UMassAmherst

Refactoring Java Generics by Inferring Wildcards, In Practice

John Altidor
&
Yannis Smaragdakis

Outline

- Background on variance
- Motivate adding Java wildcards to code
- Refactoring tool for adding Java wildcards
  - Using definition-site variance inference
  - Type influence flow analysis
    - Generalizing partial interfaces
- Case-study refactoring six, large, popular Java libraries (e.g. Oracle JDK, Guava, Apache, etc.)
  - How general can interfaces be without access to the source code of 3rd party libraries?
- Summary

Generics – Parametric Polymorphism

type parameter

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

- `List<Animal> ≡ ListOfAnimals`
- `List<Dog> ≡ ListOfDogs`

Subtype (is-a) relationships for generics

- Consider an **Animal** class hierarchy
  - **Dog** **<u>is an</u>** **Animal** (Dog **extends** Animal)
  - **Cat** **<u>is an</u>** **Animal** (Cat **extends** Animal)
- **List<Dog>** <u>is a</u> **List<Animal>**?

No!

- Can add a **Cat** to a **List<Animal>**
- <u>Cannot</u> add a **Cat** to a **List<Dog>**

Variance Introduction

- When is **C<***Expr1***>** a subtype of **C<***Expr2***>**?
- Variance allows **two different instantiations** of a generic to be subtype-related
- Supports more reusable software: Apply one piece of code to multiple instantiations

Variance Introduction

- When is `C<`*Expr1*`>` a subtype of `C<`*Expr2*`>`?

```
class List<X>
{
   void add(X x) { … }

   X get(int i) { … }

   int size() { … }
}
```

- **`List<Dog>`** <u>is not a</u> **`List<Animal>`**

Variance Introduction

- When is `C<`*`Expr1`*`>` a subtype of `C<`*`Expr2`*`>`?

```
class RList<X>
{
   X get(int i) { … }

   int size() { … }
}
```

Can **read** from but not **write** to

Yes!

- `RList<Dog>` <u>is a</u> `RList<Animal>`?

Variance Introduction

- When is **C<*Expr1*>** a subtype of **C<*Expr2*>**?

```
class RList<X>
{
   X get(int i) { … }

   int size() { … }
}
```

Can **read** from but not **write** to

- **RList<Dog>** is a **RList<Animal>**
- **RList<Dog>** can do everything **RList<Animal>** can do
- Java wildcards enable variant subtyping

Use-Site Variance (Java Wildcards)

```
class List<X>
{
   void add(X x) { … }

   X get(int i) { … }

   int size() { … }
}
```

Use-Site Variance (Java Wildcards)

```
class List<X>
{
   void add(X x) { … }

   X get(int i) { … }

   int size() { … }
}
```

List<? extends X>

covariant
version of List:
List<Dog>
<u>is a</u>
List<? extends Animal>

Use-Site Variance (Java Wildcards)

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

```
List<? super X>
```

contravariant
version of List:
List<Animal>
is a
List<? super Dog>

Use-Site Variance (Java Wildcards)

```
class List<X>
{
   void add(X x) { … }

   X get(int i) { … }

   int size() { … }
}
```

List<?>

bivariant
version of List:
List<T> <u>is a</u> List<?>,
for any T

What Is This Paper About?

- Refactoring tool to automatically add wildcards
  - Result is a more reusable interface
  - Code can be applied to multiple instantiations
- Users can select a subset of declarations (e.g., variables, method arguments, return types) to generalize their types
  - Allows programmers to choose where to add wildcards

Client of Generic Class Without Wildcards

```
void performSpeak(
        List<Animal> list)
{
  Animal animal =
    list.get(0);
  animal.speak();
}
```

Can only be a `List<Animal>`.

Client of Generic Class Without Wildcards

```
void performSpeak(
        List<Animal> list)
{
  Animal animal =
    list.get(0);
  animal.speak();
}
```

The `add` method is not invoked on `list`.

