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

UNIVERSITEIT GENT

ROSAEC center
Research On Software Analysis for Error-free Computing
http://rosaec.snu.ac.kr

Programming Research Laboratory
http://ropas.snu.ac.kr

SEOUL NATIONAL UNIVERSITY

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

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

Constraint: elements of type A must be orderable!

Generic Programming

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

constraints

- Implicit instantiation recovered in source languages
- Concepts and type-classes tangle resolution and implicit instantiation

I: Resolution

Inspired by Logic Programming:

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

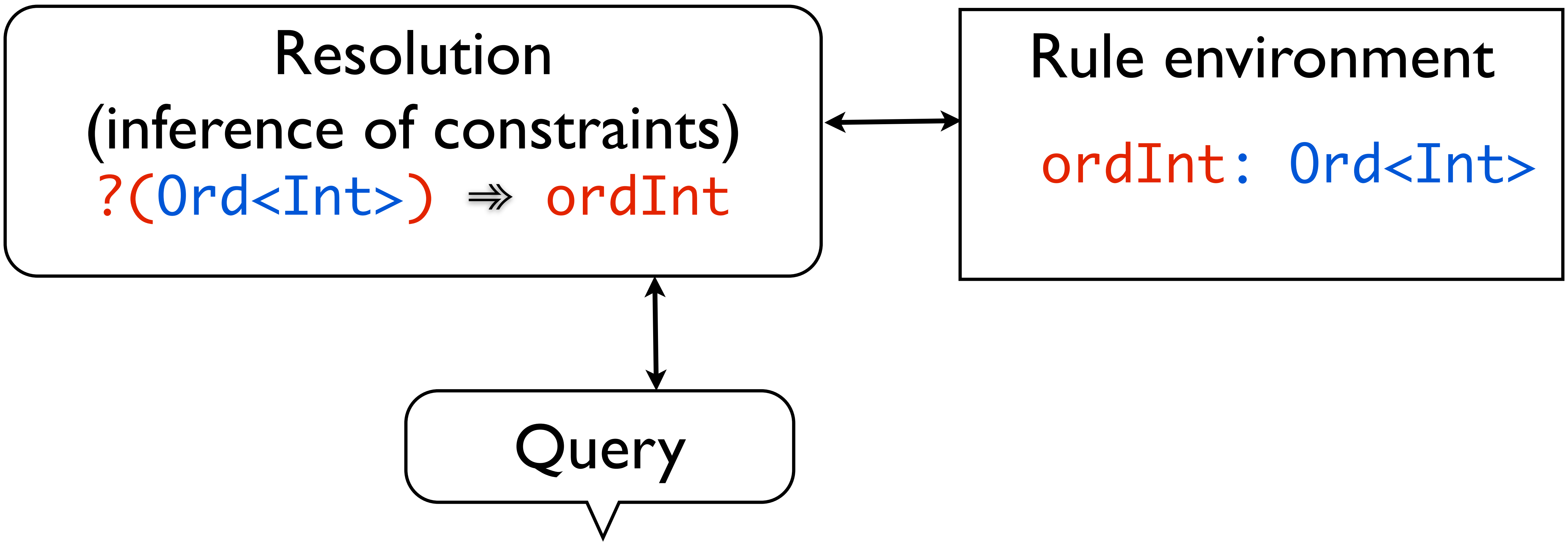
1: Resolution

A simpler generic sort first:

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

interface Ord<A> {
  (==) : A → A → Bool
  (<)  : A → A → Bool
}
```

1: Resolution




```
sort<Int> ?(Ord<Int>) [3,1,2]
```

1: Recursive Resolution

Resolution
(inference of constraints)

```
?(Ord<List<Int>>) ⇛
ordList(ordInt)
```

↔

Rule environment

```
ordInt: Ord<Int>
ordList:∀ A. Ord<A>
⇒ Ord<List<A>>
```

↕

Query

```
sort<List<Int>> ?(Ord<List<Int>>) [[2,3],[1,5]]
```

2: Scoping

Inspired by conventional λ-binders:

- Lexical and local scoping
- Rule abstractions define rules
- Rule applications apply rules

2: Scoping

Another version of generic sort:

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

2: Scoping

Rule (abstraction)

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

Extending environment

Rule application

The Implicit Calculus

(Simple) Types	τ	$::=$	$\alpha \mid Int \mid \tau_1 \rightarrow \tau_2 \mid \rho$
Rule Types	ρ	$::=$	$\forall \vec{\alpha}.\bar{\rho} \Rightarrow \tau$
Expressions	e	$::=$	$n \mid x \mid \lambda x : \tau . e \mid e_1\ e_2$
		$\mid$	$?\rho \mid (\!\mid e : \rho \mid\!) \mid e[\vec{\tau}] \mid e\ \mathbf{with}\ \overline{e : \rho}$

Syntactic Sugar:

$$\mathbf{implicit}\ \overline{e : \rho}\ \mathbf{in}\ e_1 : \tau \stackrel{\text{def}}{=} (\!\mid e_1 : \bar{\rho} \Rightarrow \tau \mid\!)\ \mathbf{with}\ \overline{e : \rho}$$

Source Languages

From source:

```
sort [[2,3],[1,5]]
```

query (more type-) inference

To core:

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

Conventional 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

Comparison

The following definitions:

Constraint used as a type!

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

log: PrintStream ⇒ String → ()
```

Type used as a constraint!

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=(|...:∀ A. Ord<A> ⇒ 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