Typeclass Introduction

型クラスの目的

- クラスの継承とかと同じ様に多相性を実現すること
- つまりDogとかCatとかをAnimalとしてみなせるかみたいな
- ただし実現できる多相性の種類がクラスの継承とかと異なる

言葉としての型クラス

- そもそも「型クラス=Typeclass」という名前は何なのか
- 最初に聞いたときの思ひで
  - 「型」と「クラス」って同じじゃないの?
- なんで同じ言葉並べてるんや?
  - 「型」と「クラス」は型クラスの文脈では明確に違う意味を持つ
- 型: いわゆるデータ型、Int、String、Dogとか
- クラス: 型を分類するもの
- つまり型クラスとはデータ型を何らかの特徴に基づいて分類し、その分類に基づいて多態性を実現する機能

Scalaにおける型クラスの実装

型クラスの実態

- Scalaではtraitを使用して型クラスを実現する
- 題材としてcirceのEncoderを考える

```
// 説明に必要ない部分は一旦省いた
trait Encoder[A] extends Serializable {
  self =>
  /**
   * Convert a value to JSON.
   */
  def apply(a: A): Json
}
```

- ここで定義した Encoder trait自体を型クラスと呼ぶ
- この型クラスが言いたいのは何らかの型 AがあってそれがcirceのJsonにできるということ
- つまり型 Aになにか新しい操作を定義しているような感じ

型クラスのインスタンス

- 実際にEncoder型クラスを使って具体的な型にencode能力を与えるためには各々型への実装を用意する

```
val intEncoder = new Encoder[Int] {
  override def apply(a: Int) = Json.fromInt(a)
}

val stringEncoder = new Encoder[String] {
  override def apply(a: String) = Json.fromString(a)
}
```

- この様に特定の型に対しての型クラスの実装を「型クラスのインスタンス」と呼ぶ
- 型クラスが型を分類するためのものだったのに対して、特定の型に対して型クラスが実装されたので抽象度が下がっているためこう呼べる

多相性の整理

- そもそも多相性とは違うものを同じものとみなしてプログラムを構造化してプログラミングコストを下げるためのテクニック
- 多相性とは言っても種類がある

パラメトリック多相

- いわゆるジェネリクスだと思っておけばいい

```
List(1, 2, 3)
List("a", "b", "c")
// val res0: List[Int] = List(1, 2, 3)
// val res1: List[String] = List(a, b, c)
```

サブタイピング

- Javaでいうクラスの継承
- tsみたいなstructural subtypingだと構造として同じかどうか（多分これはサブタイピングとみなしていい）
- Javaのオブジェクト指向が言ってる多相性はこれのこと

```
trait Figure
case class Circle(radius: Double) extends Figure
case class Rectangle(width: Double, height: Double) extends Figure

val fig1: Figure = Circle(1.0)
val fig2: Figure = Rectangle(1.0, 2.0)
```

アドホック多相

- 処理対象のデータ型によって処理内容を変更する多相性のこと
- 関数のオーバーロードとかこれ
- 型クラスが実現する多相性もこれ

```
object Concat {
  def concat(a: String, b: String) = a + b
  def concat(a: Int, b: Int) = a + b
}

Concat.concat("a", "b") // => ab
Concat.concat(1, 2) // => 3
```

型クラスを使用したアドホック多相

- 型クラスとそのインスタンス、多相性について整理できたので実際にアドホック多相を実現する

インスタンスの定義

```
import io.circe.{Encoder, Json}

implicit val intEncoder: Encoder[Int] = new Encoder[Int] {
  override def apply(a: Int) = Json.fromInt(a)
}

implicit val stringEncoder: Encoder[String] = new Encoder[String] {
  override def apply(a: String) = Json.fromString(a)
}

case class Dog(name: String, age: Int)

implicit val dogEncoder = new Encoder[Dog] {
  override def apply(a: Dog) = Json.obj(
    ("name", Json.fromString(a.name)),
    ("age", Json.fromInt(a.age))
  )
}
```

アドホック多相な関数の作成

```
def toJson[T](a: T)(implicit encoder: Encoder[T]): Json = encoder(a)

// 処理対象のデータ型によって型クラスのインスタンスが解決されて異なる処理が実行されている
toJson(1)
// val res0: io.circe.Json = 1

toJson("hello")
// val res1: io.circe.Json = "hello"

toJson(Dog("john", 3))
// val res2: io.circe.Json =
// {
//   "name" : "john",
//   "age" : 3
// }

case class Cat(name: String, age: Int)

// 型クラスのインスタンスが定義されていないものはコンパイルが通らない
toJson(Cat("Garfield", 38))
// could not find implicit value for parameter encoder: io.circe.Encoder[Cat]
```

- この様に toJson という関数が処理対象のデータ型に対して多相的に振る舞う

型クラスとサブタイピング

- 型クラスでアドホック多相ができることはわかったのだがこれがなぜ必要なのかがわからない
- クラス継承を用いたサブタイピングでも同じようなことができそうな気がする
- 型クラスが解決するサブタイピングの課題をいくつか紹介する

