SUBTYPING WITH GENERICS: A UNIFIED APPROACH

A Dissertation Presented

by

JOHN G. ALTIDOR

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2014

School of Computer Science

SUBTYPING WITH GENERICS: A UNIFIED APPROACH

A Dissertation Presented

by

JOHN G. ALTIDOR

Approved as to style and content by:

_____
Yannis Smaragdakis, Chair

_____
Jack C. Wileden, Member

_____
Neil Immerman, Member

_____
Ian Grosse, Member

_____
Lori Clarke, Department Chair
School of Computer Science

ACKNOWLEDGMENTS

ABSTRACT

SUBTYPING WITH GENERICS: A UNIFIED APPROACH

SEPTEMBER 2014

JOHN G. ALTIDOR

B.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yannis Smaragdakis

Reusable software increases programmers' productivity and reduces repetitive code and software bugs. Variance is a key programming language mechanism for writing reusable software. Variance is concerned with the interplay of parametric polymorphism (i.e., templates, generics) and subtype (inclusion) polymorphism. Parametric polymorphism enables programmers to write abstract types and is known to enhance the readability, maintainability, and reliability of programs. Subtyping promotes software reuse by allowing code to be applied to a larger set of terms. Integrating parametric and subtype polymorphism while maintaining type safety is a difficult problem. Existing variance mechanisms enable greater subtyping between parametric types, but they suffer from severe deficiencies: They are unable to express several common type abstractions. They can cause a proliferation of types and redundant code. They are difficult for programmers to use due to their inherent complexity.

This dissertation aims to improve variance mechanisms in programming languages supporting parametric polymorphism. To address the shortcomings of current mechanisms, we will combine two popular approaches, definition-site variance and use-site variance, in a single programming language. We have developed formal languages or *calculi* for reasoning about variance. The calculi are example languages supporting both notions of definition-site and use-site variance. They enable stating precise properties that can be proved rigorously. The *VarLang* calculus demonstrates fundamental issues in variance from a language-neutral perspective. The *VarJ* calculus illustrates realistic complications by modeling a mainstream programming language, Java. VarJ not only supports both notions of use-site and definition-site variance but also language features with complex interactions with variance such as F-bounded polymorphism and wildcard capture.

A mapping from Java to VarLang is implemented in software that *infers* definition-site variance for Java. Large, standard Java libraries (e.g., Oracle's JDK 1.6) were analyzed using the software to compute metrics measuring the benefits of adding definition-site variance to Java, which only supports use-site variance. Applying this technique to six Java generic libraries shows that 21-47% (depending on the library) of generic definitions are inferred to have single-variance; 7-29% of method signatures can be relaxed through this inference, and up to 100% of existing wildcard annotations are unnecessary and can be elided.

Although the VarJ calculus proposes how to extend Java with definition-site variance, no mainstream language currently supports both definition-site and use-site variance. To assist programmers with utilizing both notions with existing technology, We developed a refactoring tool that refactors Java code by inferring definition-site variance and adding wildcard annotations. This tool is practical and immediately applicable: It assumes no changes to the Java type system, while taking into account all its intricacies. This system allows users to select declarations (variables,

method parameters, return types, etc.) to generalize. It performs a flow analysis to determine the set of declarations that require updating given a user's selection. We evaluated our technique on six Java generic libraries. We found 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. The result of applying this refactoring is a more general interface that supports greater software reuse.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Writing reusable software is vital to programmer productivity and software safety. Productivity increases when programmers are able to apply a reusable and well-tested solution. Repeating most of an existing implementation may introduce new bugs in the reimplementation. Duplicate code is also difficult to maintain because updates must be repeated. A software fix, for example, must be applied to every repetition of the buggy code. The ability to reuse code, thus, is key to software development.

Abstraction is the fundamental mechanism for writing reusable code. Code can be generalized by abstracting components of the software that can vary. This supports the idea of software modularity, where software components can be swapped with others without modifying existing code. Extending software with new features without changing existing code is another important goal of reusable design.

1.1   Subtype Polymorphism

Programming languages provide tools for introducing abstractions and rewriting reusable software. *Polymorphism* is a broad category of abstraction mechanisms. It refers to the ability to apply one piece of code to multiple types. *Subtype polymorphism*, also known as *inclusion polymorphism*, is a kind of polymorphism where multiple classes can *extend* or *implement* another class or interface. In Java, for example, a class `Dog` may extend another class `Animal` as in the following code:

```
class Animal { void speak() { ... } }
class Dog extends Animal { void fetchFrisbee() { ... } }
class Client {
```

```
  void foo(Animal a) { a.speak(); }
  void bar(Dog d) { d.fetchFrisbee(); }
}
```

The extending class `Dog` is called a *subclass* of `Animal`. The extended class `Animal` is called a *superclass* or a *parent* of the extending class `Dog`. Java supports *inheritance*: subclasses *inherit* all members (e.g., methods and fields) from their parent class. In this example, because `Dog` extends `Animal`, `Dog` inherits the method `speak` from its parent `Animal`. As a result, `speak` can be invoked on any instance of `Dog` as well as any instance of `Animal`. Furthermore, a subclass and its superclass are said to be in an *is-a* relationship. Because class `Dog` extends class `Animal`, a `Dog` is an `Animal`. That is, every instance of `Dog` is also considered to be an instance of `Animal` because `Dog`s inherit the ability to perform all of the operations of `Animal`s. Therefore, method `foo` in class `Client` can take in instances of `Dog` because they are also instances of `Animal`.

Although a `Dog` is an `Animal`, the inverse is not true: An `Animal` is *not* a `Dog` because a `Dog` can do something that an `Animal` cannot. Not every instance of `Animal` supports a `fetchFrisbee` method but every instance of `Dog` does. As a result, the `bar` method in class `Client` accepts every instance of `Dog`, but it does not accept every instance of `Animal`.

Java interfaces [28, Chapter 9] provide another example of subtype polymorphism. They enable developers to program to an interface rather than an implementation. A class may provide an implementation of an interface. Any implementation of an interface can be used where an instance of the interface is expected without modifying clients of the interface.

Classes, interfaces, and similar kinds of software modules are modeled in programming language research as *types*. *Subtyping* establishes when one type can be used where another is expected. Subtyping is defined using a binary relation $<:$ between types. $T <: U$ is read as "T is a subtype of U"; this signals that an instance of T may

be provided where a `U` is expected. Furthermore, every instance of `T` is also said to be an instance of `U`.

`U` is also called a supertype of `T`, when `T` <: `U`. This vocabulary comes from viewing a type as the set of all of its instances. Under this interpretation, `U` is a superset of `T`.

The subtyping relation should be defined to satisfy the *subsumption principle*: Whenever `T` <: `U` is established, any operation that an instance of supertype `U` can perform should also be able to be performed with an instance of subtype `T`. For example, a standard Java compiler concludes `C` <: `I`, if class `C` implements interface `I`. This subtype relationship is safe because class `C` implements all of the methods/operations declared in interface `I`. Since class `C` may implement additional methods not in interface `I`, it is not safe to assume the reverse, `I` <: `C`. In this case, an instance of `I` may not contain a method that is supported in class `C`; moreover, assuming `I` <: `C` would violate the subsumption principle.

1.2 Parametric Polymorphism

Parametric polymorphism is one of the most significant programming language advances of the past 40 years. It is another mechanism for writing reusable software. This language feature occurs in many programming languages and appears in Java as *generics*. A generic class can declare type variables that abstract type expressions that occur in the implementation of the class. First, consider the non-generic class `ListOfAnimal` below:

```
class ListOfAnimal {
  Animal get(int index) { ... }
  void add(Animal elem) { ... }
  int size() { ... }
}
```

This class represents a list of any type of `Animal`s. It supports reading `Animal`s from the list using the `get` method and adding `Animal`s to the list using `add`. The `size` method returns the number of `Animal`s in the list.

Because instances of this class can contain any type of `Animal`, type errors can result when a more specific type of list is desired. If a list should only contain `Dogs`, for example, but a client mistakenly added an instance of class `Cat` to the list, a runtime error may result.[1] One way of avoiding this kind of error is making another class that only allows creating lists of `Dogs` such as the class below.

```
class ListOfDogs {
  Dog get(int index) { ... }
  void add(Dog elem) { ... }
  int size() { ... }
}
```

If one also wanted to create a list that only contained `Cats`, a similar class to `ListOfDogs` would be needed, where occurrences of the type `Dog` are replaced with `Cat`. Creating a new class each time a list of a new type is desired is a clear case of duplicated code that is error prone and difficult to maintain. Generics were invented to support this type variability without requiring code to be duplicated. The Java generic class below removes the need for the classes `ListOfAnimal` and `ListOfDog`.

```
class List<X> {
  X get(int index) { ... }
  void add(X elem) { ... }
  int size() { ... }
}
```

In this generic `List` class, a type parameter/variable `X` has been declared. It abstracts the type of elements in instances of `List`. Lists of specific types of elements can be emulated by instantiating the parameter with the desired type. For example, the type `List<Animal>` simulates class `ListOfAnimal`. Type `List<Animal>` can be thought of as a new version of class `List` with occurrences of type variable `X` replaced by the type argument, `Animal`.

---

[1]A runtime error would certainly have to result if a method only in class `Dog` (e.g., `fetchFrisbee`) is invoked on the `Cat` added to the list.

Similarly, type `List<Dog>` emulates the class `ListOfDogs`. Instances of `List<Dog>` can only contain `Dogs`. A `Cat` cannot be added to a `List<Dog>`. This prevents the runtime error that can occur when we use `ListOfAnimal` to create a list that should only contain `Dogs`.

1.3 Variance Introduction

Generics and subtyping are two key programming language mechanisms for writing reusable software. Although they work well in isolation, utilizing both features simultaneously suffers from severe practical limitations. In this dissertation, we are concerned with *variance*, the ability to write *one* piece of code that applies to *multiple instantiations* of a generic. For example, a Java class `Dog` may extend a class `Animal`. In this case, `Dog` is considered to be a subtype of `Animal`. However, Java does not conclude `List<Dog>` is a subtype of `List<Animal>`. Assuming this subtype relationship can result in a runtime error. In this case, the supertype `List<Animal>` can add a `Cat` to itself but the subtype `List<Dog>` cannot, which violates the subsumption principle.

This dissertation investigates mechanisms for improving variance in programming languages. Variance mechanisms are the keystone of safe genericity in modern programming languages, as they attempt to develop the exact rules governing the interplay of the two major forms of polymorphism: parametric polymorphism (i.e., generics or templates) and subtype (inclusion) polymorphism. Concretely, variance mechanisms aim to answer the question "under what conditions for type expressions *Exp1* and *Exp2* is `C<`*Exp1*`>` a subtype of `C<`*Exp2*`>`?"

The conventional answer to this question has been *definition-site variance*: the definition of generic class `C<X>` determines its variance [4, 18, 24]. Depending on how type parameter `X` is used in the class, `C` can have one of four flavors of variance: it can be *covariant*, meaning that `C<S>` is a subtype of `C<T>` if `S` is a subtype of `T`; it can be *contravariant*, meaning that `C<S>` is a subtype of `C<T>` if `T` is a subtype of `S`;

it can be *bivariant*, meaning that `C<S>` is always a subtype of `C<T>`, for any two types `S` and `T`; or it can be *invariant*, meaning that `C<S>` is a subtype of `C<T>` only if types `S` and `T` are "equivalent" or subtypes of each other.

Languages like C# [31] and Scala [51] support a type system with definition-site variance: at the point of defining the generic type `C<X>` we state its subtyping policy, and the type system attempts to prove that our assertion is statically safe. For instance, a C# definition `class C<out X> ...` means that `C` is *covariant*: `C<S>` is a subtype of `C<T>` if `S` is a subtype of `T`. The type system's obligation is to ensure that type parameter `X` of `C` is used in the body of `C` in a way that guarantees type safety under this subtyping policy. For example, `X` cannot appear as the argument type of a public method in `C`—a rule colloquially summarized as "the argument type of a method is a contravariant position".

By contrast, the type system of Java employs the concept of *use-site variance* [34]: a class does not itself state its variance when it is defined. Uses of the class, however, 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)` can accept arguments 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.

Use-site variance is a truly elegant idea. Producing automatically all different variance flavors from a single class definition is an approach of hard-to-dispute flexibility. The idea was quickly integrated in Java in the form of *wildcards* [28, Section 4.5.1] and it is widely used in standard Java libraries. Despite the conceptual elegance, however, the practical deployment of wildcards has been less than entirely successful.

Among opponents, "wildcards" has become a virtual synonym for a language design mess. (E.g., Josh Bloch's presentation at Javapolis 2008 emphasized "We simply cannot afford another wildcards" [8].) The reason is that use-site variance results in conceptual complexity, requires anticipation of generality at all usage points, and postpones the detection of overly restrictive type signatures until their use.

However, the traditional approach of definition-site variance, as used in Scala and C#, is also hardly free of usability problems. For a class that is not purely covariant or contravariant, the only way to achieve full genericity is by introducing specialized interfaces that correspond to the class's co-, contra-, and bivariant parts. Consequently, users have to remember the names of these interfaces, library designers must anticipate genericity, and a combinatorial explosion in the number of introduced interfaces is possible. (E.g., for a type `Triple<X,Y,Z>`, we may need an interface for each of the $3^3 = 27$ possible access combinations, such as "covariant with respect to `X`, contravariant with respect to `Y` and `Z`". The number is 3^3 and not 4^3 only because bivariance is not allowed as an explicit annotation.)

It is worth noting that, although definition-site variance is arguably simpler than use-site variance, it was purposely left out of the recent programming language Dart. Every generic in Dart is assumed to be covariant in its type parameters. The Dart programming language specification [36] states the following:

> The type system is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring.

Generally, each flavor of variance has its own advantages. Use-site variance is arguably a more advanced idea, yet it suffers from specific usability problems because it places the burden on the *user* of a generic type. (Although one should keep in mind that the users of one generic type are often the implementors of another.)

Definition-site variance may be less expressive, but leaves the burden of specifying general interfaces with the *implementor* of a generic. A natural idea, therefore, is to combine the two flavors in the same language design and allow full freedom: For instance, when a type is naturally covariant, its definition site can state this property and relieve the user from any further obligation. Conversely, when the definition site does not offer options for fully general treatment of a generic, a sophisticated user can still provide fully general signatures.

This dissertation addresses the shortcomings of current variance mechanisms by providing theoretical and practical foundations for combining definition-site and use-site variance in a single language. The thesis of this dissertation is stated below.

