Table of Contents for Programming Scala, 2nd Edition

Chapter 15. Scala's Type System, Part II

This chapter continues the survey of the type system that we started in the previous chapter. The type features discussed here are the ones you'll encounter eventually, but you don't need to understand them right away if you're new to Scala. As you work on Scala projects and use third-party libraries, if you encounter a type system concept that you haven't seen before, you'll probably find it covered here. (For more depth than we can cover here, see *The Scala Language Specification*.) Still, I recommend you skim the chapter. For example, you'll see a few examples of *path-dependent* types in more advanced examples later in the book, although you won't need a "deep" understanding of them.

Path-Dependent Types

Scala, like Java before it, lets you nest types. You can access nested types using a *path* expression.

Consider the following example:

```
// src/main/scala/progscala2/typesystem/typepaths/type-
path.scalaX
package progscala2.typesystem.typepaths

class Service {
// ❶
  class Logger {
    def log(message: String): Unit = println(s"log: $message")       //
❷
  }
  val logger: Logger = new Logger
}

val s1 = new Service
val s2 = new Service { override val logger = s1.logger }     // ERROR!
❸
```

❶

Define a class `Service` with a nested class `Logger`.

❷

Use `println` for simplicitly.

❸

Compilation error!

Compiling this file produces the following error on the last line:

```
error: overriding value logger in class Service of type
this.Logger;
 value logger has incompatible type
       val s2 = new Service { override val logger = s1.logger }
                                      ^
```

Shouldn't the two `Logger`s' types be considered the same? No. The error message says it's expecting a logger of type `this.Logger`. In Scala, the type of each `Service` instance's logger is considered a different type. In other words, the actual type is *path-dependent*. Let's discuss the kinds of type paths.

C.this

For a class `C1`, you can use the familiar `this` inside the body to refer to the current instance, but `this` is actually a shorthand for `C1.this` in Scala:

```
// src/main/scala/progscala2/typesystem/typepaths/path-
expressions.scala

class C1 {
  var x = "1"
  def setX1(x:String): Unit = this.x = x
  def setX2(x:String): Unit = C1.this.x = x
}
```

Inside a type body, but outside a method definition, `this` refers to the type itself:

```
trait T1 {
  class C
  val c1: C = new C
  val c2: C = new this.C
}
```

To be clear, the `this` in `this.C` refers to the trait `T1`.

C.super

You can refer to the parent of a type with `super`:

```
trait X {
                                // Do
  def setXX(x:String): Unit = {} Nothing!
}
class C2 extends C1
class C3 extends C2 with X {
  def setX3(x:String): Unit = super.setX1(x)
  def setX4(x:String): Unit = C3.super.setX1(x)
  def setX5(x:String): Unit = C3.super[C2].setX1(x)
  def setX6(x:String): Unit = C3.super[X].setXX(x)
  // def setX7(x:String): Unit = C3.super[C1].setX1(x)    //
  ERROR
  // def setX8(x:String): Unit = C3.super.super.setX1(x)  //
  ERROR
}
```

`C3.super` is equivalent to `super` in this example. You can qualify which parent using `[T]`, as shown for `setX5`, which selects `C2`, and `setX6`, which selects `X`. However, you can't refer to "grandparent" types (`setX7`). You can't chain `super`, either (`setX8`).

If you call `super` without qualification on a type with several ancestors, to which type does `super` bind? The rules of *linearization* determine the target of `super` (see Linearization of an Object's Hierarchy).

