Generic programming in Scala

[Superfunky library name here]

Carlos Tomé Cortiñas   Matthew Swart   Renate Eilers

Department of Information and Computing Sciences, Utrecht University

Table of contents

An introduction to generic programming

Generic programming – Why?

Do you ever find yourself writing variations of the same program over and over?

```
data List a = Cons a (List a)
            | Nil

size :: List a → Int
size Nil = 0
size (Cons x xs) = 1 + size xs
```

```
data Tree a = Node (Tree a) (Tree a)
            | Leaf a

size :: Tree a → Int
size (Leaf n) = 1
size (Node l r) = 1 + size l + size r
```

Write functions on the *structure* of datatypes rather than on concrete instantiations.

Generic programming – What?

Most datatypes can be represented by combination of *basic types* such as **units**, **sum** and **product**:

```
data List a = Nil | Cons a (List a)
```

vs.

```
data RList a = () :+: (a :×: (List a))
```

```
data () = ()
data a :+: b = Inl a | Inl b
data a :×: b = a :×: b
```

A program written to work on these standard types can be used on *any* datatype provided a conversion between both types is supplied!

```
data Iso a b = Iso {from :: a → b, to :: b → a}
```

```
fromList :: List a → RList a
toList   :: RList a → List a
```

Shapeless

Representation of case classes/sealed traits based on HList and Coproducts

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

val gen = Generic[Shape]
gen: shapeless.Generic[Shape]{type Repr = shapeless.:+:
                      [Rectangle,shapeless.:+:[Circle,shapeless.CNil]]}
```

The pattern:

```
trait Eq[A] {
  def eq(x : A, y : A) : Boolean
}
implicit val hnilEq: Eq[HNil] = {..}
implicit def hlistEq[H, T <: HList](..) = {..}

implicit val cnilEq : Eq[CNil] = {..}
implicit def coproductEq[H, T <: Coproduct](..) = {..}

implicit def genericEq[A, R](
    implicit
      gen: Generic.Aux[A, R],
      env: Lazy[Eq[R]]
  ) : Eq[A] = {..}
```

EMGM

Criteria for generic programming libraries

	LIGD	PolyLib	SYB	SYB3	Spine	EMGM	RepLib	Smash	Uniplate
Universe Size									
Regular datatypes	●	●	●	●	●	●	●	●	●
Higher-kinded datatypes	●	○	●	●	●	●	○	●	●
Nested datatypes	●	○	●	○	●	●	●	◐	●
Nested & higher-kinded	●	○	○	○	●	◐	○	◐	○
Other Haskell 98	●	○	●	●	●	●	●	◐	●
Subuniverses	○	●	○	○	○	●	●	○	○
First-class generic functions	●	○	●	●	●	◐	●	◐	○
Abstraction over type constructors	●	◐	○	○	◐	●	●	●	○
Separate compilation	●	●	●	●	○	●	●	●	●
Ad-hoc definitions for datatypes	○	◐	●	●	○	●	●	●	●
Ad-hoc definitions for constructors	◐	○	●	●	●	●	●	●	●
Extensibility	○	◐	○	●	○	●	●	○	○
Multiple arguments	●	●	◐	◐	●	●	●	◐	○
Constructor names	●	●	●	●	●	●	●	●	○
Consumers	●	●	●	●	●	●	●	●	●
Transformers	●	●	●	●	●	●	●	◐	●
Producers	●	●	◐	◐	◐	●	●	◐	○
Performance	◐	◐	○	○	◐	●	◐	●	●
Portability	●	○	○	○	○	●	○	○	●
Overhead of library use									
Automatic generation of representations	○	○	●	◐	○	○	◐	○	●
Number of structure representations	4	1	2	2	3	4	4	8	1
Work to instantiate a generic function	◐	●	●	●	◐	◐	◐	●	●
Work to define a generic function	◐	●	●	●	●	●	◐	●	●
Practical aspects	○	◐	●	◐	○	○	●	○	●
Ease of learning and use	◐	●	○	○	●	◐	○	○	●

EMGM I

Extensible and **M**odular **G**enerics for the **M**asses.

Functions are implemented using the `Generic` class:

```
class Generic g where
    unit :: g ()
    plus :: g a → g b → g (a :+: b)
    prod :: g a → g b → g (a :×: b)
    int  :: g Int
    view :: Iso b a[1] →g a → g b
```

[1] The type `Iso b a` signifies an isomorphism between the types `a` and `b`.

EMGM II

Type representations are encoded in the Rep class.

