Algebraic Data Types

Edit this page on GitHub

The `enum` concept is general enough to also support algebraic data types (ADTs) and their generalized version (GADTs). Here is an example how an `Option` type can be represented as an ADT:

```
enum Option[+T]:
  case Some(x: T)
  case None
```

This example introduces an `Option` enum with a covariant type parameter `T` consisting of two cases, `Some` and `None`. `Some` is parameterized with a value parameter `x`. It is a shorthand for writing a case class that extends `Option`. Since `None` is not parameterized, it is treated as a normal enum value.

The `extends` clauses that were omitted in the example above can also be given explicitly:

```
enum Option[+T]:
  case Some(x: T) extends Option[T]
  case None       extends Option[Nothing]
```

Note that the parent type of the `None` value is inferred as `Option[Nothing]`. Generally, all covariant type parameters of the enum class are minimized in a compiler-generated `extends` clause whereas all contravariant type parameters are maximized. If `Option` was non-variant, you would need to give the extends clause of `None` explicitly.

As for normal enum values, the cases of an `enum` are all defined in the `enum`s companion object. So it's `Option.Some` and `Option.None` unless the definitions are "pulled out" with an import:

```
scala> Option.Some("hello")
val res1: t2.Option[String] = Some(hello)

scala> Option.None
val res2: t2.Option[Nothing] = None
```

Note that the type of the expressions above is always `Option`. Generally, the type of a enum case constructor application will be widened to the underlying enum type, unless a more specific type is expected. This is a subtle difference with respect to normal case classes. The classes making up the cases do exist, and can be unveiled, either by constructing them directly with a `new`, or by explicitly providing an expected type.

```
scala> new Option.Some(2)
val res3: Option.Some[Int] = Some(2)
scala> val x: Option.Some[Int] = Option.Some(3)
val res4: Option.Some[Int] = Some(3)
```

As all other enums, ADTs can define methods. For instance, here is `Option` again, with an `isDefined` method and an `Option(...)` constructor in its companion object.

```
enum Option[+T]:
  case Some(x: T)
  case None

  def isDefined: Boolean = this match
    case None => false
    case _    => true

object Option:

  def apply[T >: Null](x: T): Option[T] =
    if x == null then None else Some(x)

end Option
```

Enumerations and ADTs have been presented as two different concepts. But since they share the same syntactic construct, they can be seen simply as two ends of a spectrum and it is perfectly possible to construct hybrids. For instance, the code below gives an implementation of `Color` either with three enum values or with a parameterized case that takes an RGB value.

```
enum Color(val rgb: Int):
  case Red   extends Color(0xFF0000)
```

```
  case Green extends Color(0x00FF00)
  case Blue  extends Color(0x0000FF)
  case Mix(mix: Int) extends Color(mix)
```

Parameter Variance of Enums

By default, parameterized cases of enums with type parameters will copy the type parameters of their parent, along with any variance notations. As usual, it is important to use type parameters carefully when they are variant, as shown below:

The following `View` enum has a contravariant type parameter `T` and a single case `Refl`, representing a function mapping a type `T` to itself:

```
enum View[-T]:
  case Refl(f: T => T)
```

The definition of `Refl` is incorrect, as it uses contravariant type `T` in the covariant result position of a function type, leading to the following error:

```
-- Error: View.scala:2:12 --------
2 |   case Refl(f: T => T)
  |             ^^^^^^^^^
  |contravariant type T occurs in covariant position in type T => T of value f
  |enum case Refl requires explicit declaration of type T to resolve this issue
```

Because `Refl` does not declare explicit parameters, it looks to the compiler like the following:

```
enum View[-T]:
  case Refl[/*synthetic*/-T1](f: T1 => T1) extends View[T1]
```

The compiler has inferred for `Refl` the contravariant type parameter `T1`, following `T` in `View`. We can now clearly see that `Refl` needs to declare its own non-variant type parameter to correctly type `f`, and can remedy the error by the following change to `Refl`:

```
enum View[-T]:
-  case Refl(f: T => T)
+  case Refl[R](f: R => R) extends View[R]
```

Above, type `R` is chosen as the parameter for `Refl` to highlight that it has a different meaning to type `T` in `View`, but any name will do.

After some further changes, a more complete implementation of `View` can be given as follows and be used as the function type `T => U`:

```
enum View[-T, +U] extends (T => U):
  case Refl[R](f: R => R) extends View[R, R]

  final def apply(t: T): U = this match
    case refl: Refl[r] => refl.f(t)
```

Syntax of Enums

Changes to the syntax fall in two categories: enum definitions and cases inside enums. The changes are specified below as deltas with respect to the Scala syntax given here

1. Enum definitions are defined as follows:

```
TmplDef   ::=  `enum' EnumDef
EnumDef   ::=  id ClassConstr [`extends' [ConstrApps]] EnumBody
EnumBody  ::=  [nl] ‘{’ [SelfType] EnumStat {semi EnumStat} ‘}’
EnumStat  ::=  TemplateStat
            |  {Annotation [nl]} {Modifier} EnumCase
```

2. Cases of enums are defined as follows:

```
EnumCase  ::=  `case' (id ClassConstr [`extends' ConstrApps]] | ids)
```

Reference

For more information, see Issue #1970.

Enume... Transla...

Contributors to this page

Varunram odersky pikinier20 BarkingBad julienrf ShapelessCat michelou bishabosha robstoll TheElectronWill sideeffffect jducoeur dwijnand k0ala OlivierBlanvillain

Scaladoc