Type erasure and type tags

Everyone working with JVM knows that type information is erased after compilation and therefore, it is not available at runtime. This process is known as *type erasure*, and sometimes it leads to unexpected error messages.

For example, if we wanted to create an array of a certain type, which is unknown until runtime, we could write a naive implementation such as this:

```
def createArray[T](length: Int, element: T) = new Array[T](length)
```

However, because of type erasure, the above code doesn't compile:

```
error: cannot find class tag for element type T
   def createArray[T](length: Int, element: T) = new Array[T](length)
                                                 ^
```

The error message actually suggests a possible solution. We can introduce an additional implicit parameter of the type `ClassTag` to pass type information to runtime:

```
import scala.reflect.ClassTag

def createArray[T](length: Int, element: T)(implicit tag: ClassTag[T]) =
  new Array[T](length)
```

With this little adjustment, the above code compiles and works exactly as we expect:

```
scala> createArray(5, 1.0)
res1: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0)
```

In addition to the syntax shown above, there is also a shortcut that does the same thing:

```
def createArray[T: ClassTag](length: Int, element: T) = new Array[T](length)
```

Note that prior to Scala 2.10, you could achieve the same thing with `scala.reflect.Manifest`. Now this approach is deprecated and type tags are the way to go.

Existential types

In Java, type parameters were introduced only in version 1.5, and before that generic types literally didn't exist. In order to remain backwards compatible with existing code, Java still allows the use of raw types, so the following code generates a warning but compiles:

```scala
type Endpoint = PartialFunction[Request, Future[Response]]
type Application = (request: Request) => Option[Future[Response]]
```

Since the `Endpoint` now is a `PartialFunction`, we can use `orElse` to combine several of them before lifting the result to an `Application` instance.

```scala
val endpoint1: Endpoint = /* ... */
val endpoint2: Endpoint = /* ... */

val routes = endpoint1 orElse endpoint2
val app: Application = routes.lift
```

Since partial functions are not defined for all possible inputs, the server needs to deal with an empty `Option` branch, but this is trivial.

One improvement that we can make is to parametrise our API with the effect type. When we were discussing the reader monad, we encountered the following definition.

```scala
type ReaderT[F[_], A, B] = Kleisli[F, A, B]
```

Given that `A` is `Request` and `B` is `Response`, this type matches our API perfectly. The authors of `http4s` noticed that as well, and this library is exactly what we're going to discuss next.

http4s

`http4s` is a Typelevel project that helps with writing HTTP services in a purely functional and typeful way. http4s[25] supports several popular backends including Tomcat and Jetty and it also comes with its own backend called Blaze. In this book, we're going to stick to it, so let's add all necessary modules to `build.sbt`:

```scala
val http4sVersion = "0.20.0-M2"
val logbackVersion = "1.2.3"

val http4sDependencies = Seq(
  "org.http4s" %% "http4s-dsl" % http4sVersion,
  "org.http4s" %% "http4s-blaze-server" % http4sVersion,
  "org.http4s" %% "http4s-circe" % http4sVersion,
  "io.circe" %% "circe-generic" % circeVersion,
  "ch.qos.logback" % "logback-classic" % logbackVersion
)
```

[25] http://http4s.org/

```scala
trait Endpoint[A] {
 def map[B](fn: A => B): Endpoint[B]
 def toService: Service[Request, Response]
}
```

This trait has a number of methods for mapping the values and composing several `Endpoints` together. The `toService` methods connects Finch with the underlying Finagle infrastructure by converting an `Endpoint` into a Finagle `Service`.

The `io.finch.syntax` package provides DSL-like constructs for defining `Endpoints` in a convenient way. As for the output of the response, it's represented in Finch by the following (simplified) hierarchy:

```scala
sealed trait Output[+A]
case class Payload[A] extends Output[A]
case class Failure extends Output[Nothing]
case class Empty extends Output[Nothing]
```

For convenience, the `Outputs` trait provides a number of methods describing common outputs for different situations. In particular:

```scala
trait Outputs {
  def Ok[A](a: A): Output[A] = Output.payload(a, Status.Ok)
  def NotFound(cause: Exception): Output[Nothing] =
    Output.failure(cause, Status.NotFound)
}
```

For example, we can define an endpoint that returns current time in Epoch milliseconds:

```scala
object Endpoints {
  import io.finch._
  import io.finch.syntax._

  val timeE = get("time") { Ok(System.currentTimeMillis().toString) }
}
```

This endpoint doesn't do much, but let's try it out anyway. We can call the `toService` method now and pass an obtained `Service` instance to the Finagle server:

Of course, there is a method called `mkString` that does what we need, but it only works with strings and we want to build something more generic.