Thesis:

> Subtype and parametric polymorphism can be leveraged in tandem, by employing and generalizing the concepts of use-site and definition-site variance. Concretely, we will show that the definition- and use-site variance mechanisms can be combined in full generality and type-safety, to support a programming model with greater opportunity for reusability of generic code.

1.4 Illustration of Approach

To quickly indicate some of the contributions of this work, this section briefly illustrates one aspect of our approach: inferring definition-site variance. Later chapters present our approach to combining definition- and use-site variance in detail.

The variance of a class with respect to its type parameters is constrained by the variance of the positions these type parameters occur in. For instance, an argument type position is contravariant, while a return type position is covariant. However, in the presence of recursive type constraints and wildcards, no past technique reasons in a general way about the variance of a type expression in a certain position. For instance, past techniques would not infer anything other than *invariance* for classes `C` and `D`:

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

Our approach is based on assigning a variance to every type expression, and defining an operator, $\otimes$ (pronounced "transform"), used to compose variances. In our calculus, inferring the most general variance for the above type definitions reduces to finding the maximal solution for a constraint system over the standard variance lattice ($*$ is top, o is bottom, $+$ and $-$ are unordered, with a join of $*$ and a meet of o). If c stands for the (most general) variance of the definition of C<X> with respect to type parameter X, and d stands for the variance of D<Y> with respect to Y, the constraints (simplified) are:

$$c \sqsubseteq +$$

$$c \sqsubseteq - \otimes (- \sqcup c)$$

$$c \sqsubseteq - \otimes (+ \sqcup d)$$

$$d \sqsubseteq - \otimes c$$

Consider the first of these constraints. Its intuitive meaning is that the variance of class C (with respect to X) has to be at most covariance, $+$, (because X occurs as a return type of foo). Similarly, for the third constraint, the variance of C has to be at most the variance of type expression D<? extends X> transformed by the variance, $-$, of the (contravariant) position where the type expression occurs. The variance of type expression D<? extends X> itself is the variance of type D joined with the variance of the type annotation, $+$.

We will see the full rules and definition of $\otimes$, as well as prove their soundness, later, but for this example it suffices to know that $- \otimes + = -$, $- \otimes - = +$, $- \otimes * = *$,

and $-\otimes o = o$. It is easy to see with mere enumeration of the possibilities that the most general solution has $c = +$ and $d = -$. Thus, by formulating and solving these constraints, we correctly infer the most general variance: class C is *covariant* with respect to X, and class D is *contravariant* with respect to Y. We note that the interaction of wildcards and type recursion is non-trivial. For instance, removing the "? super" from the type of argument csx would make both C and D be invariant.

1.5   Dissertation Outline

This dissertation is structured as follows. Chapter 2 provides a more detailed background on definition-site and use-site variance. The presentation of our approach starts in Chapter 3. That chapter describes how definition- and use-site variance can be applied from a language-neutral perspective. It presents three fundamental problems that we address for reasoning about variance. Chapter 4 presents *VarLang*, a unifying framework for checking and inferring definition-site variance in a language that also supports use-site variance. Chapter 5 discusses realistic complications for adding definition-site variance to Java, a large, complex, main-stream programming language with intricate features that interact with variance. That chapter also provides background needed to understand Chapter 6. Chapter 6 presents *VarJ*, a model for Java with definition-site variance. Unlike VarLang, VarJ is equipped with an *operational semantics*. That is, there is a model of execution associated with programs in VarJ. The type soundness proof ensures runtime-type errors do not occur for well-typed programs. Since the type soundness proof includes a lot of detail not related to variance, Chapter 7 focuses on why the variance analysis is safe for program execution. This dissertation then switches gears to practical applications of the work and case studies. Chapter 8 describes how we inferred definition-site variance for Java. Chapter 9 describes a refactoring tool that we developed. It refactors Java code generalizing parametric types by adding wildcard annotations. The tool allows users to

select declarations (variables, method parameters, return types, etc.) to generalize and works with declarations that are not declared in available source code. The result of this refactoring is a more general interface that supports greater software reuse. Chapter 10 presents related work and compares the related work with our approach. Chapter 11 discusses the implications of this work, possible future work, and plans to adopt ideas from this work into mainstream languages in the real world.

CHAPTER 2

BACKGROUND ON VARIANCE

This chapter offers a brief background on definition- and use-site variance as well as their relative advantages.

2.1 Definition-site Variance

Languages supporting definition-site variance [31, 51] typically require each type parameter to be declared with a variance annotation. For instance, Scala [51] requires the annotation + for covariant type parameters, − for contravariant type parameters, and invariance as default. A well-established set of rules can then be used to verify that the use of the type parameter in the generic[1] is consistent with the annotation.

In intuitive terms, we can understand the restrictions on the use of type parameters as applying to *positions*. Each typing position in a generic's signature has an associated variance. For instance, method return and exception types, supertypes, and upper bounds of class type parameters are covariant positions; method argument types and class type parameter lower bounds are contravariant positions; field types are both co- and contravariant occurrences, inducing invariance. Type checking the declared variance annotation of a type parameter requires determining the variance 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.* Figure 2.1 presents the variance *lattice*. Consider the following templates of Scala classes, where v_X, v_Y, and

---

[1]We refer to all generic types (e.g., classes, traits, interfaces) uniformly as "generics".
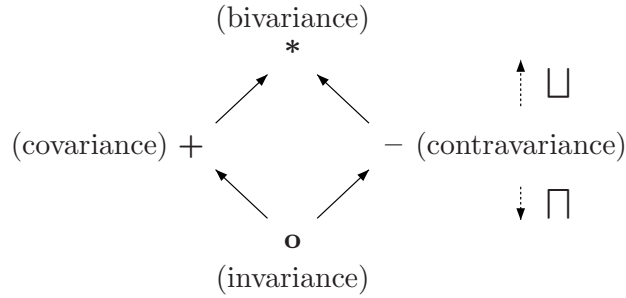
Figure 2.1. Standard variance lattice.

v_Z stand for variance annotations.[2]

```scala
abstract class RList[v_XX] { def get(i:Int):X }

abstract class WList[v_YY] { def set(i:Int, y:Y):Unit }

abstract class IList[v_ZZ] { def setAndGet(i:Int, z:Z):Z }
```

The variance v_X is the declared definition-site variance for type variable `X` of the Scala class `RList`. If $v_X = +$, the `RList` class typechecks because `X` does not occur in a contravariant position. If $v_Y = +$, the `WList` class does not type check because `Y` occurs in a contravariant position (second argument type in `set` method) but $v_Y = +$ implies `Y` should only occur in a covariant position. `IList` typechecks 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`. Thus, a read-only list of `Ts` (`RList[T]`) can always be safely thought of as a read-only list of some supertype of `Ts` (`RList[S]`, where `T<:S`). This is the exact definition of covariant subtyping and the reason why a return type is a covariant position. Thus, `RList` is covariant in `X`. Similarly, `WList` is a write-only list, and is intuitively *contravariant*.

---

[2]In Scala, a method's return type is written after the method's name and argument. Also, the Scala type `Unit` is the equivalent of Java type `void`.

Its definition supports this intuition: Objects of type `T` can be written to a write-only list of `Ts` (`WList[T]`) and written to a write-only list of `Ss` (`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`.

The variance of type variables is *transformed* by the variance of the context the variables appear in. Covariant positions *preserve* the variance of types that appear in them, whereas contravariant positions *reverse* the variance of the types that appear in them. The "reverse" of covariance is contravariance, and vice versa. The "reverse" of invariance is itself. Thus, we can consider the occurrence of a type parameter to be initially covariant. For instance, consider again the Scala classes above. In `RList`, `X` only appears as the return type of a method, which preserves the initial covariance of `X`, so `RList` is covariant in `X`. In `WList`, `Y` appears in a contravariant position, which reverses its initial covariance, to contravariance. Thus, `WList` is contravariant.

When a type parameter is used to instantiate a generic, its variance is further transformed by the declared definition-site variance of that generic. For example:

```
class SourceList[+Z] { def copyTo(to:WList[Z]):Unit }
```

Suppose the declared definition-site variance of `WList` (with respect to its single parameter) is contravariance. In `WList[Z]`, the initial covariance of `Z` is transformed by the definition-site variance of `WList` (contravariance). It is then transformed again by the contravariant method argument position. As a result, `Z` appears covariantly in this context, and `SourceList` is covariant in `Z`, as declared. Any variance transformed by invariance becomes invariance. Thus, if `Z` had been used to parameterize an invariant generic, its appearance would have been invariant. In Section 3.1 we generalize and formalize this notion of transforming variance.

We have so far neglected to discuss *bivariance*: `C<X>` is bivariant implies that `C<S><:C<T>` for any types `S` and `T`. Declaring a bivariant type parameter is not supported by the widely used definition-site variant languages. At first this seems to

not be entirely surprising. For a type parameter to be bivariant, it must only appear bivariantly in a generic. This means either it does not appear at all, or it appears only as the type argument to instantiate other bivariant generics. If a type parameter does not appear in a generic's signature at all, then it is useless to parameterize over it; if it is only used to instantiate other bivariant generics, it could just as well be replaced by any arbitrary type, since, by definition, a bivariant generic does not care what type it is instantiated with. Nevertheless, this argument ignores *type recursion.* As we discuss in Section 3.3 and in our experimental findings, several interesting interface definitions are inherently bivariant.

Finally, instead of declaring the definition-site variance of a type parameter and checking it for consistency, it is tempting to *infer* the most general such variance from the definition of a generic. This becomes hard in the presence of type recursion and supporting it in full generality is one of the contributions of our work.

2.2   Use-site Variance

An alternative approach to definition-site variance is *use-site variance* [12,34,63]. Instead of declaring the variance of `X` at its definition site, generics are assumed to be *invariant* in their type parameters. However, a type-instantiation of `C<X>` can be made co-, contra-, or bivariant using variance annotations.

For instance, using the Java wildcard syntax, `C<? extends T>` is a *covariant* instantiation of `C`, representing a type "C-of-some-subtype-of-T". `C<? extends T>` is a supertype of all type-instantiations `C<S>`, or `C<? extends S>`, where $S <: T$. In exchange for such liberal subtyping rules, type `C<? extends T>` can only access fully those methods and fields of `C` in which `X` appears covariantly. (Other methods can be used only with type-neutral values, e.g., called with `null` instead of values of type `X`.) In determining this, use-site variance applies the same set of rules used in definition-site

variance, with the additional condition that the upper bound of a wildcard is consid-
ered a covariant position, and the lower bound of a wildcard a contravariant position.

For example, consider an invariant generic class `List` that uses its type parameter
in both covariant and contravariant positions:

```
class List<X> {
  ...  // other members that don't affect variance
  void add(int i, X x) { ... }  // requires a List<? super X>
  X get(int i) { ... }          // requires a List<? extends X>
  int size() { ... }            // requires a List<?>
}
```

`List<? extends T>`, only has access to method "`X get(int i)`", but not method
"`void add(int i, X x)`". (More precisely, method `add` can only be called with `null`
for its second argument.)

Similarly, `List<? super T>` is the contravariant version of `List`, and is a supertype
of any `List<S>` and `List<? super S>`, where `T <:S`. Of course, `List<? super T>` has
access only to methods and fields in which `X` appears contravariantly or not at all.
(The `get` method returns `Object` for a `List<? super T>`.)

Use-site variance also allows the representation of the *bivariant* version of a
generic. In Java, this is accomplished through the unbounded wildcard: `List<?>`.
Using this notation, `List<S> <:List<?>`, for any `S`. The bivariant type, however, only
has full access to methods and fields in which the type parameter does not appear
at all. In definition-site variance, these methods and fields would have to be factored
out into a non-generic class.

2.3   A Comparison

Both approaches to variance have their merits and shortcomings. Definition-site
variance enjoys a certain degree of conceptual simplicity: the generic type instantia-
tion rules and subtyping relationships are clear. However, in the worst case the class
or interface designer must pay for such simplicity by splitting the definitions of data

16

types into co-, contra, and bivariant versions. This can be an unnatural exercise. For example, the data structures library for Scala contains immutable (covariant) and mutable (invariant) versions of almost every data type—and this is not even a complete factoring of the variants, since it does not include contravariant (write-only) versions of the data types.

The situation gets even more complex when a generic has more than one type parameter. In general, a generic with n type parameters needs 3^n (or 4^n if bivariance is allowed as an explicit annotation) interfaces to represent a complete variant factoring of its methods. Arguably, in practice, this is often not necessary.

Use-site variance, on the other hand, allows users of a generic to create co-, contra-, and bivariant versions of the generic on the fly. This flexibility allows class or interface designers to implement their data types in whatever way is natural. The users of these generics must pay the price, by needing to carefully consider the correct use-site variance annotations, so that the type can be as general as possible. This might not seem very difficult for a simple instantiation such as `List<? extends Number>`. However, type signatures can very quickly become complicated. For instance, the following method signature is part of the Apache Commons-Collections Library:

```
Iterator<? extends Map.Entry<? extends K,V>>
  createEntrySetIterator(Iterator<? extends Map.Entry<? extends K,V>>)
```

2.4   Generalizing the Design Space.

Our goal is to combine the positive aspects of use-site and definition-site variance, while mitigating their shortcomings. The key is to have a uniform and general treatment of definition-site and use-site variance in the same type system. This creates opportunities for interesting language designs. For instance:

- A language can combine explicit definition- and use-site variance annotations and perform type *checking* to ensure their soundness. For example, Scala or

C# can integrate wildcards in their syntax and type reasoning. This will give programmers the opportunity to choose not to split the definition of a type just to allow more general handling in clients. If, for instance, a `List` is supposed to support both reading and writing of data, then its interface can be defined to include both kinds of methods, and not need to be split into two types. The methods that use `List` can still be made fully general, as long as they specify use-site annotations. Generally, allowing both kinds of variance in a single language ensures modularity: parts of the code can be made fully general regardless of how other code is defined. This reduces the need for anticipation and lowers the burden of up-front library design.

Similarly, Java can integrate explicit definition-site variance annotations for purely variant types. This will reduce the need for use-site annotation and the risk of too-restricted types.

- A language can combine use-site variance annotations with *inference* of definition-site variance (for purely variant types). This is the approach that we implement and explore in later sections. Consider the long type signature of the `createEntrySetIterator` method mentioned above. It contains two wildcard-instantiations of `Iterator` and two more of `Map.Entry` totaling four wildcard-instantiations. Our approach can infer that `Iterator` is covariant, and `Map.Entry` is covariant in its first type parameter—without having to change the definition of either generic. Thus, the following signature in our system has exactly the same generality without any wildcards:

```
Iterator<Map.Entry<K,V>>
  createEntrySetIterator(Iterator<Map.Entry<K,V>>)
```

Furthermore, specifying the most general types proves to be challenging for even the most seasoned Java programmers: our experiments reveal that (at least) 7%

of the types in method signatures of the Java core library (`java.*`) are overly specific. We will discuss the details of our findings in Section 8.2.

