Ways to pattern match generic types in Scala

Posted by **Jaakko Pallari** on Tue, May 31, 2016

Find me on:

Occasionally in Scala, there becomes a need to pattern match on values where the type information is lost. If you know the specific type of the value you want to extract, it's easy to come up with a solution:

```
1 def extractString(a: Any): Option[String] = a match {
2   case s: String => Some(s)
3   case _ => None
4 }
```

However, when the value you want to extract contains a type parameter or is a generic type itself, the solution is not so straightforward anymore. The reason why it's so difficult comes from type erasure in the JVM: generic types only exist in the compilation phase, and not during runtime. Scala's generic types behave the same way, which is why `List[String]` is actually just a `List` in runtime. Because all lists have the same type regardless of their type parameter, it's impossible to distinguish `List[String]` from `List[Int]` by it's runtime type information only.

In this article, I demonstrate a few solutions that can be used when pattern matching against generic types. First, I'll demonstrate two ways to avoid the problem altogether. After that, I'll show how Shapeless features can be used for solving the problem. Finally, I'll show how to solve the problem using Scala's type tags.

Contents briefly:

Avoid losing the generic type

Before you try to pattern match on generic types, try to figure out if you actually need it at all. If you have the control of the code that requires pattern matching, try to structure the code in a manner that doesn't lose generic type information. Here's an example of a structure that will encounter problems in pattern matching:

```
1  sealed trait Result
2  case class Ok[T](values: List[T]) extends Result
3  case class Fail(message: String) extends Result
4
5  def handleResult(result: Result): Unit = result match {
6    case Fail(message) => println("ERROR: " + message)
7    case Ok(vs: List[String]) => println("Got strings of total length: " +
8    case _ => println("Got something else")
9  }
10
11 handleResult(Fail("something happened")) // output: "ERROR: something hap
12 handleResult(Ok(List("this", "works")))  // output: "Got strings of total
13 hanldeResult(Ok(List("doesn't", "work", 4))) // ClassCastException... oop
```

In the code above, notice the `ClassCastException` we get at runtime while attempting to pass in a list of mixed type objects. An exception like this can be fatal in a live system, but it's easy enough to write by mistake. Ideally we could verify that these kind of errors are caught during compile time rather during runtime.

Since the `Result` type didn't enforce any restrictions on the type parameter of the `Ok` type, the type parameter information is lost. We can add the type parameter to the `Result` type to allow us to better manage the types:

```
1  sealed trait Result[+T]
2  case class Ok[+T](values: List[T]) extends Result[T]
3  case class Fail(message: String) extends Result[Nothing]
4
5  def handleResult(result: Result[String]): Unit = result match {
6    case Fail(message) => println("ERROR: " + message)
7    case Ok(vs) => println("Got strings of total length: " + vs.map(_.size)
8  }
9
```

Now that the type parameter is included in the `Result` type, we can enforce boundaries to the type of the list included in the `Ok` type at compile time. This let's us avoid having to try to regain the generic parameter completely.

Making the type parameter covariant allows us to pass the `Fail` value to the `handleResult` function. Covariance in `Result` allows passing values of type `Result[T]` where `T` is a subtype of `String` to the `handleResult` function. Because `Nothing` is subtype of every other type (including `String`), `Fail` values - which are of type `Result[Nothing]` - can be passed as a parameter to `handleResult`. This allows us to keep types such as `Fail` type parameterless.

Avoid matching on generic type parameters

Sometimes you can't control what type of value gets passed to the function you're implementing. For example, the actors in Akka are forced to handle all types (type `Any`) of messages. Since you can't add boundaries to the incoming values at compile time, you will again lose the ability to distinguish between a `List[String]` and `List[Int]`.

```
1  def handle(a: Any): Unit = a match {
2    case vs: List[String] => println("strings: " + vs.map(_.size).sum)
3    case vs: List[Int]    => println("ints: " + vs.sum)
4    case _ =>
5  }
6
7  handle(List("hello", "world")) // output: "strings: 10"
8  handle(List(1, 2, 3))          // ClassCastException... oh no!
```

In the code above, if we attempt to pass a list of integers to the `handle` function, the function will attempt to interpret the list as a string list, which will end in a `ClassCastException` as we try to access the list as a string list. As explained in the earlier section, we'd like to weed out these unexpected failure cases during compile time.

