Refactoring Java Generics by Inferring Wildcards, In Practice

John Altidor
University of Massachusetts
jaltidor@cs.umass.edu

Yannis Smaragdakis
University of Athens
smaragd@di.uoa.gr

Abstract

Wildcard annotations can improve the generality of Java generic libraries, but require heavy manual effort. We present an algorithm for refactoring and inferring more general type instantiations of Java generics using wildcards. Compared to past approaches, our work is practical and immediately applicable: we assume no changes to the Java type system, while taking into account all its intricacies. Our system allows users to select declarations (variables, method parameters, return types, etc.) to generalize and considers declarations not declared in available source code. It then performs an inter-procedural flow analysis and a method body analysis, in order to generalize type signatures. We evaluate our technique on six Java generic libraries. We find that 34% of available declarations of variant type signatures can be generalized—i.e., relaxed with more general wildcard types. On average, 146 other declarations need to be updated when a declaration is generalized, showing that this refactoring would be too tedious and error-prone to perform manually.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors

Keywords variance; definition-site variance; use-site variance; wildcards; generics; polymorphism; refactoring

1. Introduction

The maintainability, safety, and reliability of existing Java programs improve when libraries are refactored to define generic classes. Generics enable clients to inform the compiler of the types of elements in a collection and improve safety by eliminating unsafe casts.

Unfortunately, generics restrict subtyping, which is a key feature of object-oriented languages for enabling general, reusable functionality. A method that takes in an instance of a class `Animal`, for example, will also accept an instance of a subclass of `Animal` that does not yet exist. A method with an argument type of `List<Animal>`, however, would not accept a `List<Dog>` nor a `List` of any other subclass of `Animal` to preserve type soundness.

Variance mechanisms in modern programming languages try to address this problem by enabling two *different* instantiations of a generic to be subtype related. Given a generic type `C<X>`, when is a type-instantiation `C<`*Exp1*`>` a subtype of another type instantiation `C<`*Exp2*`>`? Variance (more precisely: subtype variance with respect to generic type parameters) is the study of this question. Variance is a key topic in language design, since it develops the exact rules governing the interplay of the two major forms of polymorphism: parametric polymorphism (i.e., generics or templates) and subtype (inclusion) polymorphism.

The Java language type system employs the concept of *use-site variance* [13]: uses of a class can choose to specify that they are referring to a *covariant*, *contravariant*, or *bivariant* version of the class. For instance, a method `void meth(C<? extends T> cx)` accepts a covariant version of `C`. The method argument can be of type `C<T>` but also `C<S>` where `S` is a subtype of `T`. An object with type `C<? extends T>` may not offer the full functionality of a `C<T>` object: the type system ensures that the body of method `meth` employs only such a subset of the functionality of `C<T>` that would be safe to use on any `C<S>` object (again, with `S` a subtype of `T`). This can be viewed informally as automatically projecting class `C` and deriving per-use versions.

Our past work [1] presented an approach that *infers* general use-site variance annotations for every use of a generic `C<X>`, by leveraging the *definition-site* variance of `C`, i.e., its most liberal safe variance based on all its methods. In this way, all generic types are classified as inherently (i.e., based on their definition) covariant, contravariant, bivariant, or invariant with respect to a type parameter. Then, this inherent variance can be employed at every use-site of the generic type. For example, our past approach would infer that the interfaces `java.util.Iterator<X>` and `java.util.Comparator<X>` are covariant and contravariant, respectively, with respect to their type parameters. Hence, it would be reasonable to change all oc-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '14, October 19–21, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2660193.2660203

currences of Iterator<T> (for any type expression T) into Iterator<? extends T>: it is, e.g., safe to pass an instance of Iterator<Dog> to a method expecting an Iterator<Animal>, assuming Dog $<:$ Animal. Similarly, one can reasonably change all occurrences of Comparator<X> into Comparator<? super X>.

This paper is based on the observation that this past work needs significant extension to yield benefits with existing programs and without changing the Java type system. First, the approach ignores practical complexities. Preserving the original program's semantics with additional wildcards may require adding wildcards to syntactically-illegal locations (e.g., wildcards are not allowed in the outermost type arguments in parent type declarations in Java). Second, our past approach misses opportunities for more general types by analyzing method bodies. Most importantly, however, past work assumes that the entire program and all its libraries get rewritten. Even if this was what the programmer desired, it is an impossible requirement in practice: part of the code is unavailable for a rewrite (e.g., fixed signatures of native methods). Instead, we need an approach that is aware of which type occurrences cannot be generalized and integrates this knowledge into its variance inference. Furthermore, the use mode of a variance inference algorithm is typically local: the programmer wants help in safely generalizing a handful of type occurrences, as well as any other types that are essential in order to generalize the former.

We present a modular approach that addresses the above need, by leveraging an inter-procedural flow analysis. Our technique has an incremental usage mode: we only perform program rewrites based on program sites that the programmer selected for refactoring (and on other sites these depend on), and not on an entire, closed code base. Our type generalization fully takes into account the peculiarities of the Java type system, as well as other constraints (e.g., generic native methods) that render some type occurrences off-limits for generalization. Furthermore, we perform a method body analysis that can infer more general types than mere method signature analysis. The result is a refactoring algorithm that allows *safely* inferring more general types for any subset of a program's type occurrences and for any pragmatic environment restrictions. Our approach yields more general types than past work [8, 15] and assumes no changes to Java (unlike, e.g., [2]).

In outline, our work makes the following contributions:

- To assist the programmer with utilizing variance in Java, we present a refactoring approach that automatically rewrites Java code with more general wildcard types. Our tool allows users to select which declarations to generalize the type signatures of.
- Our approach works in a context where not all types can be rewritten because, for example, they are declared in a third-party library for which the source code is unavailable. The user may also select declarations to exclude from rewriting if keeping the more specific type is desired to support future code updates.
- Our approach handles the entire Java language and preserves the behavior of programs employing intricate Java features, such as generic methods, method overrides, and wildcard capture.
- We evaluate our tool on six Java generic libraries. We find that 34% of available declarations of variant type signatures (generic types that may promote a wildcard) can be generalized—i.e., relaxed with more general wildcard types. On average, 146 declarations will need to be updated if a declaration is generalized, showing that this refactoring would be too tedious and error-prone to perform manually.
- We offer both empirical evidence and a proof that the refactoring algorithm is sound. The six large Java generic libraries that were analyzed were also refactored by our tool. The refactored code of the libraries was compiled using javac. Section B in the Appendix formally argues why the refactoring algorithm preserves the ability to compile programs.

2. Illustration

We next illustrate the impact and intricacies of inferring variance annotations in a pragmatic setting. Figure 1 presents an example program before and after automatic refactoring by our tool. This program declares two classes: WList, a write-only list, and MapEntryWList, a specialized WList of map entries. Our tool allows a user to select declarations whose types should be generalized. For this example, suppose the user selects method arguments source, dest, strings, and entry (lines 7, 11, 18, and 23, respectively).

1. Consider generalizing the type of the argument dest, declared on line 11, of the addAndLog method. In general, the interface java.util.List is invariant in its type parameter because it allows both reading elements from a list and writing elements to a list. (We explain such background in more detail in Section 3.) However, in the addAndLog method, no elements are read from the list dest. Within this method, only the add method is invoked on dest to write to this list. The type signature of List.add contains the type parameter of List only in the argument type, which is a contravariant position. Hence, only a contravariant version of List is required by dest, and its type can be safely promoted to List<? super T>.
2. The user has selected generalizing the type of the argument source of the addAll method declared on line 7. Only the iterator method is invoked on source within this method, which returns an Iterator<E>. As mentioned in Section 1, Iterator is covariant in its type parameter, and our tool infers this. As a result, the type parameter of List in the type signature of List.iterator occurs only covariantly. Because of the limited use of source in addAll we can

```
1  import java.util.*;
2  class WList<E> {
3   private List<E> elems =  new LinkedList<E>();
4   void add(E elem) {
5     addAll(Collections.singletonList(elem));
6   }
7   void addAll(List<E> source ) {
8     addAndLog(source.iterator(), this.elems);
9   }
10  static <T> void
11  addAndLog(Iterator<T> itr, List<T> dest ) {
12    while(itr.hasNext()) {
13     T elem = itr.next();
14     log(elem);
15     dest.add(elem);
16    }
17  }
18  static void client(WList<String> strings ) { ... }
19 }
20 class MapEntryWList<K,V>
21  extends WList<Map.Entry<K,V>> {
22  @Override
23  void add(Map.Entry<K, V> entry )  { }
24 }
```

```
1  import java.util.*;
2  class WList<E> {
3   private List<E> elems = new LinkedList<E>();
4   void add(E elem) {
5     addAll(Collections.singletonList(elem));
6   }
7   void addAll(List<? extends E> source) {
8     addAndLog(source.iterator(), this.elems);
9   }
10  static <T> void
11  addAndLog(Iterator<? extends T> itr, List<? super T> dest) {
12    while(itr.hasNext()) {
13     T elem = itr.next();
14     log(elem);
15     dest.add(elem);
16    }
17  }
18  static void client(WList<? super String> strings) { ... }
19 }
20 class MapEntryWList<K,V>
21  extends WList<Map.Entry<K,V>> {
22  @Override
23  void add(Map.Entry<K, V> entry) { }
24 }
```

Figure 1. Code comparison. Original code on the left. Refactored code on the right. Declarations that were selected for generalization are shaded in the original version.

safely infer that the type of `source` can be promoted to the more general type `List<? extends E>`.

3. If only the type of `source` changes to `List<? extends E>`, then the refactored program will no longer compile. After changing `source`'s type, `source.iterator()` no longer returns an `Iterator<E>` but instead an `Iterator<? extends E>`. The method call to `addAndLog` on line 8 would cause a type error because this method expects a stricter type, `Iterator<E>`.[1] As a result, to perform this refactoring without introducing compilation errors, we must perform a flow analysis to determine if generalizing the type of one declaration requires changing the types of other declarations. This flow analysis requires careful reasoning as dependency relationships arise from many non-trivial language features in Java. In this example, the type of the `itr` parameter (line 11) is also generalized to `Iterator<? extends T>`.

4. The user has selected for generalization the type of the `strings` argument of method `client`, declared on line 18. The type of this argument is promoted to `WList<? super String>`. The method body is elided for brevity, but let us assume that all non-static methods of `WList` are dispatched on variable `strings` in this method. The refactoring of the type of `strings` is safe because the tool can infer that `WList` is contravariant, but only after performing the earlier refactorings. In the original version, the occurrence of the type parameter `E` in `List<E>`, the type of `source`, constrained the inferred definition-site variance of `WList` to be invariance because the inferred definition-site variance of `List` is invariance. Changing the type of `source` to `List<? extends E>`, however, allows the definition-site variance of `WList` to be contravariance.[2] This contravariance of `WList` tells us that it is safe to add the use-site annotation `? super` to the type of `strings`. Note how this type generalization is done for different reasons than that of `source`, earlier: type `WList` is inherently contravariant (after earlier refactorings), therefore it does not matter how `strings` is used. In contrast, the generalization of the type of `source` was possible only because of the way `source` was used in method `addAll`.

5. The argument `entry` of the *overriding* method `add` is declared on line 23. Our tool infers that `Map.Entry` is covariant in its first type parameter. Therefore, changing the type of `entry` to `Map.Entry<? extends K, V>` would not cause a runtime error. Our tool does not apply this update, however, for the following reasons:

(a) Java (`javac`) would no longer infer that `MapEntryWList.add` overrides `WList.add`. Because of the '`@Override`' annotation, `javac` would flag a compilation error.

(b) Removing the '`@Override`' annotation would seem not cause a compilation error. Instead since `MapEntryWList.add` does not override `WList.add`, Java now considers that `MapEntryWList.add` *overloads* `WList.add`, where the arguments types of `MapEntryWList.add` and `WList.add` are `Map.Entry<? extends K, V>` and `Map.Entry<K, V>`, respectively. However, the *erasures* [11, Chapter 4.6] of the

[1] The inferred type parameter passed in the invocation of the generic method `addAndLog` is `E`. Thus, the first argument type of `addAndLog` is `Iterator<E>`.

[2] The occurrence of the type parameter `E` in the type of the field `elems`, declared on line 3, does not constrain the definition-site variance of `E` in `WList` because `elems` is what is called *object-private* in the Scala language [16, Sections 5.2 and 4.5]: `elems` is not only private to `WList` but also only accessed from a `this` qualifier on line 8.

type signatures of both methods are the same: Both argument types erase to `Map.Entry`. Overloaded methods that have the same erasure result in a compilation error.

(c) Even if the Java compiler did not flag a compilation error as a result of generalizing the type of `entry`, performing this refactoring is undesirable because it could change the runtime behavior of the program. Client code that previously invoked `MapEntryWList.add` at runtime may now invoke `WList.add` instead, since `MapEntryWList.add` would no longer override `WList.add`.

(d) Another option to allowing the type of `entry` to be generalized would be to change the parent type declaration of `MapEntryWList`. Our tool does not add wildcards to parent type declarations for a number of reasons that we discuss in Section 4.6. For instance, the most straightforward generalization would change the parent type of `MapEntryWList` to `WList<? super Map.Entry<K, V>>`, since we inferred the refactored version of `WList` to be contravariant. This change is not legal in Java because wildcards are not allowed in the outermost type arguments in parent type declarations in Java. (The type in question is not a class type but rather a reference type [11, Chapter 4.3].)

