

---

# Datatype Generic Programming with Scala 3

---

YOSHIMURA Hikaru

[hikaru\\_yoshimura@r.recruit.co.jp](mailto:hikaru_yoshimura@r.recruit.co.jp)



*April 15-16, 2023 @ Sponsor session of ScalaMatsuri 2023*

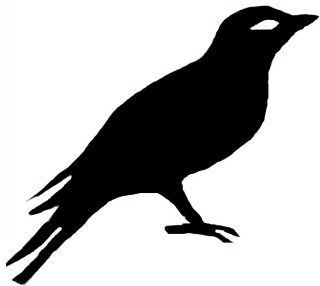
<https://github.com/y-yu/scalamatsuri2023> (5b54a4a)

## Datatype Generic Programming with Scala 3

- こちらのページは日本語の説明で、実際の発表では表示されません
- ただオーディエンス向けに事前に配っておく資料的な感じで利用する予定です

- ➊ Introduction
- ➋ Overview of datatype generic programming
- ➌ Datatype generic programming in Scala 3
- ➍ Conclusion

# Who am I?



- Recruit Co., Ltd.
  - StudySapuri ENGLISH server side
- Quantum Information & Algorithms
- Cryptography & Security
- Programming &  $\text{\LaTeX}$  typesetting
  - Scala, Rust, Go, Swift

Twitter [@\\_yyu\\_](#)

Qiita [yyu](#)

GitHub [y-yu](#)

## TestObject: generating fixtures for unit tests

- We use `TestObject` on our product, which is a utility for generating dummy objects(also known as *fixtures*) for unit tests.

```
case class StudySapuriSession(  
  /* very complicated! */  
)  
val dummyData = TestObject.get[StudySapuriSession]
```

- Type class `TestObject[A]` provides us with a way to generate some value of `A`.

```
trait TestObject[A] {  
  def generate: State[Int, A]  
}
```

```
implicit val strInstance: TestObject[String] = new TestObject {  
  def generate: State[Int, A] = State(s => (s + 1, s.toString))  
}
```

- In a naive way, we would have to define too many `TestObject` implicit instances for every type used in our product, but it's not possible or reasonable.

- このトークではこの `TestObject` の話をする
- これは単体テストなどのモッキング用に、任意の型 `A` のダミー値を作成することができる
- 実態としてはステートモナドになっており、この `Int` のステートを元にして型 `A` の値を作成する
- naïf には、このような `implicit` インスタンスを、プロダクトに存在する型のぶんだけ大量に定義していく必要があるが、そのような作業は無理である

# TestObject and datatype generic programming

- The many `TestObject` instances can be provided by *datatype generic programming*, rather than manually.
  - We can easily obtain `dummyData: StudySapuriSession` once we define `TestObject` instances for primitive or Java types,
  - And then datatype generic programming generates the other instances for our defined data structures(= case objects).
- Datatype generic programming is one of the ways of meta-programming.
- In this talk I'll explain datatype generic programming with Scala 3.

- このようなときに *datatype generic programming* を使うことができる。手作業でのインスタンス定義を回避して自動的にインスタンスを提供させる
  - ユーザーが手でやるのはプリミティブな型や Java の型だけでよい
- Datatype generic programming はメタプロク的一种である
- しかし datatype generic programming のやり方が Scala 2 と 3 で相当変わってしまった。今回のトークでは Scala 3 でのやり方について解説する

# Datatype generic programming vs. raw macros

- Unfortunately if we were to use raw macros, we could create a new syntax like LISP's S-expression in Scala source, but we don't want that.
- Almost every data structure can be classified as either “tuple”-like or “enum”-like:

```
case class TupleLike(  
  field1: Int, field2: String  
)
```

```
sealed trait EnumLike  
case class Pattern1(v: Int) extends EnumLike  
case class Pattern2(v: String) extends EnumLike
```

- `TupleLike` requires both two values of `Int` and `String`, whereas `EnumLike` requires either an `Int` value or a `String` value.
- Datatype generic programming provides us with the following two functions: ① converting a value to the analogy tuple or enum ② and reverting it to the original type.