Client of Java Wildcards

```
void performSpeak(
        List<? extends Animal> list)
{
  Animal animal =
    list.get(0);
  animal.speak();
}
```

Can be a `List<Dog>`, `List<Cat>`, or list of any subclass of `Animal`.

Client of Java Wildcards

```
void performSpeak(
        List<? extends Animal> list)
{
  Animal animal =
    list.get(0);
  animal.speak();
}
```

Rewrite performed by refactoring tool

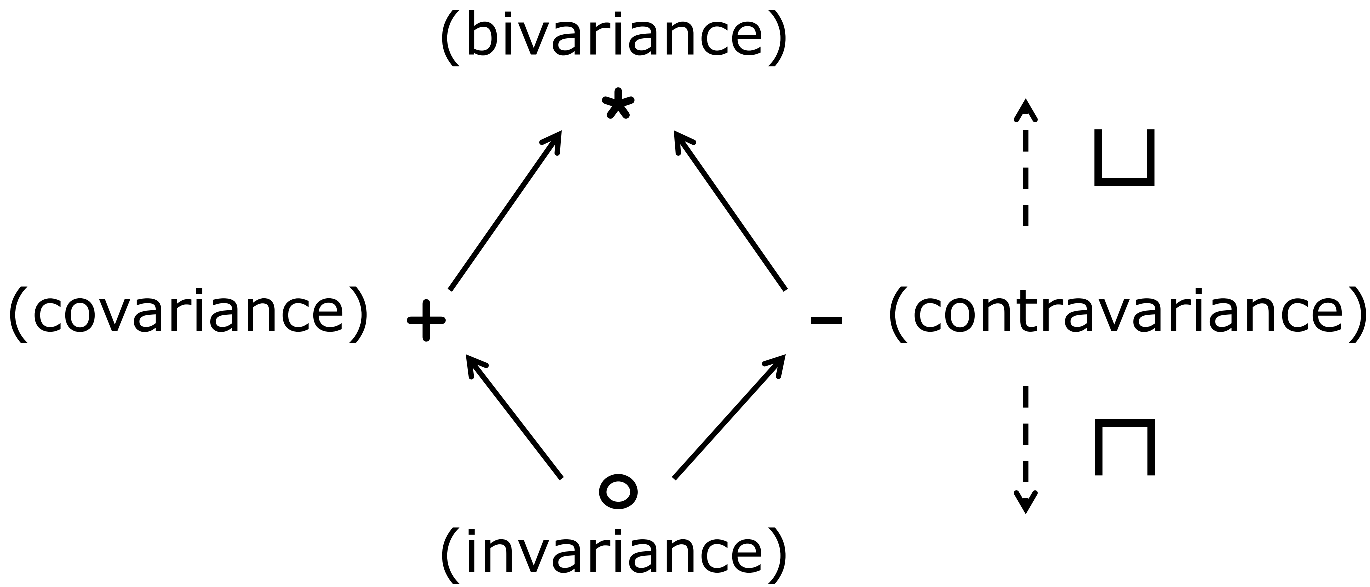
How Do We Generalize Variances?

- Every generic type parameter has a maximum inherent (“definition-site”) variance
  - We saw: **`RList<X>`** inherently covariant

How Do We Generalize Variances?

- Every generic type parameter has a maximum inherent ("definition-site") variance
  - We saw: **`RList<X>`** inherently covariant
- Our prior work: definition-site variance inference for Java [PLDI'11]
- Example inferences:
  - **`java.util.Iterator<E>`** is covariant
  - **`java.util.Comparator<T>`** is contravariant
- In theory, can perform substitution:
  `Iterator<T>` ➔ **`Iterator<? extends T>`**
- Topic of this work: problems in practice!