Our tool will generalize the type signature of an overridden method only if all of the methods it overrides and vice versa can also be generalized, so that all overriding relationships in the original program are maintained. More generally, our tool ensures the behavior of programs is preserved.

Although the above example is small, it illustrates how generalizing types with Java wildcards requires intricate, tedious, and error-prone reasoning. The complexity of the refactoring, thus, warrants automation.

3. Background

Our refactoring tool infers definition-site variance to generalize types. The tool safely rewrites `Iterator<Animal>` to `Iterator<? extends Animal>` because it infers `Iterator` to be covariant. This section provides a brief background of the core variance analysis performed for determining wildcard annotations. Further details can found in past work [1, 2].

3.1 Variance From A Generic Definition

Our tool adds wildcards to applications of a generic that are inherently variant, based on its definition. Languages supporting definition-site variance enable programmers to declare type parameters of a generic with variance annotations. For instance, Scala [16] requires the annotation + for covariant type parameters, - for contravariance, and invariance is the default. A well-established set of rules can then be used to verify that the use of the type parameter in the generic is consistent with the annotation. We first explain how definition-site variances of type parameters are constrained using a slight extension of Java that includes Scala's syntax for definition-site variance annotations.

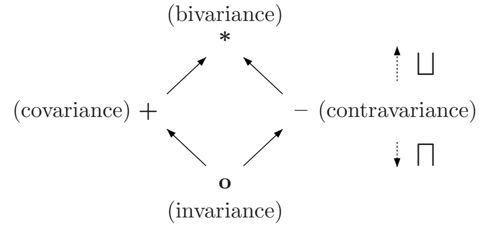

Figure 2. Standard variance lattice.

In intuitive terms, definition-site variances are constrained by the use of type parameters in certain "positions". Each typing position in a generic's signature has an associated variance. For instance, method return types are covariant positions and method argument types are contravariant positions. Type checking the declared variance annotation of a type parameter requires determining the variances of the positions the type parameter occurs in. *The variance of all such positions should be at least the declared variance of the type parameter.* The ordering of variance values is presented in the variance lattice in Figure 2. Consider the following Java interfaces, where v_X, v_Y, and v_Z stand for definition-site variance annotations.

```
interface RList<vX X> { X get(int i); }
interface WList<vY Y> { void set(int i, Y y); }
interface IList<vZ Z> { Z setAndGet(int i, Z z); }
```

The variance v_X is the (imaginary) declared definition-site variance for type variable `X` of interface `RList`. If $v_X = +$, the `RList` interface type checks because `X` does not occur in a contravariant position. If $v_Y = +$, the `WList` interface does not type check because `Y` does occur in a contravariant position (the second argument type in `set` method), but $v_Y = +$ implies `Y` should only occur in a covariant position. `IList` type checks only if $v_Z = o$ because `Z` occurs in both a covariant and a contravariant position.

Intuitively, `RList` is a read-only list: it only supports retrieving objects. The return type of a method indicates this "retrieval" capability. Retrieving objects of type `T` can be safely thought of as retrieving objects of any supertype of `T`. A read-only list of `T`s (`RList<T>`) can always be safely thought of as a read-only list of some supertype of `T`s (`RList<S>`, where `T` $<:$ `S`). Thus, a return type is a covariant position and `RList` is covariant in `X`. Similarly, `WList` is a write-only list, and is intuitively *contravariant*. Objects of type `T` can be written to a write-only list of `T`s (`WList<T>`), but also to a write-only list of `S`s (`WList<S>`), where `T` $<:$ `S`, because objects of type `T` are also objects of type `S`. Hence, a `WList<S>` can safely be thought of as a `WList<T>`, if `T` $<:$ `S`.

On the contrary, it would *not* be safe to assume `WList` is covariant because the *subsumption (is-a)* principle would be violated. (Informally, anything one can do with the supertype, one can also do with the subtype.) Assuming, for example, a `WList<Dog>` *is a* `WList<Animal>` violates the subsumption principle because a `WList<Animal>` can add a `Cat`

to itself, but a `WList<Dog>` cannot without introducing a possible runtime type error.

Finally, `C<X>` is *bivariant* implies that `C<S>`$<:$`C<T>` for any types `S` and `T`. Previous work [1] discusses several interesting interface definitions that are inherently bivariant.

3.2 Computing Most General Definition-Site Variance

We infer definition-site variance by first generating a set of constraint expressions on definition-site variances. Definition-site variances are referred to in expressions by variables of the form *dvar*(`X`; `C`), where `C` is the name of a generic definition and `X` is the name of a type parameter of the generic. We then compute the maximum values for the variables that satisfy the constraint expressions. This computation requires fixed point iteration as variance variables can be recursively constrained. Consider the following two classes, for example:

```
class C<X> {
  X    foo (C<? super X>   csx) { ... }
  void bar (D<? extends X> dsx) { ... }
}
class D<Y> { void baz (C<Y> cx) { ... } }
```

The maximum variances for the type parameters of classes `C` and `D` are constrained by the *variances of the type expressions* that occur in the class signatures. We would generate the following constraints for classes `C` and `D`:

$$\texttt{foo}\text{ return type} \implies \mathit{dvar}(\texttt{X};\texttt{C}) \sqsubseteq + \otimes \underbrace{+}_{\mathit{var}(\texttt{X};\texttt{X})}$$

$$\texttt{foo}\text{ arg type} \implies \mathit{dvar}(\texttt{X};\texttt{C}) \sqsubseteq - \otimes \underbrace{(\mathit{dvar}(\texttt{X};\texttt{C}) \sqcup -)}_{\mathit{var}(\texttt{X};\texttt{C<-X>})}$$

$$\texttt{bar}\text{ arg type} \implies \mathit{dvar}(\texttt{X};\texttt{C}) \sqsubseteq - \otimes \underbrace{(\mathit{dvar}(\texttt{Y};\texttt{D}) \sqcup +)}_{\mathit{var}(\texttt{X};\texttt{D<+X>})}$$

$$\texttt{baz}\text{ arg type} \implies \mathit{dvar}(\texttt{Y};\texttt{D}) \sqsubseteq - \otimes \underbrace{(\mathit{dvar}(\texttt{X};\texttt{C}) \sqcup o)}_{\mathit{var}(\texttt{Y};\texttt{C<oY>})}$$

The function *var*(`X`; `T`) assigns a variance value to each type expression `T` with respect to type variable `X`. Method `foo`'s return type, `X`, results in the upper bound $+\otimes$ *var*(`X`; `X`) because the variance of its type *var*(`X`; `X`) was *transformed* using the operator $\otimes$, by covariance, the variance of the position that the type occurs in. (Section 3.3 provides details of the $\otimes$ operator.) Also, existing wildcards in the code (e.g., "`? extends X`" for `D`) result in a lattice join operation ($\sqcup$) of the variance of the wildcard and the definition-site variance of the generic (e.g., "*dvar*(`Y`; `D`) $\sqcup$+") [1, 2].

Solving the constraints (i.e., computing the greatest fixed-point) for the above example yields *dvar*(`X`; `C`) = $+$ and *dvar*(`Y`; `D`) = $-$.

3.3 Variance Transformation

The variance of type variables is *transformed* by the variance of the context that the variables appear in. Figure 3 summarizes the behavior of the transform operator $\otimes$. Given two generic types `A<X>` and `B<X>` with variances v_A and v_B for their parameters, we can compute the variance of type `A<B<X>>` with respect to `X` as $v = v_A \otimes v_B$. To sample why the definition of the transform operator makes sense, we show one example case:

Case $+ \otimes - = -$: This means that *var*(`X`; `C<`E`>`) $= -$ when generic `C` is covariant in its type parameter and type expression E is contravariant in `X`. This is true because, for any two types T_1 and T_2, $T_1 <: T_2$

$$\implies E\texttt{[}T_2\texttt{/X]} <: E\texttt{[}T_1\texttt{/X]} \qquad \text{(contravariance of } E\text{)}$$

$$\implies \texttt{C<}E\texttt{[}T_2\texttt{/X]>} <: \texttt{C<}E\texttt{[}T_1\texttt{/X]>} \qquad \text{(covariance of C)}$$

$$\implies \texttt{C<}E\texttt{>[}T_2\texttt{/X]} <: \texttt{C<}E\texttt{>[}T_1\texttt{/X]}$$

Hence, `C<E>` is contravariant with respect to `X`.

$+\otimes+=+$	$-\otimes+=-$	$*\otimes+=*$	$o\otimes+=o$
$+\otimes-=-$	$-\otimes-=+$	$*\otimes-=*$	$o\otimes-=o$
$+\otimes*=*$	$-\otimes*=*$	$*\otimes*=*$	$o\otimes*=*$
$+\otimes o=o$	$-\otimes o=o$	$*\otimes o=*$	$o\otimes o=o$

Figure 3. Definition of variance transformation: $\otimes$. This operator is commutative.

3.4 Rewrites using Definition-Site Inference

Given inferred definition-site variances, the refactoring tool generalizes type signatures by replacing *specified* use-site annotations with *inferred* use-site annotations. Each inferred use-site annotation is the inferred definition-site variance of the generic *joined* with the specified use-site annotation. Consider classes `C` and `D` from Section 3.1 that were inferred to be covariant and contravariant, respectively, in their type parameters. Given these inferred def-site variances, the refactoring tool can safely rewrite those definitions to the code below. (Again, the inferred variances are lattice-joined with the variance of the wildcard that pre-existed in the code. E.g., covariance joined with contravariance becomes bivariance and the inferred variance is $*$, corresponding to the wildcard '`?`'.) All client code of classes `C` and `D` would still compile. The classes have been generalized without sacrificing type safety.

```
class C<X> {
  X    foo ( C<?>  csx) { ... }
  void bar ( D<?>  dsx) { ... }
}
class D<Y> { void baz ( C<? extends Y>  cx) { ... }}
```

4. Type Influence Flow Analysis

The act of generalizing occurrences of types in a program introduces a tradeoff. On the one hand, we want to assist programmers with generalizing their interfaces in a type safe manner—i.e., to replace type occurrences with more general

types. More general types for the interface of a class, however, entail fewer operations for implementations of this interface. For instance, promoting the type of an object from List<String> to the type List<? extends String> results in the inability of that object to add String objects to itself. Furthermore, in Java, an overriding method must have the same type signature as the overridden method. Hence, generalizing the types in a method's signature also restricts the ability of subclasses to provide alternative implementations.

To enable library designers to manage this tradeoff, our tool allows users to choose which declarations to update instead of always generalizing all (rewritable) types. A refactoring tool should not introduce new compilation errors for practicality and should preserve the semantics of the original program. Thus, automating the update of a fragment of the program requires a flow analysis to determine, given a type occurrence to update, the set of other type occurrences that also need to be updated. We say that *a declaration A influences a declaration B* if making A's type more general requires making B's type more general. Moreover, we coin the term "type influence flow analysis" (or just "influence analysis") for the program analysis computing (an over-approximation of) this information.

We implement our influence analysis by building a directed flow graph where nodes represent declarations in the program. A flow graph is constructed so that if declaration A influences declaration B, then the graph will contain a path from A to B. Thus, our global "influence" relation is just the transitive closure of primitive influences (directed edges in the flow graph) induced by the program text. For example, an edge from variable A to variable B would be generated for an assignment expression from A to B: generalizing the type of A would require B's type to also be generalized. Subsequent subsections provide further details.

4.1 Influence Nodes

Our refactoring tool generalizes interfaces by generalizing types of declarations. Nodes in our flow graph represent declarations in the program.

The Java language constructs that can be nodes in our influence graph are given by the syntactic category InfluenceNode:

```
InfluenceNode  ::=  MethodDecl | Variable
Variable       ::=  VariableDeclaration
                 |  FieldDeclaration
                 |  ParameterDeclaration
```

MethodDecl, VariableDeclaration, FieldDeclaration and ParameterDeclaration are the syntactic entities that their names suggest, as defined in the JLS [11]. Both static and instance fields are instances of FieldDeclarations. VariableDeclarations are local variable declarations, which occur in blocks, method bodies, initialization statements of for-loops, etc. Formal value arguments from methods and constructors are ParameterDeclarations. Arguments of catch blocks are ignored; they cannot be parametric types [11, Chapter 8.1.2]. MethodDecl nodes in the flow graph are used to capture the influences on return types of method declarations. Return types may need to be generalized if the types of variables occurring in return statements are generalized. Generalizing the return type of a method can influence the type of other declarations; for instance, a variable can be assigned the result of a method invocation.

Auxiliary functions used in this presentation are defined over a language similar to Featherweight Generic Java [12], which we will call FGJ*, rather than the full Java language, in order to focus the presentation on the essential elements. FGJ*'s syntax is presented in Figure 4. We skip the definitions of some syntactic elements such as statements (s) and names of type variables (X or Y) or methods (m). We follow the FGJ convention of using $\lhd$ to abbreviate the extends keyword and $\overline{\mathtt{A}}$ denotes the possibly empty vector $\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n$. Reference types allow use-site annotations, which denote wildcard annotations; types C<? extends T> and C<T>, for example, are expressed in the syntax as C<+T> and C<oT>. Although method invocations in the FGJ* syntax contain specified type arguments ($\overline{\mathtt{T}}$) and a qualifier component ("e."), we allow invocations where both can be skipped.