CHAPTER 3

REASONING ABOUT VARIANCE

In order to meet our goal of a general, unified framework (for both checking and inference of both use-site and definition-site variance) we need to solve three fundamental problems. The first is that of composing variances, the second deals with the integration of use-site annotations in definition-site reasoning, and the third concerns the handling of recursive types. The concepts from this chapter establish formal foundations that are applied throughout the remainder of this dissertation. To the best of our knowledge, this work is the first to solve all three problems in their full generality.

3.1   Variance Composition

In earlier variance formalisms, reasoning about nested types, such as `A<B<X>>`, has been hard. Igarashi and Viroli pioneered the treatment of variant types as unions of sets of instances. Regarding nested types, they note (Section 3.3 of [34]): "We could explain more complicated cases that involve nested types but it would get harder to think of the set of instances denoted by such types." The first observation of our work is that it is quite easy to reason about nested types, not as sets of instances but based on variance composition. That is, given two generic types `A<X>` and `B<X>`, if the (definition-site) variances of `A` and `B` (with respect to their type parameters) are known, then we can compute the variance of type `A<B<X>>`.[1] This

---

[1]This relies on a natural extension of the definition of variance, to include the concept of a variance of an arbitrary type expression with respect to a type variable. E.g., type expression E is

composition property generalizes to arbitrarily complex-nested type expressions. The basis of the computation of composed variances is the *transform* operator, $\otimes$, defined in Figure 3.1. The relation $v_1 \otimes v_2 = v_3$ intuitively denotes the following: If the variance of a type variable X in type expression E is v_2 and the definition-site variance of the type parameter of a class C is v_1,[2] then the variance of X in type expression C<E> is v_3.

Definition of variance transformation: $\otimes$			
$+ \otimes + = +$	$- \otimes + = -$	$* \otimes + = *$	$o \otimes + = o$
$+ \otimes - = -$	$- \otimes - = +$	$* \otimes - = *$	$o \otimes - = o$
$+ \otimes * = *$	$- \otimes * = *$	$* \otimes * = *$	$o \otimes * = o$
$+ \otimes o = o$	$- \otimes o = o$	$* \otimes o = *$	$o \otimes o = o$

Figure 3.1. Variance transform operator.

The behavior of the transform operator is simple: invariance transforms everything into invariance, bivariance transforms everything into bivariance, covariance transforming a variance leaves it the same, and contravariance reverses it. (The reverse of bivariance is itself, the reverse of invariance is itself.) To sample why the definition of the transform operator makes sense, let us consider some of its cases. (The rest are covered exhaustively in our proof of soundness.)

- Case $+ \otimes - = -$: This means that type expression C<E> is contravariant with respect to type variable X when generic C is covariant in its type parameter and type expression E is contravariant in X. This is true because, for any T_1, T_2:

---

covariant in X iff $T_1 <: T_2 \implies E[T_1/\text{X}] <: E[T_2/\text{X}]$. (These brackets denote substitution of a type for a type variable and should not be confused with the Scala bracket notation for generics, which we shall avoid except in pure-Scala examples.)

[2]For simplicity, we often refer to generics with a single type parameter. For multiple type parameters the same reasoning applies to the parameter in the appropriate position.

$$T_1 <: T_2 \implies \qquad \text{(by contravariance of } E)$$

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

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

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

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

- Case $* \otimes v = *$: This means that type expression `C<E>` is bivariant with respect to type variable `X` when generic `C` is bivariant in its type parameter, regardless of the variance of type expression E (even invariance). This is true because:

$$\text{for any types } S \text{ and } T \implies \qquad \text{(by bivariance of } \texttt{C})$$

$$\texttt{C<}E[S/\texttt{X}]\texttt{>} <: \texttt{C<}E[T/\texttt{X}]\texttt{>} \implies$$

$$\texttt{C<}E\texttt{>}[S/\texttt{X}] <: \texttt{C<}E\texttt{>}[T/\texttt{X}]$$

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

As can be seen by inspection of all cases in Figure 3.1, operator $\otimes$ is associative. The operator would also be commutative, except for the case $* \otimes o = * \neq o = o \otimes *$. This is a design choice, however. With the types-as-sets approach that we follow in our formalization, operator $\otimes$ would be safe to define as a commutative operator, by changing the case $o \otimes *$ to return $*$. To see this, consider the meaning of $o \otimes *$. When generic `C` is invariant with respect to its type parameter `X` and type expression E is bivariant in `X`, should type expression `C<E>` be bivariant or invariant with respect to `X`? The answer depends on what we mean by "invariance". We defined invariance earlier as "`C<S><:C<T>` only if $S = T$". Is the type equality "$S = T$" syntactic or semantic? (I.e., does type equality signify type identity or equivalence, as can be

established in the type system?) If type equality is taken to be syntactic, then the only sound choice is $o \otimes * = o$:

$$C<E>[S/X] <: C<E>[T/X] \implies$$

$$C<E[S/X]> <: C<E[T/X]> \implies \qquad \text{(by invariance of C)}$$

$$E[S/X] = E[T/X] \implies \qquad \text{(assuming X occurs in } E)$$

$$S = T$$

Hence, $C<E>$ is invariant with respect to X. If, however, the definition of invariance allows for type equivalence instead of syntactic equality, then it is safe to have $o \otimes * = *$: By the bivariance of E, $E[S/X] <: E[T/X]$ and $E[T/X] <: E[S/X]$. Hence, $E[S/X]$ is equivalent to $E[T/X]$ and consequently $C<E>[S / X]$ can be shown equivalent to $C<E>[T / X]$ (assuming a natural extensionality axiom in the type system).

We chose the conservative definition, $o \otimes * = o$, in Figure 3.1 to match that used in our implementation of a definition-site variance inference algorithm for Java, discussed later. Since, in our application, bivariance is often inferred (not stated by the programmer) and since Java does not naturally have a notion of semantic type equivalence, we opted to avoid the possible complications both for the user and for interfacing with other parts of the language.

Similar reasoning to the transform operator is performed in Scala to check definition-site variance annotations. [51, Section 4.5] defines the variance position of a type parameter in a type or template and states "Let the opposite of covariance be contravariance, and the opposite of invariance be itself." It also states a number of rules defining the variance of the various type positions such as "The variance position of a method parameter is the opposite of the variance position of the enclosing parameter clause." The $\otimes$ operator is a generalization of the reasoning stated in that section; it adds the notion of bivariance and how the variance of a context transforms the

variance of a type actual in general instead of defining the variance of a position for language-specific constructs.

3.2 Integration of Use-Site Variance

The second new element of our work is the integration of use-site annotations in the reasoning about definition-site variance. Earlier work such as [34] showed how to reason about use-site variance for Java. Emir et al [24] formalized definition-site variance as it occurs in C#.[3] However, no earlier work explained how to formally reason about variance in a context including both definition-site and use-site variance. For example, suppose Scala is extended with support for use-site variance, v is a variance annotation ($+$, $-$, or o), and the following are syntactically legal Scala classes.

```
abstract class C[vX] {
 def set(arg1:X):Unit
}
abstract class D[+X] {
 def compare(arg2:C[+X]):Unit
}
```

Section 2.1 gave an overview of how declared definition-site variance annotations are type checked in Scala. Since class C only contains the set method, it typechecks with v = - because X only appears contravariantly in the type signature of the set method. However, type checking class D with the compare method requires reasoning about the variance of X in the argument type expression C[+X].

In our unified framework, a use-site annotation corresponds to a join operation in the standard variance lattice (Figure 2.1). That is, if generic C<X> has definition-site variance v_1 with respect to X, then the type expression C[v_2X] has variance $v_1 \sqcup v_2$ with respect to X.

---

[3]Their calculus is an extension of C# minor [40].

Intuitively, this rule makes sense: When applying use-site variance annotations, it is as if we are removing from the definition of the generic the elements that are incompatible with the use-site variance. For instance, when taking the covariant version, `C[+X]`, of our Scala class `C`, above, we can only access the members that use type parameter `X` covariantly—e.g., method `set` would be inaccessible. Hence, if class `C` is naturally contravariant in `X` (meaning that `X` only occurs contravariantly in the body of `C`), then `C[+X]` is a type that cannot access any member of `C` that uses `X`. Thus, `C[+X]` is bivariant in `X`: the value of the type parameter cannot be used. This is precisely what our lattice join approach yields: $+ \sqcup - = *$. As a result, any declared definition-site variance for class `D` would be legal.

To see more rigorously why the lattice-join approach is correct, let us consider the above case formally. (Other cases are covered exhaustively in our proof of soundness.) Given a contravariant generic `C`, why is it safe to infer that `C<+X>` (`C[+X]` in Scala syntax) is bivariant in `X`? We start from the Igarashi and Viroli approach to variance: All types are in a lattice with subtyping as its partial order and the meaning of `C<+T>` is $\bigsqcup_{T' <: T} $ `C<T'>`. This definition yields the standard variance theorems $T <: T' \Rightarrow$ `C<+T>` $<:$ `C<+T'>` and `C<T>` $<:$ `C<+T>`. Consider then the bottom element of the type lattice. (This lattice depends on the type system of the language and is not to be confused with the simple variance lattice of Figure 2.1.) We have:

$$\bot <: \text{T} \Rightarrow \qquad \text{(by first theorem above)}$$

$$\text{C<+}\bot\text{>} <: \text{C<+T>} \qquad\qquad (1)$$

But also, for any type `T'`:

$$\bot <: \text{T}' \Rightarrow \qquad (\text{C contravariant})$$

$$\text{C<T'>} <: \text{C<}\bot\text{>} \qquad\qquad (2)$$

25

Therefore:

$$C\texttt{<+T>} =\qquad\qquad\qquad\qquad\text{(by variance def)}$$

$$\bigsqcup_{T' <: T} C\texttt{<T'>} <:\qquad\qquad\qquad\text{(by (2), above)}$$

$$C\texttt{<}\bot\texttt{>} <:\qquad\qquad\text{(by second theorem above)}$$

$$C\texttt{<+}\bot\texttt{>}\qquad\qquad\qquad\qquad\qquad(3)$$

Hence, from (1) and (3), all `C<+T>` are always subtype-related, i.e., have type `C<*>`.

A Note on Scala:

To enable interoperability between Scala and Java, Scala represents Java wildcard types as existential types. For example, a Java `Iterator<?>` could be written as `Iterator[T] forSome { type T }` or more compactly as `Iterator[_]`. Similarly, the type Java `Iterator<? extends Comparator>` maps to the Scala type `Iterator[_ <: Comparator]`, and the type Java `Iterator<? super Comparator>` maps to the Scala type `Iterator[_ >: Comparator]`. However, Scala variance reasoning with existential types is too conservative because it just assumes that the use-site variance annotation overrides the definition-site variance instead of reasoning about how they both interact. For example, consider the Scala traits below.

```
trait GenType[+Y] { def get(i:Int):Y }
trait Wild[-X] {
  def add(elem:X):Unit
  // flagged as error but actually safe
  def compare(w:GenType[_ >: X]):Unit
}
```

The Scala compiler flags an error because it assumes the variance of `X` in `GenType[_ >: X]` is contravariance. This contravariance occurrence is then negated (transformed by contravariance) to covariance because it occurs in an argument (contravariant) position. Because the Scala compiler assumes `X` occurs in a covariant position in

26

`compare`'s argument type but the definition-site of `X` in trait `Wild` is contravariance, Scala flags this occurrence as an error. However, it is safe to assume that the variance of `X` in `GenType[_ >: X]` is bivariance. Because `GenType` is covariant in its type parameter, the contravariant version of `GenType` essentially provides no members of `GenType` that contain `GenType`'s type parameter in their type signature. Our joining of the definition-site and use-site variances takes advantage of this reasoning enabling more safe code to type check.

3.3 Recursive Variances

The third novel element of our approach consists of reasoning about recursive type definitions. This is particularly important for inferring (instead of just checking) definition-site variance. With type recursion, the unknown variance becomes recursively defined and it is not easy to compute the most general solution. Furthermore, type recursion makes the case of bivariance quite interesting. In contrast to non-recursive types, recursive types can be bivariant even when their type parameter is used. For instance the following type is safely bivariant:

```
interface I<X> {  I<X> foo (I<X> i); }
```

To appreciate the interesting complexities of reasoning about type recursion, we discuss some cases next.

3.3.1 Recursive Variance Type 1

The following interface demonstrates a most simple form of recursive variance:

```
interface C1<X> {  C1<X> foo1 (); }
```

The variance of `C1` depends on how `X` appears in its signature. The only appearance of `X` is in the return type of `foo1`, a covariant position, as the argument to `C1`. Thus, the variance of `X` in this appearance is its initial covariance, transformed by the

27

variance of `C1`—the very variance we are trying to infer! This type of recursive variance essentially says that the variance of `C1` is the variance of `C1`, and thus can be satisfied by any of the four variances: covariance, contravariance, invariance, or bivariance. Without any more appearances of `X`, the most liberal form of variance for `C1` is bivariance.

If `X` does appear in other typing positions, however, the variance of its declaring generic is completely determined by the variance of these appearances:

```
interface C2<X> extends C1<X> {  void  bar2 (X x);  }
interface C3<X> extends C1<X> {  X     bar3 ();     }
```

The definition-site variance of `C2` is constrained by the variance of `C1`, as well as `X`'s appearance as a method argument type—a contravariant appearance. Since `C1`'s variance is completely unconstrained, `C2` is simply contravariant. Similarly, `C3` is only constrained by `X`'s appearance as a method return type—a covariant appearance—and is thus covariant, as well.

The above pattern will be common in all our recursive variances. Without any constraints other than the recursive one, a generic is most generally bivariant. When other constraints are factored in, however, the real variance of `C1` can be understood informally as "can be either co- or contravariant".

3.3.2   Recursive Variance Type 2

The next example shows a similar, but much more restrictive form of recursive variance:

```
interface D1<X> {  void foo1 (D1<X> dx);  }
```

The variance of `D1` is again recursively dependent on itself, only this time `X` appears in `D1<X>` which is a method argument. If a recursive variance did not impose any restrictions in a covariant position, why would it be any different in a contravariant position? Interestingly, the contravariance means that the variance of `D1` is the

28

variance of `D1` transformed by the contravariance. This means the variance of `D1` must be the *reverse* of itself!

The only two variances that can satisfy such a condition are bi- and invariance. Again, without any other uses of `X`, `D1<X>` is most generally bivariant.