We can achieve our goal by turning our `Monoid` definition into a type class. First, let's write an implementation of `Monoid` for strings:

```
object DefaultMonoids {
  implicit val stringConcatMonoid = new Monoid[String] {
    override def compose(a: String, b: String): String = s"$a$b"
    override def empty: String = ""
  }
}
```

And then, let's define an operation called `combineAll` that relies only on methods from the `Monoid` trait:

```
object Operations {
  def combineAll[A](list: List[A])(implicit monoid: Monoid[A]): A = {
    list.foldRight(monoid.empty)((a, b) => monoid.compose(a, b))
  }
}
```

With the above definitions in place, we can combine a list of strings easily:

```
import DefaultMonoids._

val result = Operations.combineAll(List("a", "b", "cc"))
println(result)      // prints "abcc"
```

The good thing about combining objects generically is that you can use the same operation for different implementations. If, at some point, you need to combine multiple objects of, say, type `Area` to calculate the total, the only thing that will be needed is a `Monoid[Area]` instance.

Monoids in Cats

A library called Cats[10] contains many abstractions from category theory including the `Monoid` type class.

The library is split into several modules, but in order to start using it simply add the following to your dependencies:

[10] http://typelevel.org/cats/

```
trait Monad[F[_]] extends FlatMap[F] {
  def flatMap[A, B](fa: F[A])(f: (A) => F[B]): F[B]
  def pure[A](x: A): F[A]
}
```

Since Cats 0.7.0, in addition to `pure` and `flatMap`, users are also required to implement the `tailRecM` method:

```
trait FlatMap[F[_]] {
  def tailRecM[A, B](a: A)(f: A => F[Either[A, B]]): F[B]
}
```

It was introduced to support stack-safe monadic transformations by means of `flatMap`. The good news is that once you have a proper implementation of `tailRecM`, you don't need to worry about stack-safety. The bad news is that, well, you have to write it.

Fortunately, there are only several distinct monadic shapes and therefore, only a handful of different `tailRecM` implementations. For example, our dummy `Cell` class is actually pretty similar to `Option`, and that means that we can implement `tailRecM` in a similar fashion.

Let's add monadic features to our `Cell` class.

```
case class Cell[A](value: A) {
  def bind[B](f: A => Cell[B]): Cell[B] = f(value)
}
```

Here we're removing the `map` function, because Cats will give it to us for free. Also, we're defining a helper function called `bind` that we will use for implementing a `Monad` instance for our class. Basically, it does the same thing as `flatMap`, but are using a different name to avoid confusion.

Now let's define a `Monad[Cell]`:

```
implicit val cellMonad = new Monad[Cell] {
  override def flatMap[A, B](fa: Cell[A])(f: (A) => Cell[B]):
    Cell[B] = fa.bind(f)
  override def pure[A](x: A): Cell[A] = Cell(x)
  @tailrec override def tailRecM[A, B](a: A)(f:
    (A) => Cell[Either[A, B]]): Cell[B] = f(a) match {
        case Cell(Left(a1)) => tailRecM(a1)(f)
        case Cell(Right(next)) => pure(next)
      }
}
```

```
val ioa: IO[Unit] = for {
  _ <- contextShift.shift
  _ <- IO { println("Enter your name: ") }
  _  <- IO.shift(blockingCtx)
  name <- IO { scala.io.StdIn.readLine() }
  _ <- contextShift.shift
  _ <- IO { println(s"Hello $name!") }
} yield ()
```

Note that the second call to the parameterless `shift` method moves the computation back from the blocking thread pool.

Alternatively, we could use `evalOn` method defined on `ContextShift`:

```
val ioa: IO[Unit] = for {
  _ <- contextShift.shift
  _ <- IO { println("Enter your name: ") }
  name <- contextShift.evalOn(blockingCtx)(
    IO { scala.io.StdIn.readLine() }
  )
  _ <- IO { println(s"Hello $name!") }
  _ <- IO { blockingService.shutdown() }
} yield ()
```

The two pieces of code are equivalent.

- `IterateeModule`
- `EnumeratorModule`
- `EnumerateeModule`

If you look at their source code, you will see that all modules accept `F[_]` as a type parameter. The `F[_]` type constructor chooses the monadic context that our program will be using. This choice is usually made via imports, and there are several built-in options available:

import	monad
`io.iteratee.modules.id._`	`cats.Id`
`io.iteratee.modules.eval._`	`cats.Eval`
`io.iteratee.modules.option._`	`scala.Option`
`io.iteratee.modules.either._`	`scala.util.Either`
`io.iteratee.monix.task._`	`monix.eval.Task`

Note that for `Either` the actual monad type is slightly more involved:

```
trait EitherModule extends Module[({ type L[x] = Either[Throwable, x] })#L]
```

The expression in square brackets is known as *anonymous type expression* or *type lambda*. This is somewhat similar to *lambda expressions* and achieves the same thing as the following:

```
type EitherMonad[A] = Either[Throwable, A]
trait EitherModule extends Module[EitherMonad]
```

The good thing about module organization in `iteratee-io` is that it allows users to work with one generic API and switch monad types by changing only one line of code.