$\mathtt{v}, \mathtt{w} ::= + \mid - \mid * \mid o$	*use-site variance*
$\mathtt{T}, \mathtt{U}, \mathtt{S} ::= \mathtt{X} \mid \mathtt{N} \mid \mathtt{R}$	*types*
$\mathtt{N} ::= \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$	*class types*
$\mathtt{R} ::= \mathtt{C<}\overline{\mathtt{v}\ \mathtt{T}}\mathtt{>}$	*reference types*
$\mathtt{L} ::= \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X} \lhd \mathtt{U}}\mathtt{>} \lhd \mathtt{N}\ \{\ \overline{\mathtt{T}\ \mathtt{f;}}\ \ \overline{\mathtt{M}}\ \}$	*class declaration*
$\mathtt{M} ::= \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{U}}\mathtt{>}\ \mathtt{T}\ \mathtt{m(}\overline{\mathtt{T}\ \mathtt{x}}\mathtt{)}\ \{\ \mathtt{s}\ \}$	*method declaration*
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.<}\overline{\mathtt{T}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new}\ \mathtt{N(}\overline{\mathtt{e}}\mathtt{)}$	*expressions*
$\mathtt{s} ::= \mathtt{e}_1\ \mathtt{=}\ \mathtt{e}_2\mathtt{;} \mid \ldots$	*statements*
$\mathtt{X}, \mathtt{Y} ::= \ldots$	*type variables*
$\mathtt{x}, \mathtt{y} ::= \ldots$	*expression variables*

Figure 4. FGJ* Syntax

Auxiliary functions are defined in Figure 5. Some functions are defined only informally because their precise definitions are either easy to determine or are not the focus of this paper. For example, *Lookup*($\mathtt{m}$; $\overline{\mathtt{e}}$) returns the declaration of the method being called (statically) by the method invocation $\mathtt{m(}\overline{\mathtt{e}}\mathtt{)}$.[3] Detailed definitions of look up functions and other auxiliary functions that are omitted here can be found in [2, 12].

4.2 Flow Dependencies from Qualifiers

The semantics of an object-oriented language like Java entails intricate flow dependencies from qualifiers. The qualifier of a Java expression is the part of the expression

[3] Changing wildcard annotations does not affect method overloading resolution because the type signatures of two different overloaded methods are not allowed to have the same erasure.

***nodesAffectingType*:** (representative rules)

$$\frac{\begin{array}{c} \mathit{Lookup}(\texttt{m};\overline{\texttt{e}}) = M \\ \mathit{returnTypeDependsOnParams}(M) \end{array}}{\begin{array}{c} \mathit{nodesAffectingType}(\texttt{<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}) = \\ \cup_{i=1}^{|\overline{\texttt{e}}|} \mathit{nodesAffectingType}(\texttt{e}_i) \cup \{M\} \end{array}} \; (\text{N-GenericMethod})$$

$$\frac{\begin{array}{c} \mathit{Lookup}(\texttt{m};\overline{\texttt{e}}) = M \\ \neg \mathit{returnTypeDependsOnParams}(M) \end{array}}{\mathit{nodesAffectingType}(\texttt{<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}) = \{M\}} \; (\text{N-MonoMethod})$$

$$\frac{\texttt{e} \neq \texttt{m(}\overline{\texttt{e}}\texttt{)}}{\begin{array}{c} \mathit{nodesAffectingType}(\texttt{e}) = \\ \mathit{accessedNodes}(\texttt{e}) \end{array}} \; (\text{N-NonMethodCall})$$

***destinationNode*:** (representative rules)

$$\frac{\begin{array}{c} \texttt{<}\overline{\texttt{S}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \in P \\ \mathit{Lookup}(\texttt{m};\overline{\texttt{e}}) = \texttt{<}\overline{\texttt{Y} \lhd \texttt{U}}\texttt{> T m(}\overline{\texttt{T x}}\texttt{) \{ ... \}} \end{array}}{\mathit{destinationNode}(\texttt{e}_i) = \texttt{x}_i} \; (\text{D-MethodCall})$$

$$\frac{\begin{array}{c} \texttt{e}_L = \texttt{e}_R \in P \\ \mathit{varDecl}(\texttt{e}_L) = \texttt{x} \end{array}}{\mathit{destinationNode}(\texttt{e}_R) = \texttt{x}} \; (\text{D-Assignment})$$

$$\frac{\begin{array}{c} \text{“}\textbf{return}\ \texttt{e ;}\text{”} \in P \\ \mathit{enclosingMethod}(\texttt{e}) = M \end{array}}{\mathit{destinationNode}(\texttt{e}) = M} \; (\text{D-Return})$$

Lookup$(\texttt{m};\overline{\texttt{e}})$ = the declaration of the method being called (statically) by the invocation of method `m` with arguments $\overline{\texttt{e}}$.
returnTypeDependsOnParams$(M) \equiv M$ is a generic method with a type parameter `X` that syntactically occurs in both the return type and in an argument type.
varDecl$(\texttt{e})$ = the declaration referred to by expression `e` (e.g. *varDecl*$(\texttt{x.f})$ = declaration of field `f`).
accessedNodes$(\texttt{e})$ = the set of declarations accessed in expression `e` (e.g. *accessedNodes*$(\texttt{x.f}) = \{\texttt{x}, \texttt{f}\}$).
enclosingMethod$(\texttt{e})$ = the enclosing method of `e` (this function is partial).
hierarchyMethods(M) = the set of methods that either override M or are overridden by M.
hierarchyParams$(\texttt{x}) = \{i^{th}$ parameter of $M' \mid M' \in$ *hierarchyMethods*$(M)\}$, where `x` is the ith formal parameter of M.
P is the input Java program and, "$A \in P$" denotes expression or statement A syntactically occurs in program P.

Figure 5. Auxilary Functions

that identifies the host (object, type definition, or package) from which the member is accessed. In the expression `someString.charAt(0)`, for example, the subexpression `someString` is the qualifier of the method invocation `charAt(0)`. Generalizing the type of a qualifier of a method invocation may require generalizing the type signature of the method accessed. In particular, we need to add edges in the flow graph from qualifiers (declarations accessed in qualifiers) to formal method arguments when analyzing a method invocation. The following example motivates these dependencies:

```
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);
  }
}
```

The generic interfaces `C` and `D` are both safely bivariant. `D` is clearly bivariant because its type parameter does not appear in the definition of `D`. `C` is bivariant because its variance is only constrained by `D`, which is also bivariant.

Suppose the argument `arg` in method `foo` above is not rewritable (i.e., its type remains `D<X>`). Also, consider rewriting the `Client` class and assume that all method arguments in `Client` are rewritable. Then it seems that the types of variables `cstr` and `dstr` in `bar` are rewritable to `C<?>` and `D<?>`, respectively, by the bivariance of both interfaces `C` and `D`. However, this would cause `bar` to generate the following compilation error (modulo generated numbers):

```
foo(D<capture#274 of ?>) in C<capture#274 of ?>
cannot be applied to (D<capture#582 of ?>)
```

Effectively, the error states that the unknown type that the "`?`" stands for in `C<?>` is not known to be the same as the unknown type that the "`?`" stands for in `D<?>`.

This error would have been avoided if the type of `arg` in method `foo` was rewritten. More generally, wildcard annotations need to be added in type definitions in order for the inferred definition-site variance to support all of the operations of the class. In the case of interface `C`, an instance of type `C<?>` cannot access the method `foo` unless the type of `arg` is changed to `D<?>`.[4] Therefore, we need to add an influence edge from the qualifier `cstr` to the formal parameter `arg` of method `foo`.

4.3 Expression Targets

Expressions may access variables that are declared with types that were generalized. In the refactored code, the type of an expression can change as a result of changing the type of a variable or declaration accessed by the expression. In the motivating example of Section 2, the type of the parameter `source` changes from `List<E>` to `List<? extends E>` in the refactored code. The return

[4] Without changing the type of `arg` to `D<?>`, invoking `foo` on an instance of `C<?>` type checks only if `null` is passed as an argument.

type of method `List<E>.iterator` is `Iterator<E>`. Updating the type of `source` causes the type of the expression `source.iterator()` on line 8 to change from `Iterator<E>` to `Iterator<? extends E>`. In turn, changing the type of expression `source.iterator()` requires modifying the type of method parameter `itr` on line 11.

The essence of determining type influences emerging from expressions is described by two key functions: *nodesAffectingType*(e) computes the set of declarations accessed in expression e that can affect the type of e. *destinationNode*(e) is a partial function that returns the declaration that is influenced by the type of e. Figure 5 contains the definitions of these functions for the most important (and representative) elements of the FGJ* syntax. Considering the motivating example, for instance, *nodesAffectingType*(`source.iterator()`) = {`source`, `List.iterator`} and *destinationNode*(`source.iterator()`) = `WList.addAndLog.itr`. Because the expression `source.iterator()` is the first argument in the method call to `addAndLog`, the first formal parameter, `itr`, of `addAndLog` is the destination node of `source.iterator()`. Influence edges are added from nodes in *nodesAffectingType*(e) to the node returned by *destinationNode*(e). These edges signal the dependencies caused by the expression in a context such as a method invocation.

4.4 Dependencies from Inheritance

In Java, an overriding method in a subclass is required to have the same argument types as the overridden method in the superclass. We add corresponding edges between method declarations in the influence flow graph so that overrides relationships are preserved in the refactored code. In the motivating example, the `add` method in `MapEntryWList` (line 23) overrides the `add` method in the super class `WList`. Our analysis infers that the type of `MapEntryWList.add`'s argument influences the type of `WList.add`'s argument, to preserve the override. In general, we add an edge from a parameter to its corresponding parameter in an overriding method. An edge in the reverse direction is also added, since generalizing the parameter type in the overridden method requires updating the corresponding parameter's type in the subclass to preserve the override. Adding edges to method parameters in subclasses, however, requires a whole-program analysis: All of the subclasses of the input class must be known to find all of the overridding methods.

4.5 Algorithm

Algorithm 1 contains the pseudo-code of our algorithm for computing the type influence flow graph. The algorithm implements the analyses described in the preceding subsections using functions defined in Figure 5. Given the flow graph, determining if the type of one declaration influences another is performed by checking for existence of a path in the graph.

Algorithm 1 Algorithm computing influence flow graph

Input: Java program P
Output: Flow graph G on Java declarations

// Analysis from Section 4.2
1: **for each** method call $\texttt{<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \in P$ **do**
2:     *qualifierDecl* $\leftarrow$ *varDecl*(`e`)
3:     $\texttt{<}\overline{\texttt{Y} \lhd \texttt{U}}\texttt{> T m(}\overline{\texttt{T x}}\texttt{) \{ ... \}} \leftarrow$ *Lookup*$(\texttt{m}; \overline{\texttt{e}})$
4:     Add edge $(qualifierDecl, \texttt{x}_i)$ to G, *for each* $\texttt{x}_i \in \overline{x}$.
5: **end for**

// Analysis from Section 4.3
6: **for each** expression $e \in P$ **do**
7:     $D \leftarrow$ *destinationNode*(e)
8:     Add edge (N, D) to G, *for each* $N \in$ *nodesAffectingType*(e).
9: **end for**

// Analysis from Section 4.4
10: **for each** method declaration $M \in P$ **do**
11:     Add edge (M', M) to G, *for each* $M' \in$ *hierarchyMethods*(M).
12:     **for each** parameter $\texttt{x} \in$ *formalParams*(M) **do**
13:         **for each** parameter $\texttt{y} \in$ *hierarchyParams*$(\texttt{x})$ **do**
14:             Add edge $(\texttt{y}, \texttt{x})$ to G
15:         **end for**
16:     **end for**
17: **end for**
18: **return** G

4.6 Non-rewritable Overrides

The motivating example illustrates the need to determine when types cannot be further generalized. Clearly, types of declarations from binary files (e.g., jar files) are not rewritable because we do not have access to the source code.[5] Consider the example interface `java.util.List<E>`, which declares a method `iterator` with return type `Iterator<E>`. A class implementing `List<E>` cannot override `iterator` with a return type of `Iterator<? extends E>` even though `Iterator` is covariant in its type parameter. We use the influence graph to determine if a declaration can influence a non-rewritable declaration. Any declaration that can reach a non-rewritable declaration in the graph is also considered to be non-rewritable.

The motivating example shows that we must classify some declarations from source as non-rewritable. `MapEntryWList.add`'s argument type, `Map.Entry<K, V>`, on line 23 is a parameterized type, which could be further generalized safely to `Map.Entry<? extends K, V>` by the covariance of `Map.Entry` in its first type parameter. The argument's type in the overridden method, `WList.add`, could not because it is just a type variable (`E`). Generally, we classify

[5] Bytecode is often as malleable as source code. In principle our approach could apply to bytecode. However, this would not address the issue of unavailable code—native code would still be inaccessible—and, furthermore, Java bytecode does not preserve full type information for generics.

```
public interface OrderedIterator<E> extends Iterator<E> {
  E previous();
}
protected static class EntrySetIterator<K, V>
  extends LinkIterator<K, V>
  implements OrderedIterator<Map.Entry<K, V>>,
             ResettableIterator<Map.Entry<K, V>>
{
  public Map.Entry<K, V> previous() { ... }
}
```

```
EntrySetIterator<K,V>
  <: ResettableIterator<Map.Entry<K,V>>
  <: Iterator<Map.Entry<K,V>>.
EntrySetIterator<K,V>
  <: OrderedIterator<Map.Entry<? extends K,V>>
  <: Iterator<Map.Entry<? extends K,V>>.
```

Figure 6. Simplified code example from Apache collections library on the left. Subtyping (interface-implements) relationships on the right, if we annotate `K` with "`? extends`" only in the parent type `OrderedIterator<Map.Entry<K,V>>`.

an argument type or return type (of a non-static and non-final method) as not rewritable if the type is just a type variable.

