Type Class Derivation

Type class derivation is a way to automatically generate given instances for type classes which satisfy some simple conditions. A type class in this sense is any trait or class with a type parameter determining the type being operated on. Common examples are `Eq`, `Ordering`, or `Show`. For example, given the following `Tree` algebraic data type (ADT),

```
enum Tree[T] derives Eq, Ordering, Show:
  case Branch(left: Tree[T], right: Tree[T])
  case Leaf(elem: T)
```

The `derives` clause generates the following given instances for the `Eq`, `Ordering` and `Show` type classes in the companion object of `Tree`,

```
given [T: Eq]       : Eq[Tree[T]]    = Eq.derived
given [T: Ordering] : Ordering[Tree] = Ordering.derived
given [T: Show]     : Show[Tree]     = Show.derived
```

We say that `Tree` is the *deriving type* and that the `Eq`, `Ordering` and `Show` instances are *derived instances*.

Types supporting `derives` clauses

All data types can have a `derives` clause. This document focuses primarily on data types which also have a given instance of the `Mirror` type class available. Instances of the `Mirror` type class are generated automatically by the compiler for,

- enums and enum cases
- case classes and case objects
- sealed classes or traits that have only case classes and case objects as children

`Mirror` type class instances provide information at the type level about the components and labelling of the type. They also provide minimal term level infrastructure to allow higher level libraries to provide comprehensive derivation support.

```
sealed trait Mirror:

  /** the type being mirrored */
  type MirroredType

  /** the type of the elements of the mirrored type */
  type MirroredElemTypes

  /** The mirrored *-type */
  type MirroredMonoType

  /** The name of the type */
  type MirroredLabel <: String

  /** The names of the elements of the type */
  type MirroredElemLabels <: Tuple

object Mirror:

  /** The Mirror for a product type */
  trait Product extends Mirror:

    /** Create a new instance of type `T` with elements      *  taken from proc
     */
    def fromProduct(p: scala.Product): MirroredMonoType

  trait Sum extends Mirror:

    /** The ordinal number of the case class of `x`.      *  For enums, `ordina
     */
    def ordinal(x: MirroredMonoType): Int

end Mirror
```

Product types (i.e. case classes and objects, and enum cases) have mirrors which are subtypes of `Mirror.Product`. Sum types (i.e. sealed class or traits with product children, and enums) have mirrors which are subtypes of `Mirror.Sum`.

For the `Tree` ADT from above the following `Mirror` instances will be automatically provided by the compiler,

```
// Mirror for Tree
new Mirror.Sum:
  type MirroredType = Tree
  type MirroredElemTypes[T] = (Branch[T], Leaf[T])
  type MirroredMonoType = Tree[_]
  type MirroredLabel = "Tree"
  type MirroredElemLabels = ("Branch", "Leaf")

  def ordinal(x: MirroredMonoType): Int = x match
    case _: Branch[_] => 0
    case _: Leaf[_] => 1

// Mirror for Branch
new Mirror.Product:
  type MirroredType = Branch
  type MirroredElemTypes[T] = (Tree[T], Tree[T])
  type MirroredMonoType = Branch[_]
  type MirroredLabel = "Branch"
  type MirroredElemLabels = ("left", "right")

  def fromProduct(p: Product): MirroredMonoType =
    new Branch(...)

// Mirror for Leaf
new Mirror.Product:
  type MirroredType = Leaf
  type MirroredElemTypes[T] = Tuple1[T]
  type MirroredMonoType = Leaf[_]
  type MirroredLabel = "Leaf"
  type MirroredElemLabels = Tuple1["elem"]

  def fromProduct(p: Product): MirroredMonoType =
    new Leaf(...)
```

Note the following properties of `Mirror` types,

- Properties are encoded using types rather than terms. This means that they have no runtime footprint unless used and also that they are a compile time feature for use with Scala 3's metaprogramming facilities.
- The kinds of `MirroredType` and `MirroredElemTypes` match the kind of the data type the mirror is an instance for. This allows `Mirror`s to support ADTs of all kinds.
- There is no distinct representation type for sums or products (ie. there is no `HList` or `Coproduct` type as in Scala 2 versions of Shapeless). Instead the collection of child types of a data type is represented by an ordinary, possibly parameterized, tuple type. Scala 3's metaprogramming facilities can be used to

work with these tuple types as-is, and higher level libraries can be built on top of them.
- For both product and sum types, the elements of `MirroredElemTypes` are arranged in definition order (i.e. `Branch[T]` precedes `Leaf[T]` in `MirroredElemTypes` for `Tree` because `Branch` is defined before `Leaf` in the source file). This means that `Mirror.Sum` differs in this respect from Shapeless's generic representation for ADTs in Scala 2, where the constructors are ordered alphabetically by name.
- The methods `ordinal` and `fromProduct` are defined in terms of `MirroredMonoType` which is the type of kind-`*` which is obtained from `MirroredType` by wildcarding its type parameters.

Type classes supporting automatic deriving

A trait or class can appear in a `derives` clause if its companion object defines a method named `derived`. The signature and implementation of a `derived` method for a type class `TC[_]` are arbitrary but it is typically of the following form,

```
import scala.deriving.Mirror

def derived[T](using Mirror.Of[T]): TC[T] = ...
```

That is, the `derived` method takes a context parameter of (some subtype of) type `Mirror` which defines the shape of the deriving type `T`, and computes the type class implementation according to that shape. This is all that the provider of an ADT with a `derives` clause has to know about the derivation of a type class instance.

Note that `derived` methods may have context `Mirror` parameters indirectly (e.g. by having a context argument which in turn has a context `Mirror` parameter, or not at all (e.g. they might use some completely different user-provided mechanism, for instance using Scala 3 macros or runtime reflection). We expect that (direct or indirect) `Mirror` based implementations will be the most common and that is what this document emphasises.

Type class authors will most likely use higher level derivation or generic programming libraries to implement `derived` methods. An example of how a `derived` method might be implemented using *only* the low level facilities described above and Scala 3's general metaprogramming features is provided below. It is not anticipated that type class authors would normally implement a `derived` method in this way, however this walkthrough can be taken as a guide for authors of the higher level derivation

libraries that we expect typical type class authors will use (for a fully worked out example of such a library, see Shapeless 3).

How to write a type class `derived` method using low level mechanisms

The low-level method we will use to implement a type class `derived` method in this example exploits three new type-level constructs in Scala 3: inline methods, inline matches, and implicit searches via `summonInline` or `summonFrom`. Given this definition of the `Eq` type class,

```
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 given instance for `Eq[T]` given a `Mirror[T]`. Here is a possible implementation,

```
import scala.deriving.Mirror

inline given derived[T](using m: Mirror.Of[T]): Eq[T] =
  val elemInstances = summonAll[m.MirroredElemTypes]           // (1)
  inline m match                                               // (2)
    case s: Mirror.SumOf[T]     => eqSum(s, elemInstances)
    case p: Mirror.ProductOf[T] => eqProduct(p, elemInstances)
```

Note that `derived` is defined as an `inline` given. This means that the method will be expanded at call sites (for instance the compiler generated instance definitions in the companion objects of ADTs which have a `derived Eq` clause), and also that it can be used recursively if necessary, to compute instances for children.

The body of this method (1) first materializes the `Eq` instances for all the child types of type the instance is being derived for. This is either all the branches of a sum type or all the fields of a product type. The implementation of `summonAll` is `inline` and uses Scala 3's `summonInline` construct to collect the instances as a `List`,

```
inline def summonAll[T <: Tuple]: List[Eq[_]] =
  inline erasedValue[T] match
    case _: EmptyTuple => Nil
    case _: (t *: ts) => summonInline[Eq[t]] :: summonAll[ts]
```

with the instances for children in hand the `derived` method uses an `inline match` to dispatch to methods which can construct instances for either sums or products (2).

Note that because `derived` is `inline` the match will be resolved at compile-time and only the left-hand side of the matching case will be inlined into the generated code with types refined as revealed by the match.

In the sum case, `eqSum`, we use the runtime `ordinal` values of the arguments to `eqv` to first check if the two values are of the same subtype of the ADT (3) and then, if they are, to further test for equality based on the `Eq` instance for the appropriate ADT subtype using the auxiliary method `check` (4).

```
import scala.deriving.Mirror

def eqSum[T](s: Mirror.SumOf[T], elems: List[Eq[_]]): Eq[T] =
  new Eq[T]:
    def eqv(x: T, y: T): Boolean =
      val ordx = s.ordinal(x)                          // (3)
      (s.ordinal(y) == ordx) && check(elems(ordx))(x, y) // (4)
```

In the product case, `eqProduct` we test the runtime values of the arguments to `eqv` for equality as products based on the `Eq` instances for the fields of the data type (5),

```
import scala.deriving.Mirror

def eqProduct[T](p: Mirror.ProductOf[T], elems: List[Eq[_]]): Eq[T] =
  new Eq[T]:
    def eqv(x: T, y: T): Boolean =
      iterator(x).zip(iterator(y)).zip(elems.iterator).forall {  // (5)
        case ((x, y), elem) => check(elem)(x, y)
      }
```

Pulling this all together we have the following complete implementation,

```
import scala.deriving.*
import scala.compiletime.{erasedValue, summonInline}

inline def summonAll[T <: Tuple]: List[Eq[_]] =
  inline erasedValue[T] match
    case _: EmptyTuple => Nil
    case _: (t *: ts) => summonInline[Eq[t]] :: summonAll[ts]

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

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

```
  def check(elem: Eq[_])(x: Any, y: Any): Boolean =
    elem.asInstanceOf[Eq[Any]].eqv(x, y)

  def iterator[T](p: T) = p.asInstanceOf[Product].productIterator

  def eqSum[T](s: Mirror.SumOf[T], elems: => List[Eq[_]]): Eq[T] =
    new Eq[T]:
      def eqv(x: T, y: T): Boolean =
        val ordx = s.ordinal(x)
        (s.ordinal(y) == ordx) && check(elems(ordx))(x, y)

  def eqProduct[T](p: Mirror.ProductOf[T], elems: => List[Eq[_]]): Eq[T] =
    new Eq[T]:
      def eqv(x: T, y: T): Boolean =
        iterator(x).zip(iterator(y)).zip(elems.iterator).forall {
          case ((x, y), elem) => check(elem)(x, y)
        }

  inline given derived[T](using m: Mirror.Of[T]): Eq[T] =
    lazy val elemInstances = summonAll[m.MirroredElemTypes]
    inline m match
      case s: Mirror.SumOf[T]     => eqSum(s, elemInstances)
      case p: Mirror.ProductOf[T] => eqProduct(p, elemInstances)
end Eq
```

we can test this relative to a simple ADT like so,

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

@main def test(): Unit =
  import Opt.*
  val eqoi = summon[Eq[Opt[Int]]]
  assert(eqoi.eqv(Sm(23), Sm(23)))
  assert(!eqoi.eqv(Sm(23), Sm(13)))
  assert(!eqoi.eqv(Sm(23), Nn))
```

In this case the code that is generated by the inline expansion for the derived `Eq` instance for `Opt` looks like the following, after a little polishing,

```
given derived$Eq[T](using eqT: Eq[T]): Eq[Opt[T]] =
  eqSum(
    summon[Mirror[Opt[T]]],
    List(
      eqProduct(summon[Mirror[Sm[T]]], List(summon[Eq[T]])),
      eqProduct(summon[Mirror[Nn.type]], Nil)
```

```
    )
  )
```

Alternative approaches can be taken to the way that `derived` methods can be defined. For example, more aggressively inlined variants using Scala 3 macros, whilst being more involved for type class authors to write than the example above, can produce code for type classes like `Eq` which eliminate all the abstraction artefacts (eg. the `Lists` of child instances in the above) and generate code which is indistinguishable from what a programmer might write by hand. As a third example, using a higher level library such as Shapeless the type class author could define an equivalent `derived` method as,

