Polymorphic Symmetric Multiple Dispatch with Variance

Gyunghee Park, Jaemin Hong, Guy L. Steele Jr., and Sukyoung Ryu

Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)

1 Background

1.1 Method Overloading

- Allows defining multiple methods with the same name.

```
add(x: Int, y: Int): Int = …
add(x: List, y: List): List = …
add(x: SortedList, y: List): SortedList = …
add(x: SortedList, y: SortedList): SortedList = …
```

- A method dispatch mechanism chooses a method to invoke.

```
a: List = SortedList(1, 3, 5)
b: List = SortedList(2, 4, 6)          (SortedList <: List)
add(a, b)
```

1.2 Static Dispatch

- Selects a method based on the compile-time types of arguments.

```
add(x: List, y: List): List = …
```

1.3 Single Dispatch

- Considers the run-time type of a single argument, usually a receiver.

```
add(x: SortedList, y: List): SortedList = …
```

- Is widely used in modern object-oriented languages.
- Causes the binary method problem.

1.4 Multiple Dispatch

- Considers the run-time types of more than one argument.
- Can be either symmetric or asymmetric.

```
add(x: SortedList, y: SortedList): SortedList = …
```

- Provides an intuitive solution for the binary method problem.

2 Motivation

2.1 The Goals of Method Dispatch

- Every method invocation should be unambiguous.
- The most specific declaration exists among applicable declarations.

```
add(x: List, y: List): List = …
add(x: SortedList, y: List): SortedList = …
add(x: SortedList, y: SortedList): SortedList = …
```

(more specific)

- Every method invocation should be type-sound.
- A dynamically chosen method preserves the return type of a statically chosen method.

```
add(x: List, y: List): List = …
add(x: SortedList, y: SortedList): SortedList = …
```

2.2 Issues in Symmetric Multiple Dispatch

- Run-time types are unknown at compile time.
- Cannot check unambiguity and type preservation statically.

```
add(x: List, y: SortedList): SortedList = …
add(x: SortedList, y: List): SortedList = …

a: SortedList = SortedList(1, 3, 5)
b: List = SortedList(2, 4, 6)
add(a, b)
```

- At compile time, it is unambiguous, but not at run time.
- Needs a disambiguating declaration.

```
add(x: SortedList, y: SortedList): List = …
```

- It does not preserve static return type.
- Needs a fix for the return type.

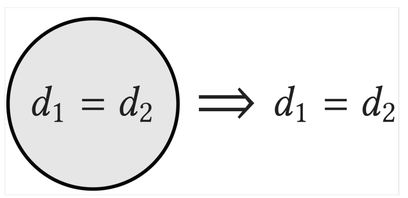
```
add(x: SortedList, y: SortedList): SortedList = …
```

- We can check validities of overloaded method sets statically.

3 Overloading Rules

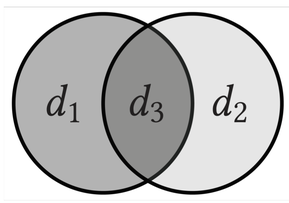
3.1 No Duplicate Rule

- Rules out duplicate signatures.



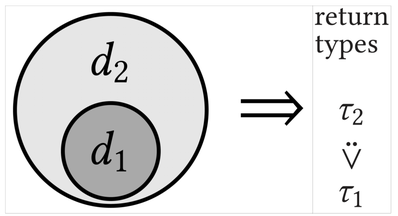
3.2 Meet Rule

- Ensures existence of the most specific declaration.



3.3 Return Type Rule

- Ensures type preservation.



4 Additional Issues in the Paper

4.1 Parametric Polymorphism

- Parametrizes classes and methods by types.

```
add[P](x: List[P], y: List[P]): List[P] = …
```

- We introduce quantified types to formally define "more specific".
- Variance defines additional subtype relation for parametric types.
  - If `T` is covariant, then `A <: B → T[A] <: T[B]`.
  - If `T` is contravariant, then `A <: B → T[B] <: T[A]`.

4.2 Dynamic Dispatch Algorithm

- Determines a method to invoke.
- Infers type arguments for parametric methods.
- We formally defined the algorithm and proved its correctness.

이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. NRF- 2017R1A2B3012020 와 2017M3C4A7068177).

KAIST

PLRG Programming Language Research Group