However, if `X` does appear either co- or contravariantly in combination with this type of recursive variance, the resulting variance can only be *invariance*:

```
interface D2<X> extends D1<X> {  void  bar2 (X x); }
interface D3<X> extends D1<X> {  X     bar3 ();    }
```

In the above example, `X` appears contravariantly in `D2`, as the argument of `bar2`. At the same time, the variance of `X` must be the opposite of itself, as constrained by the recursive variance in supertype `D1`. This is equivalent to `X` appearing covariantly, as well. Thus, the only reasonable variance for `D2` is invariance. A similar reasoning results in the invariance of `D3`.

Thus, recursive variance of this type can be understood informally as "cannot be either co- or contravariant" when other constraints are taken into account.

3.3.3 Recursive Variance Type 3

The following example shows yet a third kind of recursive variance:

```
interface E1<X> {  E1<E1<X>> foo1 ();  }
```

The variance of `E1` is the same as `X`'s variance in `E1<E1<X>>`. That is, the initial covariance of `X`, transformed by the variance of `E1`—twice. This type of recursive variance can, again, like the previous two, be satisfied by either in- or bivariance. However, the key insight is that, no matter whether `E1` is contra- or covariant, any variance transformed by `E1` twice (or any even number of times, for that matter) is always preserved. This is obvious if `E1` is covariant. If `E1` is contravariant, being transformed by `E1` twice means a variance is reversed, and then reversed again, which

still yields a preserved variance. Thus, unless `E1` is bi- or invariant, `X` in `E1<E1<X>>` is always a covariant appearance.

Thus, when other appearances of `X` interact with this form of recursive variance, its informal meaning becomes "cannot be contravariant". In other words, when this recursive variance is part of the constraints of a type, the type can be bivariant, covariant, or invariant. The following examples demonstrate this:

```
interface E2<X> extends E1<X> {  void bar2 (X x); }
interface E3<X> extends E1<X> {  X    bar3 ();    }
```

`X` appears contravariantly in `E2`, eliminating bivariance and covariance as an option for `E2`. However, `X` also appears in `E1<E1<X>>` through subtyping, which means it cannot be contravariant. Thus, `E2` is invariant.

In `E3`, `X` appears covariantly, and `X` in `E1<E1<X>>` can still be covariant. Thus, `E3` can safely be covariant.

3.3.4   Recursive Variance Type 4

Our last example of recursive variance is also twice constrained by itself. But this time, it is further transformed by a contravariance:

```
interface F1<X> {  int foo1(F1<F1<X>> x); }
```

The variance of `F1` is the same as `X`'s variance in `F1<F1<X>>`, then transformed by the contravariant position of the method argument type. That is, `X`'s initial covariance, transformed twice by the variance of `F1`, then reversed. Like all the other recursive variances, bi- and invariance are options. However, since the twice-transformation by any variance preserves the initial covariance of `X` in `F1<F1<X>>`, the transformation by the contravariance produces a contravariance. Thus, if `F1` cannot be bivariant, it must be contravariant (or invariant).

In other words, along with other constraints, `F1` has the informal meaning: "cannot be covariant". For instance:

```
interface F2<X> extends F1<X> {  void bar2 (X x); }
interface F3<X> extends F1<X> {  X    bar3 ();    }
```

In `F2`, `X` appears contravariantly as a method argument. Combined with the recursive variance through subtyping `F1<X>`, `F2` can be contravariant. In `F3`, however, `X` appears covariantly. With bivariance and contravariance no longer an option, the only variance satisfying both this covariant appearance and the recursive variance of `F1<F1<X>>` is invariance. Thus, `F3` is invariant in `X`.

3.3.5   Handling Recursive Variance.

The above list of recursive variances is not exhaustive, although it is representative of most obvious cases. It should be clear that handling recursive variances in their full generality is hard. The reason our approach can handle recursive variance well is that all reasoning is based on constraint solving over the standard variance lattice. Constraints are simple inequalities ("below" on the lattice) and can capture type recursion by having the same constant or variable (in the case of type inference) multiple times, both on the left and the right hand side of an inequality.

3.3.6   A Note on Scala:

Scala's reasoning about recursive variances is limited because it does not have the notion of bivariance; it does not allow the most general types to be specified. Consider the three following traits.

`trait C1[$v_X$X] { def foo:C1[X] }`

`trait C2[$v_Y$Y] extends C1[Y] { def bar(arg:Y):Unit }`

`trait C3[$v_Z$Z] extends C1[Z] { def baz:Z }`

Because trait `C1` has type 1 recursive variance, if Scala supported bivariant annotations, it would be safe to set the definition-site variances as follows: $v_X = *$, $v_Y = -$, and $v_Z = +$. Since Scala does not support bivariant annotations, no assignments allow both trait `C2` to be contravariant and trait `C3` to be covariant. For example,

setting $v_X = +$ implies attempting to compile trait `C2` will generate an error because Scala infers `Y` occurs covariantly in the base type expression occurring in "`C2[-Y]` `extends C1[Y]`"; since `Y` is declared to be contravariant, `Y` should not occur in a covariant position in the definition of `C2`. Below are the only three assignments allowed by the Scala compiler.

$$v_X = - \qquad\qquad v_Y = - \qquad\qquad v_Z = o$$

$$v_X = + \qquad\qquad v_Y = o \qquad\qquad v_Z = +$$

$$v_X = o \qquad\qquad v_Y = o \qquad\qquad v_Z = o$$

CHAPTER 4

VARLANG: A CORE LANGUAGE AND CALCULUS

We combine all the techniques of the previous section into a unified framework for reasoning about variance. We introduce a core language, VarLang [1], for describing the various components of a class that affect its variance. Reasoning is then performed at the level of this core language, by translating it to a set of constraints.

4.1 Syntax

A sentence S in VarLang is a sequence (treated as a set) of modules, the syntax of which is given in Figure 4.1.

$$\texttt{M} \in \textit{Module} ::= \texttt{module C<}\overline{\texttt{X}}\texttt{> \{ } \overline{\texttt{T}v} \texttt{ \}}$$
$$\texttt{T} \in \textit{Type} ::= \texttt{X} \mid \texttt{C<}\overline{v\texttt{T}}\texttt{>}$$
$$\texttt{v} \in \textit{Variance} ::= + \mid - \mid * \mid o$$
$$\texttt{C} \in \textit{ModuleNames} \text{ is a set of module names}$$
$$\texttt{X} \in \textit{VariableNames} \text{ is a set of variable names}$$

Figure 4.1. Syntax of VarLang

Note that variance annotations, v, $(+/-/*/o)$ can appear in two places: at the top level of a module, as a suffix, and at the type level, as a prefix. Informally, a v at the top level means that the corresponding type appears covariantly/contravariantly/invariantly (i.e., in a covariant/contravariant/invariant position). A v on a type means that the type parameter is qualified with the corresponding use-site variance

annotation, or no annotation (for invariance). For instance, consider the VarLang sentence:

```
module C<X> { X+, C<-X>-, void+, D<+X>- }
module D<Y> { void+, C<oY>- }
```

This corresponds to the example from Section 1.4. That is, the informal meaning of the VarLang sentence is that:

- In the definition of class `C<X>`, `X` appears covariantly; `C<? super X>` appears contravariantly; `void` appears covariantly; `D<? extends X>` appears contravariantly.

- In the definition of class `D<Y>`, `void` appears covariantly; `C<Y>` appears contravariantly.

4.2 VarLang Translation

Our reasoning approach consists of translating a VarLang sentence S into a set of constraints over the standard variance lattice (Figure 2.1). The constraints are "below"-inequalities and contain variables of the form $var(\mathtt{X};\mathtt{T})$ and $var(\mathtt{X};\mathtt{C})$, pronounced "variance of type variable `X` in type expression `T`" and "(definition-site) variance of type variable `X` in generic `C`". The constraints are then solved to compute variances, depending on the typing problem at hand (checking or inference). The following rules produce the constraints. (Note that some of the constraints are vacuous, since they establish an upper bound of $*$, but they are included so that the rules cover all syntactic elements of VarLang and the translation from a VarLang sentence to a set of constraints is obvious.)

$$var(\mathtt{X};\mathtt{C}) \sqsubseteq \mathtt{v}_i \otimes var(\mathtt{X};\mathtt{T}_i), \forall i,$$

$$\text{where } \mathtt{module\ C<}\overline{\mathtt{X}}\mathtt{>\ \{\ }\overline{\mathtt{Tv}}\mathtt{\ \}} \in S$$

(4.1)

$$var(\mathtt{X}; \mathtt{C<>}) \sqsubseteq * \tag{4.2}$$

$$var(\mathtt{X}; \mathtt{Y}) \sqsubseteq *, \text{ where } \mathtt{X} \neq \mathtt{Y} \tag{4.3}$$

$$var(\mathtt{X}; \mathtt{X}) \sqsubseteq + \tag{4.4}$$

$$var(\mathtt{X}; \mathtt{C<\overline{vT}>}) \sqsubseteq (\mathtt{v}_i \sqcup var(\mathtt{Y}; \mathtt{C})) \otimes var(\mathtt{X}; \mathtt{T}_i), \forall i,$$

$$\tag{4.5}$$

where $\mathtt{Y}$ is the i-th type variable in the definition of $\mathtt{C}$.

Rule 4.1 specifies that for each type $\mathtt{T}_i$ in module $\mathtt{C}$, the variance of the type variable $\mathtt{X}$ in $\mathtt{C}$ must be below the variance of $\mathtt{X}$ in $\mathtt{T}_i$ transformed by $\mathtt{v}_i$, the variance of the position that $\mathtt{T}_i$ appears in. This corresponds to the traditional reasoning about definition site variance from Section 2.1.

Rules 4.2 and 4.3 specify that the $\mathtt{X}$ can have any variance in a type expression for which it does not occur in. Rule 4.4 constrains the initial variance of a type variable to be at most covariance.

Rule 4.5 is the most interesting. It integrates our reasoning about how to compose variances for complex expressions (using the transform operator, as described in Section 3.1) and how to factor in use-site variance annotations (using a join in the variance lattice, as described in Section 3.2).

Note that the rules use our transform operator in two different ways: to combine the variance of a position with the variance of a type, and to compose variances.

We prove the soundness of the above rules denotationally—that is, by direct appeal to the original definition and axioms of use-site variance [34]. The proof can be found in Appendix A.

Example

We can now revisit in more detail the example from the Introduction, containing both recursive variance and wildcards:

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

As we saw, the corresponding VarLang sentence is:

```
module C<X> { X+, C<-X>-, void+, D<+X>- }
module D<Y> { void+, C<oY>- }
```

The generated constraints (without duplicates) are:

$$var(\texttt{X};\texttt{C}) \sqsubseteq + \otimes var(\texttt{X};\texttt{X}) \qquad \text{(rule 4.1)}$$

$$var(\texttt{X};\texttt{X}) \sqsubseteq + \qquad \text{(rule 4.4)}$$

$$var(\texttt{X};\texttt{C}) \sqsubseteq - \otimes var(\texttt{X};\texttt{C<-X>}) \qquad \text{(rule 4.1)}$$

$$var(\texttt{X};\texttt{C<-X>}) \sqsubseteq (- \sqcup var(\texttt{X};\texttt{C})) \otimes var(\texttt{X};\texttt{X}) \qquad \text{(rule 4.5)}$$

$$var(\texttt{X};\texttt{C}) \sqsubseteq + \otimes var(\texttt{X};\texttt{void}) \qquad \text{(rule 4.1)}$$

$$var(\texttt{X};\texttt{void}) \sqsubseteq * \qquad \text{(rule 4.2)}$$

$$var(\texttt{X};\texttt{C}) \sqsubseteq - \otimes var(\texttt{X};\texttt{D<+X>}) \qquad \text{(rule 4.1)}$$

$$var(\texttt{X};\texttt{D<+X>}) \sqsubseteq (+ \sqcup var(\texttt{Y};\texttt{D})) \otimes var(\texttt{X};\texttt{X}) \qquad \text{(rule 4.5)}$$

$$var(\texttt{Y};\texttt{D}) \sqsubseteq + \otimes var(\texttt{Y};\texttt{void}) \qquad \text{(rule 4.1)}$$

$$var(\texttt{Y};\texttt{void}) \sqsubseteq * \qquad \text{(rule 4.2)}$$

$$var(\texttt{Y};\texttt{D}) \sqsubseteq - \otimes var(\texttt{Y};\texttt{C<oY>}) \qquad \text{(rule 4.1)}$$

$$var(\texttt{Y};\texttt{C<oY>}) \sqsubseteq (o \sqcup var(\texttt{X};\texttt{C})) \otimes var(\texttt{Y};\texttt{Y}) \qquad \text{(rule 4.5)}$$

$$var(\texttt{Y};\texttt{Y}) \sqsubseteq + \qquad \text{(rule 4.3)}$$

4.3   Revisiting Recursive Type Variances

Armed with our understanding of variance requirements as symbolic constraints on a lattice, it is quite easy to revisit practical examples and understand them quickly.

36

For instance, what we called type 2 recursive variance in Section 3.3 is just an instance of a recursive constraint $c \sqsubseteq - \otimes c$, where c is some variable of the form $var(\texttt{X};\texttt{C})$. This is a case of a type that recursively (i.e., inside its own definition) occurs in a contravariant position. (Of course, the recursion will not always be that obvious: it may only become apparent after other constraints are simplified and merged.) It is easy to see from the properties of the transform operator that the only solutions of this constraint are o and $*$; i.e., "cannot be either co- or contravariant" as we described in Section 3.3. If $c = +$, then the constraint generated by type 2 recursive variance would be violated, since $c = + \not\sqsubseteq - \otimes c = - \otimes + = -$. Similar reasoning shows c cannot be $-$ and satisfy the constraint.

4.4 Constraint Solving

Checking if a variance satisfies a constraint system (i.e., the constraints generated for a VarLang module) corresponds to checking definition-site variance annotations in type definitions that can contain use-site variance annotations. Analogously, inferring the most general definition-site variances allowed by a type definition corresponds to computing the most general variances that satisfy the constraint system representing the type definition. The trivial and least general solution that satisfies a constraint system is assigning the definition-site variance of all type parameters to be invariance. Assigning invariance to all type parameters is guaranteed to be a solution, since invariance is the bottom element, which must be below every upper bound imposed by the constraints. Stopping with this solution would not take advantage of the subtyping relationships allowed by type definitions. Fortunately, the most general solution is always unique and can be computed efficiently by fixed-point computation running in polynomial time of the program size (number of constraints generated).

The only operators in constraint systems are the binary operators $\sqcup$ and $\otimes$. Both of these are monotone, as can be seen with the variance lattice and Figure 3.1.