- たとえば我々が生のマクロを使うと、Scala ソースコード内で LISP の S 式を書けるような構文を創造できるが、あまりそれは望まれていない
  - 構文が創造されれば、Scala の他にその創造された構文の勉強も必要になってしまい、事実上 1 つのプログラム言語のために複数の言語を学ぶようなことになる
- ほぼ (?) すべてのデータ構造はタプルのような構造と、enum のような構造のどちらかである
  - タプルは含まれている型の全ての値が必要で、一方で enum は含まれる型のうちどれかしら 1 つを要請する
  - この例だと `TupleLike` は `Int` と `String` の両方が必要だが、一方で `EnumLike` はどちらがあればインスタンス化できる
- こういう感じで全てのデータ構造についてタプルと enum に分けていき、そのタプルを操作して再び元のデータ構造に戻すみたいな抽象化を提供するのが Datatype generic programming である

# Meta-programming using datatype generic programming

- Almost all meta-programming can be done using such datatype abstraction and ad-hoc polymorphism, without the need to create a new syntax.
  - ① First, convert user-defined data structures(= case objects) to tuple or enum like using datatype generic programming.
  - ② Then, find some implicit instances based on the types included in the tuple or enum.
  - ③ Finally, revert the derived instance for tuple or enum like to one for the original data type.
- `case class TupleLike(field1: Int, field2: String)` example:

$$\frac{\text{TupleLike} \Leftrightarrow (\text{Int}, \text{String}) \quad \frac{\text{TestObject}[\text{Int}] \quad \text{TestObject}[\text{String}]}{\text{TestObject}[(\text{Int}, \text{String})]} \textcircled{2}}{\text{TestObject}[\text{TupleLike}] \textcircled{1}}$$

where `TupleLike`  $\Leftrightarrow$  `(Int, String)` is powered by datatype generic programming.

- このようにタプルや `enum` とデータ構造を行きするような抽象化と、アドホック多相があればほぼ全てのメタプログラミングをシンタックスの創造なしで行うことができる
  1. datatype generic programming でデータ構造をタプルや `enum` に変換する
  2. そのタプルや `enum` に含まれている型の implicit インスタンスを探索する
  3. そして元々の型に復活させる
- 具体的に `TupleLike` を見ていくとこういう感じになる

# Macro compatibility between Scala 2 and 3

- We use *shapeless*[1] for datatype generic programming in Scala 2.
- In our product, we compile & test both of Scala 2 and 3 for almost all of our code.
- There is no compatibility of macros between Scala 2 and 3 😊
- It follows that `TestObject` implemented on Scala 2 won't work well in Scala 3.
  - *shapeless* 3[2] for Scala 3 is being developed but unfortunately it doesn't have compatibility of *shapeless* for Scala 2 😊

- Eventually we(mainly ScalaNinja) began to develop another `TestObject` implementation for Scala 3



Fig 1: ScalaNinja

- 我々は Scala 2 での datatype generic programming に *shapeless* をつかっている
- ところで、我々のプロダクトは Scala 3 でもほぼ全てのコードをコンパイル&テストしている
- そして Scala 2 と 3 でマクロの互換性がない
- つまり Scala2 で作ってあった `TestObject` は Scala 3 では動かない。Scala 3 対応した *shapeless* 3 も作らてはいるが、これは *shapeless* 2 とは別物というくらいに互換性がない
- そういうわけで Scala 3 版の `TestObject` を作るようになった



# Datatype generic programming in Scala 3

- Scala 3 supports datatype generic programming initially.
- Some functions can be used to convert case objects from/to tuple like without any libraries like follows:

```
import scala.compiletime.*
import scala.deriving.*
case class TupleLike(
  field1: Int, field2: String
)
```

```
scala> Tuple.fromProductTyped(TupleLike(1, "a"))
val res0: (Int, String) = (1,a)

scala> summon[Mirror.ProductOf[TupleLike]].fromProduct(res0)
val res1: TupleLike = TupleLike(1,a)
```

- Similarly some functions can be used to convert `sealed trait` to enum like.
- Meta-programming tools in Scala 3 is reinforced rather than Scala 2 👍

