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

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 **is an** Animal (Dog extends Animal)
  - Cat **is an** Animal (Cat extends Animal)

- **List<Dog> is a List<Animal>?**
  - No!

  - Can add a Cat to a List<Animal>
  - **Cannot** add a Cat to a List<Dog>
Variance Introduction

- When is \( c<Expr_1> \) a subtype of \( c<Expr_2> \)?

- Variance allows \textbf{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 $c<Expr2>$?

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

    X get(int i) { ... }

    int size() { ... }
}
```

- `List<Dog>` is not a `List<Animal>`
Variance Introduction

- When is \( c<Expr1> \) a subtype of \( c<Expr2> \)?

```java
class RList<X>
{
    X get(int i) { ... }
    int size() { ... }
}
```

Can read from but not write to

- \( RList<Dog> \) is a \( RList<Animal> \)?

Yes!
Variance Introduction

- When is $c<Expr1>$ a subtype of $c<Expr2>$?

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

    int size() { ... }
}
```

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

Can read from but not write to
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>  
is a  
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> is a 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

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

Can only be a List<Animal>.
Client of Generic Class Without Wildcards

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

The `add` method is not invoked on `list`. 
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.
void performSpeak(
    List<? extends Animal> list)
{
    Animal animal =
        list.get(0);
    animal.speak();
}
How Do We Generalize Variances?

- Every generic type parameter has a **maximum inherent ("definition-site") variance**
  - We saw: \texttt{RList\langle X\rangle} 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> \rightarrow Iterator<? extends T>`
- Topic of this work: problems in practice!
Variances “Too Small” are Refactored

- Order stems from subtyping:
  \[ v \leq w \implies C\langle vT \rangle \text{ is a } C\langle wT \rangle \]
- “Too small” defined in next slide
More General Refactoring Via [PLDI’11]

- Every type argument is annotated with a variance
  - \texttt{Iterator\langle -\text{Animal} \rangle} = \texttt{Iterator\langle ? \ super \ Animal \rangle}
  - \texttt{Iterator\langle o\text{Animal} \rangle} = \texttt{Iterator\langle \text{Animal} \rangle}

- Let \( v_d \) be the definition-site variance of generic \( C \)

- More general refactoring:
  \[ C\langle vT \rangle \Rightarrow C\langle (v \sqcup v_d)T \rangle \]

- Ex: \texttt{Iterator\langle ? \ super \ T \rangle} \Rightarrow \texttt{Iterator\langle ? \rangle},
  since \( - \sqcup + = * \).
  - Contravariant use-site annotation removes covariant part
  - \texttt{Iterator} is covariant (only has covariant part)
  - Only the bivariant part of \texttt{Iterator} is left over
Why Is This Hard in Practice?

- Cannot generalize all wildcards
- Changing one type may require updating others
- Source code from 3\textsuperscript{rd} 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

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

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

- 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 bivariant
  - Type parameter Y does not appear in definition of D
- User selected generalizing type of cstr
Complication 2: Dependencies To Reflect Variance

```java
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);
    }
}
```

- Interfaces C and D are both **bivariant**
  - 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);
    }
}

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

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

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

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

- Generalizing type of `arg` causes compiler error

`Iterator<? extends T>`
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.

```java
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.

```java
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 with 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?”

<table>
<thead>
<tr>
<th>Library</th>
<th># P-Decls</th>
<th>Rewritable</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>4900</td>
<td>87%</td>
<td>12%</td>
</tr>
<tr>
<td>Java</td>
<td>170</td>
<td>90%</td>
<td>12%</td>
</tr>
<tr>
<td>total</td>
<td>5070</td>
<td>88%</td>
<td>12%</td>
</tr>
<tr>
<td>classes</td>
<td>1553</td>
<td>67%</td>
<td>14%</td>
</tr>
<tr>
<td>JScience</td>
<td>interfaces</td>
<td>56</td>
<td>95%</td>
</tr>
<tr>
<td>total</td>
<td>1609</td>
<td>68%</td>
<td>16%</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>classes</td>
<td>17907</td>
<td>75%</td>
<td>11%</td>
</tr>
<tr>
<td>Total</td>
<td>interfaces</td>
<td>352</td>
<td>89%</td>
</tr>
<tr>
<td>total</td>
<td>18259</td>
<td>76%</td>
<td>11%</td>
</tr>
</tbody>
</table>

- 11% of parameterized decls can be generalized
Statistics Over All Generic Type Appearances: “How Many Type Expressions Are Too Conservative?”