Discussion: parent types are not rewritable. As mentioned in Section 2, our analysis does not generalize parent type declarations (i.e., `extends` and `implements` clauses). We chose not to rewrite parent types in order to improve the usability of the refactoring tool and to simplify the analysis. We next discuss the rationale in detail.

If we were to generalize parent types, the influence analysis would be far less intuitive to users of the refactoring tool as dependencies would no longer be traceable by flows from only variable/member declarations. Rewriting parent types would significantly complicate the analysis, may cause decidability issues, and would not significantly increase the number of declarations that could be rewritten. We explain the issues using the example in Figure 6, which is a simplified version of a code segment from the Apache collections library [3].

Consider rewriting the type `OrderedIterator<Map.Entry<K,V>>` in the `implements` clause of `EntrySetIterator`. We inferred that `OrderedIterator` is covariant in its type. However, rewriting `OrderedIterator<Map.Entry<K,V>>` to `OrderedIterator<? extends Map.Entry<K,V>>` in a parent type declaration is not legal in Java since wildcards are not allowed in the outermost type arguments in parent type declarations [11, Chapter 4.3].

Now consider rewriting the first parent-interface type to `OrderedIterator<Map.Entry<? extends K,V>>`. The latter is a class type and legal in a parent type declaration. This causes a compile error because it implies that `EntrySetIterator<K,V>` implements two different instantiations of the same generic: `Iterator<Map.Entry<? extends K,V>>` and `Iterator<Map.Entry<K,V>>`; Figure 6 also shows how this was derived.[6]

Another possibility is to rewrite the second parent-interface type to `ResettableIterator<Map.Entry<? extends K,V>>` in addition to promoting the first parent-interface type to `OrderedIterator<Map.Entry<? extends K,V>>`. Then, `EntrySetIterator<K,V>` only implements a single instantiation: `Iterator<Map.Entry<? extends K,V>>`. However, this is safe only if `ResettableIterator` is covariant in its type parameter. If `ResettableIterator` is invariant, then `ResettableIterator<Map.Entry<K,V>>` and `ResettableIterator<Map.Entry<? extends K,V>>` are not subtype-related. We only want to replace types with more general supertypes without sacrificing functionality. Hence, determining if one declared parent type can be generalized *not only depends on all the other parent types but also on whether the argument types being generalized are passed to covariant type constructors*. (Adding wildcards to a type used to parameterize another is safe only if the parameterized type is covariant.)

Further complicating matters, it has been shown in past work [14, 18] that introducing the wildcard annotation "`? super`" in parent type declarations makes deciding the subtyping relation (determining whether one given type is a subtype of another given type) highly likely undecidable.[7] Rewriting parent types would then require our tool to check if the generalized parent type would be within a decidable fragment. We expect that most programmers would find the dependencies involving parent types to be severely non-intuitive. This makes it difficult for users to choose which types they want to rewrite and the types they want preserved. To make the influence analysis more intuitive and avoid decidability issues, we restrict rewriting to types of variable declarations and of members of classes and interfaces.

5. Method Body Analysis

We can infer safe definition-site variances of type parameters solely from the interfaces or member type signatures of a generic. Only analyzing type signatures, however, is more restrictive than necessary because a programmer can specify a more specialized type than needed. The class below presents a non-trivial example.

```
class Body<X extends Comparable<X>> {
  int compareFirst(List<X> lx, X other) {
    X first = lx.get(0);
    return first.compareTo(other); } }
```

[6] Interface `ResettableIterator<E>` also extends `Iterator<E>`.

[7] Subtyping in the presence of definition-site variance and contravariant type constructors in parent type declarations was shown to be undecidable in [14]. [14, Appendix A] contained simple Java programs with "`? super`" annotations in parent types that crashed Java 1.5 and 1.6 compilers (`javac`) when checking example subtype relations. [18] identified a decidable fragment that does not allow "`? super`" in parent types.

Only analyzing the interface of the `Body` class restricts the greatest definition-site variance we can infer for the type parameter to be invariance. The variance of `Body` is constrained by the invariance of `List` in the first argument of the `compareFirst` method. By taking into account how variables are actually used in a program, however, we may detect when the type of a variable can be promoted to a more liberal type. In the `compareFirst` method body, only the `get` method is invoked on the `lx` argument. In the type signature of `get`, the type parameter of `List` occurs only in the return type, a covariant position. Hence, only the covariant version of `List` is required for the `lx` variable, and its type can be promoted to `List<? extends X>`.[8] Assuming this new type for `lx`, we can also safely infer that the definition-site variance of `Body` is now contravariance. We shall use this reasoning for illustration next.

High-level picture. To infer use-site variance, we generate a set of constraint inequalities between variance expressions in similar fashion to that described in Section 3.2. Use-site variances can now change (they integrate the result of a method body analysis, whereas earlier they consisted of only the wildcard annotation on the type). Hence, we add to the syntax of variance expressions a new kind of variable: If `y` is a method argument declared with type $\texttt{C<}\overline{\texttt{vT}}\texttt{>}$ and $\texttt{X}_i$ is the ith type parameter of generic `C`, then $uvar(\texttt{X}_i;\, \texttt{C};\, \texttt{y})$ denotes the inferred use-site annotation for the ith type argument in the type of `y`.

Treating use-site variances as variables, in turn, relaxes constraints on definition-site variances. Consider the constraint on the inferred def-site variance of `Body`'s type parameter that is generated from analyzing the type of method argument `lx` if we only analyzed type signatures.

$$\begin{aligned} dvar(\texttt{X};\texttt{Body}) &\sqsubseteq - \otimes var(\texttt{X};\texttt{List<}o\texttt{X>}) \\ &= - \otimes (dvar(\texttt{E};\texttt{List}) \sqcup o) \\ &= - \otimes dvar(\texttt{E};\texttt{List}) = - \otimes o = o, \\ &\text{where } \texttt{E} \text{ is the type parameter of } \texttt{List}. \end{aligned}$$

$dvar(\texttt{E};\, \texttt{List})$ refers to the definition-site variance of the type parameter `E` of the `List` interface; it can only (safely) be invariance. This upper bound constrains $dvar(\texttt{X};\, \texttt{Body})$ to invariance and is too restrictive considering the limited use of `lx`. Our method body analysis would replace this constraint on $dvar(\texttt{X};\, \texttt{Body})$ with the following more relaxed one. The specified use-site annotation o has been joined with $uvar(\texttt{E};\, \texttt{List};\, \texttt{lx})$. Constraints on this variable are generated based on the limited use of `lx`. In this example, we would infer $uvar(\texttt{E};\, \texttt{List};\, \texttt{lx}) = +$ and $dvar(\texttt{X};\, \texttt{Body}) = -$.

$$\begin{aligned} dvar(\texttt{X};\texttt{Body}) &\sqsubseteq - \otimes \big(dvar(\texttt{E};\texttt{List}) \sqcup o \sqcup uvar(\texttt{E};\texttt{List};\texttt{lx})\big) \\ &= - \otimes (o \sqcup o \sqcup uvar(\texttt{E};\texttt{List};\texttt{lx})) \\ &= - \otimes uvar(\texttt{E};\texttt{List};\texttt{lx}) \end{aligned}$$

[8] The type signature of method `List<X>.get` = `(int)` $\rightarrow$ `X`.

Further details of the constraint generation process performed during the method body analysis can be found in Section A of the Appendix.

6. Type Influence Graph Optimizations

As the number of declarations in the flow graph increases, so may the number of unmodifiable declarations. In turn, fewer declarations will be rewritten because more paths to unmodifiable declarations will exist in the graph. To allow more rewritable declarations to be detected, the analysis ignores (i.e. does not add to the flow graph) declarations that are not affected by the generalizations performed by the tool. It is safe to ignore such declarations because, even if they were rewritable, their types would not change from rewrites performed by the refactoring tool. Considering the `java.util.List.size` method, for example, which returns an `int`, adding wildcards to any instantiation of the `List` interface would never cause the `size` method to return anything but an `int`. The expression `l.size()` returns `int`, whether `l` has type `List<Animal>` or `List<?>`, for instance.

The influence analysis also ignores declarations of parameterized types by using results from the definition-site and use-site variance inference. Using the inference we separate parameterized types into two categories. A *variant type* is a parametric type $\texttt{C<}\overline{\texttt{vT}}\texttt{>}$, where generic `C` is safely (definition-site) variant (covariant, contravariant, or bivariant) in at least one of its type parameters; otherwise, we call $\texttt{C<}\overline{\texttt{vT}}\texttt{>}$ an invariant type. Because `Iterator` is covariant in its type parameter, for example, `Iterator<Animal>` is a variant type. Only variant types can be refactored with wildcards, when inferring definition-site variance solely from type signatures.

The influence analysis ignores declarations of the following types. Below we list the types and explain why they are safe to ignore.

1. Primitive types (e.g., `int`, `char`) and monomorphic class types (e.g., `String`, `Object`). These types are not affected by adding wildcards.
2. Type variables that are declared to be the types of declarations that do not affect method overriding. These types cannot be further generalized by wildcards. For example, a field or local variable declared with a type that is a type variable would not be added to the flow graph. However, as explained in Section 4.6, we cannot ignore argument types and returns types of non-static and non-final methods if they are just type variables. Declarations of these types are added to the flow graph.
3. Parametric types that are only specified with bivariant use-site annotations (e.g., `List<?>`). These types cannot be further generalized no matter the rewrites performed.
4. Parametric types that only contain specified use-site annotations that are greater-than or equal-to (according to the ordering of Figure 2) the inferred use-site annotations. The

rewrites performed by the refactoring tool never cause the type of any declaration to require a use-site annotation that is greater-than the inferred use-site annotation. The inferred definition-site variances only assume that the inferred use-site annotations are written in type definitions. As a result, when inferring definition-site variance from only type signatures, variables declared with invariant types are also ignored; in this case, adding a wildcard to a variant type will never cause a wildcard to be added to an invariant type. For example, assuming `Iterator` and `List` are covariant and invariant, respectively, changing a declaration's type from `Iterator<Animal>` to `Iterator<? extends Animal>` will never require a wildcard to be added to the type of a variable declared with `List<Animal>`. In the definition of `Iterator`, `List` could not be applied to `Iterator`'s type parameter without causing `Iterator` to be invariant and no longer be covariant in its type parameter. The variance analysis ensures that only parameterized declarations with an over-specified use-site annotation need to be rewritten.

When performing the method body analysis to infer use-site annotations (as described in Section 5), the inferred use-site annotation in an invariant type may be greater than invariance. For example, a method argument may be declared with an invariant type (`List<Animal>`), but its use of the invariant type may be limited and may support a greater use-site annotation than specified in the original program. If a declaration has an inferred use-site annotation that is greater than the specified annotation, then that declaration will be added to the flow graph. As a result, performing the method body analysis may cause more declarations to be added to the flow graph than with the signature-only analysis because now some declarations of invariant types may be added to the graph. In turn, the number of rewritable declarations may decrease.

7. Evaluation

Our refactoring tool allows users to modularly generalize classes by selecting which declarations (of local variables, fields, method arguments, and return types) to rewrite. Parametric types are generalized by adding wildcard annotations based on inferred definition-site variances.

Past work [1] showed that the majority (53%) of interfaces and a large proportion (27%) of classes in popular, large Java generic libraries are variant even though they were not designed with definition-site variance in mind. This demonstrates the potential impact of the refactoring tool if all declarations were rewritable even for users who are not familiar with definition-site variance.

Not all declarations are rewritable, however, as discussed in previous sections. Changing the type of one variable, for example, may require changing the type of a method argument that is not declared in available source code. To evaluate the potential impact of the refactoring tool, we calculated how many declarations of parameterized types are rewritable. We applied the refactoring tool to six Java libraries, including the core Java library from Oracle's JDK 1.6, i.e., the classes and interfaces of `java.*`. The other libraries are JScience [9], a Java library for scientific computing; Guava [4], a superset of the Google collections library; GNU Trove [10]; Apache Commons-Collection [3]; and JPaul [17], a library supporting program analysis.

The results of our experiment appear in Figures 7 and 8. Overall, we found significant potential for generalizing types, even under the constraints of our flow analysis, which only allows generalization if the type in question does not influence unmodifiable library types. When considering all parameterized types with a method body analysis, 73% of "parameterized decls" or "p-decls" are rewritable or do not influence an unmodifiable type. Figure 8's table contains statistics for "variant decls" or "V-Decls", which are the subset of declarations that are declared with variant types. The majority of variant declarations (57%) can also be rewritten.