- 実は Scala 3 はライブラリーなしでこのようにケースクラスをタプルに変換するなどの機構が用意されている
- こんな感じで `scala.compiletime` や `scala.deriving` を引っ張ってくることで、任意のケースクラスをそのフィールドの型に対応するタプルにしたり、タプルから構成したりできる
- 今回は紹介しない部分も含めて、Scala 3 はメタプログラミングが Scala 2 に比べて強化されている
- TODO: “reinforced” はなんかもっといい英語を探す

# TestObject implementation on Scala 3

- We'll define `derive` method as the final goal, like this:

```
inline implicit def derive[A]: TestObject[A]
```

- `derive` provides `TestObject` instances for all `A`.
- This is an overview of the `derive` behavior:
  - ① Check if the instance for the input type has been defined.
  - ② If not found, pattern match the type into either tuple like or enum like.
  - ③ Collect the *ill-typed* list of `TestObject` for each types contained in ② using `erasedValue`.
  - ④ Finally, make the instance for the input type using ③ instances list.
- Let's see the details!

- 今回の目標は Scala 3 のメタプロ機構で任意の型 `A` に対する `TestObject[A]` を提供する `derive` メソッドをつくること
- `derive` は
  1. すでに定義されているインスタンスがないか探して
  2. もしなかったら、タプルか enum のケースへパターンマッチする
  3. `erasedValue` をつかって、上記のタプルか enum に入っている型のインスタンスを集めてきて型なしリストに詰め込む
  4. そして最後に 🍷 のリストで材料をつくって最終的なインプットされた型のインスタンスをつくる

# ① Check if the instance for the input type has been defined

- `summonFrom` searches the `TestObject` instance for type `A`.

```
inline implicit def derive[A]: TestObject[A] =  
  summonFrom {  
    case x: TestObject[A] =>  
      x  
    case _ =>  
      create[A] // we'll define next page!  
  }
```



Fig 2: Image of `summonFrom`

- `summonFrom`をつかってインスタンスを探すことができる
  - もし見つかったらそれを返せばいい
  - そうでない場合、`create`メソッドをつかって定義していく
- If `summonFrom` finds a `TestObject[A]` instance, then the instance will be assigned to the variable `x`.
    - In this case, it's unnecessary to define a new instance so `derive` returns `x`.
  - In the latter case, we call `create` method to define `TestObject[A]`.

## ② Pattern matching if `A` is `ProductOf[A]` or `SumOf[A]`

- Since there is no `TestInstance[A]` instance yet, `create` finds `ProductOf[A]` or `SumOf[A]` instance using `summonFrom` again.

```
inline final def create[A]: TestObject[A] =  
  summonFrom {  
    case _: Mirror.ProductOf[A] =>  
      deriveProduct[A] // 1  
    case _: Mirror.SumOf[A] =>  
      deriveSum[A]      // 2  
  }
```

- This means that:
  - `A` is a tuple-like type (i.e. case classes) if there is a `ProductOf[A]` instance,
  - `A` is an enum-like structure (i.e. sealed traits). if there is a `SumOf[A]` instance.

- `TestInstance[A]`が見つからなかったので、`create`では再び `summonFrom`を使って `ProductOf[A]`か `SumOf[A]`のインスタンスを探す
  - もし `ProductOf[A]`が見つかったら、型 `A`はケースクラスなどのタプル的なデータ構造であり、
  - 一方で `SumOf[A]`のインスタンスが見つければ、型 `A`は sealed trait などの enum 的なデータ構造である

### ③ Making a list `List[TestObject[?]]` of *ill-typed* instances

- Before seeing `deriveProduct` and `deriveSum`, we need to prepare the way to collect all instances for types being contained in `A`.
- For example `TupleLike`, we need the both instances of `TestObject[Int]` and `TestObject[String]`.
- `erasedValue` allows us to search and collect all instances recursively as follows:

```
inline def deriveRec[T <: Tuple]: List[TestObject[?]] =  
  inline erasedValue[T] match {  
    case _: EmptyTuple =>  
      Nil  
    case _: (t *: ts) =>  
      derive[t]/* mutual recursion */ :: deriveRec[ts]  
  }
```

```
case class TupleLike(  
  field1: Int, field2: String  
)
```

- There is no type compatibility among the instances, `deriveRec` cannot help but to return *ill-typed* list 😊


- Additionally, `*:` is type-level tuple constructor provided since Scala 3.

