Type Class: The Ultimate Ad Hoc

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

August 23, 2017



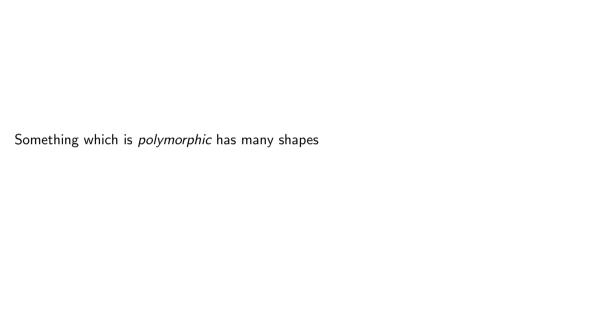
Type classes are a language feature

- ► Haskell
- Purescript
- EtaClean

Scala

- or sometimes a design pattern

Polymorphism



Polymorphism is good

- ▶ less duplication
- more reuse
- many other benefits

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.

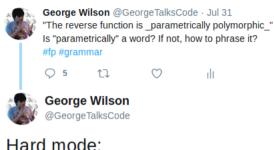
Example:

reverse :: [a] -> [a]

An	ad-hocly	polymorphic	type ca	n be	instantiated	to	some	different	types,
and	l may beh	nave different	ly for ea	ich t	ype				

Example:

==



"The sum function is _ad-hocly polymorphic_"?
Surely that can't be cromulent.

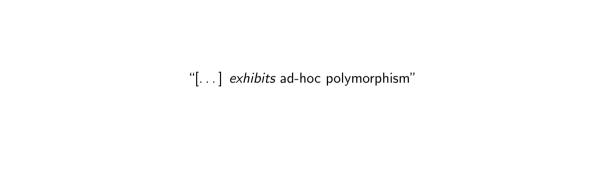
4:52 PM - 31 Jul 2017

 \bigcirc

Ĺ.

0

- 1



Programmers discussing programming



Programmers discussing grammar





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

```
interface Equal<A> {
   public boolean eq(A other);
}
class Person {
   public int age;
```

public String name;

```
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);
```

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

return false;

```
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...
```

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

public boolean eq(List<A> other) {
```

// ... but how do we compare A for equality?

// implementation...

Type Classes

class Equal a where

eq :: a -> a -> Bool

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

data Person = Person {
  age :: Int
```

, name :: String

```
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
```

(h:t) -> eq a h || elementOf a t

[] -> False

Instances can be constrained

eq (x:xs) (y:ys) = eq x y && eq xs ys

eq [] (y:ys) = False

Instances can be constrained

eq (x:xs) (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

Type classes have restrictions in order to enforce type class coherence

Informally, coherence means:

- ▶ for a given type class for a given type, there is zero or one instance
 - ▶ no matter how you ask for an instance, you get the same one
 - ▶ if an instance exists, you can't not get it

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

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

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

```
Equal.hs

class Equal a where

eq :: a -> a -> Bool
```

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

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

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

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

Equal.hs

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

```
Equal.hs
class Equal a where
eq :: a -> a -> Bool
```

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

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

```
Equal.hs

class Equal a where

eq :: a -> a -> Bool
```

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

"Orphan instance"
Orphan instances can break coherence

Type class coherence gives sane benefits:

- ▶ Instances never depends on imports
- ► The semantics are what we expect

Type class coherence gives sane benefits:

- ► Instances never depends on imports
- ▶ The semantics are what we expect

Type class coherence precludes:

- Custom local instances
- Multiple instances selectable by import

(But there are other solutions to those things)

Implicits

More Flexible Than TypeclassesTM

....

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

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

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

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

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

```
def eq(a: Person, b: Person): Boolean =
    a.age == b.age && a.name == b.name
}
```

implicit def equalPerson: Equal[Person] = new Equal[Person] {

```
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

- ▶ 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

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

```
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
}
```

```
case object LT extends Ordering
case object EO extends Ordering
case object GT extends Ordering
trait Order[A] {
  def compare(a: A, b: A): Ordering
implicit def joyDivisionWithoutIan = 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
```

sealed trait Ordering

```
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".

```
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")
```

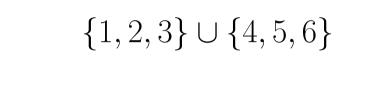
```
sort (
 List (
    Person(30, "Robert")
  , Person(20, "John")
  , Person(40, "Alfred")
==>
List (
 Person(40, "Alfred")
, Person(20, "John")
, Person(30, "Robert")
```

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

// what happens?
sort(persons)

```
// both in scope
implicit def orderPersonByAge: Order[Person] = ...
implicit def orderPersonByName: Order[Person] = ...
// what happens?
sort(persons)
```

Hopefully a compiler error!



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

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

```
Something.scala

import Persons.{p1, 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!

Aspect	Interfaces	Type classes	Implicits
Instance types you control	✓	✓	√
Instance types you don't control	X	✓	✓
Instances can depend on other instances	X	✓	✓
Type-directed	✓	✓	sort of
Custom local instances	X	X	✓
Coherent	✓	✓	X