential Types, forSome
2. Structural Types (Static Duck-Typing)
3. "Easy" Reflection
4. Refinement Types
5. Self Types and Self Aliases
6. Infix Type Notation
7. Scala Enumerations

Type Parameters are not Optional

- When using a parameterized type in a method or definition, you cannot skip the parameter:

```
def lengthOfList(xs: List) = xs.length
> Main.scala:23: class List takes type parameters
>               def lengthOfList(xs: List) = xs.length
>                                      ^
```

- We could use a generic to satisfy it:

```
def lengthOfList[T](xs: List[T]) = xs.length
```

- But we don't use `T` anywhere or care what it is in this case

Existential Types

- Existential types allow us to skip the definition of a type parameter in the method:

```
def lengthOfList(xs: List[T forSome {type T}]) = xs.length
```

- The `T forSome {type T}` means "I know there is a type parameter here, I just don't care about it"

- There is a shorthand form (because that is a lot of typing):

```
def lengthOfList(xs: List[_]) = xs.length
```

- So `[_]` replaces `[T forSome {type T}]` and is equivalent except that we cannot refer to the name T any more (which is rarely, but occasionally, needed)

Bounding Existential Types

- You can still use bounds with existential types, e.g. to write

```
def fruitNames[T <: Fruit](fruits: List[T]) = fruits.map(_.name)
```

- you can use existentials, like this:

```
def fruitNames(fruits: List[T forSome {type T <: Fruit}]) = fruits.map(_.name)
```

- or alternatively the placeholder shorthand:

```
def fruitNames(fruits: List[_ <: Fruit]) = fruits.map(_.name)
```

- In all three cases you can use `.name` since we know it is a Fruit subtype
- but only with the full generic form can you use T for another parameter, or for a return type

Structural Types

- Although a different concept, these are often used with existential types, like this:

```
def maxSizeInSeq(xs: Seq[_ <: {def size: Int}]) = xs.map(_.size).max
```

- The `_ <: {def size: Int}` (which could be written with `forSome`) means any type that defines a method called `size`, with no parameters, returning an `Int`

- Within the `maxSizeInSeq` method, we only know one thing about the contents of xs, which is that they have a `size` method resulting in an Int. Therefore, we can call that

- This is often referred to as *static duck typing* in Scala (if it quacks like a duck…) but it is fully compile time verified still unlike dynamically typed languages which use duck typing

- Behind the scenes, reflection is used to invoke the `.size` method, so you should be aware of the performance costs

Tricks with Structural Types

- Structural types provide a very useful shortcut for actual reflection:

```
val s = "hello"
s.length
> res12_0: Int = 5
val obj1: AnyRef = s
obj1.length  // won't work - wrong type
> Main.scala:25: value length is not a member of AnyRef
```

- The second call on `AnyRef` will not compile since it doesn't know if `obj1` has a `.length` on it

Tricks with Structural Types

- We could use the reflection API to call it (several lines of code) or we can use a cast to a structural type like this:

```
obj1.asInstanceOf[{def length: Int}].length
> res13: Int = 5
```

- This literally means "cast to a structural type with one length method returning an Int, then call that method"
- If the method `length` is not available, you will receive a reflection `NoSuchMethodException` at runtime
- You should also `import scala.language.reflectiveCalls` to use structural types (or get a compile warning)

Refinement Types

- Let's take a look at our type parameters FruitBowl example again:

```
case class FoodBowlT[+F <: Food](item: F)
val appleBowl = FoodBowlT[Apple](fiji)
val muesliBowl = FoodBowlT[Muesli](alpen)

def feedToFruitEater(bowl: FoodBowlT[Fruit]) = println(s"Yummy ${bowl.item.name}")
```

- We can now control what our fruit eater eats, the compile will not let us provide a bowl of Muesli:

```
feedToFruitEater(appleBowl)
> Yummy fiji
feedToFruitEater(muesliBowl)
> Main.scala:27: type mismatch;
>  found   : FoodBowlT[Muesli]
>  required: FoodBowlT[Fruit]
> feedToFruitEater(muesliBowl)
>                   ^
```

- Can we achieve the same thing with type members?

Refinement Types

```scala
abstract class FoodBowl {
    type FOOD <: Food
    val item: FOOD
}

class AppleBowl extends FoodBowl {
    type FOOD = Apple
    val item = fiji
}

class MuesliBowl extends FoodBowl {
    type FOOD = Muesli
    val item = alpen
}

def feedToFruitEater(bowl: FoodBowl) = println(s"yummy ${bowl.item.name}")

feedToFruitEater(new AppleBowl)   // yummy fiji
feedToFruitEater(new MuesliBowl)  // yummy alpen -- oops!
```

- Since both AppleBowl and MuesliBowl extend FoodBowl, both compile
- We want to prevent a fruit eater eating anything but Fruit
- But we don't want to restrict them to just Apples (e.g. AppleBowl). How do we achieve this?

Refinement Types

- We can limit the type of FoodBowl that is acceptable using a refinement type:

```
// refinement type!
def safeFeedToFruitEater(bowl: FoodBowl { type FOOD <: Fruit }) =
  println(s"yummy ${bowl.item.name}")

safeFeedToFruitEater(new AppleBowl)
> yummy fiji
safeFeedToFruitEater(new MuesliBowl)
> Main.scala:27: type mismatch;
>  found   : MuesliBowl
>  required: FoodBowl{type FOOD <: Fruit}
> safeFeedToFruitEater(new MuesliBowl)
                     ^
```

- Now our method specifies that only `FoodBowls` with a `FOOD` subtype of `Fruit` is acceptable

- Because the inner type FOOD refines what is acceptable, we call this a refinement type

Self Types and Aliases

Consider an embedded class:

```
case class Person(name: String) {
    case class LifePartner(name: String) {
        def describe: String = s"$name loves $name"
    }
}
```

What we are trying to do is refer to both the LifePartner name and the Person name in the same inner class, but both refer to the LifePartner name

We can do this with:

```
case class Person(name: String) { outer =>
    case class LifePartner(name: String) {
        def describe: String = s"${this.name} loves ${outer.name}"
    }
}
```

the `outer =>` provides an alias for the outer class `this` that we can use later, we could also have said `case class LifePartner(name: String) { inner =>` for the inner class if we wanted to, which would alias the inner `this`

Self Types with type requirements

```
import com.typesafe.scalalogging._

trait Loves { this: LazyLogging =>
    def loves(i1: AnyRef, i2: AnyRef) = logger.error(s"$i1 loves $i2")
}
```

- Here we separate out a `Loves` trait, and log that someone loves someone else
- In order to use this trait, the class using it **must also provide** `LazyLogging` when it is defined

Self Types with type requirements

- Unlike `extends`, using a self type does not affect inheritance order, only asserting that the `LazyLogging` must be in the concrete class somewhere

To use:

```
case class Lovers(name1: String, name2: String) extends Loves with LazyLogging {
  def describe: Unit = loves(name1, name2)
}
```

- The `Lovers` implementation must provide `LazyLogging` or this will not compile
- However it may choose to provide a *sub-type* of `LazyLogging` also, giving a great deal of control

Infix Type Notation

Scala method infix rules:

- Infix can be used for any instance method with a single parameter, e.g.

```
val s = "hello"
s.charAt(1)
> res1: Char = 'e'
s charAt 1
> res2: Char = 'e'
```

- This can be used for any `item.method(singleParam)` and is particularly effective with symbolic methods

Infix Type Notation

- There is similar syntactic sugar for types with two type parameters:

```
class Loves[T1, T2] {
    def describe(i1: T1, i2: T2) = s"$i1 loves $i2"
}
```

```
case class NamedLoves(p1: String, p2: String) extends Loves[String, String] {
    def sayIt: String = describe(p1, p2)
}
```

is equivalent to

```
case class PersonLoves(p1: Person, p2: Person) extends (Person Loves Person) {
    def sayIt: String = describe(p1, p2)
}
```

and

```
def sayItWithRoses(lovers: Person Loves Person) = ???
```

Scala Enumerations

In Scala these are implemented using Path Dependent Types (see previous module)

```
object Color extends Enumeration {
  val Red, Green, Yellow, Blue = Value
}

object Size extends Enumeration {
  val S = Value(1, "Small")
  val M = Value(2, "Medium")
  val L = Value(3, "Large")
  val XL = Value(4, "Extra Large")
}
```

By default, Value increments the Int value by 1 on each call.

Scala Enumerations

Being path dependent types, the enumerated values are now type safe:

```
Color.values
// Color.ValueSet = Color.ValueSet(Red, Green, Yellow, Blue)

Size.values
// Size.ValueSet = Size.ValueSet(Small, Medium, Large, Extra Large)

Color.Green.id   // 1
Size.S.id        // 1

def shirt(color: Color.Value, size: Size.Value): String =
  s"A nice hawaiian shirt, color $color, size $size"

shirt(color = Color.Red, size = Size.XL)  // fine

shirt(color = Size.S, size = Color.Yellow)  // will not compile
```

Scala Enumerations

- Occasionally useful
- More limited than Java Enumerations
- If you need more power, use a sealed trait and case classes/objects