Just as for `this`, you can use `super` to refer to the parent type in a type body outside a method:

```
class C4 {
  class C5
}
class C6 extends C4 {
  val c5a: C5 = new C5
  val c5b: C5 = new super.C5
}
```

path.x

You can reach a nested type with a period-delimited path expression. All but the last elements of a type path must be *stable*, which roughly means they must be packages, singleton objects, or type declarations that alias the same. The last element in the path can be unstable, including classes, traits, and type members. Consider this example:

```
package P1 {
  object O1 {
    object O2 {
      val name = "name"
    }
    class C1 {
      val name = "name"
    }
  }
}
class C7 {
                                  // Okay  - a reference to a
  val  name1 = P1.O1.O2.name      field
  type C1    = P1.O1.C1
// Okay  - a reference to a "leaf"
class
                                  // Okay  - same
  val  c1    = new P1.O1.C1       reason
  // val name2 = P1.O1.C1.name    // ERROR - P1.O1.C1 isn't
  stable.
}
```

The `C7` members `name1`, `C1`, and `c1` all use stable elements until the last position, while `name2` has an unstable element (`C1`) before the last position.

You can see this if you uncomment the `name2` declaration, leading to the following compilation error:

```
[error] .../typepaths/path-expressions.scala:52: value C1 is not a member
of
  object progscala2.typesystem.typepaths.P1.O1
[error]     val name2 = P1.O1.C1.name
[error]                          ^
```

Of course, avoiding complex paths in your code is a good idea.

Dependent Method Types

A new feature added in Scala 2.10 is *dependent method types*, a form of path-dependent typing that is useful for several design problems.

One application is the *Magnet Pattern*, where a single processing method takes an object, called a *magnet*, which ensures a compatible return type. For a detailed example of this technique, see the *spray.io* blog. Let's work through an example:

```
// src/main/scala/progscala2/typesystem/dependentmethodtypes/dep-
method.sc

import scala.concurrent.{Await, Future}                  // ❶
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

case class LocalResponse(statusCode: Int)                // ❷
case class RemoteResponse(message: String)
...
```

❶

Import `scala.concurrent.Future` and related classes for asynchronous computation.

❷

Define two case classes used to return a “response” from a computation, either a local (in-process) invocation or a remote service invocation. Note that they do not share a common supertype. They are completely distinct.

We’ll explore the use of `Future`s in depth in Futures. For now, we’ll just sketch the details as needed. Continuing on:

```
sealed trait Computation {
  type Response
  val work: Future[Response]
}

case class LocalComputation(
    work: Future[LocalResponse]) extends Computation {
  type Response = LocalResponse
}
case class RemoteComputation(
    work: Future[RemoteResponse]) extends Computation
{
  type Response = RemoteResponse
}
...
```

A sealed hierarchy for `Computation` covers all the kinds of “computation” performed by our service, local and remote processing. Note that the `work` to be done is wrapped in a `Future`, so it runs asynchronously. Local processing returns a corresponding `LocalResponse` and remote processing returns a corresponding `RemoteResponse`:

```
object Service {
  def handle(computation: Computation): computation.Response = {
    val duration = Duration(2, SECONDS)
    Await.result(computation.work, duration)
  }
}

Service.handle(LocalComputation(Future(LocalResponse(0))))
// Result: LocalResponse =
LocalResponse(0)
Service.handle(RemoteComputation(Future(RemoteResponse("remote call"))))
// Result: RemoteResponse = RemoteResponse(remote
call)
```

Finally, a service is defined with a single entry point `handle`, which uses `scala.concurrent.Await` to wait for the future to complete. `Await.result` returns the `LocalResponse` or `RemoteResponse`, corresponding to the input `Computation`.

Note that `handle` doesn't return an instance of a common superclass, because `LocalResponse` and `RemoteResponse` are unrelated. Instead, it returns a type dependent on the argument. It's also not possible for a `RemoteComputation` to return a `LocalResponse` and vice versa, because either combination won't type check.

Type Projections

Let's revisit our `Service` design problem in Path-Dependent Types. First, let's rewrite `Service` to extract some abstractions that would be more typical in real applications:

```
// src/main/scala/progscala2/typesystem/valuetypes/type-projection.scala
package progscala2.typesystem.valuetypes

trait Logger {
//  ❶
  def log(message: String): Unit
}

class ConsoleLogger extends Logger {
//  ❷
  def log(message: String): Unit = println(s"log: $message")
}

trait Service {
//  ❸
  type Log <: Logger
  val logger: Log
}

class Service1 extends Service {
//  ❹
  type Log = ConsoleLogger
  val logger: ConsoleLogger = new ConsoleLogger
}
```

❶

A `Logger` trait.