```
class Rep g a where
  rep :: g a

instance (Generic g) ⇒ Rep g Int where
  rep = int
```

EMGM III

To write a generic function we define a type and make it an instance of `Generic`. As an example, let's define a generic encoding function:

```
newtype Encode a =  Enc {enc :: a → [Bit]}

instance Generic Encode where
    unit       = Enc (const [])
    plus a b   = Enc (λ x →case x of
                  Inl l → 0:enc a l
                  Inr r → 1:enc b r
    prod a b   = Enc (λ (x :+: y) →
                  enc a x ++ enc b y
    int        = Enc encodeInt
    view iso a = Enc (λ x →enc a (from iso x)}
```

Implementation

Converting Haskell code to Scala: type classes

Haskell's type classes can be simulated by Scala's traits:

```
class Generic g where
    unit :: g Unit
    plus :: g a → g b →
      g (a :+: b)
    prod :: g a → g b →
      g (a :×: b)
    int  :: g Int
    view :: Iso b a →
      g a → b a
```

```
trait Generic[G[_]] {
    def unit: G[Unit]
    def plus[A, B]
      (a: G[A], b: G[B]): G[Plus[A, B]]
    def prod[A, B]
      (a: G[A], b: G[B]): G[Prod[A, B]]
    def int: G[Int]
    def view[A, B]
      (iso: Iso[B, A], a: () => G[A]): G[B]
}
```

Converting Haskell code to Scala: class constraints

Haskell's class constraints can be modeled through Scala's implicits:

```
newtype Encode a =  Enc {enc :: a → [Bit]}

instance Generic Encode where
    unit       = Enc (const [])
    prod a b   = Enc (λ (x :+: y) →  enc a x ++ enc b y
    view iso a = Enc (λ x →enc a (from iso x)}
```

vs.

```
abstract class Encode[A] {
    def enc :  A => List[Bit]
  }

implicit object Encode extends Generic[Encode] {
    def unit : Encode[Unit] = new Encode[Unit] {def enc = const(Nil)}
    def prod[A,B](a : Encode[A], b : Encode[B]) : Encode[Prod[A,B]] = {
      new Encode[Prod[A,B]] { def enc = x => x match {
          case  Prod(l,r) => (a.enc(l)) ++ (b.enc(r))
        }
      }
    def view[A,B](iso : Iso[B,A],  a : () => Encode[A]) : Encode[B] = {
      new Encode[B] { def enc = x => a().enc(iso.from(x))
      }
    }
  }
```

Difficulties

- Many generic functions require functionality that is not standard in Scala (e.g. type-level currying, higher-kinded types)

```
trait Collect[F[_],B,A] {
  def collect_ : A => F[B]
}
implicit def CollectG[F[_],B] (implicit altf : Alternative[F]) =
  new Generic[({type C[X] = Collect[F,B,X]})#C]{...}
```

- Scala's ability to do type inference is a lot weaker than Haskell's, resulting in failure to resolve implicit values.

```
type C[X] = Collect[List,Int,X]
implicit val i = implicitly[GRep[C,List[Plus[Int,Char]]]]
```

- Scala call-by-value by default has to be overcomed to deal with recursive types, which some functions in the EMGM library depend upon.

Results

Results

The library currently comes with the following function types[2]:

- `Map a b`: for mapping values of type `a` inside a functor to `b`.
- `Crush b a`: a generalization of `fold`.
- `Collect f b a`: for collected all values of type `b` from a type `a` into some container-type `f`.
- `Everywhere a b`: for applying a function of type `a` $\rightarrow$ `a` for everything of type `a` that is encountered within a structure of type `b`.

[2] `Collect` and `Everywhere` have not yet been fully tested

Example

```
scala> val tree : BinTree[Int] = Bin(Bin(Leaf(1),Leaf(2)),Bin(Leaf(3),Leaf(4)))
tree: Data.BinTree[Int] = Bin(Bin(Leaf(1),Leaf(2)),Bin(Leaf(3),Leaf(4)))

scala> val list : List[Int] = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> map((x: Int) ⇒ x + 1) (list)
res0: List[Int] = List(2, 3, 4, 5)

scala> map((x: Int) ⇒ x + 1) (tree)
res1: Data.BinTree[Int] = Bin(Bin(Leaf(2),Leaf(3)),Bin(Leaf(4),Leaf(5)))
```

Conclusion

Conclusion

Overall, the Scala language is expressive enough to support generic programming, **but**:

- Generic programming in Scala is not straightforward, some functionalities have to be 'hacked' in (e.g. non-strict evaluation, type-level currying)
- We just focused on the functional part of Generic programming.
- Code turns very long and unelegant very easily, in part because the programmer is required to supply a lot of type information.