Variances "Too Small" are Refactored



- Order stems from subtyping:

  $\mathbf{v} \leq \mathbf{w} \Longrightarrow$ `C<vT>` is a `C<wT>`
- "Too small" defined in next slide

More General Refactoring Via [PLDI'11]

- Every type argument is annotated with a variance
  - `Iterator<-Animal> = Iterator<? super Animal>`
  - `Iterator<oAnimal> = Iterator<Animal>`
- Let v_d be the definition-site variance of generic **C**
- <u>More general refactoring</u>:

  C<**vT**> ➔ C<**(v ⊔ v_d) T**>
- Ex: **`Iterator<? super T>`** ➔ **`Iterator<?>`**, since **- ⊔ + = ***.
  - Contravariant use-site annotation removes covariant part
  - **`Iterator`** is covariant (only has covariant part)
  - Only the bivariant part of **`Iterator`** is left over

Why Is This Hard in Practice?

- Cannot generalize all wildcards
- Changing one type may require updating others
- Source code from 3rd party libraries may not be available for refactoring
- Programmer may not want all code to be refactored:
  Users can select a subset of declarations (e.g., variables, method arguments, return types) to generalize
- Preserving original program semantics
  (e.g. method overrides)

Complication 1: Type Influence Flow Analysis

Iterator<? extends T>

```
void foo(Iterator<T> arg) {
  Iterator<T> itr = arg;
  bar(itr);
}

void bar(Iterator<T> arg2);
```

FlowsTo(x) =
set of decls
influenced by x

- Edges between decls signal type influence
- FlowsTo(x) = set of decls reachable from x in type influence graph
- FlowsTo(**arg**) = { **itr**, **arg2** }

Complication 2: Dependencies To Reflect Variance

```
interface C<X> { void foo(D<X> arg); }
interface D<Y> { int getNumber(); }
class Client {
  void bar(C<String> cstr,
           D<String> dstr) {
    cstr.foo(dstr);
} }
```

- Interfaces **C** and **D** are both <u>bivariant</u>
  - Type parameter **Y** does not appear in definition of **D**
- User selected generalizing type of **cstr**

Complication 2: Dependencies To Reflect Variance

```
interface C<X> { void foo(D<X> arg); }
interface D<Y> { int getNumber(); }
class Client {
  void bar(C<?> cstr,
           D<String> dstr) {
    cstr.foo(dstr);
} }
```

Type Rewrite

- Interfaces **C** and **D** are both <u>bivariant</u>
  - Type parameter **Y** does not appear in definition of **D**
- User selected generalizing type of **cstr**

UMassAmherst

Complication 2: Dependencies To Reflect Variance

```
interface C<X> { void foo(D<X> arg); }
interface D<Y> { int getNumber(); }
class Client {
  void bar(C<?> cstr,
           D<String> dstr) {
    cstr.foo(dstr);
} }
```

Type Rewrite

```
foo(D<capture#274 of ?>) in C<capture#274 of ?>
cannot be applied to (D<String>)
```

- Resulting error message above
- Unknown type in `C<?>` not in `D<String>`

Complication 2: Dependencies To Reflect Variance

Type Rewrite

```
interface C<X> { void foo(D<?> arg); }
interface D<Y> { int getNumber(); }
class Client {
  void bar(C<?> cstr,
           D<String> dstr) {
    cstr.foo(dstr);
} }
```

✓

- Needed to perform rewrite to allow bivariant use of interface **C**

Complication 3: Not All Source Code Available

Iterator<? extends T>

```
void foo(Iterator<T> arg) {
   thirdPartyFunction(arg);
}

void thirdPartyFunction(Iterator<T> arg2);
```

Source code not available

- Generalizing type of `arg` causes compiler error

More Wildcards via Method Body Analysis

- Use-site annotation greater than definition-site variance may suffice for a method
- **`List`** is invariant. **`Iterator`** is covariant.

```
Animal first(List<Animal> l) {
  Iterator<Animal> itr =
    l.iterator();
  return itr.next();
}
```