If you can control the type of the values passed into the function (e.g. you can control what type of messages you sent to your actor), you can avoid the problem by boxing the input which has a type parameter with a container that specifies the type parameter:

```
def handle(a: Any): Unit = a match {
  case Strings(vs) => println("strings: " + vs.map(_.size).sum)
  case Ints(vs)    => println("ints: " + vs.sum)
  case _ =>
}

handle(Strings(List("hello", "world"))) // output: "strings: 10"
handle(Ints(List(1, 2, 3)))             // output: "ints: 6"
handle(Strings(List("foo", "bar", 4)))  // compile time error
```

In the example, we define concrete types `Strings` and `Ints` to manage the generic types for us. They ensure that you cannot build lists of mixed values, so that the `handle` function can safely use their lists without having to know the type parameters of the lists at runtime.

Introducing Typeable and TypeCase

Shapeless provides handy tools for dealing with type safe casting: `Typeable` and `TypeCase`.

Typeable is a type class that provides the ability to cast values from `Any` type to a specific type. The result of the casting operation is an `Option` where the `Some` value will contain the successfully casted value, and the `None` value represents a cast failure. Shapeless provides the Typeable capability for all Scala's primitive types, case classes, sealed traits hierarchies, and at least some of the Scala's collection types and basic classes out-of-the-box. Here are some examples of Typeable in action:

```
import shapeless._

case class Person(name: String, age: Int, wage: Double)

val stringTypeable = Typeable[String]
val personTypeable = Typeable[Person]

stringTypeable.cast("foo": Any) // result: Some("foo")
stringTypeable.cast(1: Any)     // result: None
personTypeable.cast(Person("John", 40, 30000.0): Any) // result Some(...)
personTypeable.cast("John": Any) // result: None
```

TypeCase bridges Typeable and pattern matching. It's essentially an extractor for `Typeable` instances. TypeCase and Typeable allow implementing the example in the previous section without boxing:

```
val intList    = TypeCase[List[Int]]

def handle(a: Any): Unit = a match {
  case stringList(vs) => println("strings: " + vs.map(_.size).sum)
  case intList(vs)    => println("ints: " + vs.sum)
  case _ =>
}

val ints: List[Int] = Nil

handle(List("hello", "world")) // output: "strings: 10" so far so good
handle(List(1, 2, 3))          // output: "ints: 6" yay!
handle(ints)                   // output: "strings: 0" wait... what? We'l
```

Instead of boxing the list values, the `TypeCase` instances can be used for pattern matching on the input. TypeCase will automatically use any `Typeable` instance it can find for the the given type to perform the casting operation. If the casting operation fails (produces `None`), the pattern isn't matched, and the next pattern is tried.

Keeping the generic type "generic"

While Typeable can be used for pattern matching on specific types, its true power is the ability to pattern match on types where the type parameter of the extracted type is kept generic. Here is an example use of this ability:

```
import shapeless._

def extractCollection[T: Typeable](a: Any): Option[Iterable[T]] = {
  val list = TypeCase[List[T]]
  val set  = TypeCase[Set[T]]
  a match {
    case list(l) => Some(l)
    case set(s)  => Some(s)
    case _       => None
  }
}

val l1: Any = List(1, 2, 3)
val l2: Any = List[Int]()
val s:  Any = Set(1, 2, 3)

extractCollection[Int](l1)    // Some(List(1, 2, 3))
```

In this example, we've created function `extractCollection` to extract all lists and sets of any generic type. We declare that the function type parameter `T` should have a `Typeable` instance when we call the function. The presence of the instance allows us to create extractors for a list of `T` and a set of `T`. We can then use these extractors to extract only values that conform to the types `List[T]` or `Set[T]`. For all the other values, we produce no value.

In order to use the function, we need to give a hint to the compiler what type of values we want to extract. This is done by specifying the type of extracted values as the type parameter for the function.

By keeping the extracted type generic, we've successfully separated part of the extraction logic from the type that we want to extract. This allows us to reuse the same function for all types that have a `Typeable` instance.

Typeable's secret sauce

While Typeable may look like it has what it takes to solve type erasure, it's still subject to the same behaviour as any other runtime code. This can be seen in the last lines of the previous code examples where empty lists were recognized as string lists even when they were specified to be integer lists. This is because Typeable casts are based on the values of the list. If the list is empty, then naturally that is a valid string list and a valid integer list (or any other list for that matter). Depending on the use-case, the distinction between different types of empty lists might or might not matter, but it can certainly catch the user off guard, if they're not familiar with how Typeable operates.