❷

A concrete `Logger` that logs to the console, for simplicity.

❸

A `Service` trait that defines an abstract type alias for the `Logger` and declares a field for it.

❹

A concrete service that uses `ConsoleLogger`.

Suppose we want to “reuse” the `Log` type defined in `Service1`. Let’s try a few possibilities in the REPL:

```
// src/main/scala/progscala2/typesystem/valuetypes/type-projection.sc

scala> import progscala2.typesystem.valuetypes._

       val l1: Service.Log   = new
scala> ConsoleLogger
<console>:10: error: not found: value Service
       val l1: Service.Log   = new ConsoleLogger
               ^

       val l2: Service1.Log  = new
scala> ConsoleLogger
<console>:10: error: not found: value Service1
       val l2: Service1.Log  = new ConsoleLogger
               ^

       val l3: Service#Log   = new
scala> ConsoleLogger
<console>:10: error: type mismatch;
 found   : progscala2.typesystem.valuetypes.ConsoleLogger
 required: progscala2.typesystem.valuetypes.Service#Log
       val l3: Service#Log   = new ConsoleLogger
                               ^

       val l4: Service1#Log  = new
scala> ConsoleLogger
l4: progscala2.typesystem.valuetypes.ConsoleLogger =
  progscala2.typesystem.valuetypes.ConsoleLogger@6376f152
```

Using `Service.Log` and `Service1.Log` means that Scala is looking for an *object* named `Service` and `Service1`, respectively, but these companion objects don’t exist.

However, we can *project* the type we want with `#`. The first attempt doesn’t type check. Although both `Service.Log` and `ConsoleLogger` are both subtypes of `Logger`, `Service.Log` is abstract so we don’t yet know if it will actually be a supertype of `ConsoleLogger`. In other words, the final concrete definition could be another subtype of `Logger` that isn’t compatible with `ConsoleLogger`.

The only one that works is `val l4 = Service1#Log = new ConsoleLogger`, because the types check statically.

Finally, all the simpler type specifications we write every day are called *type designators*. They are actually shorthand forms for type projections. Here are a few examples of designators and their longer projections, adapted from *The Scala Language* Specification, Section 3.2:

```
                    //
Int                 scala.type#Int
                    //
scala.Int           scala.type#Int
package pkg {
  class MyClass {
    type t
// pkg.MyClass.type#t
  }
}
```

Singleton Types

We learned about *singleton objects* that are declared with the `object` keyword. There is also a concept called *singleton types*. Any instance `v` that is a subtype of `AnyRef`, including `null`, has a unique *singleton type*. You get it using the expression `v.type`, which can be used as types in declarations to narrow the allowed instances to *one*, the corresponding instance itself. Reusing our `Logger` and `Service` example from before:

```
// src/main/scala/progscala2/typesystem/valuetypes/type-types.sc

scala> val s11 = new Service1
scala> val s12 = new Service1

scala> val l1: Logger = s11.logger
l1: ...valuetypes.Logger = ...valuetypes.ConsoleLogger@3662093

scala> val l2: Logger = s12.logger
l2: ...valuetypes.Logger = ...valuetypes.ConsoleLogger@411c6639

scala> val l11: s11.logger.type = s11.logger
l11: s11.logger.type = progscala2.typesystem.valuetypes.ConsoleLogger@3662093

scala> val l12: s11.logger.type = s12.logger
<console>:12: error: type mismatch;
 found   : s12.logger.type (with underlying type ...valuetypes.ConsoleLogger
)
 required: s11.logger.type
       val l12: s11.logger.type = s12.logger
                                      ^
```

The only possible assignment to `l11` and `l12` is `s11.logger`. The type of `s12.logger` is incompatible.

