Dérivation générique de *Type Classes*

AVEC SHAPELESS

...présentée par

Thomas Dufour

Scala, Elm @ EPITECH Lille

 chwthewke

 chwthewke

 thomas_d sur lillefp.slack.com

Motivation

DÉRIVATION GÉNÉRIQUE

DÉRIVATION AUTOMATIQUE

Type classes vs boilerplate

Les type classes sont un mécanisme d'abstraction puissant

Mais écrire des instances est répétitif

En particulier les type classes concernant les types de données (* -> *)

Eq, Show, Order, Monoid, Semigroup, circe.Decoder/Encoder, etc.

Comment peut-on simplifier l'écriture de ces instances ?

Peut-on (et veut-on) les produire automatiquement ?

Programmation générique

Generic programming is the kinds of polymorphism your language does not support yet.

Miles Sabin

La programmation générique, ce sont les formes de polymorphisme qui ne sont pas encore permises directement par le langage.

À qui s’adresse la dérivation générique ?

Aux auteurs de bibliothèque qui fournissent des *type classes*

À ceux qui veulent comprendre comment ça fonctionne dans

- Circe
- Doobie
- Pureconfig
- Decline
- Monocle
- Parboiled
- Scodec

Aux utilisateurs de type classes fournies sans dérivation générique

Exemple d'application : auto-diff

```scala
case class Address( street: String, city: String )
case class Person( name: String, age: Int, address: Address )

import fr.thomasdufour.autodiff._, Difference._, generic.auto._

implicitly[Diff[Person]]
  .apply( Person( "Jean Martin", 29, Address( "2 rue Pasteur", "Lille" ) ),
          Person( "Jean Martin", 55, Address( "2 rue Pasteur", "Lyon" ) ) )
  .foreach( r => println( Pretty.Colorized2.show( r ) ) )
```

```
in Person
- age
  - 29 -> 55
- address
  - in Address
    - city
      - Lille -> Lyon
```

La *type class* Diff

```scala
trait Diff[A] {
  def apply( left: A, right: A ): Option[Difference]
}

sealed trait Difference

object Difference {

  final case class Value( left: String, right: String )                  extends Difference
  final case class Coproduct( name: String, difference: Difference )     extends Difference
  final case class Product( name: String, fields: NonEmptyList[Field] )  extends Difference
  final case class Tuple( name: String, fields: NonEmptyList[Index] )    extends Difference
  final case class Seq( name: String, diffs: NonEmptyList[Index] )       extends Difference

  final case class Field( name: String, difference: Difference )
  final case class Index( index: Int, difference: Difference )
}
```

Pour quels types générer des instances ?

Autant qu'on pourra

Mais sans s'interdire de varier les approches

Pour les types primitifs ?

- Instances ad hoc (n ≈ 10^1)

Pour les collections, et les amis Option, Either ?

- Dérivation ad hoc
  - `implicit def collectionInstance[A](implicit eltInstance: TC[A]): TC[Coll[A]]`
- L'approche générique est également possible (pour certains types)

Pour les tuples (de 1 à 22) ?

- Euuuuuuh… codegen ? ¯_(ツ)_/¯ (méthodes manuelle ou générique également possibles)

Pour les ADTs (case classes, types "union") ?

- Dérivation générique

Shapeless : un aperçu

PROGRAMMATION GÉNÉRIQUE EN SCALA

HList : listes hétérogènes

```
type T = Int :: String :: HNil // T <: HList

type T = ::[Int, ::[String, HNil]]


val t: T = 1 :: "abc" :: HNil

// val s: T = 1 :: 2 :: "abc" :: HNil
// [error]  found   : Int :: Int :: String :: shapeless.HNil
// [error]  required: T
// [error]     (which expands to)  Int :: String :: shapeless.HNil
// [error]   val s: T = 1 :: 2 :: "abc" :: HNil
```

Dérivation de Diff pour HList

```scala
implicit val hnilDiff: Diff[HNil] =
  new Diff[HNil] {
    override def apply( left: HNil, right: HNil ): Option[Difference] = None
  }

implicit def hconsDiff[H, T <: HList]( implicit diffHead: Diff[H], diffTail: Diff[T]
    ): Diff[H :: T] =
  new Diff[H :: T] {
    def combineDifferences( h: Option[Difference], t: Option[Difference]
        ): Option[Difference] = ???

    override def apply( left: H :: T, right: H :: T ): Option[Difference] = {
      val headDiff = diffHead.apply( left.head, right.head )
      val tailDiff = diffTail.apply( left.tail, right.tail )
      combineDifferences( headDiff, tailDiff )
    }
  }
```