Moreover, since Typeable inspects the values of a collection to determine whether the collection can be cast or not, it will take longer to cast a large collection than a small one. Let's do some rudimentary profiling to see how the size of the collection affects casting.

```
1  import shapeless._
2
3  def time[T](f: => T): T = {
4    val t0 = System.currentTimeMillis()
5    val result = f
6    val t1 = System.currentTimeMillis()
7    println(s"Elapsed time: ${t1 - t0} ms")
8    result
9  }
```

```
val list4 = (1 to 100000).toList
val list5 = (1 to 1000000).toList
val list6 = (1 to 10000000).toList

val listTypeable = Typeable[List[Int]]
time { listTypeable.cast(list1: Any) } // 0 ms
time { listTypeable.cast(list2: Any) } // 1 ms
time { listTypeable.cast(list3: Any) } // 5 ms
time { listTypeable.cast(list4: Any) } // 4 ms
time { listTypeable.cast(list5: Any) } // 7 ms
time { listTypeable.cast(list6: Any) } // 70 ms
```

In the above example, we created lists of various sizes, and measured how long it took to cast them. Notice how the time required increases as the size of the collection expands.

Custom type class instance for Typeable

Occasionally you might encounter a type that you can't automatically use Typeable against. These are usually standard classes (as opposed to case classes) that have type parameters. For example, you can't automatically use Typeable on the following type:

```
class Funky[A, B](val foo: A, val bar: B) {
  override def toString: String = s"Funky($foo, $bar)"
}

```

Instead, we have to provide our own custom `Typeable` instance to allow casting `Funky` values. The `Typeable` interface requires us to implement two methods: `cast` and `describe`. The `cast` method does the real casting work, while the `describe` provides human readable information about the type being cast.

```
implicit def funkyIsTypeable[A: Typeable, B: Typeable]: Typeable[Funky[A,
  new Typeable[Funky[A, B]] {
    private val typA = Typeable[A]
    private val typB = Typeable[B]

    def cast(t: Any): Option[Funky[A, B]] = {
      if (t == null) None
      else if (t.isInstanceOf[Funky[_, _]]) {
```

```
13            } yield o.asInstanceOf[Funky[A, B]]
14          } else None
15        }
16
17        def describe: String = s"Funky[${typA.describe}, ${typB.describe}]"
18      }
```

The type class instance is parametrized with `Typeable` instances for the `Funky` class's type parameters. This allows us to use the instance for all `Funky` types where the type parameters are also `Typeable`.

With the help of the instance parameters, we can create a cast method that attempts to cast the given value to a `Funky[A, B]` when it can also cast the values inside `Funky`.

As we can see from the example, even with two type parameters and two fields, the casting process is already complex. Adding more fields and type parameters requires even more casting steps. Moreover, the casting is not enforced by the compiler (e.g. you can easily miss a casting step for a field), which means that it's exposed to casting failures.

Type tags

Another way do typesafe casting is to use Scala's type tags. A type tag is a full type description of a Scala type as a runtime value generated at compile time. Similar to Shapeless, type tags are also materialized through a type class. The type class provides the ability to access the generic type parameter's type information during runtime.

Unlike in Shapeless, the casting is not based on checking the elements inside the class. It's instead based on comparing type tags together. A type tag can tell us whether the type of a type tag conforms to the type of another type tag. Type tags allow checking if the types are equal or if there is a subtype relationship.

```
1  import scala.reflect.runtime.universe._
2
3  def handle[A: TypeTag](a: A): Unit =
4    typeOf[A] match {
5      case t if t =:= typeOf[List[String]] =>
6        // list is a string list
7        val r = a.asInstanceOf[List[String]].map(_.length).sum
8        println("strings: " + r)
```

```
      println("ints: " + r)
    case _ => // ignore rest
  }

val ints: List[Int] = Nil

handle(List("hello", "world")) // output: "strings: 10"
handle(List(1, 2, 3))          // output: "ints: 6"
handle(ints)                   // output: "ints: 0" it works!
```

In the example, we implement the old familiar `handle` function from previous examples with the help of type tags. We declare the input to have a type tag, and then compare the type to some existing known types: string lists and integer lists. If there's a match between the types, we can safely perform a cast using `asInstanceOf`. Since the type parameter is matched by type, an empty list of integers will be recognized as list of integers instead of list of strings.