Singleton *objects* define both an instance and a corresponding type:

```
// src/main/scala/progscala2/typesystem/valuetypes/object-
types.sc

case object Foo { override def toString = "Foo says Hello!" }
```

If you want to define methods that take arguments of this type, use `Foo.type`:

```
scala> def printFoo(foo: Foo.type) = println(foo)
printFoo: (foo: Foo.type)Unit

scala> printFoo(Foo)
Foo says Hello!
```

Types for Values

Every value has a type. The term *value types* refers to all the different forms these types may take, all of which we've encountered along the way.

Warning

In this section, we are using the term *value type* following the usage of the term in *The Scala Language Specification*. However, elsewhere in the book we use the term in the more conventional sense to refer to all subtypes of `AnyVal`.

For completeness, the value types are *parameterized types*, *singleton types*, *type projections*, *type designators*, *compound types*, *existential types*, *tuple types*, *function types*, and *infix types*. Let's review the last three types, because they provide convenient syntax alternatives to the conventional way of writing the types. We'll also cover a few details that we haven't seen already.

Tuple Types

We've learned that Scala allows you to write `Tuple3[A,B,C]` as `(A,B,C)`, called a *tuple type*:

```
val t1: Tuple3[String, Int, Double] = ("one", 2, 3.14
)
val t2: (String, Int, Double)       = ("one", 2, 3.14
)
```

This is convenient for more complex types to reduce the number of nested brackets and it's a bit shorter because the `TupleN` is not present. In fact, it's rare to use the `TupleN` form of the type signature. Contrast `List[Tuple2[Int,String]]` with `List[(Int,String)]`.

Function Types

We can write the type of a function, say a `Function2`, using the arrow syntax:

```
                                                "int $i, double
val f1: Function2[Int,Double,String] = (i,d) => s$d"
                                          "int $i, double
val f2: (Int,Double) => String = (i,d) => s$d"
```

Just as it's uncommon to use the `TupleN` syntax to specify a tuple, it's rare to use the `FunctionN` syntax.

Infix Types

A type that takes two type parameters can be written in infix notation. Consider these examples using `Either[A,B]`:

```
val left1:  Either[String,Int] = Left("hello"
)
val left2:  String Either Int  = Left("hello"
)
val right1: Either[String,Int] = Right(1)
val right2: String Either Int  = Right(2)
```

You can nest infix types. They are left-associative, unless the name ends in a colon (:), in which case they are right-associative, just like for terms (we haven't emphasized this, but if an expression isn't a type, it's called a *term*). You can override the default associativity using parentheses:

```
// src/main/scala/progscala2/typesystem/valuetypes/infix-types.sc

       val xll1:  Int Either Double  Either String  = Left(Left(1
scala> ))
xll1: Either[Either[Int,Double],String] = Left(Left(1))

       val xll2: (Int Either Double) Either String  = Left(Left(1
scala> ))
xll2: Either[Either[Int,Double],String] = Left(Left(1))


       val xlr1:  Int Either Double  Either String  = Left(Right(3.14
scala> ))
xlr1: Either[Either[Int,Double],String] = Left(Right(3.14))

       val xlr2: (Int Either Double) Either String  = Left(Right(3.14
scala> ))
xlr2: Either[Either[Int,Double],String] = Left(Right(3.14))


       val xr1:   Int Either Double  Either String  = Right("foo"
scala> )
xr1: Either[Either[Int,Double],String] = Right(foo)

       val xr2:  (Int Either Double) Either String  = Right("foo"
scala> )
xr2: Either[Either[Int,Double],String] = Right(foo)


       val xl:   Int Either (Double Either String)  = Left(1
scala> )
xl: Either[Int,Either[Double,String]] = Left(1)

       val xrl:  Int Either (Double Either String)  = Right(Left(3.14
scala> ))
xrl: Either[Int,Either[Double,String]] = Right(Left(3.14))

       val xrr:  Int Either (Double Either String)  = Right(Right("bar"
scala> ))
xrr: Either[Int,Either[Double,String]] = Right(Right(bar))
```

Obviously, it can become complicated quickly.

Now, let's move on to a big and important, if sometimes challenging topic, *higher-kinded types*.

Higher-Kinded Types

We're accustomed to writing methods like the following for `Seq` instances:

```
def sum(seq: Seq[Int]): Int = seq reduce (_ + _)

                          // Result:
sum(Vector(1,2,3,4,5))    15
```

First, let's generalize the notion of addition to a *type class* (recall Type Class Pattern), which allows us to generalize the element type:

```
// src/main/scala/progscala2/typesystem/higherkinded/Add.scala
package progscala2.typesystem.higherkinded

trait Add[T] {
// ❶
  def add(t1: T, T2: T): T
}

object Add {
// ❷
  implicit val addInt = new Add[Int] {
    def add(i1: Int, i2: Int): Int = i1 + i2
  }

  implicit val addIntIntPair = new Add[(Int,Int)] {
    def add(p1: (Int,Int), p2: (Int,Int)): (Int,Int) =
      (p1._1 + p2._1, p1._2 + p2._2)
  }
}
```

❶

A trait that defines addition as an abstraction.

❷

A companion object that defines instances of the trait as implicit values of `Add` for `Int`s and pairs of `Int`s.

Now, let's try it out:

```
// src/main/scala/progscala2/typesystem/higherkinded/add-seq.sc
import progscala2.typesystem.higherkinded.Add            // ❶
import progscala2.typesystem.higherkinded.Add._

def sumSeq[T : Add](seq: Seq[T]): T =                    // ❷
  seq reduce (implicitly[Add[T]].add(_,_))

sumSeq(Vector(1 -> 10, 2 -> 20, 3 -> 30))
// Result:
(6,60)
sumSeq(1 to 10)
// Result:
55
sumSeq(Option(2))                                        // ❸ Error!
```

❶

Import the `Add` trait, followed by the implicits defined in the `Add` companion object.

❷

Use a *context bound* and `implicitly` (see Using implicitly) to “sum” the elements of a sequence.

❸

It’s an error to pass an `Option`, because `Option` is not a subtype of `Seq`.

The `sumSeq` method can “sum” any sequence for which an implicit `Add` instance is defined.

However, `sumSeq` still only supports `Seq` subtypes. What if a container isn’t a `Seq` subtype, but implements `reduce`? We would like a `sum` that’s more generic.

Scala supports *higher-kinded types*, which let us abstract over parameterized types. Here’s one possible way to use them:

```
//
src/main/scala/progscala2/typesystem/higherkinded/Reduce.scala
package progscala2.typesystem.higherkinded
import scala.language.higherKinds
// ❶

trait Reduce[T, -M[T]] {
// ❷
  def reduce(m: M[T])(f: (T, T) => T): T
}

object Reduce {
// ❸
  implicit def seqReduce[T] = new Reduce[T, Seq] {
    def reduce(seq: Seq[T])(f: (T, T) => T): T = seq reduce f
  }

  implicit def optionReduce[T] = new Reduce[T, Option] {
    def reduce(opt: Option[T])(f: (T, T) => T): T = opt reduce f
  }
}
```

❶

Higher-kinded types are considered an optional feature. A warning is issued unless you import the feature.

❷

A trait that defines “reduction” as an abstraction for higher-kinded types, `M[T]`. Using `M` as the name is an informal convention in many libraries.

❸

Define implicit instances for reducing `Seq` and `Option` values. For simplicity, we’ll just use the `reduce` methods these types already provide.

`Reduce` is declared with `M[T]` *contravariant* (the `-` in front). Why? If we make it invariant (no `+` or `-`), implicit instances where `M[T]` is `Seq` won’t get used for subtypes of `Seq`, such as `Vector`. (Try removing the `-`, then running the example that follows.) Note that the `reduce` method passes a container of type `M[T]` as an argument. As we saw in Functions Under the Hood and again in Lower Type Bounds, arguments to methods are in *contravariant position*. So, we need `Reduce` to be *contravariant* in `M[T]`.

Comparing to `Add` before, the implicits `seqReduce` and `optionReduce` are methods, rather than values, because we still have the type parameter `T` that needs to be inferred for specific instances. We can't use implicit `val`s like we could for `Add`.

