Type Class: The Ultimate Ad Hoc

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

August 7, 2017

Type classes are a language feature

- Haskell
- Purescript
- Eta
- Clean

or sometimes a design pattern

- Scala

Polymorphism

Something which is *polymorphic* has many shapes

Polymorphism is good

- less duplication
- more reuse
- fewer possible implementations

Broadly speaking there are two major forms of polymorphism in programming:

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

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

Example:

```
reverse :: [a] -> [a]
```

An *ad-hocly polymorphic* type can be instantiated to some different types, and may behave differently for each type

Example:

==

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


elementOf(me, functionalProgrammers);
// true
```

```java
package java.lang;

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

```
package java.lang;

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

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




}
```

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

```
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

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)
```

```
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

```
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

- You can write instances for types you did not write
- Instances can depend on other instances

There are exactly two places a type class instance is allowed to exist

Person.hs

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

instance Equal Person where
  eq p1 p2 = ...
```

Equal.hs

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

There are exactly two places a type class instance is allowed to exist

Person.hs

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

Equal.hs

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


instance Equal Person where
  eq p1 p2 = ...
```

Person.hs

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

Equal.hs

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

EqualInstances.hs

```haskell
instance Equal Person where
  eq p1 p2 = ...
```

Person.hs

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

Equal.hs

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

EqualInstances.hs

```haskell
instance Equal Person where
  eq p1 p2 = ...
```

"Orphan instance"

Orphan instances break *type class coherence*

Type class coherence gives many sane benefits:

- There can only be zero or one instance
- Using an instance never depends on imports

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

Implicits

More Flexible Than Typeclasses™

```
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

- No restriction on orphan instances
- No restriction on number of instances

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

```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
    }
}
```

```scala
def sort[A](list: List[A])(implicit orderA: Order[A]): List[A] = {
  // quicksort goes here
}
```

```
sort(
  List(
    Person(30, "Robert")
  , Person(20, "John")
  , Person(40, "Alfred")
  )
)
```

```
sort(
  List(
    Person(30, "Robert")
  , Person(20, "John")
  , Person(40, "Alfred")
  )
)


==>

List(
  Person(20, "John")
, Person(30, "Robert")
, Person(40, "Alfred")
)
```

Then the boss says “I want those sorted by name”.

Then the boss says “I want those sorted by name”.

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

```
sort(
  List(
    Person(30, "Robert")
  , Person(20, "John")
  , Person(40, "Alfred")
  )
)
```

```
sort(
  List(
    Person(30, "Robert")
  , Person(20, "John")
  , Person(40, "Alfred")
  )
)


==>

List(
  Person(40, "Alfred")
, Person(20, "John")
, Person(30, "Robert")
)
```

```scala
// both in scope
implicit def orderPerson: Order[Person] = ...
implicit def orderPersonByName: Order[Person] = ...

// what happens?
sort(persons)
```

```scala
// both in scope
implicit def orderPerson: Order[Person] = ...
implicit def orderPersonByName: Order[Person] = ...

// what happens?
sort(persons)
```

Hopefully a compiler error!

Set.scala

```scala
def emptySet[A]: Set[A]

def insert[A](a: A, set: Set[A])(implicit o: Order[A]): Set[A]

def union[A](s1: Set[A], s2: Set[A])(implicit o: Order[A]): Set[A]

def isElement[A](a: A, set: Set[A])(implicit o: Order[A]): Boolean
```

Persons.scala

```scala
implicit def orderPersonByAge: Order[Person] = ...

def persons: Set[Person] =
  insert(p1, insert(p2, insert(p3, emptySet)))
```

Something.scala

```scala
import Persons.persons

implicit def orderPersonByName: Order[Person] = ...

val x = isElement(p1, persons) // FALSE!
```

Recommendations when writing implicits:

- Only create instances in the file that defines the type or the "type class"
- Disallow creating more than one instance (regardless of which file you're in)

Recommendations when writing implicits:

- Only create instances in the file that defines the type or the “type class”
- Disallow creating more than one instance (regardless of which file you're in)

What about implicits in external libraries?

- Assess their usage of implicits. Do they use them as like type classes?
- If you distrust their implicits, pass everything of theirs explicitly

Type classes:

- Big wins in flexibility, expressiveness, and modularity
- Restrictions are straightforward and compiler checked
- Coherence keeps things sane

Thanks for listening!

Questions?