Type tags and unknown types

The downside of the approach shown in the previous example is that we must provide a type tag to perform the type matching with. In the example, we rely on the type tag instance provided to the function to recover the type information for the generic type. In many cases, the function signature could be limited to just a `Any => Unit` without any type tag information. One way to get around this problem is to provide the type tag information as part of the type.

```
import scala.reflect.runtime.universe._

class Funky[A, B](val foo: A, val bar: B) {
  override def toString: String = s"Funky($foo, $bar)"
}

final case class FunkyCollection[A: TypeTag, B: TypeTag](funkySeq: Seq[Fu
  val selfTypeTag = typeTag[FunkyCollection[A, B]]

  def hasType[Other: TypeTag]: Boolean =
    typeOf[Other] =:= selfTypeTag.tpe

  def cast[Other: TypeTag]: Option[Other] =
    if (hasType[Other])
```

```
19
20 val a: FunkyCollection[String, Int] = FunkyCollection(Seq(new Funky("foo'
21 val b: FunkyCollection[_, _] = a
22
23 b.hasType[FunkyCollection[String, Int]] // true
24 b.hasType[FunkyCollection[Int, String]] // false
25 b.cast[FunkyCollection[String, Int]]    // Some(a)
26 b.cast[FunkyCollection[Int, String]]    // None
```

In this example, we've created a wrapper to a collection of `Funky` objects from the last example. Besides a sequence of `Funky` objects, the construction of `FunkyCollection` requires type tags for the type parameters as part of the construction. The type tags are then used to materialize a type tag for the `FunkyCollection` itself which can be used for comparing against other types.

The `cast` method is used for performing a type safe cast based on the relationship of the given type and the type tag stored in the object. If the types match, we can cast the object using `asInstanceOf`.

Unfortunately, it's possible to accidentally get the wrong type tag in your object. If another class extends `FunkyCollection`, the type tag will remain the same in that class. Thus all `hasType` and `cast` comparisons will be made against `FunkyCollection` rather than the type extending `FunkyCollection`. This can be prevented by overriding the `selfTypeTag` to use the type tag for the extending class. However, in doing so, a hidden requirement is given to which ever class extends `FunkyCollection`, thus it increases the potential for introducing new bugs. Therefore, you may want to seal `FunkyCollection` from extensions using `final` keyword to prevent the problem.

We can also create an extractor based on the `cast` method.

```
1  import scala.reflect.runtime.universe._
2
3  object FunkyCollection {
4    def extractor[A: TypeTag, B: TypeTag] = new FunkyExtractor[A, B]
5  }
6
7  class FunkyExtractor[A: TypeTag, B: TypeTag] {
8    def unapply(a: Any): Option[FunkyCollection[A, B]] = a match {
9      case kvs: FunkyCollection[_, _] => kvs.cast[FunkyCollection[A, B]]
10     case _ => None
11   }
```

```
val b: FunkyCollection[_, _] = a

b match {
  case stringIntExt(collection) =>
    // `collection` has type `FunkyCollection[String, Int]`
    ...

  case _ =>
    ...
}
```

In the example, we have a special class, `FunkyExtractor`, that provides the `unapply` method for extracting `FunkyCollection` values. The class is parametrized with type tags, which are used in combination with the `FunkyCollection` type for performing the cast operation on `FunkyCollection` values.

Extracting the boilerplate

The pattern for embedding a type tag and creating a cast method is pretty much the same across all types. Let's extract those features into a trait:

```
trait TypeTaggedTrait[Self] { self: Self =>
  val selfTypeTag: TypeTag[Self]

  def hasType[Other: TypeTag]: Boolean =
    typeOf[Other] =:= selfTypeTag.tpe

  def cast[Other: TypeTag]: Option[Other] =
    if (hasType[Other])
      Some(this.asInstanceOf[Other])
    else
      None
}

abstract class TypeTagged[Self: TypeTag] extends TypeTaggedTrait[Self] {
  val selfTypeTag: TypeTag[Self] = typeTag[Self]
}
```

cast to as the type parameter, the trait requires the trait implementation to extend the type parameter. This is done by adding the type parameter as the self type annotation.

The `selfTypeTag` for the trait can be provided implicitly using an abstract class `TypeTagged`. By extending the `TypeTagged` class, classes can automatically provide the correct type tag through the type parameter.