Note that `Seq` is not given a type parameter in this expression, `seqReduce[T] = new Reduce[T, Seq] {…}` (and similarly for `Option` in `optionReduce`). The type parameter is inferred from the definition of `Reduce`. If you add it, e.g., `… new Reduce[T, Seq[T]]`, you get a confusing error message, "Seq[T] takes no type parameters, expected: one."

Let's use `sum2` to reduce `Option` and `Seq` instances:

```
// src/main/scala/progscala2/typesystem/higherkinded/add.sc
import scala.language.higherKinds
import progscala2.typesystem.higherkinded.{Add, Reduce}     // ❶
import progscala2.typesystem.higherkinded.Add._
import progscala2.typesystem.higherkinded.Reduce._

def sum[T : Add, M[T]](container: M[T])(                    // ❷
  implicit red: Reduce[T,M]): T =
    red.reduce(container)(implicitly[Add[T]].add(_,_))

sum(Vector(1 -> 10, 2 -> 20, 3 -> 30))
// Result:
(6,60)
sum(1 to 10)
// Result:
55
sum(Option(2))                                              // Result:
                                                            2
sum[Int,Option](None)                                       // ❸ ERROR!
```

❶ Import the `Add` and `Reduce` traits, followed by the implicits defined in their companion objects.

❷ Define a `sum` method that works with higher-kinded types (details to follow).

❸ It's an error to sum (reduce) an empty container. The type signature is added to the `sum` call to tell the compiler to interpret `None` as `Option[Int]`. Otherwise, we get a compilation error that it can't disambiguate between `addInt` and `addIntIntPair` for the `T` in `Option[T]`. With the explicit types, we get the real, runtime error we expect—that you can't call `reduce` on `None` (which is true for all empty containers).

The `sum` implementation is not trivial. We have the same context bound `T : Add` we had before. We would like to define a context bound for `M[T]`, such as `M[T] : Reduce`, but we can't because `Reduce` takes two type parameters and context bounds only work for the case of one and only one parameter. Hence, we add a second argument list with an implicit `Reduce` parameter, which we use to call `reduce` on the input collection.

We can simplify the implementation a bit more. We can redefine `Reduce` with one type parameter, the higher-kinded type, allowing us to use it in a context bound like we wanted to do before:

```
// src/main/scala/progscala2/typesystem/higherkinded/Reduce1.scala
package progscala2.typesystem.higherkinded
import scala.language.higherKinds

trait Reduce1[-M[_]] {
// ❶
  def reduce[T](m: M[T])(f: (T, T) => T): T
}

object Reduce1 {
// ❷
  implicit val seqReduce = new Reduce1[Seq] {
    def reduce[T](seq: Seq[T])(f: (T, T) => T): T = seq reduce f
  }

  implicit val optionReduce = new Reduce1[Option] {
    def reduce[T](opt: Option[T])(f: (T, T) => T): T = opt reduce f
  }
}
```

❶

The `Reduce1` abstraction with one type parameter, `M`, which is still contravariant, but the type parameter is not specified. Hence, it's an *existential type* (see Existential Types). Instead, the `T` parameter is moved to the `reduce` method.

❷

The `seqReduce` and `optionReduce` implicits are now values, rather than methods.

Whereas before we needed implicit methods so the type parameter `T` could be inferred, now we have just single instances that defer inference of `T` until `reduce` is called.

The updated `sum` method is simpler, too, and it produces the same results (not shown):

```
//
src/main/scala/progscala2/typesystem/higherkinded/add1.sc
...

def sum[T : Add, M[_] : Reduce1](container: M[T]): T =
    implicitly[Reduce1[M]].reduce(container)(implicitly[Add[T]].add(_,_
))
```

We now have two context bounds, one for `Reduce1` and one for `Add`. The type parameters given on `implicitly` disambiguate between the two implicit values.

In fact, most uses of higher-kinded types you'll see will look more like this example, with `M[_]` instead of `M[T]`.

Due to the extra abstraction and code sophistication that higher-kinded types introduce, should you use them? Libraries like Scalaz and Shapeless use them extensively to compose code in very concise and powerful ways.

However, always consider the capabilities of your team members. Be wary of making code that's *so* abstract it's hard to learn, test, debug, evolve, etc.

Type Lambdas

A *type lambda* is analogous to a function nested within another function, only at the type level. They are used for situations where we need to use a parameterized type that has too many type parameters for the context. This is a coding idiom, rather than a specific feature of the type system.

Let's see an example using `map`, with a slightly different approach than we used for `reduce` in the previous section:

```
//
src/main/scala/progscala2/typesystem/typelambdas/Functor.scala
package progscala2.typesystem.typelambdas
import scala.language.higherKinds

trait Functor[A,+M[_]] {                                          //
❶
  def map2[B](f: A => B): M[B]
}
object Functor {                                                  //
❷
  implicit class SeqFunctor[A](seq: Seq[A]) extends Functor[A,Seq] {
    def map2[B](f: A => B): Seq[B] = seq map f
  }
  implicit class OptionFunctor[A](opt: Option[A]) extends Functor[A,Option] {
    def map2[B](f: A => B): Option[B] = opt map f
  }

  implicit class MapFunctor[K,V1](mapKV1: Map[K,V1])              // ❸
    extends Functor[V1,({type λ[α] = Map[K,α]})#λ] {              // ❹
      def map2[V2](f: V1 => V2): Map[K,V2] = mapKV1 map {
        case (k,v) => (k,f(v))
      }
    }
}
```

❶

The name "Functor" is widely used for types with `map` operations. We'll discuss why in The Functor Category in Chapter 16. Unlike our previous `Reduce` types, this one does not pass the collection as an argument to the method. Rather, we'll define implicit conversions to `Functor` classes that provide the `map2` method. The "2" prevents confusion with the normal `map` method. This means we don't need `M[T]` to be contravariant and in fact it's useful to make it covariant now.

❷

Define implicit conversions for `Seq` and `Option` in the usual way. For simplicitly, just use their `map` methods in the implementations of `map2`. Because `Functor` is covariant in `M[T]`, the implicit conversion for `Seq` will get used for all subtypes, too.

❸

The core of the example: define a conversion for `Map`, where we have two type parameters, instead of one.

❹

Use a *type lambda* to handle the extra type parameter.

In `MapFunctor`, we “decide” that mapping over a `Map` means keeping the keys the same, but modifying the values. The actual `Map.map` method is more general, allowing you to modify both. (In fact, we’re effectively implementing `Map.mapValues`). The syntax of the *type lambda* idiom is somewhat verbose, making it hard to understand at first. Let’s expand it to understand what it’s doing:

```
... Functor[V1,                     // ❶
  (                                 // ❷
    {                               // ❸
      type λ[α] = Map[K,α]          // ❹
    }                               // ❺
  )#λ                               // ❻
]
```

❶

`V1` starts the list of type parameters, where `Functor` expects the second one to be a container that takes *one* type parameter.

❷

Open parenthesis for expression that’s finished on line 6. It starts the definition of the second type parameter.

❸

Start defining a *structural type* (see Structural Types).

❹

Define a type member that aliases `Map`. The name λ is arbitrary (as always for type members), but it’s widely used, giving this pattern its name.[] The type has its own type parameter α (also an arbitrary name), used for the `Map` key type in this case.

❺

End the structural type definition.

❻

Close the expression started on line 2 with a type projection of the type λ out of the structural type (recall Type Projections). The λ is an alias for `Map` with an embedded type parameter that will be inferred in subsequent code.

Hence, the type lambda handles the extra type parameter required for `Map`, which `Functor` doesn’t support. The α will be inferred in subsequent code. We won’t need to reference λ or α explicitly again.

