Type Classes as Objects and Implicits

Bruno C. d. S. Oliveira
(joint work with Adriaan Moors and Martin Odersky)
ROSAEC Center Workshop (Milestone talk)

Introduction

- Type classes are good for
  - Retroactive extension
  - Concept-style generic programming (a la C++)
- How can we enjoy the benefits of type classes in OO?
- Scala's answer: Standard class system + Implicits

Type Classes (Haskell)

First Role: Requirements on types

$$\textbf{class } Ord\ a \textbf{ where}$$
$$\leqslant \quad :: a \to a \to Bool$$

$$\textbf{instance } (Ord\ a, Ord\ b) \Rightarrow Ord\ (a, b) \textbf{ where}$$
$$(xa, xb) \leqslant (ya, yb) = xa \leqslant ya \vee (xa \equiv ya \wedge xb \leqslant yb)$$

Type Classes

Second Role: Implicit arguments

$$sort :: Ord\ a \Rightarrow [a] \rightarrow [a]$$

$$Prelude > List.sort\ [(3,3),(2,4),(3,4)]$$
$$[(2,4),(3,3),(3,4)]$$

Lightweight OO Approach

- Use existing class system for doing type classes
  - This covers the first role of type classes
- Add implicits:
  - This covers the second role of type classes

Scala's Class System

- Traits : Like interfaces but allowing for a disciplined form of multiple inheritance.
- Classes: Similar to Java classes
- Objects: Singleton instances

Ordering Type Class

traits describe type classes

$$\textbf{trait}\ Eq[T]\ \{$$
$$\quad \textbf{def}\ equal\ (a:T, b:T):Boolean$$
$$\}$$

$$\textbf{trait}\ Ord[T]\ \textbf{extends}\ Eq[T]\ \{$$
$$\quad \textbf{def}\ compare\ (a:T, b:T):Boolean$$
$$\quad \textbf{def}\ equal\ (a:T, b:T):Boolean =$$
$$\quad\quad compare\ (a, b) \wedge compare\ (b, a)$$
$$\}$$

Ordering Instances

```
class IntOrd extends Ord[Int] {
  def compare (a:Int, b:Int) = a <= b
}

class ListOrd[T] (ordD:Ord[T]) extends Ord[List[T]] {
  def compare (l1:List[T], l2:List[T]) =
    (l1, l2) match {
      case (x::xs, y::ys) =>
        if (ordD.equal (x,y)) compare (xs,ys)
        else ordD.compare (x,y)
      case (_, Nil)       => false
      case (Nil, _)       => true
    }
}
```

classes/objects model instances

Using Ordering

$$\textbf{def}\ sort\,[T]\ (xs : List\,[T])\ (ordT : Ord\,[T]) : List\,[T] = \ldots$$

$$\textbf{val}\ l1 = List\ (7,2,6,4,5,9)$$

$$\textbf{val}\ l2 = List\ (2,3)$$

Inconvenient!

$$\textbf{val}\ test = \textbf{new}\ ListOrd\ (\textbf{new}\ IntOrd\ ()).compare\ (l1,l2)$$

$$\textbf{val}\ test2 = \textbf{new}\ ListOrd2\ (\textbf{new}\ IntOrd\ ()).compare\ (l1,l2)$$

$$\textbf{val}\ test3 = sort\ (l1)\ (\textbf{new}\ ListOrd\ (\textbf{new}\ IntOrd\ ()))$$

Implicit Orderings

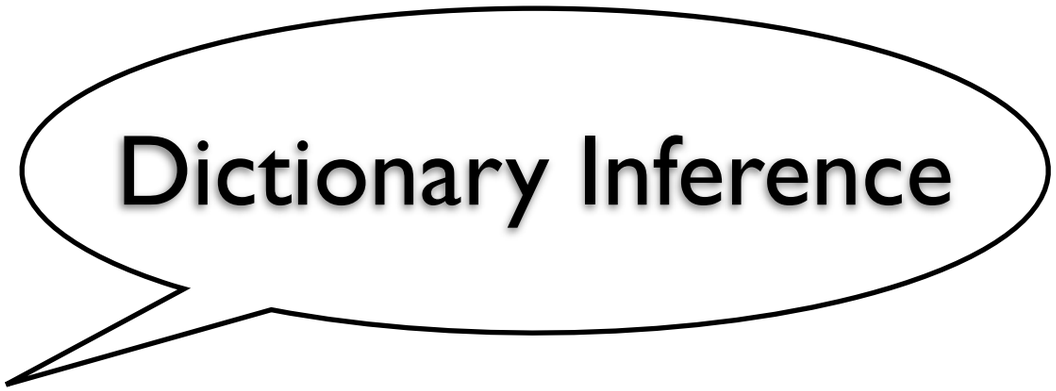
Implicits for automatically providing instances