"Rewritten P-decls" (and V-decls) are parameterized declarations that not only can be rewritten with wildcards but were also actually generalized by the tool because they contain a specified use-site annotation that is less general than the corresponding inferred use-site annotation. For example, a rewritable declaration can be declared with type, `Iterator<? extends Animal>`; this type does not require a rewrite, however, because the inferred use-site annotation, $+$, is not greater than the specified use-site annotation, $+$.

Even in these sophisticated generic libraries written by experts, who are more disciplined with specifying use-site annotations, we found significant potential for generalizing types. Under the most conservative scenario (considering all parameterized types, examining only type signatures), 11% of all the types that appear anywhere in these libraries are less general than they could be! This number grows to 34% if only variant types are considered. Programmers can use our refactoring to safely perform such rewrites. In these libraries, some variant types were used with more discipline, such as the interface `com.google.common.base.Function<F,T>`, which is contravariant in its "argument" type `F` and covariant in its "return" type `T`. In the class `com.google.common.util.concurrent.Futures`, for example, for many declarations of type `Function`, programmers specified wildcard annotations to reflect the inferred definition-site variances (e.g., `Function<? super I, ? extends O>`). Other variant types had many declarations where use-site annotations were skipped, such as `org.apache.commons.collections15.Transformer<I,O>`, `java.util.Iterator<E>`, and `java.util.Comparator<T>`.

The last two columns in the first table (Figure 7) list the average sizes of the flows-to sets for parameterized declarations. A flows-to set for a declaration `x` is the set of declarations that `x` influences according to our type influence analysis. The flows-to sizes are quite large (146.1 on average), showing that manually checking if a declaration's

Library		# Parameterized Decl Total	# Rewritable P-Decl Total	Rewriteable P-Decl %	Rewritten Total	Rewritten Percentage	Flowsto Avg. Size	Flowsto-R Avg. Size
Java	classes	4900	4284	87%	569	12%	61.10	1.23
		4900	4193	86%	584	12%	61.37	1.16
	interfaces	170	153	90%	20	12%	39.91	2.75
		170	148	87%	34	20%	40.08	2.76
	total	5070	4437	88%	589	12%	60.39	1.29
		5070	4341	86%	618	12%	60.66	1.21
JScience	classes	1553	1042	67%	217	14%	52.04	5.42
		1553	1017	65%	229	15%	54.59	5.49
	interfaces	56	53	95%	43	77%	10.21	0.66
		56	53	95%	44	79%	10.27	0.66
	total	1609	1095	68%	260	16%	50.59	5.19
		1609	1070	67%	273	17%	53.05	5.25
Apache	classes	3357	2567	76%	565	17%	119.81	0.69
		3357	2491	74%	600	18%	122.31	0.72
	interfaces	46	38	83%	1	2%	84.61	0.61
		46	38	83%	16	35%	84.63	0.61
	total	3403	2605	77%	566	17%	119.33	0.69
		3403	2529	74%	616	18%	121.80	0.71
Guava	classes	5794	3973	69%	355	6%	289.27	0.97
		5794	3690	64%	384	7%	313.20	0.93
	interfaces	69	57	83%	2	3%	134.59	3.70
		69	56	81%	2	3%	154.91	3.73
	total	5863	4030	69%	357	6%	287.45	1.01
		5863	3746	64%	386	7%	311.34	0.97
Trove	classes	953	531	56%	127	13%	13.93	0.26
		953	531	56%	139	15%	13.95	0.28
	interfaces	0	0	0%	0	0%	N/A	N/A
		0	0	0%	0	0%	N/A	N/A
	total	953	531	56%	127	13%	13.93	0.26
		953	531	56%	139	15%	13.95	0.28
JPaul	classes	1350	1085	80%	137	10%	15.60	0.50
		1350	1067	79%	187	14%	15.98	0.53
	interfaces	11	11	100%	0	0%	0.73	0.73
		11	11	100%	1	9%	0.73	0.73
	total	1361	1096	81%	137	10%	15.48	0.50
		1361	1078	79%	188	14%	15.86	0.53
Total	**classes**	**17907**	**13482**	**75%**	**1970**	**11%**	**139.21**	**1.28**
		17907	**12989**	**73%**	**2123**	**12%**	**147.74**	**1.26**
	interfaces	**352**	**312**	**89%**	**66**	**19%**	**58.36**	**2.23**
		352	**306**	**87%**	**97**	**28%**	**62.44**	**2.24**
	total	**18259**	**13794**	**76%**	**2036**	**11%**	**137.65**	**1.30**
		18259	**13295**	**73%**	**2220**	**12%**	**146.10**	**1.28**

Figure 7. Variance rewrite statistics for all declarations with generic types. Rewritable decls are those that do not affect unmodifiable code, per our flow analysis. Rewritten decls are those for which we can infer a more general type than the one already in the code. Shaded results are for the method body analysis, unshaded for the signature-only analysis.

Library		# Variant Decls	Rewritable V-Decl Total	Rewritable V-Decl %	Rewritten V-Decl Total	Rewritten V-Decl %
Java	classes	1115	746	67%	563	50%
		1115	708	63%	529	47%
	interfaces	47	31	66%	20	43%
		47	31	66%	21	45%
	total	1162	777	67%	583	50%
		1162	739	64%	550	47%
JScience	classes	717	349	49%	217	30%
		720	350	49%	218	30%
	interfaces	51	48	94%	43	84%
		51	48	94%	43	84%
	total	768	397	52%	260	34%
		771	398	52%	261	34%
Apache	classes	1197	759	63%	544	45%
		1201	730	61%	532	44%
	interfaces	6	2	33%	1	17%
		6	2	33%	1	17%
	total	1203	761	63%	545	45%
		1207	732	61%	533	44%
Guava	classes	1906	1088	57%	355	19%
		1906	990	52%	336	18%
	interfaces	11	8	73%	2	18%
		11	8	73%	2	18%
	total	1917	1096	57%	357	19%
		1917	998	52%	338	18%
Trove	classes	367	226	62%	127	35%
		367	226	62%	127	35%
	interfaces	0	0	0%	0	0%
		0	0	0%	0	0%
	total	367	226	62%	127	35%
		367	226	62%	127	35%
JPaul	classes	253	139	55%	137	54%
		260	146	56%	144	55%
	interfaces	0	0	0%	0	0%
		0	0	0%	0	0%
	total	253	139	55%	137	54%
		260	146	56%	144	55%
Total	**classes**	**5555**	**3307**	**60%**	**1943**	**35%**
		5569	**3150**	**57%**	**1886**	**34%**
	interfaces	**115**	**89**	**77%**	**66**	**57%**
		115	**89**	**77%**	**67**	**58%**
	total	**5670**	**3396**	**60%**	**2009**	**35%**
		5684	**3239**	**57%**	**1953**	**34%**

Figure 8. Variance rewrite statistics for declarations with variant types (i.e., using generics that are definition-site variant). Rewritable decls are those that do not affect unmodifiable code, per our flow analysis. Rewritten decls are those for which we can infer a more general type than the one already in the code. Shaded results are for the method body analysis, unshaded for the signature-only analysis. There are slightly more variant decls in the method body analysis because more generics are variant.

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

```
Animal first(List<Animal> l) {
 Iterator<Animal> itr =
   l.iterator();
 return itr.next(); }
```
After Kiezun et al. refactoring

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

Figure 9. Refactoring resulting from applying Kiezun et al.'s and then our refactoring tool

Library	Signature-Only Runtime	Method-Body Runtime
Java	122.557	132.180
JScience	1.696	2.444
Apache	5.383	6.144
Guava	14.907	18.021
Trove	2.773	2.996
JPaul	0.610	0.973

Figure 10. Time in seconds to infer definition-site variance of generic type parameters and refactor libraries.

type is rewritable is tedious and error-prone. The "Flowsto-R" column lists the average sizes of flows-to sets only for declarations that are rewritable. As expected, the rewritable declarations typically influence fewer declarations than non-rewritable ones.

To show that our refactoring tool is efficiently usable (i.e., runs in seconds rather than hours), Figure 10 lists the runtimes of the refactoring tool for each library for both types of analyses. Refactoring the JDK only took about two minutes even though it contains over 198K lines of code, thousands of parameterized declarations, and 182 declared generic type parameters to infer definition-site variance for.

8. Comparison to Related Work

We next compare our work to past approaches that infer use-site variance annotations.

Kiezun et al. [15] offered an automated approach to adding type parameters to existing class definitions. The introduction of type parameters to a class often requires instantiating generics or determining the type arguments to instantiate uses of a generic. Kiezun et al.'s approach may instantiate a generic with a wildcard annotation but only when it is *required*, for example, to preserve a method override. Consider a non-generic class `D` that (1) extends the non-generic version of the class `TreeSet` and (2) contains a method `addAll(Collection c1)` that overrides a method in `TreeSet`. In the generic version of `TreeSet<E>`, the `addAll` method has the signature `addAll(Collection<? extends E> c2)`. Class `D` can be parameterized with type parameter `E` and then can extend the generic version, `TreeSet<E>`. Preserving the method override of `addAll` requires changing the method argument `c1`'s type to `Collection<? extends E>`.

A new wildcard is introduced only when an existing wildcard from the original program requires the new wildcard to preserve the ability to compile the code and to preserve method overrides. Kiezun et al.'s approach does not infer definition-site variance and would not introduce a wildcard if the original program does not access a declaration with a wildcard in its type. However, Kiezun et al.'s proposed refactoring would be a useful preprocessing step to our refactoring tool. After classes are parameterized with type parameters, our tool can take advantage of the variance inference to add wildcards to support greater reuse. This series of steps, for instance, could perform the refactoring in Figure 9. Kiezun et al.'s approach would not add wildcards in this example because wildcards are not required for the program to compile. Our refactoring tool can infer that `Iterator` is covariant and that method argument `l` is only using the covariant operations of `List` in `foo`'s method body.

Craciun et al. [7, 8] offered an approach to inferring use-site annotations, where a parametric type is modeled as an interval type with two bounds: A lower bound and an upper bound for stating the types of objects that can be written to or read from, respectively, an instance of a generic. In this calculus, supertypes have wider ranges: $\texttt{List<[T}_L\texttt{,T}_U\texttt{]>} <: \texttt{List<[S}_L\texttt{,S}_U\texttt{]>}$, if $\texttt{S}_L <: \texttt{T}_L$ and $\texttt{T}_U <: \texttt{S}_U$. Types `List<[T,T]>`, $\texttt{List<[}\bot\texttt{,T]>}$, and $\texttt{List<[T,}\top\texttt{]>}$ are abbreviated as $\texttt{List<}\odot\texttt{T>}$, $\texttt{List<}\oplus\texttt{T>}$, and $\texttt{List<}\ominus\texttt{T>}$, where $\bot$ and $\top$ are subtypes and supertypes of every type, respectively. Furthermore, the type of the `this` reference must be specified for each instance method in a generic. An example class in their language is provided below (with the leftmost type of each signature corresponding to the type of `this`):

```
class List<A> {
 List<⊕A> | A getFst() { ... }
 List<⊖A> | void setFst(A a) { ... }
 public final Comparator<⊖A> comp;
}
```

When fields are declared with parametric types instead of type variables, however, Craciun et al.'s approach does not infer the greatest use-site variance that supports all of the available operations. A contravariant instantiation of the `List` class above, $\texttt{List<}\ominus\texttt{T>}$, should be able to read the `comp` field as an instance of $\texttt{Comparator<}\ominus\texttt{T>}$. However, Craciun et al.'s type promotion technique [7, Section 4.2], would only be able to read a $\texttt{Comparator<}\oplus\texttt{T>}$ from the `comp` field of a $\texttt{List<}\ominus\texttt{T>}$. A $\texttt{Comparator<}\oplus\texttt{T>}$ cannot access its `compare` method. Furthermore, if a method argument `x` of type `List<T>` was used in the method body only in the assignment "`Comparator<⊖T> c = x.comp;`", the use-site an-

notation in the type of x inferred by the approach would be invariance. Our tool, however, would rewrite the type of x with a greater (contravariant) use-site annotation; more generally, our approach infers more liberal use-site variances when fields are of parametric types.

9. Conclusions

We presented an automated approach to generalizing Java generics with wildcards. We developed a refactoring tool that infers definition-site variance and adds wildcard annotations to the types of declarations. It allows users to select a subset of declarations to generalize and performs a type influence analysis. The analysis automatically determines which declarations to generalize and which to ignore. In six sophisticated Java generic libraries written by experts, we found that 34% of the uses of variant generics could have been specified with more general wildcards. The result of this refactoring is a more general interface that supports greater software reuse.

Acknowledgments

We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant (PADECL) and a European Research Council Starting/Consolidator grant (SPADE); and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award (MORPH-PL).

References

[1] J. Altidor, S. S. Huang, and Y. Smaragdakis. Taming the wildcards: Combining definition- and use-site variance. In *Programming Language Design and Implementation (PLDI)*, 2011.

[2] J. Altidor, C. Reichenbach, and Y. Smaragdakis. Java wildcards meet definition-site variance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[3] Apache. Apache commons-collections library. http://larvalabs.com/collections/. Version 4.01.

[4] K. Boumillion and J. Levy. Guava: Google core libraries for Java 1.5+. http://code.google.com/p/guava-libraries/. Release 8.

[5] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2008.

[6] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 1989.