The following script verifies that the code works:

```
// src/main/scala/progscala2/typesystem/typelambdas/Functor.sc
import scala.language.higherKinds
import progscala2.typesystem.typelambdas.Functor._

List(1,2,3) map2 (_ * 2)                     // List(2, 4, 6)
Option(2) map2 (_ * 2)                       // Some(4)
val m = Map("one" -> 1, "two" -> 2, "three" -> 3)
m map2 (_ * 2)
// Map(one -> 2, two -> 4, three -> 6)
```

You don’t need the type lambda idiom often, but it’s a useful technique for the problem described. A future release of Scala may provide a simpler syntax for this idiom.

Self-Recursive Types: F-Bounded Polymorphism

Self-recursive types, technically called *F-bounded polymorphic types*, are types that refer to themselves. A classic example is Java’s Enum abstract class, the basis for all Java enumerations. It has the following declaration:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Most Java developers are mystified by the `Enum<E extends Enum<E>>` syntax, but it has a few important benefits. You can see one in the signature for the `compareTo` method that `Comparable<E>` declares:

```
int compareTo(E obj)
```

It is a compilation error to pass an object to `compareTo` that isn’t one of the enumeration values defined for the same type. Consider this example with two subtypes of `Enum` in the JDK, `java.util.concurrent.TimeUnit` and `java.net.Proxy.Type` (some details omitted):

```
scala> import java.util.concurrent.TimeUnit
scala> import java.net.Proxy.Type

scala> TimeUnit.MILLISECONDS compareTo TimeUnit.SECONDS
res0: Int = -1

scala> Type.HTTP compareTo Type.SOCKS
res1: Int = -1

scala> TimeUnit.MILLISECONDS compareTo Type.HTTP
<console>:11: error: type mismatch;
 found   : java.net.Proxy.Type(HTTP)
 required: java.util.concurrent.TimeUnit
              TimeUnit.MILLISECONDS compareTo Type.
HTTP

^
```

In Scala, recursive types are also handy for defining methods whose return types are the same as the type of the caller, even in a type hierarchy. Consider this example where the `make` method should return an instance of the caller's type, not the `Parent` type that declares `make`:

```
// src/main/scala/progscala2/typesystem/recursivetypes/f-bound.sc

trait Parent[T <: Parent[T]] {
// ❶
  def make: T
}

case class Child1(s: String) extends Parent[Child1] {                //
❷
                                "Child1: make:
  def make: Child1 = Child1(s$s"               )
}

case class Child2(s: String) extends Parent[Child2] {
                                "Child2: make:
  def make: Child2 = Child2(s$s"               )
}

                                  // c1: Child1 =
val c1  = Child1("c1")            Child1(c1)
                                  // c2: Child2 =
val c2  = Child2("c2")            Child2(c2)
val c11 = c1.make
// c11: Child1 = Child1(Child1: make:
c1)
val c22 = c2.make
// c22: Child2 = Child2(Child2: make:
c2)

                                  // p1: Parent[Child1] =
val p1: Parent[Child1] = c1       Child1(c1)
                                  // p2: Parent[Child2] =
val p2: Parent[Child2] = c2       Child2(c2)
val p11 = p1.make
// p11: Child1 = Child1(Child1: make:
c1)
val p22 = p2.make
// p22: Child2 = Child2(Child2: make:
c2)
```

❶

`Parent` has a recursive type. This syntax is the Scala equivalent of Java's syntax that we saw for `Enum`.

❷

Derived types must follow the signature idiom `X extends Parent[X]`.

Note the type signatures shown in the comments of the values created at the end of the script. For example, `p22` is of type `Child2`, even though we called `make` on a reference to a `Parent`.

Recap and What's Next

Perhaps the best example of a project that pushes the limits of the type system is Shapeless. Many advanced type concepts are also used extensively in Scalaz. They are worth studying as you master the type system and they provide many innovative tools for solving design problems.

It’s important to remember that you don’t have to master all the intricacies of Scala’s rich type system to use Scala effectively. However, the better you understand the details of the type system, the easier it will be to exploit third-party libraries that use them. You’ll also be able to build powerful, sophisticated libraries of your own.

Next we’ll explore more advanced topics in functional programming.