Dérivation de Diff pour HList (2)

```scala
implicit val hnilDiff: Diff[HNil] = ???

implicit def hconsDiff[H, T <: HList]( implicit diffHead: Diff[H], diffTail: Diff[T]
): Diff[H :: T] = ???

implicit def diffInt: Diff[Int]       = ???
implicit def diffString: Diff[String] = ???

implicitly[Diff[Int :: String :: HNil]]
```

Et alors ? Et alors Generic !

```scala
trait Generic[A] {
  type Repr

  def to( a: A ): Repr
  def from( r: Repr ): A
}
```

Generic (2)

```scala
trait Color
case class Person( name: String, age: Int, favourite: Option[Color] )

val g = Generic[Person] // ~= implicitly[Generic[Person]]
// g: Generic[Person]{type Repr = String :: Int :: Option[Color] :: HNil} = …@635b6454
// La représentation de `Person` est
//     String :: Int :: Option[Color] :: Hnil

//  scala> g.to( Person("Thomas", 37, None) )
//  res0: g.Repr = Thomas :: 37 :: None :: HNil
//
//  scala> g.from(res0)
//  res1: Person = Person(Thomas,37,None)
```

Dérivation avec Generic

```scala
implicit def diffGeneric[A](
    implicit g: Generic[A], d: Diff[g.Repr] ): Diff[A] =
  new Diff[A] {
    override def apply( left: A, right: A ): Option[Difference] =
      d.apply( g.to( left ), g.to( right ) )
  }
```

Dérivation avec Generic (fixed)

```
implicit def diffGeneric[A, R <: HList](
    implicit g: Generic[A] { type Repr = R }, d: Diff[R] ): Diff[A] =
  new Diff[A] {
    override def apply( left: A, right: A ): Option[Difference] =
      d.apply( g.to( left ), g.to( right ) )
  }
```

Dérivation avec Generic (fixed)

```scala
implicit def diffGeneric[A, R <: HList](
    implicit g: Generic.Aux[A, R], d: Diff[R] ): Diff[A] =
  new Diff[A] {
    override def apply( left: A, right: A ): Option[Difference] =
      d.apply( g.to( left ), g.to( right ) )
  }
```

Checkpoint

Étant donné une case class CC qui contient

- des types simples
- des collections, tuples
- ou des case classes
- etc, récursivement

On pourra dériver une représentation en HList avec Generic[CC]

On pourra (par induction) dériver des instances de Diff pour les types de cette HList

- Et donc pour ce type de HList lui-même

Ce qui nous fournit l'instance pour le type CC, en combinant avec l'instance de Generic

Coproduct

```
// Rappel HList
type T = Int :: String :: HNil // <: HList
val t: T = 1 :: "abc" :: Hnil

type S = Int :+: String :+: CNil // <: Coproduct
val s1: S = Inl( 1 )
val s2: S = Inr( Inl( "abc" ) )

// val s3: S = Inl( "abc" )
// <console>:30: error: type mismatch;
//  found   : String("abc")
//  required: Int
//        val s3: S = Inl( "abc" )
```

Generic pour Coproduct

```scala
sealed trait Color
final case class ByName( name: String )                          extends Color
final case class CMYK( c: Float, m: Float, y: Float, k: Float ) extends Color
final case class RGBA( r: Float, g: Float, b: Float, a: Float ) extends Color

val g = Generic[Color]
// g: Generic[Color]{
//   type Repr = ByName :+: CMYK :+: RGBA :+: Cnil
// } = anon$macro$2$1@1f8db9e3
// La représentation de `Color` est :
//   ByName :+: CMYK :+: RGBA :+: CNil
```

Dérivation de Diff pour Coproduct

```scala
implicit def cnilDiff: Diff[CNil] = new Diff[CNil] {
  override def apply( left: CNil, right: CNil ): Option[Difference] = left.impossible
}

implicit def cconsDiff[H, T <: Coproduct](
    implicit diffHead: Diff[H], diffTail: Diff[T] ): Diff[H :+: T] =
  new Diff[H :+: T] {
    override def apply( left: H :+: T, right: H :+: T ): Option[Difference] =
      ( left, right ) match {
        case ( Inl( h ), Inl( i ) ) => diffHead( h, i )
        case ( Inr( t ), Inr( s ) ) => diffTail( t, s )
        case _                      => Some( differentTypes(left, right) )
      }

    def differentTypes(left: H :+: T, right: H :+: T): Difference = ???
  }
```

Dérivation pour les types “union”

```
implicit def sealedTraitDiff[T, C <: Coproduct](
    implicit g: Generic.Aux[T, C], diff: Diff[C] ): Diff[T] =
  new Diff[T] {
    override def apply( left:T, right:T ): Option[Difference] =
      diff.apply( g.to( left ), g.to( right ) )
  }
```

Récapitulatif

Les mécanismes de Scala (résolution récursive d'implicits) nous permettent de dériver des *type classes* pour des types génériques

- Collections
- Tuples
- HList
- Coproduct

Shapeless nous donne une représentation de nos types "shape-full" (case classes et types union) en termes de HList et Coproduct

…

Profit!

Limitations

Temps de compilation pour les “grands” ADT

- -Yinduction-heuristics avec Typelevel Scala 2.12, ou Scala 2.13

Non-portabilité (macros)

Priorité des implicits

- Contournable avec une hiérarchie de traits
  - Ou bien des macros
- PR “import implicit”

Récursion non-terminale

```scala
sealed trait MyList[+A]
case object MyNil                                  extends MyList[Nothing]
case class MyCons[+A]( head: A, tail: MyList[A] ) extends MyList[A]

implicit val diffInt: Diff[Int] = ???

implicit val diffHNil: Diff[HNil]                                                  = ???
implicit def diffHCons[H, T <: HList]( implicit H: Diff[H], T: Diff[T] ): Diff[H :: T]  = ???
implicit val diffCNil: Diff[CNil]                                                  = ???
implicit def diffCCons[H, T <: Coproduct]( implicit H: Diff[H], T: Diff[T] ): Diff[H :+: T] = ???
implicit def diffGen[A, R]( implicit G: Generic.Aux[A, R], D: Diff[R] ): Diff[A]   = ???

implicitly[Diff[MyList[Int]]]
// [error] diverging implicit expansion for type Diff[MyList[Int]]
// [error] starting with method diffGen
```

Le problème avec MyList

MyList[Int]

(via Generic)

MyNil :+: MyCons[Int] :+: Cnil

(via Generic, pour MyCons)

Int :: MyList[Int] :: HNil

Lazy

```
sealed trait MyList[+A]
case object MyNil                                  extends MyList[Nothing]
case class MyCons[+A]( head: A, tail: MyList[A] ) extends MyList[A]

implicit val diffInt: Diff[Int] = ???

implicit val diffHNil: Diff[HNil]                                                  = ???
implicit def diffHCons[H, T <: HList]( implicit H: Diff[H], T: Diff[T] ): Diff[H :: T]      = ???
implicit val diffCNil: Diff[CNil]                                                  = ???
implicit def diffCCons[H, T <: Coproduct]( implicit H: Diff[H], T: Diff[T] ): Diff[H :+: T] = ???
implicit def diffGen[A, R]( implicit G: Generic.Aux[A, R], D: Lazy[Diff[R]] ): Diff[A]      = ???

implicitly[Diff[MyList[Int]]]
```

Poursuivre avec auto-diff ;-)

1. S’armer de “The Type Astronaut’s Guide to Shapeless”
2. Explorer le code source https://github.com/chwthewke/auto-diff

Features :

- Conception modulaire
- Dérivation générique optionnelle, automatique ou semi-automatique
- Utilisation de shapeless.LabelledGeneric pour observer les noms
- Instances pour des conteneurs, y compris Set et Map
- Diffs “customisables”

TODOs :

- 0.2.1 avec quelques tweaks
- Un meilleur README ☹

Références

The Type Astronaut's Guide to Shapeless

Shapeless is Dead, Long Live Shapeless

CoRecursive podcast w/ Miles Sabin

Generic Derivation: the Hard Parts – Travis Brown

circe/circe pour le "pattern" semiauto

propensive/magnolia (alternative à shapeless)

Ces slides à https://github.com/chwthewke/lillefp10