Type Class: The Ultimate Ad Hoc

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

August 3, 2017

Type classes are a language feature

- Haskell
- Eta
- Purescript
- Clean

Type classes are a language feature

- Haskell
- Eta
- Purescript
- Clean

or sometimes a design pattern

- Scala
- OCaml

Polymorphism

A type is *polymorphic* iff it can be applied at multiple types.

Polymorphism is good

- less duplication
- more reuse
- fewer possible implementations

Broadly speaking there are two major forms of polymorphism:

- *parametric* polymorphism
- *ad-hoc* polymorphism

A type is `parametrically polymorphic` iff it has at least one *type parameter* which can be instantiated to *any type*.

```
reverse :: [a] -> [a]
id :: a -> a
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

A type which is ad-hocly polymorphic can be instantiated to different types, and may behave differently at each type

Type Classes

```haskell
class Equal a where
  eq :: a -> a -> Bool
```

```haskell
class Equal a where
  eq :: a -> a -> Bool


data Person = Person {
  age :: Int
, name :: String
}
```

```haskell
class Equal a where
  eq :: a -> a -> Bool

data Person = Person {
  age :: Int
, name :: String
}

instance Equal Person where
  eq p1 p2 = eq (age p1) (age p2) && eq (name p1) (name p2)
```

```haskell
elementOf :: Equal a => a -> [a] -> Bool
elementOf a list =
  case list of
    []    -> False
    (h:t) -> eq a h || elementOf a t
```

Instances can be constrained

```haskell
instance (Equal a) => Equal [a] where
  eq []     []     = True
  eq (x:xs) []     = False
  eq []     (y:ys) = False
  eq (x:xs) (y:ys) = eq x y && eq xs ys
```

Instances can be constrained

```
instance (Equal a) => Equal [a] where
  eq []     []     = True
  eq (x:xs) []     = False
  eq []     (y:ys) = False
  eq (x:xs) (y:ys) = eq x y && eq xs ys
```

We can add type class instances for types we didn't write

- Writing an instance for your type can unlock many functions
- Writing functions with typeclass constraints is easy
- You can write instances for types you did not write
- Instances can depend on other instances if necessary

Interfaces

```
interface Equal<A> {
  public boolean eq(A other);
}
```

```java
interface Equal<A> {
  public boolean eq(A other);
}

class Person implements Equal<Person> {
  public int age;
  public String name;

  public boolean eq(Person other) {
    return this.age == other.age && this.name.equals(other.name);
  }
}
```

```java
static <A extends Equal<A>> boolean elementOf(A a, List<A> list) {
  for (A element : list) {
    if (a.eq(element)) return true;
  }
  return false;
}
```

```java
class String {
  private char[] value;
  // other definitions
}
```

```
class String implements Equal<String> {
  private char[] value;
  // other definitions
}
```

```
class List<A> {
  // implementation details




}
```

```
class List<A> implements Equal<List<A>> {
  // implementation details




}
```

```java
class List<A> implements Equal<List<A>> {
  // implementation details

  public boolean eq(List<A> other) {
    // implementation...

  }
}
```

```java
class List<A> implements Equal<List<A>> {
  // implementation details

  public boolean eq(List<A> other) {
    // implementation...
    // ... but how do we compare A for equality?
  }
}
```

- Interface implementation can't be conditional
- We can only implement interfaces for types we control

I argue this makes type classes more modular and more flexible

TODO show why type classes aren't perfectly flexible
(no custom local instances, maybe show newtypes for sum and product or something)

Implicits

```scala
case class Person(age: Int, name: String)
```

```scala
case class Person(age: Int, name: String)


trait Equal[A] {
  def eq(a: A, b: A): Boolean
}
```

```scala
case class Person(age: Int, name: String)

trait Equal[A] {
  def eq(a: A, b: A): Boolean
}

implicit def equalPerson: Equal[Person] = new Equal[Person] {
  def eq(a: Person, b: Person): Boolean =
    a.age == b.age && a.name == b.name
}
```

```scala
def elementOf[A](a: A, list: List[A])
                (implicit equalA: Equal[A]): Boolean = {
  list match {
    case Nil => false
    case (h::t) => equal.eq(a, h) || elementOf(a, t)
  }
}
```

```scala
implicit def equalList(implicit equalA: Equal[A]): Equal[List[A]] =
  new Equal[List[A]] {
    def eq(a: List[A], b: List[A]): Boolean = {
      (a,b) match {
        case (Nil,    Nil)    => true
        case (x::xs, Nil)    => false
        case (Nil,    y::ys) => false
        case (x::xs, y::ys) => equalA.eq(x,y) || eq(xs,ys)
      }
    }
  }
```

- We can define implicits for types we did not write
- We can write implicits that depend on implicits

```scala
sealed trait Ordering
case object LT extends Ordering
case object EQ extends Ordering
case object GT extends Ordering


trait Order[A] {
  def compare(a: A, b: A): Ordering
}
```

```scala
sealed trait Ordering
case object LT extends Ordering
case object EQ extends Ordering
case object GT extends Ordering


trait Order[A] {
  def compare(a: A, b: A): Ordering
}


implicit def orderPerson: Order[Person] = new Order[Person] {
  def compare(a: Person, b: Person): Ordering =
    intOrder.compare(a.age, b.age) match {
      case LT => LT
      case EQ => stringOrder.compare(a.name, b.name)
      case GT => GT
    }
}
```