Every constraint system has a unique maximal solution because there is guaranteed to be at least one solution (assign every type parameter invariance) and solutions to constraint systems are closed under point-wise $\sqcup$; we get a maximal solution by joining all of the greatest variances that satisfy each constraint. Because operators $\sqcup$ and $\otimes$ are monotone, we can compute the maximal solution efficiently with fixed point iteration halting when the greatest fixed point of the equations has been reached. We demonstrate this algorithm below by applying it to the example Java classes `C` and `D` from Section 4.2.

First, because we are only interested in inferring definition-site variance, we only care about computing the most general variances for terms of the form $var(\texttt{X}; \texttt{C})$ but not $var(\texttt{X}; \texttt{T})$. We can expand $var(\texttt{X}; \texttt{T})$ terms with their upper bounds containing only unknowns of the form $var(\texttt{X}; \texttt{C})$ Consider the constraint generated from `foo`'s argument type: $var(\texttt{X}; \texttt{C}) \sqsubseteq - \otimes var(\texttt{X}; \texttt{C<-X>})$. Because we are computing a maximal solution and because of the monotonicity of $\sqcup$ and $\otimes$, we can replace $var(\texttt{X}; \texttt{C<-X>})$ and $var(\texttt{X}; \texttt{X})$ by their upper bounds, rewriting the constraint as:

$$var(\texttt{X}; \texttt{C}) \sqsubseteq - \otimes \underbrace{(- \sqcup var(\texttt{X}; \texttt{C})) \otimes \overbrace{+}^{var(\texttt{X};\texttt{X})}}_{var(\texttt{X};\texttt{C<-X>})}$$

Lastly, we can ignore type expressions that do not mention a type parameter because they impose no real upper bound; their upper bound is the top element (e.g., $var(\texttt{X}; \texttt{void}) \sqsubseteq *$). This leads to the following constraints generated for the example two Java classes:

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

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

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

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

For each expanded constraint $r \sqsubseteq l$ in a constraint system, r is a $var(\texttt{X};\texttt{C})$ term and l is an expression where the only unknowns are $var(\texttt{X};\texttt{C})$ terms. The greatest fixed-point of a constraint system is solved for by, first, assigning every $var(\texttt{X};\texttt{C})$ term to be $*$ (top). Each constraint $r \sqsubseteq l$ is then transformed to $r \leftarrow l \sqcap r$, since r need not increase from the value it was lowered to by other assignments. The last step is to iterate through the assignments for each constraint until the $var(\texttt{X};\texttt{C})$ terms no longer change, which results in computing the greatest fixed-point. Finally, computing the greatest fixed-point runs in at most $2n$ iterations, where n is the number of constraint inequalities, since for each $r \leftarrow l \sqcap r$, r can decrease at most 2 times to invariance (bottom) from initially being bivariance (top).

CHAPTER 5

TOWARDS INDUSTRIAL STRENGTH LANGUAGES

The VarLang calculus of Chapter 4 proposes a unifying framework for checking and inferring both definition- and use-site variance in a language. That proposal is not accompanied by a language *operational semantics* however—its proof of soundness is expressed as a meta-theorem, i.e., under assumptions over what an imaginary language's type system should be able to prove about sets of values. This meta-theorem is welcome as an intuition about why it makes sense to combine variances in a certain way. VarLang is designed to ignore language specific constructs that are orthogonal to the denotational meaning of variance and would make our soundness proof be of the same complexity as in e.g., TameFJ [12]. Due to the novelty of both the problem we were addressing and our approach, we followed the philosophy stated in classic literature [32]:

> Often it is sensible to choose a model that is less complete but more compact, offering maximum insight for minimum investment.

However, VarLang did not establish a firm connection with any real programming language. For instance, in VarLang programs, the *variance of a position* that a type occurs in is *given*. This is not case in realistic languages like Java. This causes a straightforward, VarLang-based algorithm for inferring definition-site variance to be overly *conservative*. For instance, our early application of VarLang [1] did not try to infer the most general variance induced by polymorphic methods: if a class type parameter appears at all in the upper bound of a type parameter of a polymorphic method, we considered this to be an instance of an *invariant* position, the most

40

restrictive kind. As a result, a generic would be inferred to be invariant relative to its type parameter if this type parameter occurred in an upper bound of a (polymorphic) method's type parameter.

Futhermore, VarLang is not defined with an operational semantics. In other words, programs written in VarLang have no execution behavior associated with them. The proof of VarLang's soundness Theorem (3) does *not* show how to prove that particular rules for variant subtyping (subtyping between instantiations of a single type) do not cause runtime type errors. Since proving the absense of runtime type errors is a key goal of a type system, this vital question is not addressed by VarLang.

We next investigate realistic complications by developing the *VarJ* calculus, a type system based on Java, presented in Chapter 6. We also provide background on existential types and other concepts required to understand Chapter 6. VarJ achieves a safe synergy of use-site and definition-site variance, while supporting the full complexities of the Java realization of variance, including F-bounded polymorphism and wildcard capture. We show that the interaction of these features with definition-site variance is non-trivial and offer a full proof of soundness—the first in the literature for an approach combining variance mechanisms. The work on VarJ makes several contributions. At the high level:

- Compared to the type systems of Java, C#, or Scala, our combination of definition-site and use-site variance allows the programmer to pick the best tool for the job. Libraries can avoid offering different flavors of interfaces just to capture the notion of, e.g., "the covariant part of a list" vs. "the contravariant part of a list". Conversely, users can often use purely-variant types more easily and with less visual clutter if the implementor of that type had the foresight to declare its variance.

- Our approach maintains other features of the Java type system, namely full support for wildcards, which are a mechanism richer than plain use-site variance (e.g., [33]) and allow uses directly inspired by existential types.

- We provide a framework for determining the variance of the various positions in which a type can occur. (For example, why is the upper bound of the type parameter of a polymorphic method a contravariant position?)

Also, at the technical level:

- We show how definition-site variance interacts in interesting ways with advanced typing features, such as existential types, F-bounded polymorphism, and wild-card capture. A naive application of our earlier work [1] to Java would result in unsoundness, as we show with concrete examples. (Our earlier approach avoided unsoundness when applied to actual Java code by making several over-conservative assumptions to completely eliminate any interaction between, e.g., definition-site variance and F-bounded polymorphism.)

- We clarify and extend the TameFJ formalism with definition-site variance. TameFJ is a thorough, highly-detailed formalism and extending it is far from a trivial undertaking. The result is that we offer the first full formal modeling and proof of soundness for a language combining definition- and use-site variance.

Realistic Complications

The main contribution of VarJ consists of formalizing and proving sound the combination of definition- and use-site variance in the context of Java. In order to do so, we need to reason about the interaction of definition-site variance with many complex language features, such as F-bounded polymorphism, polymorphic methods, bounds on type parameters, and existential-types (arising in the use of wildcards). This interaction is highly non-trivial, as we see in examples next.

5.1 Generic Methods

In addition to classes having type parameters, Java methods may also be declared with type parameters. Such methods are known as polymorphic methods or generic methods [28, Section 8.4.4]. Variance complications involving generic methods will be presented in Section 5.2.3. An example generic method `toList` is defined in the following code segment:

```
class Utils {
  <X extends String> List<X> toList(X elem) {
    List<X> newList = new LinkedList<X>;
    newList.add(elem);
    int numOfCharacters = elem.length();
    System.out.println(numOfCharacters);
    return newList;
  }

  boolean foo() {
    String str = "a string";
    List<String> list1 = this.<String>toList(str);
    List<String> list2 = this.toList(str);
    return list1.equals(list2);
  }
}
```

The generic method `toList` is declared with a type parameter `X`. It takes in an element of type `X` and returns a new `List<X>` that only contains the input element. The abbreviated type signature of `toList` without mentioning the name of its value argument (`elem`) is `<X extends String> (X)` $\rightarrow$ `List<X>`. To connect this notation with that used commonly in programming language literature, type signatures of generic methods can be written as universal types [53, Section 23], where type parameters are universally quantified variables. The corresponding universal type of `toList` is $\forall$`X` $<:$ `String`.$($`X` $\rightarrow$ `List<X>`$)$. Java wildcards are modeled in VarJ as existential types rather than universal types. We explain in Section 5.2 why existential types are a more appropriate model for Java wildcards than universal types.

Method type parameters can be referenced anywhere in the type signature of the method (return type and value argument type) and in the body of method. They

can be declared with upper bounds. Because the upper bound of `X` is `String`, any instance of `X` can invoke methods in the `String` class. The `toList` method invokes the method `length` that is defined in the `String` class to retrieve the number of characters in the string bound to `elem`.

Invoking a generic method *always* requires specifying both type arguments and value arguments either by the programmer or by the compiler. First, we define terminology to clarify references to syntactic elements related to generic methods. Type arguments/parameters are variables bound to types. Value arguments are bound to values. Arguments in type signatures are called *formal* arguments. Arguments in method invocations are called *actual* arguments. Consider the type signature of `toList` and the generic method invocation `this.<String>toList(str)` in the body of the `foo` method. In that invocation, the actual type argument `String` is bound to formal type argument `X`. Similarly, the actual value argument `str` is bound to formal value `elem`. We skip the qualifiers such as "formal" and "actual" when refering to arguments when the desired qualifier is clear from the context.

In the method invocation `this.toList(str)`, an actual type argument is not specified by the programmer. For such invocations, the compiler automatically *infers* an actual type argument. This process is known as type inference. Actual type arguments are inferred first using the types of actual value arguments [28, Section 15.12.2.7]. If there are method type arguments that were not inferred from the type of the actual value arguments, the context of the method invocation is used to infer the remaining type arguments [28, Section 15.12.2.8].[1] For this invocation, the `javac` compiler infers the missing type argument to be `String` using the fact that the value argument `str` is of type `String`.

---

[1]For example, the method invocation may be the right-hand side of an assignment expression.

The interaction between generic methods, type inference, Java wildcards, and variance raises complex issues that must be addressed to support definition-site variance in Java. These issues are discussed later in this chapter.

Contrasting Use-Site Variance and Generic Methods

This section explains why generic method type parameters with upper type bounds *cannot* emulate use-site variance. Use-site variance allows truly unknown types at compile time. It enables greater abstraction on method type signatures than just introducing type variables with type bounds. For example, suppose `List` is invariant. It may seem that the method signature "`(int)` → `List<? extends Animal>`" can always be replaced by the generic method type signature "`<Y extends Animal> (int)` → `List<Y>`". A method of either of these types returns a `List` of some subtype of `Animal`. However, the following method shows that this is not the case.

```
// This method typechecks/compiles
List<? extends Animal> createList(int num) {
  if(num % 2 == 0) // if num is an even number
    return new List<Dog>();
  else // else num is an odd number
    return new List<Cat>();
}

// This method does not type check/does not compile
<Y extends Animal> List<Y> createList2(int num) {
  if(num % 2 == 0)
    // next line does not type check because
    // List<Dog> is not a subtype of List<Y>
    return new List<Dog>();
  else
    // next line does not type check because
    // List<Dog> is not a subtype of List<Y>
    return new List<Cat>();
}
```

The `createList` method returns a `List<Dog>` if the input value argument `num` is even; otherwise, it returns a `List<Cat>`. Since `List` is invariant, neither `List<Dog>` nor

List<Cat> is a subtype of List<Animal>. Both List<Dog> and List<Cat> are subtypes of List<? extends Animal>, which allows method createList to type check.

Although methods createList and createList2 have the same method body (except for comments), createList2 does not type check. In Java, a method body of a generic method is type checked once and for all legal instantiations of the method's type parameters. This ensures that a method body is type safe for every instantiation of the method's type parameters that is within the upper bounds of the type parameters. createList2 does not type check because a compiler cannot establish that for every instantiation of type parameter Y that is a subtype of Animal, List<Y> is a supertype of both List<Dog> and List<Cat>.

An invocation of createList2 would not type check even if only instantiated method bodies are type checked. In C++ [37], uninstantiated bodies of generic methods[2] are compiled without type checking. Only instantiated methods resulting from generic method invocations are type checked in C++. By the invariance of List, no instantiation of List is a supertype of both List<Dog> and List<Cat>. Hence, no instantiation of createList2 typechecks for the entire method body, and any invocation of createList2 will not type check.

Without use-site variance, type arguments must always be specified *statically* or at compile time in method invocations. Although type inference allows type actuals to be specified by the compiler instead of by the programmer, these types must be specified statically. This example shows that generic methods with upper bounds on type parameters do not give the expressiveness of wildcards. It also highlights a key difference between universally-quantified type variables and existentially-quantified type variables. Types abstracted by wildcards are existentially quantified. Details of existential types and their correspondence with Java wildcards will be given next, in

---

[2]Generic methods are known as template methods in C++.

Section 5.2. For now, we just mention existential type variables are never instantiated at compile time. On the contrary, applying terms of universal types such as generic methods requires instantiating universally-quantified type variables (either manually or automatically, by the compiler) at compile time.[3]

5.2 Existential Types

Java wildcards are not merely use-site variance, but also include mechanisms inspired by existential typing mechanisms. This section explains these mechanisms and motivates the usage of existential types in the VarJ calculus to model Java wildcards. We will show that existential types model aspects of Java wildcards that cannot be represented using VarLang types.

During the type checking phase of the compiler, Java wildcards are *captured* or converted to fresh type variables; this process is known as capture conversion [28, Section 5.1.10]. Each wildcard in the input program generates a distinct type variable. For example, the type `Pair<?, ?>` is capture converted to a type `Pair<Y, Z>`, where `Y` and `Z` are fresh, distinct type variables.

Wildcard types are modeled in the TameFJ calculus [12] as existential types. Type variables resulting from capture conversion are modeled as existentially quantified type variables. The type `Pair<?, ?>` is modeled in TameFJ as the existential type $\exists$`Y,Z.Pair<Y, Z>`. Further details of how Java wildcards are translated to existential types are in [12, Section 4]. The remainder of this section will motivate the usage of existential types to model Java wildcards using the code example in Figure 5.1.

---

[3]An encoding of existential types as universal types in an extension of the lambda calculus is presented in [30, Chapter 21]. In that encoding, $\exists(t.\sigma) = \forall(t'.\forall(t.\sigma \to t') \to t')$, where t is a type variable and σ is a type expression. In that encoding, the newly created type variable t' is instantiated at compile time in an *open* expression. However, the existentially-quantified type variable t is never instantiated at compile time.

```
class Client
{
 <X> Pair<X, X> make(List<X> l) { ... }
 <X> Boolean compare(Pair<X, X> p) { ... }

 <X> void exchangeFirsts(List<X> l1, List<X> l2) {
   X tmp = l1.get(0);     // get first element of l1
   l1.set(0, l2.get(0)); // set l1's first element to l2's
   l2.set(0, tmp);        // set l2's first element to l1's
 }

 <X> void uselessExchange(List<X> l) {
   exchangeFirsts(l, l);
 }

 void foo() {
   Pair<?, ?> pair;
   List<?> list;

   compare(pair);    // 1, error, does not type check/compile
   compare(make(list));   // 2, OK, type checks/compiles

   exchangeFirsts(list, list);    // 3, error, does not type check
   uselessExchange(list);   // 4, OK, type checks/compiles
 }
}
```

Figure 5.1. Example Java program that motivates the usage of existential types to model Java wildcards. This program is based on an example from [12, Section 2.1].