```
given eqSum[A](using inst: => K0.CoproductInstances[Eq, A]): Eq[A] with
  def eqv(x: A, y: A): Boolean = inst.fold2(x, y)(false)(
    [t] => (eqt: Eq[t], t0: t, t1: t) => eqt.eqv(t0, t1)
  )

given eqProduct[A](using inst: K0.ProductInstances[Eq, A]): Eq[A] with
  def eqv(x: A, y: A): Boolean = inst.foldLeft2(x, y)(true: Boolean)(
    [t] => (acc: Boolean, eqt: Eq[t], t0: t, t1: t) =>
      Complete(!eqt.eqv(t0, t1))(false)(true)
  )

inline def derived[A](using gen: K0.Generic[A]): Eq[A] =
  gen.derive(eqProduct, eqSum)
```

The framework described here enables all three of these approaches without mandating any of them.

For a brief discussion on how to use macros to write a type class `derived` method please read more at How to write a type class `derived` method using macros.

Deriving instances elsewhere

Sometimes one would like to derive a type class instance for an ADT after the ADT is defined, without being able to change the code of the ADT itself. To do this, simply define an instance using the `derived` method of the type class as right-hand side. E.g, to implement `Ordering` for `Option` define,

```
given [T: Ordering]: Ordering[Option[T]] = Ordering.derived
```

Assuming the `Ordering.derived` method has a context parameter of type `Mirror[T]` it will be satisfied by the compiler generated `Mirror` instance for

`Option` and the derivation of the instance will be expanded on the right hand side of this definition in the same way as an instance defined in ADT companion objects.

Syntax

```
Template          ::=  InheritClauses [TemplateBody]
EnumDef           ::=  id ClassConstr InheritClauses EnumBody
InheritClauses    ::=  [‘extends’ ConstrApps] [‘derives’ QualId {‘,’ QualId}]
ConstrApps        ::=  ConstrApp {‘with’ ConstrApp}
                     |  ConstrApp {‘,’ ConstrApp}
```

Note: To align `extends` clauses and `derives` clauses, Scala 3 also allows multiple extended types to be separated by commas. So the following is now legal:

```
class A extends B, C { ... }
```

It is equivalent to the old form

```
class A extends B with C { ... }
```

Discussion

This type class derivation framework is intentionally very small and low-level. There are essentially two pieces of infrastructure in compiler-generated `Mirror` instances,

- type members encoding properties of the mirrored types.
- a minimal value level mechanism for working generically with terms of the mirrored types.

The `Mirror` infrastructure can be seen as an extension of the existing `Product` infrastructure for case classes: typically `Mirror` types will be implemented by the ADTs companion object, hence the type members and the `ordinal` or `fromProduct` methods will be members of that object. The primary motivation for this design decision, and the decision to encode properties via types rather than terms was to keep the bytecode and runtime footprint of the feature small enough to make it possible to provide `Mirror` instances *unconditionally*.

Whilst `Mirrors` encode properties precisely via type members, the value level `ordinal` and `fromProduct` are somewhat weakly typed (because they are defined in terms of `MirroredMonoType`) just like the members of `Product`. This means that code for generic type classes has to ensure that type exploration and value selection

proceed in lockstep and it has to assert this conformance in some places using casts. If generic type classes are correctly written these casts will never fail.

As mentioned, however, the compiler-provided mechanism is intentionally very low level and it is anticipated that higher level type class derivation and generic programming libraries will build on this and Scala 3's other metaprogramming facilities to hide these low-level details from type class authors and general users. Type class derivation in the style of both Shapeless and Magnolia are possible (a prototype of Shapeless 3, which combines aspects of both Shapeless 2 and Magnolia has been developed alongside this language feature) as is a more aggressively inlined style, supported by Scala 3's new quote/splice macro and inlining facilities.

‹ Imple... How to... ›

Contributors to this page

 pikinier20  BarkingBad  julienrf  KazuyaMiyashita

 IvannKurchenko  kubukoz  odersky  ShapelessCat

 michelou  ayushworks  b-studios  liufengyun  ysthakur

 szymon-rd