How to write a type class `derived` method using macros

In the main derivation documentation page, we explained the details behind `Mirror`s and type class derivation. Here we demonstrate how to implement a type class `derived` method using macros only. We follow the same example of deriving `Eq` instances and for simplicity we support a `Product` type e.g., a case class `Person`. The low-level method we will use to implement the `derived` method exploits quotes, splices of both expressions and types and the `scala.quoted.Expr.summon` method which is the equivalent of `summonFrom`. The former is suitable for use in a quote context, used within macros.

As in the original code, the type class definition is the same:

```
trait Eq[T]:
  def eqv(x: T, y: T): Boolean
```

we need to implement a method `Eq.derived` on the companion object of `Eq` that produces a quoted instance for `Eq[T]`. Here is a possible signature,

```
given derived[T: Type](using Quotes): Expr[Eq[T]]
```

and for comparison reasons we give the same signature we had with `inline`:

```
inline given derived[T]: (m: Mirror.Of[T]) => Eq[T] = ???
```

Note, that since a type is used in a subsequent stage it will need to be lifted to a `Type` by using the corresponding context bound. Also, not that we can summon the quoted

`Mirror` inside the body of the `derived` this we can omit it from the signature. The body of the `derived` method is shown below:

```
given derived[T: Type](using Quotes): Expr[Eq[T]] =
  import quotes.reflect.*

  val ev: Expr[Mirror.Of[T]] = Expr.summon[Mirror.Of[T]].get

  ev match
    case '{ $m: Mirror.ProductOf[T] { type MirroredElemTypes = elementTypes }}
      val elemInstances = summonAll[elementTypes]
      val eqProductBody: (Expr[T], Expr[T]) => Expr[Boolean] = (x, y) =>
        elemInstances.zipWithIndex.foldLeft(Expr(true: Boolean)) {
          case (acc, (elem, index)) =>
            val e1 = '{$x.asInstanceOf[Product].productElement(${Expr(index)})}
            val e2 = '{$y.asInstanceOf[Product].productElement(${Expr(index)})}
            '{ $acc && $elem.asInstanceOf[Eq[Any]].eqv($e1, $e2) }
        }

      '{ eqProduct((x: T, y: T) => ${eqProductBody('x, 'y)}) }

  // case for Mirror.ProductOf[T]
  // ...
```

Note, that in the `inline` case we can merely write `summonAll[m.MirroredElemTypes]` inside the inline method but here, since `Expr.summon` is required, we can extract the element types in a macro fashion. Being inside a macro, our first reaction would be to write the code below. Since the path inside the type argument is not stable this cannot be used:

```
'{
  summonAll[$m.MirroredElemTypes]
}
```

Instead we extract the tuple-type for element types using pattern matching over quotes and more specifically of the refined type:

```
case '{ $m: Mirror.ProductOf[T] { type MirroredElemTypes = elementTypes }} =>
```

Shown below is the implementation of `summonAll` as a macro. We assume that given instances for our primitive types exist.

```
def summonAll[T: Type](using Quotes): List[Expr[Eq[_]]] =
   Type.of[T] match
      case '[String *: tpes] => '{ summon[Eq[String]] } :: summonAll[tpes]
```

```
      case '[Int *: tpes]    => '{ summon[Eq[Int]] }    :: summonAll[tpes]
      case '[tpe *: tpes]    => derived[tpe] :: summonAll[tpes]
      case '[EmptyTuple]     => Nil
```

One additional difference with the body of `derived` here as opposed to the one with `inline` is that with macros we need to synthesize the body of the code during the macro-expansion time. That is the rationale behind the `eqProductBody` function. Assuming that we calculate the equality of two `Person` s defined with a case class that holds a name of type `String` and an age of type `Int`, the equality check we want to generate is the following:

```
true
   && Eq[String].eqv(x.productElement(0),y.productElement(0))
   && Eq[Int].eqv(x.productElement(1), y.productElement(1))
```

Calling the derived method inside the macro

Following the rules in Macros we create two methods. One that hosts the top-level splice `eqv` and one that is the implementation. Alternatively and what is shown below is that we can call the `eqv` method directly. The `eqGen` can trigger the derivation.

```
extension [T](inline x: T)
   inline def === (inline y: T)(using eq: Eq[T]): Boolean = eq.eqv(x, y)

inline given eqGen[T]: Eq[T] = ${ Eq.derived[T] }
```

Note, that we use inline method syntax and we can compare instance such as `Sm(Person("Test", 23)) === Sm(Person("Test", 24))` for e.g., the following two types:

```
case class Person(name: String, age: Int)

enum Opt[+T]:
   case Sm(t: T)
   case Nn
```

The full code is shown below:

```
import scala.deriving.*
import scala.quoted.*
```

```
trait Eq[T]:
   def eqv(x: T, y: T): Boolean

object Eq:
   given Eq[String] with
      def eqv(x: String, y: String) = x == y

   given Eq[Int] with
      def eqv(x: Int, y: Int) = x == y

   def eqProduct[T](body: (T, T) => Boolean): Eq[T] =
      new Eq[T]:
         def eqv(x: T, y: T): Boolean = body(x, y)

   def eqSum[T](body: (T, T) => Boolean): Eq[T] =
      new Eq[T]:
         def eqv(x: T, y: T): Boolean = body(x, y)

   def summonAll[T: Type](using Quotes): List[Expr[Eq[_]]] =
      Type.of[T] match
         case '[String *: tpes] => '{ summon[Eq[String]] } :: summonAll[tpes]
         case '[Int *: tpes]    => '{ summon[Eq[Int]] }    :: summonAll[tpes]
         case '[tpe *: tpes]    => derived[tpe] :: summonAll[tpes]
         case '[EmptyTuple]     => Nil

   given derived[T: Type](using q: Quotes): Expr[Eq[T]] =
      import quotes.reflect.*

      val ev: Expr[Mirror.Of[T]] = Expr.summon[Mirror.Of[T]].get

      ev match
         case '{ $m: Mirror.ProductOf[T] { type MirroredElemTypes = elementType
            val elemInstances = summonAll[elementTypes]
            val eqProductBody: (Expr[T], Expr[T]) => Expr[Boolean] = (x, y) =>
               elemInstances.zipWithIndex.foldLeft(Expr(true: Boolean)) {
                  case (acc, (elem, index)) =>
                     val e1 = '{$x.asInstanceOf[Product].productElement(${Expr(
                     val e2 = '{$y.asInstanceOf[Product].productElement(${Expr(

                     '{ $acc && $elem.asInstanceOf[Eq[Any]].eqv($e1, $e2) }
               }
            '{ eqProduct((x: T, y: T) => ${eqProductBody('x, 'y)}) }

         case '{ $m: Mirror.SumOf[T] { type MirroredElemTypes = elementTypes }
            val elemInstances = summonAll[elementTypes]
            val eqSumBody: (Expr[T], Expr[T]) => Expr[Boolean] = (x, y) =>
               val ordx = '{ $m.ordinal($x) }
```

```
                val ordy = '{ $m.ordinal($y) }

                val elements = Expr.ofList(elemInstances)
                '{ $ordx == $ordy && $elements($ordx).asInstanceOf[Eq[Any]].eqv(

          '{ eqSum((x: T, y: T) => ${eqSumBody('x, 'y)}) }
   end derived
end Eq

object Macro3:
   extension [T](inline x: T)
      inline def === (inline y: T)(using eq: Eq[T]): Boolean = eq.eqv(x, y)

   inline given eqGen[T]: Eq[T] = ${ Eq.derived[T] }
```

‹ Type Cl... Multiv... ›

Contributors to this page

 pikinier20  BarkingBad  julienrf  odersky  michelou

 nicolasstucki  ghostdogpr  ysthakur