5.2.1 Expressible But Not Denotable Types

Expressions can be assigned with types that are expressible but cannot be written in Java's syntax. The method invocation `this.make(list)` from the above code example returns a `Pair` of two elements that are both of the same unknown type. This type can be modeled in TameFJ as $\exists$`X.Pair<X, X>`. This type cannot be written by a programmer in Java's syntax. Each occurrence of a wildcard (`?`) in the syntax causes a fresh type variable to be generated during capture conversion. The type `Pair<?, ?>` represents a pair of two elements that can be of two distinct types. However, the `compare` method requires a pair where both elements are of the same type. The first method invocation `this.compare(pair)` does not type check, as a result. The second method invocation `this.compare(this.make(list))` does type check because the capture converted type of the expression `this.make(list)` is `Pair<X, X>`, where `X` is a fresh type variable. Hence, types such as $\exists$`X.Pair<X, X>` are needed to model types that can arise during type checking with wildcards. VarLang does not support types that can model a pair of the same unknown type. VarJ extends TameFJ and supports both definition-site variance and existential types.

5.2.2 Scope of Wildcards

In Java, the types of two expressions never share a type variable resulting from capture conversion. This holds even when both of the two expressions are exactly the same.

Consider the method `exchangeFirsts` from Figure 5.1. It expects two lists that store elements of the same type. It swaps the first element in both of the lists with each other. Each time a wildcard type is capture converted, the generated type variables are distinct from variables occurring in all other types that result from capture conversion. The method invocation `exchangeFirsts(list, list)` in Figure 5.1 does *not* type check for that reason. Although both actual arguments are

the same expression, the types assigned to the two occurrences of `list` are `List<Y>` and `List<Z>`, respectively, where type variables `Y` and `Z` are distinct from each other and from all other type variables that occur in different types. As a result, the Java compiler assumes that two lists of two different types were passed as arguments in `exchangeFirsts(list, list)`.

Rejecting this method call may seem too conservative because it seems to be safe from runtime type errors. In Java, the dynamic type of `list` may change from updating the value that `list` is bound to. The previous dynamic type may not be subtype-related with the new dynamic type. When a method call is executed, the actual arguments of the method call are evaluated (from left to right in Java) before executing the code of the method body. `list` may initially be set to `new List<Dog>()` and then set to `new List<Cat>()` before the method body of `exchangeFirsts` is evaluated. For example, the following expression is legal in Java:

```
exchangeFirsts((list = new List<Dog>()), (list = new List<Cat>()))
```

In Java, an assignment $(x = e)$ is an expression. The value returned by an assignment is the value of the expression on the right-hand side. The type of an assigment is the declared type of the variable on the left-hand side. The type of both actual arguments is `list`. Although this example is contrived, many programs implement *threads* [28, Chapter 17] that modify shared memory. The execution of `exchangeFirsts(list, list)` may be interrupted by a thread that changes the value of `list`.

As a result, evaluating `exchangeFirsts(list, list)` may reduce it to `exchangeFirsts(new List<Dog>(), new List<Cat>())`. Since `List` is invariant, there does not exists a type instantiation of `List` that is a supertype of both argument types,

`List<Dog>` and `List<Cat>`.[4] A runtime type error can result if `exchangeFirsts(list,` `list)` was permitted to execute. Passing a `List<Dog>` and a `List<Cat>` would add a `Dog` to a `List<Cat>` and vice versa. We describe a type-safe way of passing two occurrences of `list` indirectly to a call to `exchangeFirsts` in Section 5.2.3.

The scope of existential type variables is used to model that the types of two Java expressions never share a type variable that resulted from a wildcard. An existential type $\exists\Delta.$`R` in TameFJ consists of an environment Δ and a body `R`. Existential type variables are declared in the environment and are bound (are in scope) in the body `R`. Existential type variables are not global variables and are bound only within the existential type that they are declared in. The grammar of existential types in VarJ is given in Figure 6.1.

In TameFJ, in the method invocation `exchangeFirsts(list, list)`, both of the actual arguments could have been assigned the type $\exists$`X.List<X>`. However, the occurrences of `X` in the bodies of the existential types $\exists$`X.List<X>` and $\exists$`X.List<X>` refer to two different *binders* or *declarations* of type variables. Type checking should not depend on the specific names chosen for binders. For example, the type of the second occurrence of `list` in `exchangeFirsts(list, list)` could have been either $\exists$`X.List<X>` and $\exists$`Y.List<Y>`. The types $\exists$`X.List<X>` and $\exists$`Y.List<Y>` are *alpha-equivalent* [53, Section 5.3] because they only differ in the names of their binders. Many proofs of language properties depend on the ability to swap syntactic terms that are alpha-equivalent without invalidating a proof [64]. The type safety proofs for TameFJ and VarJ also depend on this capability. Barendregt's variable convention is explained in more detail in Section 10.2.6.

---

[4]Although `List<?>` is a supertype of both `List<Dog>` and `List<Cat>`, '?' is not a type. Thus, `List<?>` is not a type instantiation of `List`.

5.2.3 Wildcard Capture

Wildcard capture [28, Section 15.12.2.7] is the process of passing an unknown type, hidden by a wildcard, as a type parameter in a method invocation. Consider the method invocation `uselessExchange(list)` from Figure 5.1. Although a programmer may want to pass an object of type `List<?>` as a value argument to the `uselessExchange` method, the type parameter to pass for `X` cannot be manually specified because the type hidden by `?` cannot be named by the programmer. That is, a programmer cannot specify a type `T` such that `this.<T>uselessExchange(list)` typechecks. However, passing a `List<?>` typechecks because Java allows the compiler to automatically generate a name for the unknown (capture converted) type and use that name in a method invocation. VarJ models wildcard capture and its interaction with definition-site variance. This interaction requires significant changes in our formalism relative to TameFJ [12].

Recall that the expression `exchangeFirsts(list, list)` does not type check. However, `uselessExchange(list)` seems to perform the same behavior as the former expression. The difference is that throughout the execution of the method body of `uselessExchange`, the formal argument `l` is always bound to a single instantiation of `List`. `l` can only be set to a different instance of `List<X>` within the method body. Changing the value `list` does not change the value of `l` because they are two different references.

We conclude this section by showing how wildcard capture can simplify the type signatures of methods that perform type independent operations. [9, Chapter 5, Item 28] states the rule "if a type parameter appears only once in a method declaration, replace it with a wildcard." Example code following this rule is below. Method `swapLastTwo` swaps the order of the two elements at the top of a stack. Method call `swapLastTwoHelper(stack)` typechecks because of wildcard capture. The signa-

ture of `swapLastTwo` is arguably simpler than `swapLastTwoHelper` because invoking

`swapLastTwo` does not require a type argument.

```
public void swapLastTwo(Stack<?> stack) { swapLastTwoHelper(stack); }

private <E> void swapLastTwoHelper(Stack<E> stack) {
  E elem1 = stack.pop();
  E elem2 = stack.pop();
  stack.push(elem2);
  stack.push(elem1);
}
```

5.3   F-bounded polymorphism

Another language feature that significantly complicates variance reasonining is F-bounded polymorphism [13]. An F-bound is a recursive bound in a subtype constraint on a type parameter `X`, where the type bound `T` includes an occurrence of `X`.[5] Consider the following definition:

```
interface Trouble<P extends List<P>> extends Iterator<P> {}
```

The type `Trouble<P>` extends `Iterator<P>`, which is assumed in the example to be covariant (exporting a method "`P next()`", per the Java library convention for iterators). It would stand to reason that `Trouble` is also covariant: an object of type `Trouble<P>` does precisely what an `Iterator<P>` object does, since it simply inherits methods. Consider, however, the type `Trouble<? super A>`. This is a contravariant use of a covariant generic. According to our approach for combining variances, this results in a bivariant type (due to the variance joining described in Section 3.2). For

---

[5]There are restrictions in Java for what an F-bound can be in [28, Section 4.4]. For example, a type variable `X` cannot be bounded by just itself (i.e., the constraint `X extends X` is not allowed).

example, we can derive the following subtype relationship even though the types, `MyList` and `YourList`, are not subtype-related.

Trouble<YourList> <: Trouble<Object> (by covariance assumption of `Trouble`)

<: Trouble<? super Object>

<: Trouble<? super MyList>

The problem, however, is that the bounds of type variables (`List<P>` in this case) are preserved in the existential type representing a use of `Trouble` with wildcards. This results in unsoundness because, in F-bounded polymorphism, the bound includes the hidden type, allowing its recovery and use. We can cause a problem with the following code (`ArrayList` is a standard implementation of the usual Java `List` interface, both invariant types):[6]

```
class MyList extends ArrayList<MyList> { }
class YourList extends ArrayList<YourList> {
  int i = 0;
  public boolean add(YourList list)
  {  System.out.println(list.i); return super.add(list); }
}
void foo(Trouble<? super MyList> itr) {
  itr.next().add(new MyList());
}
void main() {
  Trouble<YourList> preitr = ...;
  foo(preitr);
}
```

Function `foo` typechecks because `itr.next()` is guaranteed to return an unknown supertype, `X`, of `MyList` but also (due to the F-bound on `Trouble`) a subtype of `List<X>`. Thus, `X` has a method `add` (from `List`) which accepts `X` instances, and thus also accepts `MyList` instances (since `X` is a supertype of `MyList`).

---

[6]This example is originally due to Ross Tate.

The problem arises in the last line, `foo(preitr)`. If `Trouble<? super MyList>` were truly bivariant (as a contravariant use of a covariant generic), then that line would type check, allowing the unsound addition of a `MyList` object to a list of `YourList`s.

Note that this example is not a counterexample to VarLang's soundness Theorem 3 because `Trouble` is not covariant by the subsumption principle. For example, it is *not* safe to assume `Trouble<Dog> <: Trouble<Animal>`. An instance of `Trouble<Animal>` can return a `List<Animal>` using the `next` method but a `Trouble<Dog>` cannot. Invoking `next()` on an instance of `Trouble<Dog>` returns an instance of `List<Dog>`, which is not a subtype of `List<Animal>`, by the invariance of `List`.

This example only shows that the joining of definition- and use-site variances needs to be carefully restricted in the presence of F-bounded polymorphism. In particular, variance of bounds cannot be ignored when an F-bound occurs *within* a type expression rather than a type definition. We discuss this issue in more detail in Section 6.7.1.

CHAPTER 6

VARJ

We investigate extending Java with definition-site variance by developing the VarJ calculus [2]. VarJ is a *type system* [53] that models a subset of Java that is extended with definition-site variance. The definition of VarJ follows the standard approach to defining a type system. In this approach, a set of inference rules and mathematical functions are used to precisely state the semantics of the language. Such a rigorous definition of a language facilitates mathematical proofs of properties over the language. VarJ is defined with this approach. Using VarJ, we prove that our variance reasoning is safe or prohibits runtime type errors.

6.1   VarJ Syntax

VarJ is an extension of a past formalism, TameFJ by Cameron et al [12]. VarJ's syntax is found in Figure 6.1. A program that typechecks in TameFJ also typechecks in VarJ. Significant differences are highlighted using shading. To improve readability, some syntactic categories are overloaded with multiple meta-variables. *Existential types* range over T, U, V, and S. *Type variables* range over X, Y, and Z. *Bounds* range over B and A. *Variances* range over v and w. The bottom type, $\perp$, is used only as a lower bound.

We follow the syntactic conventions of TameFJ: all source-level type expressions are written as existential types, with an empty range for non-wildcard Java type uses and type variables written as $\exists \emptyset.$X; substitution is performed as usual except $[$T$/$X$]\exists \emptyset.$X $=$ T; $\star$ is a syntactic marker designating that a method type parameter

(i.e., for a polymorphic method) should be inferred. When a non-existential type `R` is written in a context where an existential type is expected, `R` denotes $\exists\emptyset$.`R`. For example, `List<X>` denotes $\exists\emptyset$.`List<`$\exists\emptyset$.`X>` in an appropriate context.

Class type parameters ($\overline{\mathtt{X}}$) now have definition-site variance annotations ($\overline{\mathtt{v}}$) and lower bounds ($\overline{\mathtt{B_L}}$). Method type variables now have lower bounds as well. When the bounds of a type variable are skipped, the implicit lower and upper bounds are the bottom type $\perp$ and the top type $\exists\emptyset$.`Object`, respectively. For example, $\exists\mathtt{X}$.`List<X>` denotes $\exists\mathtt{X} \rightarrow [\perp\text{-}\exists\emptyset$.`Object]`.`List<X>`. Also, when no type arguments are supplied in a parameteric type the angle brackets can be skipped; e.g., `Animal` denotes `Animal<>`.

The auxiliary function $fv(t)$ returns the set of free variables in term t; a term is string in the grammar of any syntactic category of VarJ.

The remainder of this section focuses on semantic differences between VarJ and TameFJ and new concepts from adding definition-site variance. Sections 6.2 and 6.3 formally present notions of the variance of a type expression and the variance of a type position. Section 6.4 covers subtyping with definition-site and use-site variance in VarJ. Section 6.5 discusses the updates made to allow safe interaction between wildcard capture and variant types.