[7] W.-N. Chin, F. Craciun, S.-C. Khoo, and C. Popeea. A flow-based approach for variant parametric types. In *Proceedings of Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[8] F. Craciun, W.-N. Chin, G. He, and S. Qin. An interval-based inference of variant parametric types. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, 2009.

[9] J.-M. Dautelle et al. Jscience. http://jscience.org/. Version 4.3.

[10] E. Friedman and R. Eden. Gnu Trove: High-performance collections library for Java. http://trove4j.sourceforge.net/. Version 2.1.0.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. California, USA, 7th edition, 2012.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[13] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006.

[14] A. Kennedy and B. Pierce. On decidability of nominal subtyping with variance. *FOOL/WOOD*, 2007.

[15] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[16] M. Odersky. The Scala Language Specification v 2.9. http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2014.

[17] A. Salcianu. Java program analysis utilities library. http://jpaul.sourceforge.net/. Version 2.5.1.

[18] R. Tate, A. Leung, and S. Lerner. Taming wildcards in Java's type system. In *Programming Language Design and Implementation (PLDI)*, 2011.

A. Method Body Analysis: Constraints on Use-Site Annotations

The heart of method body analysis is the production of constraints bounding use-site variances, $uvar(\texttt{X}_i;\ \texttt{C};\ \texttt{y})$. These bounds ensure that the inferred use-site annotation supports the limited use of y in its enclosing method body. The bounds on use-site variances can be more relaxed than the bounds on definition-site variances. A definition-site variance is constrained by the variance of all members of a generic. A use-site variance for a method argument y can be more general because it needs to be constrained by the variance of only those members accessed by y in the method body.

Consider the argument source of method addAll from line 7 in the motivating example (Section 2). The type of source is List<*o*E>. When the method body analysis is performed, source's inferred use-site annotation is the value of the expression: $dvar(\texttt{E}; \texttt{List}) \sqcup o \sqcup uvar(\texttt{E}; \texttt{List}; \texttt{source})$. The definition-site variance and specified use-site variance were further relaxed by $uvar(\texttt{E}; \texttt{List}; \texttt{source})$ to take advantage of the fact that not all members of List were accessed by source in the method body of addAll. We computed that only a covariant version of List was required by source; formally, we computed $uvar(\texttt{E}; \texttt{List}; \texttt{source}) = +$. In this case, $uvar(\texttt{E}; \texttt{List}; \texttt{source})$ was only constrained by the variance of the type signature of the method List.iterator. It was the only method from List used and no other uses of source occurred in the method body. As a result, the only upper bound on $uvar(\texttt{E}; \texttt{List}; \texttt{source})$ is $var(\texttt{E}; \texttt{Iterator<E>}) = +$. (Since return types are in a covariant position.) This is the same upper bound on $dvar(\texttt{E}; \texttt{List})$ that results from List.iterator alone, but $dvar(\texttt{E}; \texttt{List})$ also needs to respect other constraints.

***uvar* constraint generation:** (representative rules)

$$\frac{M = \mathit{enclosingMethod}(\mathtt{x}) \quad \text{“}\mathtt{x.f}\text{”} \in M \quad \mathit{LookupType}(\mathtt{f}) = \mathtt{T} \quad \neg\mathit{isWriteTarget}(\mathtt{x.f})}{\mathit{uvar}(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \mathit{var}(\mathtt{Y};\mathtt{T})} \ (\text{MB-FieldRead})$$

$$\frac{M = \mathit{enclosingMethod}(\mathtt{x}) \quad \text{“}\mathtt{x.f = e}\text{”} \in M \quad \mathit{LookupType}(\mathtt{f}) = \mathtt{T}}{\mathit{uvar}(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq - \otimes \mathit{var}(\mathtt{Y};\mathtt{T})} \ (\text{MB-FieldWrite})$$

$$\frac{M = \mathit{enclosingMethod}(\mathtt{x}) \quad \text{“}\mathtt{x.<\overline{S}>m(\overline{e})}\text{”} \in M \quad \mathit{Lookup}(\mathtt{m};\overline{\mathtt{e}}) = \mathtt{<\overline{Y \lhd U}>\ T\ m(\overline{T\ x})\ \{\ ...\ \}}}{\mathit{uvar}(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \sqcap_{i=1}^{|\overline{\mathtt{U}}|}\big(- \otimes \mathit{var}(\mathtt{Y};\mathtt{U})\big) \ \sqcap \mathit{var}(\mathtt{Y};\mathtt{T}) \sqcap_{i=1}^{|\overline{\mathtt{T}}|} \big(- \otimes \boxed{\mathit{localvar}(\mathtt{Y};\mathtt{T}_i;\mathtt{x}_i)}\big)} \ (\text{MB-MethodCall})$$

$$\frac{\mathtt{y} \in \mathit{hierarchyParams}(\mathtt{x}) \quad \mathit{LookupType}(\mathtt{y}) = \mathtt{C<\overline{vT}>}}{\mathit{uvar}(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \boxed{\mathit{localvar}(\mathtt{Y};\mathtt{C<\overline{vT}>};\mathtt{y})}} \ (\text{MB-Override})$$

$$\frac{M = \mathit{enclosingMethod}(\mathtt{x}) \quad \text{“}\mathtt{y = x}\text{”} \in M \quad \mathit{LookupType}(\mathtt{y}) = \mathtt{C<\overline{vT}>} \quad \mathtt{Y} = i^{\text{th}} \text{ type parameter of } \mathtt{C}}{\mathit{uvar}(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \boxed{\mathit{inferredUseSite}(\mathtt{y};\overline{\mathtt{v}};\mathtt{C};i)}} \ (\text{MB-AssignToGeneric-Same})$$

$$\frac{M = \mathit{enclosingMethod}(\mathtt{x}) \quad \text{“}\mathtt{y = x}\text{”} \in M \quad \mathit{LookupType}(\mathtt{y}) = \mathtt{D<\overline{vT}>} \quad \mathtt{C} \neq \mathtt{D}}{\mathit{uvar}(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \mathit{dvar}(\mathtt{Y};\mathtt{C})} \ (\text{MB-AssignToGeneric-Base})$$

$$\textbf{\textit{localvar}}(\mathtt{Y};\mathtt{T};\mathtt{x}) = \begin{cases} \sqcap_{i=1}^{|\overline{\mathtt{T}}|}\big(\boxed{\mathit{inferredUseSite}(\mathtt{x};\overline{\mathtt{v}};\mathtt{C};i)} \otimes \mathit{var}(\mathtt{Y};\mathtt{T}_i)\big), \text{ if } \mathtt{T} = \mathtt{C<\overline{vT}>} \text{ and } \mathit{hierarchyMaybeGeneralized}(\mathtt{x}) \\ \mathit{var}(\mathtt{Y};\mathtt{T}), \text{ otherwise} \end{cases}$$

$$\textbf{\textit{inferredUseSite}}(\mathtt{y};\overline{\mathtt{v}};\mathtt{C};\mathtt{i}) = \begin{cases} \mathit{dvar}(\mathtt{X}_i;\mathtt{C}) \sqcup \mathtt{v}_i \sqcup \boxed{\mathit{uvar}(\mathtt{X}_i;\mathtt{C};\mathtt{y})}, \text{ if } \mathtt{y} \text{ is a method argument} \\ \mathit{dvar}(\mathtt{X}_i;\mathtt{C}) \sqcup \mathtt{v}_i, \text{ otherwise, where } \mathtt{X}_i \text{ is the } i^{\text{th}} \text{ type parameter of } \mathtt{C}. \end{cases}$$

hierarchyMaybeGeneralized$(\mathtt{x}) \equiv \forall \mathtt{y} \in \mathit{hierarchyParams}(\mathtt{x}), \mathtt{y}$ is declared in available source $\wedge$ *sameGeneric*$(\mathtt{y};\mathtt{x})$.
sameGeneric$(\mathtt{x};\mathtt{y}) \equiv \mathit{LookupType}(\mathtt{x}) = \mathtt{C<\overline{vT}>} \wedge \mathit{LookupType}(\mathtt{y}) = \mathtt{C<\overline{wU}>} \wedge |\overline{\mathtt{v}}| = |\overline{\mathtt{w}}|$.
LookupType$(\mathtt{x})$ = the declared type of variable `x`.
isWriteTarget$(\mathtt{e}) \equiv$ `e` is the target (left-hand side) of an assignment.

Figure 11. Constraint Generation from Method Bodies. Shaded parts show where *uvar* constraints differ from the corresponding *dvar* constraint of the signature-only analysis.

Figure 11 contains the constraint generation rules for *uvar*s and auxiliary functions. The first three (MB) rules constrain *uvar*$(\mathtt{Y};\mathtt{C};\mathtt{x})$ by the variance of `Y` in the (non-static) members of `C` accessed by `x` in its enclosing method body. A field read is only a covariant use of its type `T`; *var*$(\mathtt{Y};\mathtt{T})$ was not transformed by $+$ in rule MB-FieldRead because $+$ is the identity element on the transform operator $\otimes$. Note that this constraint also occurs for definition-site variances; see rule W-Cls from [2, Figure 5] for details.

Constraints with *uvar*s are generated from method arguments using the auxiliary function *localvar*. *localvar* may return an expression with a *uvar* to signal that a use-site annotation can be inferred. *localvar* is not recursive and *uvar*s are only generated for top-level use-site annotations. We chose not to generalize use-site annotations in nested types for simplicity.

Section 5 did not need to mention *localvar* to describe the method body analysis at a high level. However, the actual upper bound generated for *dvar*$(\mathtt{X};\mathtt{Body})$ is $-\otimes$ *localvar*$(\mathtt{X};\mathtt{List<}o\mathtt{E>};\mathtt{lx}) = -\otimes\big(\mathit{dvar}(\mathtt{E};\mathtt{List}) \sqcup o \sqcup \mathit{uvar}(\mathtt{E};\mathtt{List};\mathtt{lx})\big)$, so the initial upper bound expression simplifies to the expression previously stated.

localvar returns an expression with a *uvar* for a method argument `x` if it is declared with a parametric type and if it satisfies *hierarchyMaybeGeneralized*$(\mathtt{x})$. *hierarchyMaybeGeneralized*$(\mathtt{x})$ is imposed to help preserve method overrides in refactored code. Use-site annotations in each position must be the same in overridden/overriding methods. We do not infer use-site annotations if an overridden method is not available from source. We also do not infer use-site annotations for `x` if one of its corresponding parameters from a overridden/overriding method is not declared with a parametric type of the same generic. One such example is method argument `entry` declared on line 23 from Section 2. Changing a use-site annotation in the type of `entry` would cause the `add` method of `MapEntryWList` to no longer override `add` in `WList`. When *hierarchyMaybeGeneralized*$(\mathtt{x})$ is satisfied, rule MB-Override ensures that each inferred use-site annotation is the same in each position for all of the overridden/overriding methods.

Rule MB-AssignToGeneric-Same handles the case when a method argument `x` is assigned to another variable `y` that are both parametric types of the same generic. Promoting a use-site annotation in the type of `x` may require promoting the type of `y`. Consider again argument `source` of method `addAll` on line 7. If the statement “`List<E> list2 = source;`” was added to the beginning of the method body of `addAll`, then changing only the type of `source` to `List<? extends E>` would cause the method to no longer type check; the type of the left-hand side of the assignment, `list2`, would no longer be a supertype of the right-hand side, `source`. The influence analysis from Section 4 would also detect that `source`’s type influences `list2`’s type. To make the upper bounds on *uvar*s less restrictive, we perform a more precise analysis for generating constraints on *uvar*s.

The expression "y = x.f" does not cause rule MB-AssignToGeneric-Same to generate a constraint using y's type even though the influence analysis detects that x's type influence y's type. Instead, rule MB-FieldRead is applied to reflect that expression "y = x.f" is only a covariant use of the field type of f. In the actual implementation, rule MB-AssignToGeneric-Same also applies when x is an expression that has a destination node (Figure 5) but x is not a qualifier in the expression. This handles the case when "**return** x ;" occurs in method M and other similar cases.

Rule MB-AssignToGeneric-Base handles the other assignment case from x to y when y is declared with a parametric type of a different generic D<$\overline{\text{wU}}$> than that used in the type of x, C<$\overline{\text{vT}}$>, where C $\neq$ D. This can occur when C<$\overline{\text{vT}}$> $<:$ D<$\overline{\text{wU}}$>. This subtyping relationship may be derived when there exists another instantiation of the base type, D<$\overline{\text{v}'\text{S}'}$>, such that (1) C<$\overline{\text{vT}}$> $<:$ D<$\overline{\text{v}'\text{S}'}$> holds not because of variance but instead by the class hierarchy and (2) C<$\overline{\text{vT}}$> $<:$ D<$\overline{\text{v}'\text{S}'}$> $<:$ D<$\overline{\text{wU}}$>. Considering the class "class Pair<X,Y> extends Box<X> {}" for example, Pair<? extends Dog, String> $<:$ Box<? extends Dog> $<:$ Box<? extends Animal>. Also, Pair<S,?> $<:$ Box<? extends S> but Pair<?,T> $\not<:$ Box<? extends S>. More generally, if y is of type Box<vS> and x is of type Pair<S,T>, generating the most relaxed but safe constraint for the assignment "y = x" requires computing the most general instantiation of Pair that is a subtype of Box<vS>. Rather than compute such an instantiation of C in rule MB-AssignToGeneric-Base, we chose to simplify the analysis by restricting the inferred use-site annotation to its corresponding definition-site variance; this is safe because definition-site variances support all uses of a generic definition.