<table>
<thead>
<tr>
<th>Library</th>
<th># P-Decls</th>
<th>Rewritable</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>4900</td>
<td>87%</td>
<td>12%</td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interfaces</td>
<td>170</td>
<td>90%</td>
<td>12%</td>
</tr>
<tr>
<td>total</td>
<td>5070</td>
<td>88%</td>
<td>12%</td>
</tr>
<tr>
<td>classes</td>
<td>1553</td>
<td>67%</td>
<td>14%</td>
</tr>
<tr>
<td>JScience</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interfaces</td>
<td>56</td>
<td>95%</td>
<td>77%</td>
</tr>
<tr>
<td>total</td>
<td>1609</td>
<td>68%</td>
<td>16%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>classes</td>
<td>17907</td>
<td>75%</td>
<td>11%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interfaces</td>
<td>352</td>
<td>89%</td>
<td>19%</td>
</tr>
<tr>
<td>total</td>
<td>18259</td>
<td>76%</td>
<td>11%</td>
</tr>
</tbody>
</table>

- 11% of parameterized decls can be more reusable
### Statistics Over Appearances of Variant Types: “Benefit in Actual Code, When Theoretically Possible”

<table>
<thead>
<tr>
<th>Library</th>
<th># V-Decls</th>
<th>Rewritable</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>classes</td>
<td>1115</td>
<td>67%</td>
</tr>
<tr>
<td>Java</td>
<td>interfaces</td>
<td>47</td>
<td>66%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>1162</td>
<td>67%</td>
</tr>
<tr>
<td></td>
<td>classes</td>
<td>717</td>
<td>49%</td>
</tr>
<tr>
<td>JScience</td>
<td>interfaces</td>
<td>51</td>
<td>94%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>768</td>
<td>52%</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>classes</td>
<td>5555</td>
<td>60%</td>
</tr>
<tr>
<td>Total</td>
<td>interfaces</td>
<td>115</td>
<td>77%</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>5670</td>
<td>60%</td>
</tr>
</tbody>
</table>

- 35% of variant decls can be more reusable
FlowsTo Set Sizes:
“Could One Do Manually What Our Tool Does?”

<table>
<thead>
<tr>
<th>Library</th>
<th># P-Decls</th>
<th>FlowsTo Avg. Size</th>
<th>FlowsTo-R Avg. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>4900</td>
<td>61.10</td>
<td>1.23</td>
</tr>
<tr>
<td>Java</td>
<td>170</td>
<td>39.91</td>
<td>2.75</td>
</tr>
<tr>
<td>total</td>
<td>5070</td>
<td>60.39</td>
<td>1.29</td>
</tr>
<tr>
<td>classes</td>
<td>1553</td>
<td>52.04</td>
<td>5.42</td>
</tr>
<tr>
<td>JScience</td>
<td>56</td>
<td>10.21</td>
<td>0.66</td>
</tr>
<tr>
<td>total</td>
<td>1609</td>
<td>50.59</td>
<td>5.19</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>classes</td>
<td>17907</td>
<td>139.21</td>
<td>1.28</td>
</tr>
<tr>
<td>Total</td>
<td>352</td>
<td>58.36</td>
<td>2.23</td>
</tr>
<tr>
<td>total</td>
<td>18259</td>
<td>137.65</td>
<td>1.30</td>
</tr>
</tbody>
</table>

- 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:  \( C<(v \sqcup v_{\text{def}})T> \text{ is a } C<vT> \)
    - Refactored Type is an Original Type