前提としてのサブタイピングを用いた仮実装

```
// 仮にこういう抽象traitが用意されていて
trait JsonEncodable {
  def toJson: Json
}

case class EncodableDog(name: String, age: Int) extends JsonEncodable {
  override def toJson: Json = Json.obj(
    ("name", Json.fromString(name)),
    ("age", Json.fromInt(age))
  )

}

case class Person(name: String) extends JsonEncodable {
  override def toJson: Json = Json.obj(
    ("name", Json.fromString(name))
  )
}

def encodeToJson(a: JsonEncodable): Json = a.toJson

encodeToJson(EncodableDog("john", 3))
encodeToJson(Person("taro"))
```

ライブラリコードに対しての抽象化

- 前述の例だとうまい具合にサブタイピングで同じようなことができているような気がする
- しかしユーザコードではないライブラリコードに対して抽象化をかまそうとするとうまく行かない
- なぜなら継承はデータ型の定義時点でしか行うことができないから
- 例えば Int に対して JsonEncodable継承はできないので Int を encodeToJson関数に突っ込むのは不可能
  - ラッパークラスを作成したりして回避することはできるけど。。。

```
// IntをJsonEncodable化する手段はない
encodeToJson(1) // compile Error!
```

- つまり型クラスのメリットとして「後付で」抽象化機構を入れ込むことができる

Implicit Derivation 1

- ジェネリックなフィールドを持つデータ型を考える
- このデータ型は `name` フィールドがジェネリックになっていて好きな型を入れられる

```
case class GenericNameSB[T](name: T, age: Int)

GenericNameSB(1, 1)

case clas Name(value: String)

GenericNameSB(Name("john"), 2)
```

Implicit Derivation 2

- `GenericNameSB`クラスを継承を使用して`JsonEncodable`化しようとすると型パラメタ`T`に対して制約をかける必要が発生する

```
// nameをjson化するためにTにはJsonEncodableを継承しているという上限境界が必要になる
case class GenericNameSB[T <: JsonEncodable](name: T, age: Int) extends JsonEncodable {
  def toJson: Json = Json.obj(
    "name" -> name.toJson, // ここでtoJsonを呼ばないとnameをJson化する方法がわからない
    "age" -> Json.fromInt(age)
  )
}
```

- しかしこれはあんまりうれしくなくて、`GenericNameSB`には`Int`などの`JsonEncodable`にしていないorできない型を入れることができなくなってしまう

```
GenericNameSB(1, 2)
// inferred type arguments [Int] do not conform to method apply's type parameter bounds [T <: JsonEncodable]
```

- これがサブタイピングで発生する限界
- すなわちあるデータ型に何らかのtraitを継承させるとき、場合によっては中のデータ型もそのtraitを継承する必要が発生する

Implicit Derivation 3

- 型クラスを使用すると前述の課題が解決できる

```
case class GenericNameTC[T](name: T, age: Int)

implicit def genericNameTCEncode[T](implicit encoder: Encoder[T]): Encoder[GenericNameTC[T]] =
  new Encoder[GenericNameTC[T]] {
    override def apply(a: GenericNameTC[T]): Json = Json.obj(
      "name" -> encoder(a.name),
      "age"  -> Json.fromInt(a.age)
    )
  }

toJson(GenericNameTC("hoge", 1))
// val res0: io.circe.Json =
// {
//   "name" : "hoge",
//   "age" : 1
// }

case class Name(value: String)

toJson(GenericNameTC(Name("john"), 1))
// could not find implicit value for parameter encoder: io.circe.Encoder[GenericNameTC[Name]]
```

Implicit Derivation 4

- 詳しく挙動を見る
- データ型の定義はそのまま
- 型クラスのインスタンス定義が複雑
- まずシグネチャを見る

```
implicit def genericNameTCEncode[T](implicit encoder: Encoder[T]): Encoder[GenericNameTC[T]] = ...
```

- このシグネチャが意味するのは任意の `T` に対して `GenericNameTC[T]` の `Encoder` インスタンスを生成するというとても広い定義
- しかしこの定義の実装時には当然 `T` をJson化する必要がある
- そのためにimplicit parameterとして `Encoder[T]` インスタンスを要求している

```
implicit def genericNameTCEncode[T](implicit encoder: Encoder[T]): Encoder[GenericNameTC[T]] =
  new Encoder[GenericNameTC[T]] {
    override def apply(a: GenericNameTC[T]): Json = Json.obj(
      "name" -> encoder(a.name), // ここで`Encoder[T]`による`T`のJson化処理が行われる
      "age" -> Json.fromInt(a.age)
    )
  }
```

Implicit Derivation 5

- 型クラス版の実装を見てみると各データ型に対する型クラスインスタンスの定義は「別々に」行われていることがわかる

```
// 各々のインスタンス定義は別のものとして実装されている
implicit val intEncoder: Encoder[Int] = ...
implicit val stringEncoder: Encoder[String] = ...
implicit def genericNameTCEncode[T](implicit encoder: Encoder[T]): Encoder[GenericNameTC[T]] = ...
```

- Scalaの型クラスがすごいのは別々に定義してあるimplicit(=型クラスのインスタンス)をコンパイラが集め、最終的にほしいデータ型のインスタンスを生成してくれること
- `Encoder[String] + Encoder[GenericNameTC[T] = Encoder[GenericNameTC[String]`のようなイメージ
- このようにアドホックにデータ型の組み合わせにおいて型クラスのインスタンスが定義されることを「Implicit Derivation」とか「型クラスの導出」とか言ったりする

参考

- https://typelevel.org/cats/typeclasses.html