B. Soundness

This section provides a sketch of a rigorous argument for why our algorithm for generalizing types with wildcards is sound. There are two important soundness questions relevant to our refactoring tool. First, are the "correct" wildcard annotations being generated? That is, will the inferred, more general, types support all of the original operations available to clients, so that the refactoring will not break any client code? Second, does the type influence graph record all of the necessary dependencies between declarations?

The second soundness question can be easily verified for a given language construct. Examining the type checking rule for an assignment expression, for example, one can verify that generalizing the type of the right-hand side may require generalizing the type of the left-hand side but not the other way around. Effectively, the type influence graph encodes an overapproximation of the dependencies in our typing rules, feature-by-feature. Therefore, answering this soundness question formally for the full Java language would be tedious, error-prone, and would focus on technicalities that do not provide fundamental insight on when code can be generalized with wildcards. Rather, we provide empirical evidence that our influence analysis is sound by recompiling the six large libraries of our study after the refactoring was performed.

Given that our influence graph records all of the necessary dependencies, it is safe to rewrite the types of all declarations in a path in the graph (assuming all declarations in the path are rewritable). Specifically, rewriting the types of all declarations in the path preserves the subtype relationships between the types of expressions from the original program. This property is a consequence of Lemma 6, which is stated later in this section.

To answer the first question, we apply many properties proven in prior work [1, 2], adapted to our refactoring setting. The essence of the proofs is similar to this past work, but the precise statements are different, due to the unique elements of our approach. First, we cannot just refer to definition-site variance, which Java does not support, but instead emulate it via refactoring-induced use-site variance. Second, we need to also integrate method body analysis, which we do later, as a separate step.

Our definition-site variance inference algorithm was proven sound in [1]. This algorithm computes how a *new* type system, ignoring Java intricacies, can *infer* definition-site variance and, thus, generalize method signatures transparently. Subsequently, the VarJ calculus [2] modeled faithfully a subset of Java that supports language features with complex interactions with variance, such as F-bounded polymorphism [6] and wildcard capture. VarJ extends the Java type system with *declared* (not inferred) definition-site variance and shows that this extension is sound. Our current system borrows from both of the above formalisms by first inferring definition-site variance [1], and then emulating it with use-site variance in a more complete setting, borrowed from VarJ [2].

The type soundness proof of VarJ requires that subtyping relationships concluded using definition-site variance annotations also satisfy the subsumption principle, where subtypes can perform the operations available to the supertype. Another consequence of satisfying that requirement is that the refactored type is not only a supertype of the original type but it is also safe to assume that the refactored type is a *subtype* of the original type. In other words, types that occur in the refactored program support all of the operations available in the original program; otherwise, the refactored code would not compile because an operation is being performed in the code that is no longer supported by a refactored type.

The proofs in [1, 2] ensure that the refactored types are subtypes of the original types according to the subtype relation with definition-site subtyping. However, Java does not support definition-site variance, and the subtype relation as defined in the JLS [11] does not conclude subtype relationships using inferred definition-site variances. To relate our subtype relation with that of the JLS, we define another sub-

type relation $<:_{\text{JLS}}$ that is the same as $<:$ except $<:_{\text{JLS}}$ does not support a definition-site subtyping rule. For example, although `Iterator` is inferred to be covariant in its type parameter, `Iterator<Dog>` $\not<:_{\text{JLS}}$ `Iterator<Animal>`. The following lemma establishes that definition-site variance can be simulated with use-site variance.

Lemma 1 (Use-site can simulate def-site)**.** If $\mathtt{T} <: \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>}$, then $\mathtt{T} <:_{\text{JLS}} \mathtt{C}\texttt{<}\overline{\mathtt{wT}}\texttt{>}$, where $\mathtt{w}_i = \mathit{dvar}(\mathtt{X}_i;\mathtt{C}) \sqcup \mathtt{v}_i$, for each $i \in |\overline{\mathtt{w}}|$.

This lemma establishes that any additional subtype relationships that hold for $<:$ but do not hold for $<:_{\text{JLS}}$ are a result of definition-site variance inference. Also, a program still compiles if types are generalized only by joining their use-site annotations with inferred definition-site variances. For example, suppose in a program that type checks that there is an assignment `x = y`, where `x` and `y` are declared with types `Iterator<Animal>` and `T`, respectively. Since the program type checks, we know $\mathtt{T} <:$ `Iterator<Animal>`. The refactoring tool may change the type of `y` to a greater supertype $\mathtt{T}'$. Since the refactored type is always a subtype of the original type according to $<:$, $\mathtt{T}' <: \mathtt{T} <:$ `Iterator<Animal>`. Lemma 1 implies that $\mathtt{T}' <:_{\text{JLS}}$ `Iterator<? extends Animal>`. So changing the type of `x` to the latter type ensures the program still compiles, as far as assignments to `x` are concerned. In this example, it could have been the case that `T = Iterator<Animal>` and $\mathtt{T}'$ = `Iterator<? extends Animal>`.

Although only the types of declarations (e.g., fields) are rewritten by the tool, we will prove that the types of *all* expressions in the refactored program can safely be subtypes of the corresponding types in the original program. This ensures nested expressions are also able to be perform operations from the original program.

The soundness proof in [1] guarantees that $\mathtt{C}\texttt{<}(v_u \sqcup v_d)\mathtt{T}\texttt{>} <: \mathtt{C}\texttt{<}v_u\mathtt{T}\texttt{>}$ is safe, where v_d is a safe definition-site variance for `C`. Considering `Iterator` is covariant, for example, this property implies that `Iterator<?>` $<:$ `Iterator<? super Animal>` is safe to conclude. The refactoring tool would replace the latter type with the former. We state this property formally in the following lemma.

Lemma 2 (Def-site joining does not further generalize)**.** Let `C` be a generic class such that $\mathtt{CT}(\mathtt{C}) =$ "`class C<`$\overline{\mathtt{X}}$`> ...`". Then $\mathtt{C}\texttt{<}(\overline{\mathtt{w} \sqcup \mathtt{v})\mathtt{T}}\texttt{>} <: \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>}$, where $\mathtt{w}_i = \mathit{dvar}(\mathtt{X}_i;\mathtt{C})$, for each $i \in |\overline{\mathtt{w}}|$.

Additionally, in order for the refactored code to compile, the operations performed on the original types must also be able to be performed on the refactored types. To describe this property precisely, first, we assume that there is an auxiliary (partial) function $\mathit{ftype}(\mathtt{f};\, \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>})$ such that given the name `f` of a field that exists in generic class `C` and an instantiation of the generic $\mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>}$, $\mathit{ftype}(\mathtt{f};\, \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>})$ returns the type of the field for that instantiation. Also, we assume that there is a (partial) function $\mathit{mtype}(\mathtt{m};\, \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>})$ such that given the name `m` of a method that exists in generic class `C` and an instantiation of the generic $\mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>}$, $\mathit{mtype}(\mathtt{m};\, \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>})$ returns the type signature of the method for that instantiation. Formal definitions of *ftype* and *mtype* can be found in [2].

Since refactored types are subtypes of original types, showing that operations can be performed on the refactored types amounts to showing that the subtyping relation satisfies the subsumption principle. This is established by the following two lemmas that were proven in [2]. The proofs of these lemmas rely on intricate reasoning involving the variances of types positions in class definitions, and the subtype lifting lemma [2, Lemma 1], which establishes the key relationship between variance and subtyping.

Lemma 3 (Subtyping Specializes Field Type)**.** If $\mathtt{T}' <: \mathtt{T}$ and $\mathit{ftype}(\mathtt{f};\mathtt{T}) = \mathtt{U}$, then $\mathit{ftype}(\mathtt{f};\mathtt{T}') <: \mathtt{U}$.

If lemma 3 were not true, the subtyping relation would violate the subsumption principle; in that case, the supertype `T` could return a `U` from its field `f` but the subtype $\mathtt{T}'$ could not.

To satisfy the subsumption principle, a method's type signature for the subtype must be a subtype of its type signature for the supertype. Lemma 4 states this precisely.

Lemma 4 (Subtyping Specializes Method Type)**.** If $\mathtt{T}' <: \mathtt{T}$ and $\mathit{mtype}(\mathtt{m};\mathtt{T}) = \texttt{<}\overline{\mathtt{Z} \lhd \mathtt{S}}\texttt{>}\ (\overline{\mathtt{U}}) \to \mathtt{V}$, then $\mathit{mtype}(\mathtt{m};\mathtt{T}') = \texttt{<}\overline{\mathtt{Z} \lhd \mathtt{S}'}\texttt{>}\ (\overline{\mathtt{U}'}) \to \mathtt{V}'$ such that (1) $\overline{\mathtt{U} <: \mathtt{U}'}$, (2) $\overline{\mathtt{S} <: \mathtt{S}'}$, and (3) $\mathtt{V}' <: \mathtt{V}$.

To formally argue that the refactoring preserves compilation, we model the refactoring tool using the FGJ* syntax and similar notation used in FGJ. Recall that an FGJ program is a pair (`CT`, `e`) of a class table `CT` that maps class names to class definitions and an expression `e` representing the main method. The refactoring tool is modeled by a function R that maps elements from an FGJ program to elements in the refactored program (R(`CT`), `e`). R is not applied to `e` because the refactoring tool does not modify term expressions. It only changes the types of declarations, which only occur in the class table.

The typing judgment $\mathtt{CT} \vdash \mathtt{e} : \mathtt{T}$ denotes that expression `e` has type `T` given class table `CT`.[9] The following key theorem is satisfied by the refactoring tool and establishes that the refactoring preserves compilation.

Theorem 1 (Refactored Types Are Safe)**.** Suppose $\mathtt{CT} \vdash \mathtt{e} : \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>}$, where $\mathtt{CT}(\mathtt{C}) =$ "`class C<`$\overline{\mathtt{X}}$`> ...`". Then $R(\mathtt{CT}) \vdash \mathtt{e} : \mathtt{C}\texttt{<}\overline{\mathtt{v}'\mathtt{T}}\texttt{>}$, where for each $i \in |\overline{\mathtt{T}}|$,

$$v_i' \sqsubseteq \begin{cases} \mathit{dvar}(\mathtt{X}_i;\mathtt{C}) \sqcup \mathtt{v}_i \sqcup \mathit{uvar}(\mathtt{X}_i;\mathtt{C};\mathtt{y}), \\ \quad \text{if } \mathtt{e} = \mathtt{y} \text{ and } \mathtt{y} \text{ is a method argument} \\ \mathit{dvar}(\mathtt{X}_i;\mathtt{C}) \sqcup \mathtt{v}_i, \text{ otherwise.} \end{cases}$$

This theorem states that the use-site variances in the type of every expression in the refactored program are bounded by the join of the use-site variances in the types in the original program and the corresponding definition-site variances, if the signature-only based variance analysis is performed.

[9] The typing judgment in FGJ takes in more parameters such as a type variable context Δ and an expression variable context Γ. We skip these parameters because the exact typing rules are not the focus of this paper.

If the method body analysis is also performed, then expressions that are method arguments may be further promoted by the inferred use-site needed to support only the operations performed in the method body. The upper bounds of the $\overline{\mathtt{v}'}$ ensure that every expression can support the operations performed in the original program. Although it is not safe to assume $\mathtt{C<\overline{v'T}>} <: \mathtt{C<\overline{vT}>}$ in general, it is safe to assume that subtype relationship for a particular method.

We will sketch the proof of this theorem. To clarify the presentation, we first prove the theorem for the case where the refactoring tools performs just the signature-only analysis. Later, we will cover the case when the method-body analysis is performed. Also, for simplicity, we ignore the type influence analysis by assuming that all declarations are declared in source, and that the types of all declarations are generalized. Finally, we assume that the type system exhibits a subsumption typing property: an expression can be typed with any supertype of its most specific type. Formally, if $\mathtt{e} : \mathtt{T}$ and $\mathtt{T} <:_{\text{JLS}} \mathtt{U}$, then $\mathtt{e} : \mathtt{U}$. Assuming this typing rule is safe because we know that the subtype relation satisfies the subsumption principle. Subsumption typing rules were defined in both type systems for TameFJ [5] and VarJ. We use relation $<:_{\text{JLS}}$ instead of $<:$ because we only want to derive typing judgments that a standard Java compiler would infer.

Since we are assuming the signature-only analysis, we can define the refactoring tool function R over all type expressions in a program: $R(\mathtt{C<\overline{wT}>}) = \mathtt{C<\overline{vT}>}$, where $\mathtt{w}_i = \mathit{dvar}(\mathtt{X}_i; \mathtt{C}) \sqcup \mathtt{v}_i$ and $\mathtt{X}_i$ is the ith type parameter of $\mathtt{C}$. Also, $R(\mathtt{X}) = \mathtt{X}$, where $\mathtt{X}$ is a type variable. Given this definition, we state two important properties:

Lemma 5 (Refactored Type is Subtype of Original)**.** $R(\mathtt{T}) <: \mathtt{T}$.

Lemma 6 (Refactoring Preserves Subtyping)**.** If $\mathtt{T} <: \mathtt{U}$, then $R(\mathtt{T}) <:_{\text{JLS}} R(\mathtt{U})$.

