Translation of Enums and ADTs

The compiler expands enums and their cases to code that only uses Scala's other language features. As such, enums in Scala are convenient *syntactic sugar*, but they are not essential to understand Scala's core.

We now explain the expansion of enums in detail. First, some terminology and notational conventions:

- We use `E` as a name of an enum, and `C` as a name of a case that appears in `E`.
- We use `< ... >` for syntactic constructs that in some circumstances might be empty. For instance, `<value-params>` represents one or more parameter lists `( ... )` or nothing at all.
- Enum cases fall into three categories:
  - *Class cases* are those cases that are parameterized, either with a type parameter section `[ ... ]` or with one or more (possibly empty) parameter sections `( ... )`.
  - *Simple cases* are cases of a non-generic enum that have neither parameters nor an extends clause or body. That is, they consist of a name only.
  - *Value cases* are all cases that do not have a parameter section but that do have a (possibly generated) `extends` clause and/or a body.

Simple cases and value cases are collectively called *singleton cases*.

The desugaring rules imply that class cases are mapped to case classes, and singleton cases are mapped to `val` definitions.

There are nine desugaring rules. Rule (1) desugars enum definitions. Rules (2) and (3) desugar simple cases. Rules (4) to (6) define `extends` clauses for cases that are missing them. Rules (7) to (9) define how such cases with `extends` clauses map into

`case class`es or `val`s.

1. An `enum` definition

   ```
   enum E ... { <defs> <cases> }
   ```

   expands to a `sealed abstract` class that extends the `scala.reflect.Enum` trait and an associated companion object that contains the defined cases, expanded according to rules (2 - 8). The enum class starts with a compiler-generated import that imports the names `<caseIds>` of all cases so that they can be used without prefix in the class.

   ```
   sealed abstract class E ... extends <parents> with scala.reflect.Enum {
     import E.{ <caseIds> }
     <defs>
   }
   object E { <cases> }
   ```

2. A simple case consisting of a comma-separated list of enum names

   ```
   case C_1, ..., C_n
   ```

   expands to

   ```
   case C_1; ...; case C_n
   ```

   Any modifiers or annotations on the original case extend to all expanded cases.

3. A simple case

   ```
   case C
   ```

   of an enum `E` that does not take type parameters expands to

   ```
   val C = $new(n, "C")
   ```

   Here, `$new` is a private method that creates an instance of `E` (see below).

4. If `E` is an enum with type parameters

   ```
   V1 T1 >: L1 <: U1 ,   ... ,   Vn Tn >: Ln <: Un      (n > 0)
   ```

   where each of the variances `Vi` is either `'+'` or `'-'`, then a simple case

```
case C
```

expands to

```
case C extends E[B1, ..., Bn]
```

where `Bi` is `Li` if `Vi = '+'` and `Ui` if `Vi = '-'`. This result is then further rewritten with rule (8). Simple cases of enums with non-variant type parameters are not permitted (however value cases with explicit `extends` clause are)

5. A class case without an extends clause

```
case C <type-params> <value-params>
```

of an enum `E` that does not take type parameters expands to

```
case C <type-params> <value-params> extends E
```

This result is then further rewritten with rule (9).

6. If `E` is an enum with type parameters `Ts`, a class case with neither type parameters nor an extends clause

```
case C <value-params>
```

expands to

```
case C[Ts] <value-params> extends E[Ts]
```

This result is then further rewritten with rule (9). For class cases that have type parameters themselves, an extends clause needs to be given explicitly.

7. If `E` is an enum with type parameters `Ts`, a class case without type parameters but with an extends clause

```
case C <value-params> extends <parents>
```

expands to

```
case C[Ts] <value-params> extends <parents>
```

provided at least one of the parameters `Ts` is mentioned in a parameter type in `<value-params>` or in a type argument in `<parents>`.

8. A value case

```
case C extends <parents>
```

expands to a value definition in `E`'s companion object:

```
val C = new <parents> { <body>; def ordinal = n }
```

where `n` is the ordinal number of the case in the companion object, starting from 0. The anonymous class also implements the abstract `Product` methods that it inherits from `Enum`.

It is an error if a value case refers to a type parameter of the enclosing `enum` in a type argument of `<parents>`.

9. A class case

```
case C <params> extends <parents>
```

expands analogous to a final case class in `E`'s companion object:

```
final case class C <params> extends <parents>
```

The enum case defines an `ordinal` method of the form

```
def ordinal = n
```

where `n` is the ordinal number of the case in the companion object, starting from 0.

It is an error if a value case refers to a type parameter of the enclosing `enum` in a parameter type in `<params>` or in a type argument of `<parents>`, unless that parameter is already a type parameter of the case, i.e. the parameter name is defined in `<params>`.

The compiler-generated `apply` and `copy` methods of an enum case

```
case C(ps) extends P1, ..., Pn
```

are treated specially. A call `C(...)` of the apply method is ascribed the

are treated specially. A call `C(ts)` of the apply method is ascribed the underlying type `P1 & ... & Pn` (dropping any transparent traits) as long as that type is still compatible with the expected type at the point of application. A call `t.copy(ts)` of `C`'s `copy` method is treated in the same way.

Translation of Enums with Singleton Cases

An enum `E` (possibly generic) that defines one or more singleton cases will define the following additional synthetic members in its companion object (where `E'` denotes `E` with any type parameters replaced by wildcards):

- A method `valueOf(name: String): E'`. It returns the singleton case value whose identifier is `name`.
- A method `values` which returns an `Array[E']` of all singleton case values defined by `E`, in the order of their definitions.

If `E` contains at least one simple case, its companion object will define in addition:

- A private method `$new` which defines a new simple case value with given ordinal number and name. This method can be thought as being defined as follows.

```
private def $new(_$ordinal: Int, $name: String) =
  new E with runtime.EnumValue:
    def ordinal = _$ordinal
    override def productPrefix = $name // if not overridden in `E`
    override def toString = $name      // if not overridden in `E`
```

The anonymous class also implements the abstract `Product` methods that it inherits from `Enum`. The `ordinal` method is only generated if the enum does not extend from `java.lang.Enum` (as Scala enums do not extend `java.lang.Enum`s unless explicitly specified). In case it does, there is no need to generate `ordinal` as `java.lang.Enum` defines it. Similarly there is no need to override `toString` as that is defined in terms of `name` in `java.lang.Enum`. Finally, `productPrefix` will call `this.name` when `E` extends `java.lang.Enum`.

Scopes for Enum Cases

A case in an `enum` is treated similarly to a secondary constructor. It can access neither the enclosing `enum` using `this`, nor its value parameters or instance members using simple identifiers.

Even though translated enum cases are located in the enum's companion object

Even though translated enum cases are located in the enum's companion object, referencing this object or its members via `this` or a simple identifier is also illegal.

The compiler typechecks enum cases in the scope of the enclosing companion object but flags any such illegal accesses as errors.

Translation of Java-compatible enums

A Java-compatible enum is an enum that extends `java.lang.Enum`. The translation rules are the same as above, with the reservations defined in this section.

It is a compile-time error for a Java-compatible enum to have class cases.

Cases such as `case C` expand to a `@static val` as opposed to a `val`. This allows them to be generated as static fields of the enum type, thus ensuring they are represented the same way as Java enums.

Other Rules

- A normal case class which is not produced from an enum case is not allowed to extend `scala.reflect.Enum`. This ensures that the only cases of an enum are the ones that are explicitly declared in it.
- If an enum case has an `extends` clause, the enum class must be one of the classes that's extended.

‹ Algebr... Contex... ›