- Not all methods from **`List`** invoked on argument **`l`**

More Wildcards via Method Body Analysis

- Use-site annotation greater than definition-site variance may suffice for a method
- `List` is invariant. `Iterator` is covariant.

```
Animal first(List<? extends Animal> l) {
  Iterator<? extends Animal> itr =
    l.iterator();
  return itr.next();
}
```

- Not all methods from `List` invoked on argument `l`

Case Study: Rewrite Analysis Over Java Libs

- How general can interfaces be without access to the source code of 3rd party libraries?
- How many declarations (e.g., variables, method arguments) can be rewritten w/ more general wildcards?
- How many declarations require updates if one declaration is rewritten (on average)?
- Analyzed six large libraries written by experts
  - Oracle JDK, Guava, Apache Collections, JScience, …
  - We expect rewritten percentage to be higher for other libraries

Statistics Over All Generic Type Appearances:
"How Many Type Expressions Are Too Conservative?"

Library		# P-Decls	Rewritable	Rewritten
	classes	4900	87%	12%
Java	interfaces	170	90%	12%
	total	5070	88%	12%
	classes	1553	67%	14%
JScience	interfaces	56	95%	77%
	total	1609	68%	16%
…	…	…	…	…
	classes	**17907**	**75%**	**11%**
Total	**interfaces**	**352**	**89%**	**19%**
	total	**18259**	**76%**	**11%**

- 11% of parameterized decls can be generalized

Statistics Over All Generic Type Appearances:

"How Many Type Expressions Are Too Conservative?"

Library		**# P-Decls**	**Rewritable**	**Rewritten**
	classes	4900	87%	12%
Java	interfaces	170	90%	12%
	total	5070	88%	12%
	classes	1553	67%	14%
JScience	interfaces	56	95%	77%
	total	1609	68%	16%
…	…	…	…	…
	classes	**17907**	**75%**	**11%**
Total	**interfaces**	**352**	**89%**	**19%**
	total	**18259**	**76%**	**11%**

- 11% of parameterized decls can be **more reusable**

Statistics Over Appearances of Variant Types:
"Benefit in Actual Code, When Theoretically Possible"

Library		# V-Decls	Rewritable	Rewritten
	classes	1115	67%	50%
Java	interfaces	47	66%	43%
	total	1162	67%	50%
	classes	717	49%	30%
JScience	interfaces	51	94%	84%
	total	768	52%	34%
…	…	…		
	classes	**5555**	**60%**	**35%**
Total	**interfaces**	**115**	**77%**	**57%**
	total	**5670**	**60%**	**35%**

- 35% of variant decls can be more reusable

FlowsTo Set Sizes:
"Could One Do Manually What Our Tool Does?"

Library		# P-Decls	FlowsTo Avg. Size	FlowsTo-R Avg. Size
	classes	4900	61.10	1.23
Java	interfaces	170	39.91	2.75
	total	5070	60.39	1.29
	classes	1553	52.04	5.42
JScience	interfaces	56	10.21	0.66
	total	1609	50.59	5.19
…	…	…	…	…
	classes	**17907**	**139.21**	**1.28**
Total	**interfaces**	**352**	**58.36**	**2.23**
	total	**18259**	**137.65**	**1.30**

- Manual refactoring too tedious and error prone

Contributions

- Refactoring tool
  - Generalizes interfaces by adding wildcards
  - Infers definition-site variance
- Type Influence Flow Analysis
  - Users select a subset of declarations to generalize
  - Optimizations to generalize more types
- Method Body Analysis
  - Add wildcards to uses of invariant types
- Soundness of Refactoring (in paper)
  - Refactoring preserves ability to perform operations
    - Safe to assume: `C<(v ⊔ v`$_{def}$`)T>` <u>is a</u> `C<vT>`
    - Refactored Type <u>is an</u> Original Type