The Implicit Calculus
A New Foundation for Generic Programming

Bruno C. d. S. Oliveira\(^1\) (presenter), Tom Schrijvers\(^2\), Wontae Choi\(^1\), Wonchan Lee\(^1\), Kwangkeun Yi\(^1\)

\(^1\)Seoul National University, \(^2\)Universiteit Gent
Introduction

• Several **generic programming** (GP) mechanisms:
  • Haskell type classes (several formal models)
  • C++0x concept proposals (some formal models)
  • Scala implicits (no formal model)

• This work: A **formal model for implicits**

• Why? Implicits **add expressiveness** and are at the same time **simpler** than other GP mechanisms.
Generic Programming

- Abstracting algorithms from specific types
- Abstraction achieved via parametrization
- Implicit instantiation of generic parameters
Generic Programming

A generic sorting algorithm on Lists

Type parameter

\[ \text{sort}<A> : \text{Ord}<A> \Rightarrow \text{List}<A> \rightarrow \text{List}<A> \]

Constraint: elements of type A must be orderable!
Using generic sorting:

sort [3,1,2]       // [1,2,3]
sort ['c','a','b']  // ['a','b','c']
sort [[2,3],[1,5]]  // [[1,5],[2,3]]

Both the type parameter and the constraint are implicitly instantiated (or inferred).
Goal

- A model of GP mechanisms (inspired by Scala implicits)
- Minimal formal calculus (language agnostic)
- Useful for language designers wanting to implement implicits in their own language
The Implicit Calculus
The Implicit Calculus

• Models 2 fundamental mechanisms:
  1. (type-directed) resolution of rules
  2. scoping of (implicit) rules

• Implicit instantiation recovered in source languages

• Concepts and type-classes tangle resolution and implicit instantiation
Inspired by Logic Programming:

- Queries for values of a certain type
- Type-directed rules to derive facts (values)
- Rule environment to collect rules
I: Resolution

A simpler generic sort first:

`sort<A> : Ord<A> → List<A> → List<A>`

`interface Ord<A> {`

  `(==) : A → A → Bool`

  `<) : A → A → Bool`

`}`
I: Resolution

Resolution (inference of constraints)

?\(\text{Ord\langle Int\rangle}\) \Rightarrow \text{ordInt}

Rule environment

\text{ordInt: Ord\langle Int\rangle}

Query

\text{sort\langle Int\rangle} ?\(\text{Ord\langle Int\rangle}\) \ [3,1,2]
Recursive Resolution

Resolution (inference of constraints)

?((\text{Ord\textunderscore List\textunderscore Int})) \Rightarrow
\text{ordList}(\text{ordInt})

Query

\text{sort\textunderscore List\textunderscore Int} ?((\text{Ord\textunderscore List\textunderscore Int})) \{[2,3],[1,5]\}

Rule environment

\text{ordInt: Ord\textunderscore Int}
\text{ordList:} \forall \text{ A. Ord\textunderscore A} \Rightarrow \text{Ord\textunderscore List\textunderscore A}
2: Scoping

Inspired by conventional $\lambda$-binders:

- **Lexical and local scoping**
- **Rule abstractions** define rules
- **Rule applications** apply rules
2: Scoping

Another version of generic sort:

$$\text{sort}<A> : \text{Ord}<A> \Rightarrow \text{List}<A> \Rightarrow \text{List}<A>$$
2: Scoping

let ordInt = (l ... : Ord<Int> l) in
implicit {ordInt} in
sort<Int> with {?(Ord<Int>)} [3,1,2]

Rule (abstraction)

Extending environment

Rule application
The Implicit Calculus

(Simple) Types \( \tau ::= \alpha \mid \text{Int} \mid \tau_1 \to \tau_2 \mid \rho \)

Rule Types \( \rho ::= \forall \vec{\alpha}. \rho \Rightarrow \tau \)

Expressions \( e ::= \ n \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \)

\[ \mid ?\rho \mid (| e : \rho |) \mid e[\vec{\tau}] \mid e \ 	ext{with} \ e : \rho \]

Syntactic Sugar:

\[ \text{implicit} \ e : \rho \ 	ext{in} \ e_1 : \tau \stackrel{\text{def}}{=} (| e_1 : \rho \Rightarrow \tau |) \ 	ext{with} \ e : \rho \]
Source Languages

From source:

\texttt{sort \ [[2,3],[1,5]]}

To core:

\texttt{\textit{sort<List<Int>>\ with\ \{?(Ord<List<Int>>)}\ [[2,3],[1,5]]}}

Conventional type-inference

query (more type-) inference
Implicit Instantiation

Implicit instantiation = resolution + (type-)inference
More in the paper

- Type System
- Elaboration semantics to System F
- Type-directed translation from source language to the Implicit calculus
- Higher-order rules and partial resolution
Comparison

- Concepts and Type Classes
  - Special interfaces for constraints
  - Implicit instantiation only for those interfaces

- Implicits
  - Implicit (and explicit) instantiation for any types
  - Constraints are just regular types
  - A general mechanism for type-directed implicit parameter passing
The following definitions:

**Constraint used as a type!**

\[ \text{sort}<A>: \text{Ord}<A> \rightarrow \text{List}<A> \rightarrow \text{List}<A> \]

**Type used as a constraint!**

\[ \text{log}: \text{PrintStream} \Rightarrow \text{String} \rightarrow () \]

are valid in a system with implicits, but invalid with type classes or concepts!
Conclusion

• Implicit calculus: *Simple formal model for GP*

• **Decoupling of various mechanisms** in existing GP mechanisms

• **Resolution and implicit instantiation** for any types
Thank You!

Questions?
2: Scoping

Rule (abstraction)

let ordList = (\...: \forall A. Ord<A> \Rightarrow Ord<List<A>> |) in
let ordInt = (\...: Ord<Int> |) in
implicit {ordInt, ordList} in
sort<List<Int>> with {?(Ord<List<Int>>)} [[2,3],[1,5]]
Haskell

- Type classes are predicates on types
- Global Scoping
- Not possible to override compiler choice
System FG

- Concepts are predicates on types
- Local Scoping
- Not possible to override compiler choice
Scala

- Type-classes/concepts are types
- Local scoping
- Overriding is possible
The Implicit Calculus

• Type-classes/concepts are types
• Local scoping
• Overriding is possible
• Higher-order rules