Building simple transformations

The simplest possible method defined on `EnumeratorModule` is probably `enumOne`:

```
def enumOne[E](e: E): Enumerator[F, E]
```

The enumerator returned by `enumOne` will produce a single value.

On the receiving side, we can also use something very simple. For example, the `takeI` method returns an iteratee that consumes the specified number of elements:

```
def takeI[E](n: Int): Iteratee[F, E, Vector[E]]
```

Applying everything we've learned so far, we can come up with the following program:

Prism

Sometimes, you have a function `A => B` that is not defined for all values in `A`. At the same time, the reverse function `B => A` also exists and it is defined for all values in `B`. This situation is often called a *relaxed isomorphism* and can be represented by an optic called prism:

```
case class Prism[A, B](
  getOption: A => Option[B],
  reverseGet: B => A
)
```

Prisms can be seen as a generalization of abstract data types and pattern matching.

Consider the following types representing configuration values:

```
sealed trait ConfigValue
case class IntValue(value: Int) extends ConfigValue
case class StringValue(value: String) extends ConfigValue
```

Using these definitions, we can create a `Prism` that extracts `Int` values from an `IntValue`:

```
val intConfP = Prism[ConfigValue, Int] {
  case IntValue(int) => Some(int)
  case _ => None
}(IntValue.apply)
```

Now, imagine that we have a `portNumber` value of type `ConfigValue` and a function `offsetPort` that adds `8000` to the passed integer:

```
val portNumber: ConfigValue = /* read by an external system */
def offsetPort(port: Int): Int = port + 8000
```

With `Prism`, we can safely lift `offsetPort` to work with `ConfigValue`s:

```
val updatedPort = intConfP.modify(offsetPort)(portNumber)
```

If `portNumber` is an instance of `IntValue`, the `updatedPort` will contain a `ConfigValue` with the updated port. If not, it will contain the old value. Instead of `modify`, you can also use `modifyOption` that returns `Some` with an updated value or `None`.

Lens

Lenses are undoubtedly the most popular optics and their usefulness is usually immediately obvious. Not surprisingly, Monocle provides a first class support for lenses.

The `Lens` is defined by two functions - `get` and `set`:

```
object Person {
  implicit val maybePerson: Option[Person] = Some(Person("User"))
}
def sayHello(implicit person: Option[Person]): String = /* ... */
```

As a result, users can define or import implicit values in the scope, but the compiler also checks object companions of associated types. We will see why this is convenient when we get to *type classes*.

Implicit conversions

Sometimes you need to change or add new methods to third-party classes. In dynamic languages this is achieved by "monkey patching", in C# or Kotlin by writing extension functions, in Scala by using *implicit conversions*.

For example, we can write an implicit conversion from a `String` to `Person`

```
case class Person(name: String) {
  def greet: String = s"Hello! I'm $name"
}
object Person {
  implicit def stringToPerson(str: String): Person = Person(str)
}
```

After importing the conversion method into scope, we will be able to treat `Strings` as `Persons` - the compiler will convert types automatically:

```
import Person.stringToPerson
"Joe".greet
// Hello! I'm Joe
```

Since conversions like these are commonly used for adding new methods, Scala also provides a shortcut:

```
implicit class StringToPerson(str: String) {
  def greet: String = s"Hello! I'm $str"
}
```

By using implicit classes we can get rid of most boilerplate.

method	description
apply	accepts a variable number of arguments and returns a Stream that will emit them
emit	accepts one argument and returns a Stream emitting it
emits	accepts a Seq and returns a Stream emitting its elements
range	accepts two values and returns a Stream emitting Ints that fit between them

For example:

```
scala> import fs2._
import fs2._

scala> val s1 = Stream(1,2,3)
s1: fs2.Stream[[x]fs2.Pure[x],Int] = Stream(..)
```

The interpreter says that the type of stream `s1` is `Stream[Pure, Int]`, which means that it emits `Int` values and doesn't evaluate any effects.

Pure streams can be converted to collections:

```
scala> s1.toList
res1: List[Int] = List(1, 2, 3)

scala> s1.toVector
res2: Vector[Int] = Vector(1, 2, 3)
```

The `Stream` class has a lot of collection-like methods such as `map`, `flatMap`, `filter`, `take`, `++` and so on. Using these methods we can organize a computation as a stream processing pipeline:

```
scala> val s2 = s1.map(_ + 1).fold(0)(_ + _)
s2: fs2.Stream[[x]fs2.Pure[x],Int] = Stream(..)

scala> s2.toVector
res2: Vector[Int] = Vector(9)
```

Note that `s2` is not actually evaluated until it's converted to `Vector`. This makes sense, because in a general case, streams are infinite and we certainly don't want to hang the program here.

In addition to already familiar collection-like methods, `Stream` introduces several new ones: