Poor Man's Type Classes

Martin Odersky
EPFL

IFIP WG2.8 working group meeting

Boston, July 2006.

Goals

Type classes are nice.

A cottage industry of Haskell programmers has sprung up around them.

Should we add type classes to OO-languages, specifically Scala?

Problem: Conceptual expense

- We have already spent the keywords **type** and **class**!
- Type classes are essentially implicitly passed dictionaries, and dictionaries are essentially objects.
- Don't want to duplicate that.

Idea: Concentrate on the delta between OO classes and type classes: *implicits*

Life without Type Classes

Some standard classes for SemiGroup and Monoid:

```
abstract class SemiGroup[a] {
    def add(x : a, y : a): a
}
abstract class Monoid[a] extends SemiGroup[a] {
    def unit : a
}
```

Two implementations of monoids:

```
object stringMonoid extends Monoid[String] {
    def add(x : String, y : String): String = x.concat(y)
    def unit : String = ""
}
object intMonoid extends Monoid[int] {
    def add(x : Int, y : Int): Int = x + y
    def unit : Int = 0
}
```

A sum method which works over arbitrary monoids:

```
def sum[a](xs : List[a])(m : Monoid[a]): a =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))
```

One invokes this sum method by code such as:

```
sum(List("a", "bc", "de"))(stringMonoid)
sum(List(1, 2, 3))(intMonoid)
```

Implicit Parameters: The Basics

The following slight rewrite of sum introduces m as an *implicit parameter*.

```
def sum[a](xs : List[a])(implicit m : Monoid[a]): a =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail)(m))
```

- Can combine normal and implicit parameters.
- However, there may only be one implicit parameter list, and it must come last.

implicit can also be used as a modifier for definitions:

```
implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}
implicit object intMonoid extends Monoid[int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

Arguments to implicit parameters can be inferred:

```
    sum(List(1, 2, 3))
```

This expands to:

```
    sum(List(1, 2, 3))(intMonoid)
```

Inferring Implicit Arguments

If an argument for an implicit parameter of type T is missing, it is inferred.

An argument value is eligible to be passed, if

- it is itself labelled **implicit**,
- it's type is compatible with T,
- one of the following holds:
  1. x is accessible at the point of call by a simple identifier (i.e. it is defined in same scope, inherited or imported)
  2. x is defined as a static value in (some superclass of) T.

If several arguments are eligible, choose most specific one.

If no most specific eligible argument exists, type error.

Locality

A consequence of implicit argument resolution is that one can have several instance definitions of the same operation at the same types.

We always pick the one which is visible at the point of call.

This is an important difference between implicits and type classes.

Rules to keep coherence:

- Scala has functions (which are objects) and methods (which are not).
- Partially applied methods are automatically converted to functions.
- Only methods can contain implicit parameters.
- Implicit parameters are instantiated where a method value is eliminated (either applied or converted to a function).

Conditional Implicits

Implicit methods can themselves have implicit parameters.

Example:

```
implicit def listMonoid[a](implicit m: Monoid[a]) =
  new Monoid[List[a]] {
    def add(xs: List[a], ys: List[a]): List[a] =
      if (xs.isEmpty) ys
      else if (ys.isEmpty) xs
      else m.add(xs.head, ys.head) :: add(xs.tail, ys.tail)
    def unit = List()
  }
```

Then:

```
println(sum(List(List(1, 2, 3), List(1, 2))))
translates to
println(sum(List(List(1, 2, 3), List(1, 2)))(listMonoid(intMonoid))
==>
List(2, 4, 3)
```

External Extensibility

Often, we like to add some new functionality to a pre-existing type.

```
trait Ordered[a] {
  def compare(that: a): Int;
  def <  (that: a): Boolean = (this compare that) <  0
  def >  (that: a): Boolean = (this compare that) >  0
  def ≤ (that: a): Boolean = (this compare that) ≤ 0
  def ≥ (that: a): Boolean = (this compare that) ≥ 0
}
```

Want to make Int, String, etc ordered.

Want to make lists ordered if their elements are.

Common solution: “open classes”.

Implicit Conversions

An implicit conversion is a unary function from S ot T, which is labelled **implicit**.

Example:

```
implicit def int2ordered(x: int) = new Ordered[int] {
    def compare(y: int) = if (x < y) −1 else if (x > y) 1 else 0
}

def sort[a](xs: Array[a])(implicit c: a ⇒ Ordered[a]): Array[a] =
    if (xs.length ≤ 1) xs
    else {
        val pivot = xs(xs.length / 2)
        Array.concat(
            sort(xs filter (pivot >))
                 xs filter (pivot ==),
            sort(xs filter (pivot <)))}

val xss: Array[List[int]]

sort(xss)
```

Translation

```
def sort[a](xs : Array[a])(implicit c : a ⇒ Ordered[a]): Array[a] =
    if (xs.length ≤ 1) xs
    else {
        val pivot = xs(xs.length / 2)
        Array.concat(
            sort(xs filter (c(pivot) >)),
                 xs filter (pivot ==),
            sort(xs filter (c(pivot) <)))
    }
val xss : Array[List[int]]
sort(xss)(list2ordered(int2ordered))
```

Applications of Implicit Conversions

An implicit conversion is applied to a term e if,

- e is not compatible with its expected type:

```
val x : Ordered[int] = 1
==>
val x : Ordered[int] = int2ordered(1)
```

- In a selection $e.m$, if e does not have a member m.

```
x.<(1)
==>
int2ordered(x).<(1)
```

- In an application $e.m(a_1, \ldots, a_n)$, if e does not have a member m which can be applied to $(a_1, \ldots, a_n)$:

```
val x = BigInt(10);
1 + x
==>
BigInt(1) + x
```

Summary

Scala implements the following analogies:

type class	=	class
instance declaration	=	implicit definition
context in a clas	=	inheritance
context in a type	=	implicit parameter
dictionary	=	object
default method in class	=	concrete member
method signature in class	=	abstract member

Conclusion

Implicit parameters are poor man's type classes.

- Conceptually lightweight.
- Piggy-backed on object system.
- Implemented in Scala version 2,
- Can also model
  - + multi-parameter type classes
  - + parametric type classes (but no general functional dependencies).
  - + associated types
  - + constructor classes (via encodings, make this convenient we should add higher-kinded type variables)