6.2   Variance of a Type

Before we embark on the specifics of the VarJ formalism, we examine the essence of variance reasoning, i.e., how variances are computed in type expressions. For now, consider the subtyping relation of our formalism as a black box—it will be defined in Section 6.4. When is a type instantiation `C<`$Exp1$`>` a subtype of another instantiation `C<`$Exp2$`>`? We answer a more general question using the predicate $var(\mathtt{X}; \mathtt{T})$, where `X` is a type variable and `T` is a type expression. The definition of var in Section 4.2 is defined only for VarLang types. We redefine var in this section over VarJ types. The definition of var in this section is more general because it is defined over a richer set

Syntax:

e	::=	x $\mid$ e.f $\mid$ e.<$\overline{\text{P}}$>m($\overline{\text{e}}$) $\mid$ new C<$\overline{\text{T}}$>($\overline{\text{e}}$)	*expressions*
s	::=	new C<$\overline{\text{T}}$>($\overline{\text{s}}$)	*values*
v	::=	$+ \mid - \mid * \mid o$	*variance*

Q	::=	class C<$\overline{\text{v X} \rightarrow \text{[B}_L\text{ -B}_U\text{]}}$> $\lhd$ N $\{$ $\overline{\text{T f;}}$ $\overline{M}$ $\}$	*class declarations*
M	::=	<$\overline{\text{X} \rightarrow \text{[B}_L\text{ -B}_U\text{]}}$> T m($\overline{\text{T x}}$) $\{$ return e; $\}$	*method declarations*

N	::=	C<$\overline{\text{T}}$>	*non-variable types*
R	::=	N $\mid$ X	*non-existential types*
T	::=	$\exists\Delta.$N $\mid$ $\exists\emptyset.$X	*existential types*
B	::=	T $\mid$ $\bot$	*type bounds*
P	::=	T $\mid$ $\star$	*method type parameter*

Δ	::=	$\overline{\text{X} \rightarrow \text{[B}_L\text{–B}_U\text{]}}$	*type ranges*
Γ	::=	$\overline{\text{x : T}}$	*var environments*
X	::=	...	*type vars*
x	::=	...	*expr vars*
C	::=	...	*class names*

Figure 6.1. VarJ Syntax

58

of types, which include existential types. Furthermore, this section provides insight into how to define *var* over types with syntactic forms that are not in the grammar of VarJ.

The goal of *var* is to determine the following: Given a type variable `X` and a type expression `T` that can contain `X`, what is the subtyping relationship between different "instantiations" of `T` with respect to (wrt) `X`, where an instantiation of `T` wrt to `X` is a substitution for `X` in `T`. For example, we want $var(\texttt{X};\texttt{T}) = +$ to imply `[U/X]T` $<:$ `[U'/X]T`, if `U` $<:$ `U'`.

To define *var*, we use predicate $\texttt{v}(\texttt{T};\texttt{T}')$ as a notational shorthand, denoting the kind of subtype relation between `T` and `T'`:

- $+(\texttt{T};\texttt{T}') \equiv \texttt{T} <: \texttt{T}'$
- $-(\texttt{T};\texttt{T}') \equiv \texttt{T}' <: \texttt{T}$
- $\texttt{o}(\texttt{T};\texttt{T}') \equiv +(\texttt{T};\texttt{T}') \wedge -(\texttt{T};\texttt{T}')$
- $*(\texttt{T};\texttt{T}') \equiv true$

Note that, by the variance lattice (in Figure 2.1), we have

$$\texttt{v} \leq \texttt{w} \implies \left[\texttt{v}(\texttt{T};\texttt{T}') \implies \texttt{w}(\texttt{T};\texttt{T}') \right] \tag{6.1}$$

In general, we want for *var* the following property, which is a generalization of the subtype lifting lemma of Emir et al.'s modeling of definition-site variance [24]:

$$var(\texttt{X};\texttt{T}) = \texttt{v} \implies \left[\texttt{v}(\texttt{U};\texttt{U}') \implies \texttt{[U/X]T} <: \texttt{[U'/X]T} \right] \tag{6.2}$$

By (6.1), (6.2) entails a more general implication:

$$\texttt{v} \leq var(\texttt{X};\texttt{T}) \implies \left[\texttt{v}(\texttt{U};\texttt{U}') \implies \texttt{[U/X]T} <: \texttt{[U'/X]T} \right] \tag{6.3}$$

We assume there is a usual class table CT that maps class identifiers `C` to their definition (i.e., $CT(\texttt{C}) = $ `class C<`$\overline{\texttt{vX} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}$`> ◁ N {` ... `}`). Similarly, we define a

variance table VT that maps class identifiers to their type parameters with their def-site variances. For example, assuming the class table mapping above, $VT(\texttt{C}) = \overline{\texttt{vX}}$. VT is overloaded to take an extra index parameter i to the ith def-site variance annotation; e.g, if $VT(\texttt{C}) = \overline{\texttt{vX}}$, then $VT(\texttt{C}, i) = \texttt{v}_i$.

Variance of Types and Ranges: $var(\texttt{X}; \phi)$, where $\phi ::= \texttt{B} \mid \texttt{R} \mid \Delta$

$var(\texttt{X}; \texttt{X}) = +$ (VAR-XX)

$var(\texttt{X}; \texttt{Y}) = *$, if $\texttt{X} \neq \texttt{Y}$ (VAR-XY)

$var(\texttt{X}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \prod_{i=1}^{n} \big(\texttt{v}_i \otimes var(\texttt{X}; \texttt{T}_i) \big)$, if $VT(\texttt{C}) = \overline{\texttt{vX}}$ (VAR-N)

$var(\texttt{X}; \bot) = *$ (VAR-B)

$var(\texttt{X}; \exists\Delta.\texttt{R}) = var(\texttt{X}; \Delta) \sqcap var(\texttt{X}; \texttt{R})$, if $\texttt{X} \notin dom(\Delta)$ (VAR-T)

$var(\texttt{X}; \overline{\texttt{Y} \rightarrow [\texttt{B}_L\text{-}\texttt{B}_U]}) = \prod_{i=1}^{n} \Big(\big(- \otimes var(\texttt{X}; \texttt{B}_{Li}) \big) \sqcap \big(+ \otimes var(\texttt{X}; \texttt{B}_{Ui}) \big) \Big)$ (VAR-R)

$\Big[var(\overline{\texttt{X}}; \phi) = \overline{\texttt{v}} \Big] \equiv \Big[\forall i, var(\texttt{X}_i; \phi) = \texttt{v}_i \Big]$, where $\phi ::= \texttt{B} \mid \texttt{R} \mid \Delta$ (VAR-SEQ)

Figure 6.2. Variance of types and ranges

The expression $var(\texttt{X}; \texttt{B})$ computes the variance of type variable $\texttt{X}$ in type expression $\texttt{B}$. Figure 6.2 contains var's definition. var's type input is overloaded for non-existential types ($\texttt{R}$) and type ranges (Δ). ($var(\overline{\texttt{X}}; \phi)$ is further overloaded in the expected way for computing variances for sequences of type variables.)

The var relation is used in our type system to determine which variance is appropriate for each type expression. Eventually our proof connects it to the subtype relation, in Lemma 1. (Proofs of all key lemmas can be found in Appendix B.)

Lemma 1 (Subtype Lifting Lemma)**.** If (a) $\overline{\texttt{v}} \leq var(\overline{\texttt{X}}; \texttt{B})$ and (b) $\Delta \vdash \overline{\texttt{v}(\texttt{T}; \texttt{U})}$ then $\overline{[\texttt{T}/\texttt{X}]}\texttt{B} <: \overline{[\texttt{U}/\texttt{X}]}\texttt{B}$.

We provide some intuition on the soundness of var's definition. One "base case" of var's definition is the VAR-XX rule. To see why it returns $+$, note that the desired implication from the subtype lifting lemma holds for this case: if $+(\texttt{T}; \texttt{U})$, which is equivalent to $\texttt{T} <: \texttt{U}$, then $[\texttt{T}/\texttt{X}]\texttt{X} = \texttt{T} <: \texttt{U} = [\texttt{U}/\texttt{X}]\texttt{X}$. The VAR-N rule computes

the variance in a non-variable type using the $\otimes$ operator, which determines how variances compose, as described in Section 3.1. VAR-R computes the variance of a type variable in a range. Computing the variance of ranges is necessary for computing the variance of constraints from type bounds on type parameters, which occur in existential types and method signatures. The domains of ranges are ignored by VAR-R. A range becomes more "specialized" as the bounds get "squeezed". Informally, a range $\overline{[\texttt{B}_L - \texttt{B}_U]}$ is a *subrange* of range $\overline{[\texttt{A}_L - \texttt{A}_U]}$ if $\overline{\texttt{A}_L <: \texttt{B}_L}$ and $\overline{\texttt{B}_U <: \texttt{A}_U}$. The variance of the lower bound is transformed by contravariance to "reverse" the subtype relation, since we want the lower bound in the subrange to be a *supertype* of the lower bound in the superrange.[1] The subtype lifting lemma can be used to entail subrange relationships:

$$var(\texttt{X}; \overline{\texttt{Y} \rightarrow [\texttt{B}_L\texttt{-B}_U]}) = \texttt{v} \textbf{ and } \texttt{v}(\texttt{T};\texttt{U})$$

$$\implies \overline{[\texttt{U}/\texttt{X}]\texttt{B}_L <: [\texttt{T}/\texttt{X}]\texttt{B}_L} \textbf{ and } \overline{[\texttt{T}/\texttt{X}]\texttt{B}_U <: [\texttt{U}/\texttt{X}]\texttt{B}_U}$$

The variance of an existential type variable is just the meet of the variances of its range (Δ) and its body (R). The VAR-T rule has the premise "$\texttt{X} \notin dom(\Delta)$" to follow *Barendregt's variable convention* [64], as in the TameFJ formalism. (*var* is undefined when this premise is not satisfied.) This convention is followed in the definitions of TameFJ and VarJ. Following this convention substantially reduces the number of places requiring alpha-conversion to be applied and allows for more elegant proofs. Furthermore, this convention ensures that predicates defined by the type system's rules are equivariant (respect alpha-renaming). In other words, changing the name of a bound variable does not invalidate any derived judgment. For example, this property holds for $var(\texttt{X};\texttt{T})$ because we can rename binders to fresh names in existential types in T without changing the variance of X in T. Without the premise

---

[1]Intuitively, the upper/lower bounds are in co-/contravariant positions, respectively.

of rule VAR-T, this property would no longer hold. So that the important premises are clearer, in the remaining rules we skip such "side-conditions" in the text and just mention that the premises for following the variable convention are implicit. Barendregt's variable convention is discussed further in Section 10.2.6.

Contrasting with *var*'s definition in Section 4.2, *var*'s definition in Figure 6.2 does not use that the lattice join operator $\sqcup$. Computing variance using the join operator is not safe in VarJ because F-bounded type variables can be declared *within* type expressions. We discuss this issue further in Section 6.7.1.

6.3 Variance of a Position

Satisfying the subsumption principle with variant subtyping requires assigning variances to positions in class definitions where types can occur. For example, return types are assumed to be in covariant positions while argument types are assumed to be in contravariant positions. These assumed variances of positions are used to typecheck class definitions and their def-site variance annotations.

The expressions "$\overline{v} \leq var(\overline{X}; B)$" and "$- \otimes v$" are used frequently in the VarJ formalism. To relate our notation to previous work, we define the following:

$$\left[\overline{vX} \vdash B\ mono\right] \equiv \left[\overline{v} \leq var(\overline{X}; B)\right] \tag{6.4}$$

$$\left[\neg v\right] = \left[- \otimes v\right] \tag{6.5}$$

A "monotonicity" judgment of the syntactic form "$\overline{vX} \vdash T\ mono$" appears originally in Emir et al.'s definition-site variance treatment [24] and later in Kennedy and Pierce [39] as "$\overline{vX} \vdash T\ ok$". The semantics of these judgments in the aforementioned sources are similar to its definition here but differs in that their work had no function similar to *var*, no $\otimes$ operator, nor a variance lattice. The negation operator $\neg$ also

appears in [24] and [39], and it is used to transform a variance by contravariance. Using the implications in Section 6.2, it is easy to show the following properties, which are important for type checking class definitions:

$$\mathtt{w} = \neg\mathtt{v} \implies \left[\mathtt{v}(\mathtt{B},\mathtt{B}') \iff \mathtt{w}(\mathtt{B}',\mathtt{B})\right] \tag{6.6}$$

$$\overline{\mathtt{vX}} \vdash \mathtt{B} \; mono \implies \left[\overline{\mathtt{v}(\mathtt{T},\mathtt{U})} \implies [\overline{\mathtt{T/X}}]\mathtt{B} <: [\overline{\mathtt{U/X}}]\mathtt{B}\right] \tag{6.7}$$

$$\overline{\neg\mathtt{vX}} \vdash \mathtt{B} \; mono \implies \left[\overline{\mathtt{v}(\mathtt{T},\mathtt{U})} \implies [\overline{\mathtt{U/X}}]\mathtt{B} <: [\overline{\mathtt{T/X}}]\mathtt{B}\right] \tag{6.8}$$

Figure 6.3 contains rules for checking class and method definitions and the definition of the *override* predicate. Premises related to type checking with definition-site variance are highlighted. Auxiliarly lookup functions, such as *mtype*, are used by the *override* definition and are used to compute the types of members (fields and methods) in class definitions. Their definitions are in Figure 6.4. These lookup functions take in non-variable types (N) instead of existential types. In the expression typing rules (in Figure 6.7), existential types are implicitly "unpacked" to non-variable types to type some expressions such as a field access. The process for packing and unpacking types is similar to the process performed in the TameFJ formalism. Section 6.5 has a brief overview of this process and an example type derivation.

The definition-site subtyping relation judgment $\Delta \vdash \mathtt{N} \prec: \mathtt{N}'$ is defined over non-variable types and considers definition-site annotations when concluding subtype relationships. For example, $VT(\mathtt{C}) = +\mathtt{X} \implies \Delta \vdash \mathtt{C}{<}\exists\emptyset.Dog{>} \prec: \mathtt{C}{<}\exists\emptyset.Animal{>}$, assuming $\Delta \vdash \exists\emptyset.Dog <: \exists\emptyset.Animal$. This relation is defined in Figure 6.6.