Now that we have extracted the `cast` method into it's own trait, we can create an extractor class around the trait:

```
1 class TypeTaggedExtractor[T: TypeTag] {
2   def unapply(a: Any): Option[T] = a match {
3     case t: TypeTaggedTrait[_] => t.cast[T]
4     case _ => None
5   }
6 }
```

Like the `FunkyExtractor` in the previous section, `TypeTaggedExtractor` creates an instance of an extractor for the given type. The extractor partially pattern matches on the `TypeTaggedTrait`, and attempts to cast to the given type if it can using the object's `cast` method.

Using the trait and the extractor, we can refactor the `FunkyCollection` to use these generalized features:

```
1 object FunkyCollection {
2   def extractor[A: TypeTag, B: TypeTag] = new TypeTaggedExtractor[FunkyCo
3 }
4
5 final case class FunkyCollection[A: TypeTag, B: TypeTag](funkySeq: Seq[Fu
6   extends TypeTagged[FunkyCollection[A, B]]
```

As mentioned in the previous `FunkyCollection` example, you may want to seal your classes that extend `TypeTagged` or `TypeTaggedTrait` to prevent incorrect type tags from appearing as the `selfTypeTag`.

Casting time relation to input size

Since the casting is based on tags instead of values, the time spent casting should remain roughly the same as input size is grown.

```
}

val coll1: Any = toFunkyCollection((1 to 100).toList)
val coll2: Any = toFunkyCollection((1 to 1000).toList)
val coll3: Any = toFunkyCollection((1 to 10000).toList)
val coll4: Any = toFunkyCollection((1 to 100000).toList)
val coll5: Any = toFunkyCollection((1 to 1000000).toList)
val coll6: Any = toFunkyCollection((1 to 10000000).toList)

time { extractor1.unapply(coll1) } // 1 ms
time { extractor1.unapply(coll2) } // 0 ms
time { extractor1.unapply(coll3) } // 0 ms
time { extractor1.unapply(coll4) } // 0 ms
time { extractor1.unapply(coll5) } // 0 ms
time { extractor1.unapply(coll6) } // 0 ms
```

Here we perform a similar kind of profiling to what we did earlier. The effects of casting can be barely seen at the millisecond scale.

However, it is only fair to point out that type tags may have thread safety or performance issues in multithreaded environments depending on what version of Scala you're using. In Scala 2.10, type tags are not thread safe. The thread safety issues were fixed in Scala 2.11 by introducing locking in critical places of the reflection API. Because the type tags use synchronization internally, the performance of casting using type tags might be much worse than when using Typeable while casting values concurrently.

Conclusions

In this article, I demonstrated a few ways to get around type erasure when doing pattern matching in Scala. I showed two examples of how to get around the whole issue by restructuring code. I also showed how to do type safe casting using Shapeless's Typeable and Scala's type tags.

Refactoring the code to not rely on pattern matching provides the cleanest solution, but it is not always possible. Some of the libraries, such as Akka, provide APIs that force its users to pattern match on the `Any` type. When interacting with a library, the problem can usually be avoided by wrapping types that have type parameters with types that don't have them.

An alternative approach for solving pattern matching on generic types is to use Shapeless's Typeable or Scala's type tags. Typeable along with TypeCase provides an easy to use API for performing type safe casting. Its

Type tags also require thread synchronization while Typeable doesn't.

I'd like to thank Miles Sabin for pointing out issues of using type tags and everyone who participated in reviewing this article. I've uploaded the code examples to Github Gist to play around with. Thanks for reading!

Topics: Scala, Shapeless

Recent Posts

This week in #DevOps (21/03/2017)

Student Hack V

Swagger with Play: All you need to know

This week in #Scala (20/03/2017)

This week in #DevOps (14/03/2017)

Posts by Topic

Scala (353)

Akka (297)

DevOps (143)

Docker (126)

Reactive (123)

see all

Posts by Author

Jan Machacek (249)

Petr Zapletal (137)

Laura Bria (117)

Chris Cundill (67)

Mark Harrison (37)

see all

CAKESOLUTIONS

Subscribe via RSS

Software Innovation DevOps Invest Experience About us Careers Blogs

Manchester Office

Houldsworth Mill
Houldsworth Street
Manchester SK5 6DA
0845 617 1200
enquiries@cakesolutions.net

London Office

CodeNode
10 South Place
London
EC2M 2RB

New York Office

Cake Solutions INC
33 Irving Place, 3rd Floor
New York, NY 10003
+1 347 708 1518
enquiries@cakesolutions.net