The Scala Type System

Part 2

Type Members, Bounds, Path-Dependent Types, F-Bounded Polymorphism

Agenda

1. Type Members vs Type Parameters
2. More Bounds
3. Path-Dependent Typing
4. F-Bounded Polymorphism and Recursive Types

Type Parameters Recap

```scala
trait Food { val name: String }
trait Fruit extends Food
trait Cereal extends Food

case class Apple(name: String) extends Fruit
case class Orange(name: String) extends Fruit
case class Muesli(name: String) extends Cereal

case class FoodBowl[+F <: Food](food: F) {
    def eat: String = s"Yummy ${food.name}"
}
```

```scala
val fiji = Apple("fiji")
val jaffa = Orange("jaffa")
val alpen = Muesli("alpen")
```

Type Parameters In Use

- Type parameter becomes part of the overall Type
- Variance governs subtyping relationships
- Compiler tracks types from outside

```
val bowlOfAlpen = FoodBowl(alpen)
bowlOfAlpen.eat

> bowlOfAlpen: FoodBowl[Muesli] = FoodBowl(Muesli("alpen"))
> res2_1: String = "Yummy alpen"

val foodInBowl = bowlOfAlpen.food
> foodInBowl: Muesli = Muesli("alpen")

val foodInBowl: Muesli = bowlOfAlpen.food
> foodInBowl: Muesli = Muesli("alpen")
```

- Compiler Knows Muesli is the type
- We can specify it explicitly

But

- What we can't do is access `F` from outside the class, e.g.

```
val foodInBowl: bowlOfAlpen.F = bowlOfAlpen.food
Main.scala:24: type F is not a member of FoodBowl[Muesli]
              val foodInBowl: bowlOfAlpen.F = { () =>
```

- The clue is in the error: `type F is not a member`
- What if it was?

Type Members

```
abstract class FoodBowl2 {
    type FOOD <: Food
    val food: FOOD
    def eat: String = s"Yummy ${food.name}"
}
```

- Type `FOOD` is abstract, but must be a subtype of `Food` (including `Food`)
- Must be overridden by a concrete class

```
class AppleBowl(val food: Apple) extends FoodBowl2 {
    type FOOD = Apple
}
```

- We can use methods available on `Food` subtypes
- We can refer to `FOOD` as a type in the class (just like a parameter)

And!

- We can refer to the `FOOD` type **outside** of the class now too

```
val apple = appleBowl.food
> apple: Apple = Apple("fiji")

val apple: Apple = appleBowl.food
> apple: Apple = Apple("fiji")

val apple: appleBowl.FOOD = appleBowl.food
> apple: appleBowl.FOOD = Apple("fiji")
```

- `appleBowl.FOOD` is now identical to `Apple` in the Scala type-system (aliased), they are interchangeable

- You can also access the type `FOOD` through the `AppleBowl` class using a projection

```
val apple: AppleBowl#FOOD = appleBowl.food
> apple: Apple = Apple("fiji")
```

- Projection access from a class uses # while member access from an instant uses .

Quick Mention, Singleton Types

- Every Scala instance has a unique type member associated with it: `.type`

```
val s = "hello"
def sayHello(str: s.type) = println(s)  // s.type is the singleton type of s
```

- `sayHello` can now **only** take the specific instance `s` as an argument, e.g.:

```
sayHello(s)
> hello

sayHello("hello")
> Main.scala:25: type mismatch;
> found   : String("hello")
> required: s.type
> sayHello("hello")
```

- This is often useful for DSLs (Domain Specific Languages) and some other advanced uses
- It's also how you can refer to the type of a singleton `object` in Scala

Using Parameters to Initialize Members

```scala
class AppleBowl(val food: Apple) extends FoodBowl2 {
    type FOOD = Apple
}
```

- We have to specify the type, it can't be inferred here
- It's also fairly awkward to specify vs a type parameter, so:

```scala
object BowlOfFood {
    def apply[F <: Food](f: F) = new FoodBowl2 {
        type FOOD = F
        override val food: FOOD = f
    }
}
```

The Best of Both Worlds?

- Can now define new `BowlOfFoods` easily:

```
val appleBowl = BowlOfFood(fiji)
val orangeBowl = BowlOfFood(jaffa)
```

- Anonymous classes are constructed with the correct `FOOD` type for the `F` inferred

```
val a: Apple = appleBowl.food
val o: Orange = orangeBowl.food
> a: appleBowl.FOOD = Apple("fiji")
> o: orangeBowl.FOOD = Orange("jaffa")
```

- So the inner types are correct, and using them is easy
- But what is the type of, say, `appleBowl`?

```
appleBowl
> res18_2: FoodBowl2{type FOOD = Apple} = BowlOfFood$$anon$1@546216e9
```

- Yikes - that's quite a mouthful, don't worry, we will see why below

Path Dependent Types

- Let's look at those type members again:

```scala
val a: Apple = appleBowl.food
// is identical too
val a: appleBowl.FOOD = appleBowl.food
```

- Remember that `appleBowl.FOOD` is indistinguishable from `Apple` by the compiler (it's an alias)

```scala
val o: appleBowl.FOOD = orangeBowl.food
> Main.scala:29: type mismatch;
> found    : orangeBowl.FOOD
>    (which expands to)  Orange
> required: appleBowl.FOOD
>    (which expands to)  Apple
```

- Certainly this is as expected, but both are `FOOD`s
- `FOOD` is only accessible through a containing class or instance
- It is known as a *path dependent type* and these are distinct types with different paths

Recap So Far

- Type members available from inside **and** outside of the class or trait
- Type parameters only available from inside of the class
- Type parameters become *part of the type* and must be specified, e.g.

```
def eatFruit(bowl: FoodBowl[Fruit])  // parameter is not optional
```

- Type members are enclosed fully in a *new type*

```
class FruitBowl extends FoodBowl { type FOOD = Fruit }
def eatFruit(bowl: FruitBowl)  // member is already embedded
```

- Type parameters can have co- and contra-variance, e.g.

```
trait FruitBowl[+F <: Food]
```

- Type members cannot have variance, and use bounds instead, e.g.

```
trait FruitBowl2 { type FOOD <: Food }
```

Types Referring to Themselves

- This is quite natural without generics, e.g.:

```
case class Distance(meters: Int) {
    def larger(other: Distance): Distance =
        if (other.meters > this.meters) other else this

    def smaller(other: Distance): Distance =
        if (other.meters < this.meters) other else this

    def >(other: Distance) = this.meters > other.meters
    def <(other: Distance) = this.meters < other.meters
}
```

- The `Distance` class definition simply uses `Distance` when being defined

```
val d1 = Distance(10)
val d2 = Distance(12)
d1 larger d2
> res2_0: Distance = Distance(12)
d1 < d2
> res2_1: Boolean = true
```

Sorting by Distance

- Here is a simple insertion sort for Distance types

```
def insertDistance(item: Distance, rest: List[Distance]): List[Distance] =
  rest match {
    case Nil                      => List(item)
    case head :: _ if item < head => item :: rest
    case head :: tail             => head :: insertDistance(item, tail)
  }

def sortDistances(xs: List[Distance]): List[Distance] = xs match {
    case Nil => Nil
    case head :: tail => insertDistance(head, sortDistances(tail))
}
```

- Most of this would be re-usable by other classes, as long as they define `<`
- But since we need to prove that to the compiler, we will need a type for that!

Recursive Typing

- Define an aspect of a type in terms of the type itself!
- Let's create a `CompareT` trait with a generic parameter

```
trait CompareT[T] {
    def >(other: T): Boolean
    def <(other: T): Boolean
}
```

- For any type `T`, if we implement this trait, we know it will have `>` and `<` methods defined for `T`
- So we can use that knowledge in our generic sort

Generic Sort Using CompareT

```
def genInsert[T <: CompareT[T]](item: T, rest: List[T]): List[T] = rest match {
    case Nil                       => List(item)
    case head :: _ if item < head => item :: rest
    case head :: tail              => head :: genInsert(item, tail)
}

def genSort[T <: CompareT[T]](xs: List[T]): List[T] = xs match {
    case Nil => Nil
    case head :: tail => genInsert(head, genSort(tail))
}
```

What's that generic type again?

```
def genSort[T <: CompareT[T]]...
```

- This means *any `T` that implements `CompareT` on itself*
- We can thus use `item < head` for `item: T` and `head: T` in `genInsert`
- An implementing class is defined like this:

```
case class Distance(meters: Int) extends CompareT[Distance] {
    def >(other: Distance) = this.meters > other.meters
    def <(other: Distance) = this.meters < other.meters
}
```

- So `Distance` implements `CompareT` of its own type, and provides `>` and `<`
- And it can be sorted with our sorter:

```
val dists = List(Distance(10), Distance(12), Distance(4))
genSort(dists)
> res8_1: List[Distance] = List(Distance(4), Distance(10), Distance(12))
```

Generics FTW

- We can now easily re-use our insert method for any new type as long as it extends `CompareT` of itself

```
case class EngineSize(ci: Int) extends CompareT[EngineSize] {
    def >(other: EngineSize) = this.ci > other.ci
    def <(other: EngineSize) = this.ci < other.ci
}

val engines = List(EngineSize(454), EngineSize(232), EngineSize(356))
genSort(engines)
// List[EngineSize] = List(EngineSize(232), EngineSize(356), EngineSize(454))
```

- This is a common pattern in Scala, and carries the name **F-Bounded Polymorphism**

- Since the type `T` refers to itself in the definition, it is also called a **Recursive Type**