- `deriveProduct`とか `deriveSum`を見ていくまえにタプルとか `enum` に所属する型のインスタンスを全部あつめてくる `deriveRec`を準備しておく
- たとえば `TupleLike`なら `TestObject[Int]`と `TestObject[String]`の両方のインスタンスを集めてくる
- `erasedValue`を使うと再帰的に集めてくることできる
- ただし `TestObject[Int]`と `TestObject[String]`みたいにインスタンスの間に型の互換性はないから、これを1つのリストに詰め込むと型が壊れる
- ちなみに `*:`は型レベルタプルで、`HList`みたいなやつ **TODO: この理解であって**  
**るか？**

## 4a In deriveProduct case

- Using `deriveRec`, we define a `TestObject` instance for `A` in `deriveProduct` case.

```
inline def deriveProduct[A](using a: ProductOf[A]): TestObject[A] = {  
  def p: TestObject[A] = {  
    val xs = deriveRec[a.MirroredElemTypes] // `a.MirroredElemTypes` is analogy tuple of `A`.  
    productImpl[A](xs, a)  
  }  
  p  
}
```

- Why does `deriveProduct` only call `productImpl` through a temporary method `p`?  

- This is ScalaNinja's remarkable and state-of-the-art technique to avoid:
  - throwing `MethodTooLargeException` due to `inline`
  - and generating too many nameless classes.

- `deriveRec`をつかって `TestObject[A]` のインスタンスを作っていく
- これは謎にメソッド `p` の中で `productImpl` 呼ぶだけとなっているのは、実は
  - `inline` のコード生成で `MethodTooLargeException` がブチあがるのを避け、
  - 無名クラスではなくてメソッドにしておくことで、無名クラスの大量生成も避ける

という ScalaNinja のテクニックとなっている

- こういうコンパイル時の効率も考慮しないといけないので、メタプロは難しい

## 4a In deriveProduct case

- First, we create a tuple naming `values` which are containing all values required by `A`.

```
final def productImpl[A](xs: List[TestObject[?]], a: ProductOf[A]): TestObject[A] =  
  new TestObject[A] {  
    def generate: IntState[A] =  
      for {  
        values <- xs.traverse(_.generate.widen[Any])  
      } yield a.fromProduct(new SeqProduct(values))  
  }
```

- It's important that `productImpl` doesn't have `inline`.
- Then, we create a `A` value using `a.fromProduct`.

- `List[TestObject[?]]`をつかってまず必要な値をつくる
- それを `a.fromProduct`で型 `A`に戻す
- ここで `productImpl`には `inline`がないのが地味に重要
  - こうしておくことで `inline`でメソッドが巨大化しすぎてエラーとなるのを回避する

## 4b In `deriveSum` case

- In `SumOf` case, we generate a value in `values`.

```
inline def deriveSum[A](using a: SumOf[A]): TestObject[A] = {  
  def s: TestObject[A] = {  
    val values = deriveRec[a.MirroredElemTypes]  
    sumImpl[A](values)  
  }  
  s  
}
```

- It's very similar to `deriveProduct`.

- `SumOf`の場合を `deriveProduct`と同じようにできる



## 4b In deriveSum case

- `sumImpl` is a very complicated function 😊

```
final def sumImpl[A](values: List[TestObject[?]]): TestObject[A] =  
  new TestObject[A] {  
    def generate: IntState[A] =  
      for {  
        allResults <- values.traverse(_.generate.widen[Any])  
        l = allResults.minBy(_.getClass.getName)  
        rOpt = allResults.tail.headOption.flatMap(  
          _ => allResults.maxByOption(_.getClass.getName)  
        )  
        s <- State.get  
      } yield rOpt match {  
        case Some(r) => if (s % 2 == 0) l.asInstanceOf[A] else r.asInstanceOf[A]  
        case None => l.asInstanceOf[A]  
      }  
  }
```

- What's the purpose of `minBy(_.getClass.getName)` and `maxByOption(_.getClass.getName)`?

