# Implicit Programming

Bruno C. d. S. Oliveira
(ongoing work jointly with Wonchan Lee, Wontae Choi,
Tom Schrijvers and Kwangkeun Yi)

# What is it?

Implicit programming is a programming style that relies on the compiler to infer certain pieces of code in programs.

# Relation to Type-Inference

- Type-inference help us by inferring types
- Implicit programming uses types to infer code.

# Related Mechanisms

- Two closely related and useful language mechanisms:
  - Type Classes (Haskell and other languages)
  - Implicits (Scala)

# Example
## (Without Implicit Programming)

# Sorting

A sorting function parametrized by a comparison function:

$$\textbf{data}\ Ordering = LT \mid EQ \mid GT$$
$$sort :: (a \to a \to Ordering) \to [a] \to [a]$$

# Comparison functions

$$cmpInt :: Int \to Int \to Ordering$$
$$cmpInt\ x\ y = compare\ x\ y$$

$$cmdPair :: (a \to a \to Ordering) \to (b \to b \to Ordering) \to$$
$$(a, b) \to (a, b) \to Ordering$$
$$cmdPair\ ca\ cb\ (x_1, y_1)\ (x_2, y_2) =$$
$$\textbf{case}\ ca\ x_1\ x_2\ \textbf{of}$$
$$LT \to LT$$
$$GT \to GT$$
$$EQ \to cb\ y_1\ y_2$$

# Client code

$$program1 :: [Int]$$
$$program1 = sort\ \boxed{cmpInt}\ [3,2,4]$$
$$program2 :: [(Int, Int)]$$
$$program2 =$$
$$\quad sort\ \boxed{(cmdPair\ cmpInt\ cmpInt)}\ [(2,3),(4,1),(2,2)]$$

# Example

## (With Implicit Programming - Type Classes)

# Sorting

- Type classes can be used for constraining generic (parametric polymorphic) functions

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

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

# Type Classes

type class

$$\textbf{class}\ Ord\ a\ \textbf{where}$$
$$compare :: a \to a \to Ordering$$

type class instance

$$\textbf{instance}\ (Ord\ a, Ord\ b) \Rightarrow Ord\ (a, b)\ \textbf{where}$$
$$compare\ (x_1, y_1)\ (x_2, y_2) = \textbf{case}\ compare\ x_1\ x_2\ \textbf{of}$$
$$LT \to LT$$
$$GT \to GT$$
$$EQ \to compare\ y_1\ y_2$$

# Code Inference

$$sort :: \boxed{Ord\ a \Rightarrow}\ [a] \rightarrow [a]$$

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

Inferred by the compiler

$$cmdPair\ cmpInt\ cmpInt$$

# Example

## (With Implicit Programming - Scala Implicits)

# Implicits

Sorting with an implicit argument:

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

Using implicits:

$$sort\ (l_1)$$

$$sort\ (List\ (3,2,1))\ (mySpecialOrd)$$

# Towards a calculus for Implicit Programming

- Why?
  - Type classes are well-studied, but have limited expressiveness;
  - Scala implicits are not formalized;
  - Understanding the essence of implicit programming: type-directed instantiation of code

# Type Classes vs Implicits

| | Type Classes | Implicits | What we want |
|---|---|---|---|
| Scoping | Global | Local | Local |
| Implicit overriding | No | Yes | Yes |
| First-class types | No | Yes | Yes |
| Constraints semantics | Set | List | Set* ? |
| Type-inference | Yes | No | Yes |
| Formalized | Yes | No | Yes |

# Current Status

- Draft operational semantics; type system and translation semantics
- Operational semantics is tricky because we need to use types
- Wonchan Lee will talk more about this

# Conclusion

- Implicit programming is useful
  - lots of applications in the Haskell and Scala community
- Type classes and Scala implicits have trade-offs (no mechanism subsumes the other)
- We believe there is a more general mechanism that generalizes both