$$\textbf{implicit}\ \textbf{val}\ \mathit{IntOrd} = \textbf{new}\ \mathit{Ord}\,[\mathit{Int}]\ \{\ldots\}$$

$$\textbf{implicit}\ \textbf{def}\ \mathit{ListOrd}\,[T]\ (\textbf{implicit}\ \mathit{ordD}{:}\,\mathit{Ord}\,[T]) =$$
$$\quad \textbf{new}\ \mathit{Ord}\,[\mathit{List}\,[T]]\ \{\ldots\}$$

$$\textbf{def}\ \mathit{cmp}\,[T]\ (x:T, y:T)\ (\textbf{implicit}\ \mathit{ord}:\mathit{Ord}\,[T]) =$$
$$\quad \mathit{ord}.\mathit{compare}\ (x, y)$$

Using Ordering



$$\textbf{val } test \;\; = cmp\ (l1, l2)$$
$$\textbf{val } test2 = cmp\ (l1, l2)\ (ListOrd2)$$
$$\textbf{val } test3 = sort\ (l1)$$

Generic Programming

Generic Programming

- Garcia et al. (JFP 2007) identified a set of criteria for language support for generic programming.
- Haskell type classes are a good mechanism to model generic programming concepts.
- How does Scala fare?

Generic Programming in the Large

	C++	*SML*	*OCaml*	*Haskell*	*Java*	*C#*	*Cecil*	*C++0X*	*G*	*JavaGI*	*Scala*
Multi-type concepts	-	●	○	●	●[2]	●[2]	◐	●	●	●	●[2]
Multiple constraints	-	◐	◐	●	●	●	●	●	●	●	●
Associated type access	●	●	◐	●	◐	◐	◐	●	●	◐	●[1]
Constraints on assoc. types	-	●	●	●	◐	◐	●	●	●	◐	●[1]
Retroactive modelling	-	●	●	●	◐[2]	◐[2]	●	●	●	●	●[23]
Type aliases	●	●	●	●	○	○	○	●	●	○	●
Separate compilation	○	●	●	●	●	●	◐	○	●	●	●
Implicit arg. deduction	●	○	●	●	◐[5]	◐[5]	◐	●	●	●	●[3]
Modular type checking	○	●	◐	●	●	●	◐	◐	●	◐	●
Lexically scoped models	○	●	○	○	○	○	○	○	●	○	●
Concept-based overloading	●	○	○	○	○	○	●	●	◐	○	◐[4]

Figure 11. Level of support for generic programming in several languages. Key: ●='good', ◐='sufficient', ○='poor' support. The rating "-" in the C++ column indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported. Notes: 1) supported via type members and dependent method types 2) supported via the CONCEPT pattern 3) supported via implicits 4) partially supported by prioritised overlapping implicits 5) decreased score due to the use of the CONCEPT pattern

Related Work

- JavaGI (Wehr et al., ECOOP 07)
  - Generalized Interfaces - type class
  - Generalized Interface implementation - type class instance
- C++0X concepts (Gregor et al., OOPSLA 06)
  - Concept declarations - type class
  - Models - type class instances

Conclusions

- A Lightweight OO approach to type classes.
- Implicits: Simple and useful mechanism
  - Could be ported to other languages
- Associated types through type members and dependent method types

Questions?

Implicits vs Haskell Type Classes

- Advantages of Implicits:
  - Local Scoping
  - Implicit arguments can be explicitly passed
  - Any values can be implicit (first-classness)
- Advantages of Type Classes
  - A bit less syntactic overhead
  - No imposed ordering of constraints