- `sumImpl`はめちゃくちゃ複雑
- けど実装は適当で、偶数か奇数かによってどれを選んでもくるか？ みたいなことをしているだけ
- ところで `minBy(_.getClass.getName)` と `maxByOption(_.getClass.getName)` はなんなのか？

## 4b In `deriveSum` case

- I don't know why, but anyway `shapeless 2` sorts the instances by their type name[3].
- On the other hand, `MirroredElemTypes` in Scala 3 are sorted by the definitions in source code.

- For instance:

```
sealed trait X
case object X3 extends X
case object X2 extends X
case object X1 extends X
```

- `shapeless 2` returns `X1 :+: X2 :+: X3`, which is sorted by alphabetical order,
- but `MirroredElemTypes` is `X3 *: X2 *: X1` 😊

- So the `minBy` and `maxByOption` are needed for the compatibility of `shapeless 2` behavior.

- 理由は不明だが `shapeless 2` は見つかったインスタンスをその名前順にソートして返すようになっている
- しかし Scala 3 の `MirroredElemTypes` は定義した順に返ってくる
- そこらの互換性のために一応順序をいじっている

# Conclusion

- The full source code for this talk has been published at 📌
  - <https://github.com/y-yu/test-object>
  - It may be useful as proof-of-concept to compare Scala 2(shapeless 2) with Scala 3.
- There is no macro compatibility between Scala 3 and 2 😊
  - And shapeless 2 and 3 don't have the same interface.
- Scala 3 supports datatype generic programming initially.
  - Is there any ways how not using ill-typed list? 🤔
- Happy datatype generic programming!

- 今回つくった `TestObject` の全体のソースは GitHub で公開してある
- Scala 2 と 3 ではマクロ（メタプロ）の互換性がないし、shapeless も 2 と 3 に互換性がない
- 一方で Scala 3 は datatype generic programming を最初からサポートしている。とはいえ ill-typed なリストを使わなくてもいいような方法とがあればいいと思う
- datatype generic programming を楽しもう！

- As March 1st, the number of lines of Scala 2 & 3 source code is 878,434 in our product.
  - This does not include generated code (such as protobuf & gRPC), so the total is approximately over 1,000,000.
- There are many microservices (Fig.3), making it a very complicated system 😊
- The number of our server-side team members is about 16.
- Meta-programming, which includes not only datatype generic programming but also *scalafix* and so on, is very useful for us.

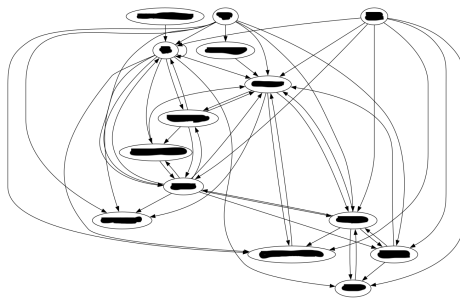


Fig 3: Very complicated micro services

- 我々はスタディーサプリ ENGLISH というプロダクトを開発していて、ソースコード規模としては3月時点でテスト込みで87万行ある。protobufやgRPCで生成されたコードを含めると100万行以上となる
- 図3にあるようにマイクロサービスの数も多く、関係も複雑となっている
- こういうものを16名程度のメンバーで維持するとなると、今回紹介したdatatype generic programmingや他にもscalafixといったメタプログラミングのテクニックはとても重要となる

# References

- [1] shapeless: generic programming for Scala (GitHub).  
<https://github.com/milessabin/shapeless>.  
Accessed: 2023-03-13.
- [2] shapeless 3 for Scala 3 (GitHub).  
<https://github.com/typelevel/shapeless-3>.  
Accessed: 2023-03-13.
- [3] shapeless 2 sorts subclasses by alphabetical order.  
<https://github.com/milessabin/shapeless/blob/da31eced505d3df9637a3a28825ff31c65a99ffe/core/shared/src/main/scala/shapeless/generic.scala#L412>.  
Accessed: 2023-03-13.

Thank you for the attention!