Lemma 5 states that the refactored type is a subtype of the original, and it holds because of lemma 2, Lemma 6 establishes that the refactoring preserves subtype relations from the original program; it is easy to show using lemmas 2 and 1. We restate theorem 1 with the signature-only based refactoring:

Theorem 2 (Refactored Types Are Safe for Sig-Only)**.** If $\mathtt{CT} \vdash \mathtt{e} : \mathtt{T}$, then $R(\mathtt{CT}) \vdash \mathtt{e} : R(\mathtt{T})$. We prove this by structural induction on expression $\mathtt{e}$.

Case: $\mathtt{e} = \mathtt{y}$. **Proof**: This proof case is trivial. Because every declaration is generalized, the refactoring tool changes the type of $\mathtt{y}$ from $\mathtt{T}$ to $R(\mathtt{T})$. □

Case: $\mathtt{e} = \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$. **Proof**: In the original program, $\mathtt{e}$ has type $\mathtt{N}$. It also has type $\mathtt{N}$ in the refactored program, if it type checks. Furthermore, $\mathtt{e}$ would have type $R(\mathtt{N})$, by the subsumption typing rule, since $\mathtt{N} <:_{\text{JLS}} R(\mathtt{N})$. So we only need to show that this expression still type checks in the refactored program. Since $\mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$ has a type, by inversion (similar to [5, Lemma 31]), we know that for each $i \in |\overline{\mathtt{e}}|$, the actual argument $\mathtt{e}_i$ also has a type that is a subtype of the type of the ith formal argument of the constructor for $\mathtt{N}$. We show that this is also the case in the refactored program.

Let $\mathtt{T}_i$ be the type of $\mathtt{e}_i$ and $\mathtt{U}_i$ by the type of the ith formal argument of the constructor for $\mathtt{N}$. By the inductive hypothesis, $R(\mathtt{CT}) \vdash \mathtt{e}_i : R(\mathtt{T}_i)$. As discussed above, it must be the case that $\mathtt{T}_i <:_{\text{JLS}} \mathtt{U}_i$. By lemma 6, this implies $R(\mathtt{T}_i) <:_{\text{JLS}} R(\mathtt{U}_i)$. Since i was arbitrary in $|\overline{\mathtt{e}}|$, $\mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$ type checks in the refactored program. Therefore, $R(\mathtt{CT}) \vdash \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}}) : \mathtt{N}$. □

Case: $\mathtt{e} = \mathtt{e}'.\mathtt{f}$. **Proof**: Since $\mathtt{CT} \vdash \mathtt{e}'.\mathtt{f} : \mathtt{T}$, then by inversion, we have the judgments (1) and (2) below:

1. $\mathtt{CT} \vdash \mathtt{e}' : \mathtt{U}$
2. $\mathit{ftype}(\mathtt{f}; \mathtt{U}) <: \mathtt{T}$
3. $R(\mathtt{CT}) \vdash \mathtt{e}' : R(\mathtt{U})$, by applying the inductive hypothesis to (1).
4. $R(\mathtt{U}) <: \mathtt{U}$, by lemma 5.
5. $\mathit{ftype}(\mathtt{f}; R(\mathtt{U})) <: \mathit{ftype}(\mathtt{f}; \mathtt{U}) <: \mathtt{T}$, by applying lemma 3 to (4) and (2).
6. $\mathit{ftype}(\mathtt{f}; R(\mathtt{U})) <:_{\text{JLS}} R(\mathtt{T})$, by applying lemma 1 to (5).
7. $R(\mathtt{CT}) \vdash \mathtt{e}'.\mathtt{f} : R(\mathtt{T})$, by (3), (6), and the subsumption typing rule. □

Case: $\mathtt{e} = \mathtt{e}'.\mathtt{<\overline{S}>m}(\overline{\mathtt{e}})$. **Proof**: Since $\mathtt{CT} \vdash \mathtt{e}'.\mathtt{<\overline{S}>m}(\overline{\mathtt{e}}) : \mathtt{T}$, then by inversion, we have judgments (1–6) below:

1. $\mathtt{CT} \vdash \overline{\mathtt{e} : \mathtt{U}}$
2. $\mathtt{CT} \vdash \mathtt{e}' : \mathtt{U}$
3. $\mathit{mtype}(\mathtt{m}; \mathtt{U}) = \mathtt{<\overline{Z \lhd V}>}\ (\overline{\mathtt{A}}) \rightarrow \mathtt{B}$
4. $\overline{\mathtt{U} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A}}$. 5. $\overline{\mathtt{S} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V}}$. 6. $[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B} <: \mathtt{T}$.
7. $R(\mathtt{CT}) \vdash \mathtt{e}' : R(\mathtt{U})$, by applying the inductive hypothesis to (2).
8. $R(\mathtt{U}) <: \mathtt{U}$, by lemma 5.

Applying lemma 4 to (3) and (8) gives the following four judgments:

9. $\mathit{mtype}(\mathtt{m}; R(\mathtt{U})) = \mathtt{<\overline{Z \lhd V'}>}\ (\overline{\mathtt{A}'}) \rightarrow \mathtt{B}'$
10. $\overline{\mathtt{A} <: \mathtt{A}'}$. 11. $\overline{\mathtt{V} <: \mathtt{V}'}$. 12. $\mathtt{B}' <: \mathtt{B}$.

We use the following standard substitution preserves subtyping lemma that was proven for many extensions of FGJ.

Lemma 7. If $\mathtt{T} <: \mathtt{T}'$, then $[\overline{\mathtt{U}/\mathtt{X}}]\mathtt{T} <: [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{T}'$.

Applying lemma 7 to (10–12) gives the next three judgments:

13. $\overline{[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A}'}$
14. $\overline{[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V}'}$
15. $[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B}' <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B}$
16. $R(\mathtt{CT}) \vdash \overline{\mathtt{e} : R(\mathtt{U})}$, by applying the inductive hypothesis to (1).

17. $\overline{R(\mathtt{U}) <: \mathtt{U}}$, by lemma 5.

18. $\overline{R(\mathtt{U}) <: \mathtt{U} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A}'}$, by (17), (4), and (13).

19. $\overline{\mathtt{S} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V} <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V}'}$, by (5) and (14).

20. $[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B}' <: [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B} <: \mathtt{T}$, by (15) and (6).

21. $\overline{\mathtt{R}(\mathtt{U}) <:_{\mathrm{JLS}} R([\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A}')}$, by applying lemma 1 to (18).

22. $\overline{\mathtt{S} <:_{\mathrm{JLS}} R([\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V}')}$, by applying lemma 1 to (19).

23. $R([\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B}') <:_{\mathrm{JLS}} R(\mathtt{T})$, by applying lemma 6 to (20).

24. $R(\mathtt{CT}) \vdash \mathtt{e}'.\mathtt{<}\overline{\mathtt{S}}\mathtt{>m}(\overline{\mathtt{e}}) : R(\mathtt{T})$, by (7), (9), (16), (21–23), and the subsumption typing rule. □

Proof of Theorem 1 with Method Body Analysis. We next sketch the proof of a theorem analogous to theorem 2 (which specializes theorem 1 specifically for the signature-only analysis), for the case of the method body analysis. Although the proof of theorem 2 assumed that the signature-only based analysis was performed, similar reasoning proves the theorem still holds if the method body analysis is performed. First, for this proof we reason with a function $R_{\mathtt{y}}(\mathtt{T})$ that refactors the type T with the method body analysis assuming that it is the declared type of method argument y:

$$R_{\mathtt{y}}(\mathtt{T}) = \begin{cases} \mathit{dvar}(\mathtt{X}_i;\mathtt{C}) \sqcup \mathtt{v}_i \sqcup \mathit{uvar}(\mathtt{X}_i;\mathtt{C};\mathtt{y}), \\ \quad \text{if } \mathtt{T} = \mathtt{C}\mathtt{<}\overline{\mathtt{vT}}\mathtt{>}, \mathtt{y} \text{ is a method argument,} \\ \quad \text{and } \mathtt{X}_i = i^{\text{th}} \text{ type parameter of } \mathtt{C}. \\ R(\mathtt{T}), \text{ otherwise.} \end{cases}$$

We define another function R^m to model applying the refactoring tool to the entire class table. $R^m(\mathtt{CT})$ is the same as $R(\mathtt{CT})$ except that when R^m is applied to the type T of method argument y, R^m returns $R_{\mathtt{y}}(\mathtt{T})$ instead of $R(\mathtt{T})$. Theorem 1 is restated with the equivalent implication below:

Theorem 3 (Refactored Types Are Safe for Body Analysis)**.** If $\mathtt{CT} \vdash \mathtt{e} : \mathtt{T}$, then:

$$\begin{cases} R^m(\mathtt{CT}) \vdash \mathtt{e} : R_{\mathtt{y}}(\mathtt{T}), \text{ if } \mathtt{e} = \mathtt{y} \text{ and } \mathtt{y} \text{ is a method argument} \\ R^m(\mathtt{CT}) \vdash \mathtt{e} : R(\mathtt{T}), \text{ otherwise.} \end{cases}$$

Note that this theorem implies that if a method argument y is used as a qualifier in an expression (e.g., "y.f"), that expression has the same type as in the refactored program where the signature-only based analysis was performed. Hence, proving this theorem amounts to showing that each kind of use of a method argument in the original program is still supported in the refactored program. Specifically, we show the following:

- If y is used in a member access expression (i.e., a field read or a method invocation), then the type of that expression in the refactored program is the same for both the signature-only based and the method body analysis.
- If y is declared with type T and is being "directly assigned" to a declaration of type U, then $R^m(\mathtt{T}) <:_{\mathrm{JLS}} R^m(\mathtt{U})$. Hence, the direct assignment still type checks in the refactored program. "Directly assigning" y to another declaration refers to the situation when y is not a qualifier in an expression but that expression has a destination declaration (node) as discussed in Section 4.3. For example, this occurs when y was directly returned from a method (i.e., "**return** y;" occurred in the method body).

In general for a type T of a method argument y, it is *not* the case that $R_{\mathtt{y}}(\mathtt{T}) <: \mathtt{T}$. However, for the *limited use* of y in the method body, it is safe to assume that $R_{\mathtt{y}}(\mathtt{T}) <: \mathtt{T}$. We define a new subtype relation where $\mathtt{T}' <:_{\mathtt{y}} \mathtt{T}$ denotes that for the limited use of type T by method argument y, T′ is a subtype of T. This assumption is safe because lemmas 3 and 4 hold for the subset of members accessed by y in the method body. In the statements of those two lemmas, we can replace the subtype T′ with $R_{\mathtt{y}}(\mathtt{T})$ and those lemmas would still hold for the particular members accessed by y. Considering method argument `source` from line 7 of Figure 1, for example, even though `List` is invariant, we have $R_{\mathtt{source}}(\mathtt{List<E>}) = \mathtt{List<?\ extends\ E>} <:_{\mathtt{source}} \mathtt{List<E>}$. The only member accessed from `source` in the method body is `iterator()`. Lemma 4 holds with the instantiations T′ = `List<? extends E>`, T = `List<E>`, and m = `iterator`.

Contrasting with lemma 6, in general we cannot establish the implication $\mathtt{T} <: \mathtt{U} \implies R^m(\mathtt{T}) <:_{\mathrm{JLS}} R^m(\mathtt{U})$ if T is the type of a method argument y. However, if $\mathtt{T} <:_{\mathrm{JLS}} \mathtt{U}$ holds in the original program because y was directly assigned to another variable of type U, then rules MB-AssignToGeneric-Same and MB-AssignToGeneric-Base from Figure 11 guarantee $R^m(\mathtt{T}) <:_{\mathrm{JLS}} R^m(\mathtt{U})$. For example, if $R^m(\mathtt{T}) = \mathtt{C}\mathtt{<}\overline{\mathtt{vT}}\mathtt{>}$ and $R^m(\mathtt{U}) = \mathtt{C}\mathtt{<}\overline{\mathtt{wT}}\mathtt{>}$, rule MB-AssignToGeneric-Same ensures $\overline{\mathtt{v} \leq \mathtt{w}}$. Moreover, if an arbitrarily complex expression e of type T occurs where an expression of type U is expected, then the implication $\mathtt{T} <:_{\mathrm{JLS}} \mathtt{U} \implies R^m(\mathtt{T}) <:_{\mathrm{JLS}} R^m(\mathtt{U})$ holds.

Given the properties above, each kind of use of a method argument is still supported in the refactored program. Furthermore, the argument above describes how to augment the proof of theorem 2 to prove theorem 3. For example, in the proof of theorem 2, for the case when $\mathtt{e} = \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$, for each $i \in |\overline{e}|$, we have $\mathtt{T}_i <:_{\mathrm{JLS}} \mathtt{U}_i$, where $\mathtt{T}_i$ is the type of the actual argument $\mathtt{e}_i$ and $\mathtt{U}_i$ is the type of the ith formal argument of the constructor for N. Since $\mathtt{e}_i$ was directly assigned to the ith formal argument, $R^m(\mathtt{T}_i) <:_{\mathrm{JLS}} R^m(\mathtt{U}_i)$. Hence, the proof of this theorem for the case when $\mathtt{e} = \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$ still holds. Augmenting the remainder of the proof is similarly straightforward.