The motivation for the assumed variances of positions is to ensure the subsumption principle holds for the subtyping hierarchy. Informally, if $\mathtt{T} <: \mathtt{U}$, then a value of type $\mathtt{T}$ may be provided whenever a value of type $\mathtt{U}$ is required. In the case of VarJ, the

Class and Method Typing:

$$\frac{\boxed{\overline{\texttt{vX} \vdash \texttt{N}, \overline{\texttt{T}} \ mono} \qquad \Delta = \overline{\texttt{X} \to [\texttt{B}_L\text{-}\texttt{B}_U]}} \\ \emptyset \vdash \Delta \ OK \qquad \Delta \vdash \texttt{N}, \overline{\texttt{T}} \ OK \qquad \vdash \overline{M} \ OK \text{ in } \texttt{C}}{\vdash \texttt{class C<}\overline{\texttt{vX} \to [\texttt{B}_L\text{-}\texttt{B}_U]}\texttt{>} \lhd \texttt{N} \ \{ \ \overline{\texttt{T f}}; \ \overline{M} \ \} \ OK}$$
$$(\text{W-Cls})$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \to [\texttt{B}_L\text{-}\texttt{B}_U]}\texttt{>} \lhd \texttt{N} \ \{ \ \dots \ \} \\ \Delta = \overline{\texttt{X} \to [\texttt{B}_L\text{-}\texttt{B}_U]} \qquad \Delta \vdash \Delta' \ OK \qquad \Delta, \Delta' \vdash \texttt{T}, \overline{\texttt{T}} \ OK \\ override(\texttt{m}; \texttt{N}; \texttt{<}\Delta'\texttt{>} \ (\overline{\texttt{T}}) \to \texttt{T}) \qquad \boxed{\overline{\neg \texttt{vX} \vdash \overline{\texttt{T}}, \Delta' \ mono} \qquad \boxed{\overline{\texttt{vX} \vdash \texttt{T} \ mono}} \\ \Delta, \Delta'; \ \overline{\texttt{x} : \texttt{T}}, \texttt{this} : \exists \emptyset.\texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e} \ : \ \texttt{T} \mid \emptyset}{\vdash \texttt{<}\Delta'\texttt{>} \ \texttt{T} \ \texttt{m}(\overline{\texttt{T x}}) \ \{ \ \texttt{return e;} \ \} \ OK \text{ in } \texttt{C}}$$
$$(\text{W-Meth})$$

$$\frac{mtype(\texttt{m}; \texttt{N}) = \texttt{<}\Delta\texttt{>} \ (\overline{\texttt{T}}) \to \texttt{T}}{override(\texttt{m}; \texttt{N}; \texttt{<}\Delta\texttt{>} \ (\overline{\texttt{T}}) \to \texttt{T})} \qquad \frac{mtype(\texttt{m}; \texttt{N}) \text{ is undefined}}{override(\texttt{m}; \texttt{N}; \texttt{<}\Delta\texttt{>} \ (\overline{\texttt{T}}) \to \texttt{T})}$$
$$(\text{Over-Def}) \qquad\qquad\qquad (\text{Over-Undef})$$

Figure 6.3. Class and Method Typing

Lookup Functions:

Shared premise for lookup rules except F-Obj:

$$CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \to [\texttt{B}_L\text{-}\texttt{B}_U]}\texttt{>} \lhd \texttt{N} \ \{ \ \overline{\texttt{S f}}; \ \overline{M} \ \}$$

$$fields(\texttt{Object}) = \emptyset \hfill (\text{F-Obj})$$

$$fields(\texttt{C}) = \overline{\texttt{g}}, \overline{\texttt{f}}, \ \text{if } \texttt{N} = \texttt{D<}\overline{\texttt{U}}\texttt{>} \text{ and } fields(\texttt{D}) = \overline{\texttt{g}} \hfill (\text{F-Super})$$

$$ftype(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = ftype(\texttt{f}; [\overline{\texttt{T/X}}]\texttt{N}), \ \text{if } \texttt{f} \notin \overline{\texttt{f}} \hfill (\text{FT-Super})$$

$$ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T/X}}]\texttt{S}_i, \hfill (\text{FT-Class})$$

$$mtype(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mtype(\texttt{m}; [\overline{\texttt{T/X}}]\texttt{N}), \ \text{if } \texttt{m} \notin \overline{M} \hfill (\text{MT-Super})$$

$$mtype(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T/X}}](\texttt{<}\Delta\texttt{>} \ (\overline{\texttt{U}}) \to \texttt{U}), \\ \quad \text{if } \texttt{<}\Delta\texttt{>} \ \texttt{U} \ \texttt{m}(\overline{\texttt{U x}}) \ \{ \ \texttt{return e;} \ \} \in \overline{M} \hfill (\text{MT-Class})$$

$$mbody(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mbody(\texttt{m}; [\overline{\texttt{T/X}}]\texttt{N}), \ \text{if } \texttt{m} \notin \overline{M} \hfill (\text{MB-Super})$$

$$mbody(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \langle \overline{\texttt{x}}.[\overline{\texttt{T/X}}]\texttt{e} \rangle, \\ \quad \text{if } \texttt{<}\Delta\texttt{>} \ \texttt{U} \ \texttt{m}(\overline{\texttt{U x}}) \ \{ \ \texttt{return e;} \ \} \in \overline{M} \hfill (\text{MB-Class})$$

Figure 6.4. Lookup Functions

Wellformed Ranges: $\Delta \vdash \Delta \ OK$

$$\cfrac{}{\Delta \vdash \emptyset \ OK} \ \text{(W-Rng-empty)}$$

$$\cfrac{\texttt{X} \notin dom(\Delta) \qquad \Delta, \texttt{X} \to [\texttt{B}_L\texttt{-B}_U], \Delta' \vdash \texttt{B}_L, \texttt{B}_U \ OK \\ \Delta \vdash ubound_\Delta(\texttt{B}_L) \sqsubseteq: ubound_\Delta(\texttt{B}_U) \\ \Delta \vdash \texttt{B}_L <: \texttt{B}_U \qquad \Delta, \texttt{X} \to [\texttt{B}_L\texttt{-B}_U] \vdash \Delta' \ OK}{\Delta \vdash \texttt{X} \to [\texttt{B}_L\texttt{-B}_U], \Delta' \ OK} \ \text{(W-Rng)}$$

Non-Variable Upper Bound: $ubound_\Delta(\texttt{B})$

$$ubound_\Delta(\texttt{B}) = \begin{cases} ubound_\Delta(\texttt{B}_U), & \text{if } \texttt{B} = \exists\emptyset.\texttt{X}, \text{ where } \Delta(\texttt{X}) = [\texttt{B}_L - \texttt{B}_U] \\ \texttt{B}, & \text{if } \texttt{B} = \exists\Delta'.\texttt{N} \end{cases}$$

Wellformed Types: $\Delta \vdash \phi \ OK$, where $\phi ::= \texttt{B} \mid \texttt{P} \mid \texttt{R}$

$$\cfrac{}{\Delta \vdash \texttt{Object<>} \ OK} \ \text{(W-Obj)} \qquad \cfrac{\texttt{X} \in dom(\Delta)}{\Delta \vdash \texttt{X} \ OK} \ \text{(W-X)} \qquad \cfrac{}{\Delta \vdash \bot \ OK} \ \text{(W-B)} \qquad \cfrac{}{\Delta \vdash \star \ OK} \ \text{(W-I)}$$

$$\cfrac{\texttt{class C<}\overline{\texttt{vX} \to [\texttt{B}_L\texttt{-B}_U]}\texttt{>} \lhd \texttt{N} \{ \quad \dots \ \} \\ \Delta \vdash \overline{[\texttt{T/X}]\texttt{B}_L <: \texttt{T}} \qquad \Delta \vdash \overline{\texttt{T} <: [\texttt{T/X}]\texttt{B}_U} \\ \Delta \vdash \overline{\texttt{T}} \ OK}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \ OK} \ \text{(W-N)} \qquad \cfrac{\Delta \vdash \Delta' \ OK \qquad \Delta, \Delta' \vdash \texttt{R} \ OK}{\Delta \vdash \exists\Delta'.\texttt{R} \ OK} \ \text{(W-T)}$$

Wellformed Expression Variable Environments $\Delta \vdash \Gamma \ OK$

$$\cfrac{}{\Delta \vdash \emptyset \ OK} \ \text{(W-Env-Empty)} \qquad \cfrac{\texttt{x} \notin dom(\Gamma) \qquad \Delta \vdash \texttt{T} \ OK \qquad \Delta \vdash \Gamma \ OK}{\Delta \vdash \Gamma, \texttt{x} : \texttt{T} \ OK} \ \text{(W-Env)}$$

Figure 6.5. Wellformedness Judgments

$$\boxed{\begin{array}{l}
\textbf{Definition-Site Subtyping: } \texttt{R} \prec: \texttt{R} \\[2mm]
\dfrac{\begin{array}{c} \texttt{class C<}\overline{\texttt{vX} \rightarrow [\texttt{B}_L\texttt{-B}_U]}\texttt{>} \lhd \texttt{N } \{ \quad \dots \} \\ \texttt{C} \neq \texttt{D} \qquad \Delta \vdash \overline{[\texttt{T/X}]}\texttt{N} \prec: \texttt{D<}\overline{\texttt{U}}\texttt{>} \end{array}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{D<}\overline{\texttt{U}}\texttt{>}} \\
\hspace{3cm} (\textsc{SD-Super}) \\[2mm]
\dfrac{\begin{array}{c} VT(\texttt{C}) = \overline{\texttt{vX}} \\ \Delta \vdash \overline{\texttt{v(T,U)}} \end{array}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{C<}\overline{\texttt{U}}\texttt{>}} \qquad \dfrac{}{\Delta \vdash \texttt{X} \prec: \texttt{X}} \\
\hspace{1cm} (\textsc{SD-Var}) \hspace{2.5cm} (\textsc{SD-X}) \\[4mm]

\textbf{Existential Subtyping: } \Delta \vdash \texttt{B} \sqsubset: \texttt{B} \\[2mm]
\dfrac{\Delta, \Delta' \vdash \texttt{N} \prec: \texttt{N}'}{\Delta \vdash \exists\Delta'.\texttt{N} \sqsubset: \exists\Delta'.\texttt{N}'} \qquad \dfrac{}{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}} \qquad \dfrac{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}' \quad \Delta \vdash \texttt{B}' \sqsubset: \texttt{B}''}{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}''} \\
\hspace{0.5cm}(\textsc{SE-SD}) \hspace{2cm} (\textsc{SE-Refl}) \hspace{2.5cm} (\textsc{SE-Tran}) \\[3mm]

\dfrac{}{\Delta \vdash \perp \sqsubset: \texttt{B}} \qquad \dfrac{\begin{array}{c} dom(\Delta') \cap fv(\exists\overline{\texttt{X} \rightarrow [\texttt{B}_L\texttt{-B}_U]}.\texttt{N}) = \emptyset \qquad fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta') \\ \Delta, \Delta' \vdash \overline{[\texttt{T/X}]\texttt{B}_L <: \texttt{T}} \qquad \Delta, \Delta' \vdash \overline{\texttt{T} <: [\texttt{T/X}]\texttt{B}_U} \end{array}}{\Delta \vdash \exists\Delta'.\overline{[\texttt{T/X}]}\texttt{N} \sqsubset: \exists\overline{\texttt{X} \rightarrow [\texttt{B}_L\texttt{-B}_U]}.\texttt{N}} \\
(\textsc{SE-Bot}) \hspace{4cm} (\textsc{SE-Pack}) \\[3mm]

\textbf{Subtyping: } \Delta \vdash \texttt{B} <: \texttt{B} \\[2mm]
\dfrac{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}'}{\Delta \vdash \texttt{B} <: \texttt{B}'} \qquad \dfrac{\Delta \vdash \texttt{B} <: \texttt{B}' \quad \Delta \vdash \texttt{B}' <: \texttt{B}''}{\Delta \vdash \texttt{B} <: \texttt{B}''} \qquad \dfrac{\Delta(\texttt{X}) = [\texttt{B}_L - \texttt{B}_U]}{\Delta \vdash \texttt{B}_L <: \exists\emptyset.\texttt{X}} \\
\hspace{0.5cm}(\textsc{ST-SE}) \hspace{2cm} (\textsc{ST-Tran}) \hspace{2.5cm} (\textsc{ST-Lbound}) \\
\hspace{6.5cm} \dfrac{}{\Delta \vdash \exists\emptyset.\texttt{X} <: \texttt{B}_U} \\
\hspace{6.5cm} (\textsc{ST-Ubound})
\end{array}}$$

Figure 6.6. Subtyping Relations

subsumption principle is established by showing appropriate subtype relationships between types of members from class definitions. Lemma 2 states a goal subsumption property, which is to have the type of field $\texttt{f}$ of the supertype $\texttt{N}'$ become a more specific type for the subtype $\texttt{N}$. Although inherited fields syntactically have the same type as in the superclass definition, definition-site subtyping allows fields to have more specific types in the subtype. Lemma 3 states the goal subsumption property for types in method signatures; the sixth conclusion of this lemma holds because of the *override* predicate.

Lemma 2 (Subtyping Specializes Field Type). *If* (a) $\vdash$ `class C<`$\overline{\texttt{vX} \to \texttt{[...]}}$`>` $\lhd$ `N`$\ldots$ *OK and* (b) $\Delta \vdash$ `C<`$\overline{\texttt{T}}$`>` $\prec:$ `N'` *and* (c) $\mathit{ftype}(\texttt{f}; \texttt{N'}) = \texttt{T}$, *then* $\Delta \vdash \mathit{ftype}(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) <: \texttt{T}$.[2]

Lemma 3 (Subtyping Specializes Method Type). *If* (a) $\vdash$ `class C<`$\overline{\texttt{vX} \to \texttt{[...]}}$`>` $\lhd$ `N`$\ldots$ *OK and* (b) $\Delta \vdash$ `C<`$\overline{\texttt{T}}$`>` $\prec:$ `N'` *and* (c) $\mathit{mtype}(\texttt{m}; \texttt{N'}) = \texttt{<}\overline{\texttt{Y} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{>}\ (\overline{\texttt{U}}) \to \texttt{U}$, *then:* (1) $\mathit{mtype}(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y} \to \texttt{[A}_L\texttt{-A}_U\texttt{]}}\texttt{>}\ (\overline{\texttt{V}}) \to \texttt{V}$, (2) $\Delta \vdash \texttt{V} <: \texttt{U}$, (3) $\Delta \vdash \overline{\texttt{U} <: \texttt{V}}$, (4) $\Delta \vdash \overline{\texttt{A}_L <: \texttt{B}_L}$, (5) $\Delta \vdash \overline{\texttt{B}_U <: \texttt{A}_U}$, *and* (6) $\overline{\mathit{var}(\overline{\texttt{Y}}; \texttt{U}) = \mathit{var}(\overline{\texttt{Y}}; \texttt{V})}$.

To satisfy the two lemmas above, we make assumptions about the variance of the positions that types can occur in. To *preserve* the subtype relationship order of a type in a member signature, we assume the type occurs in a *covariant* position (i.e., the subtype needs to have a more specific type appear in such a position). To *reverse* the subtype relationship order of a type in a member signature, we assume the type occurs in a *contravariant* position. The assumptions about the variance of the positions are reflected in the *mono* judgments in the W-Cls and W-Meth rules for checking class and method definitions. By (6.7), *not* negating the def-site variance annotations, $\overline{\texttt{v}}$, in the judgment "$\overline{\texttt{vX}} \vdash \texttt{T}$ *mono*" reflects that `T` is assumed to be in a covariant position. Since covariance, $+$, is the identity element for the $\otimes$ operator ($+ \otimes \texttt{v} = \texttt{v}$), the variances $\overline{\texttt{v}}$ do not need to be transformed by $+$. By (6.8), negating the def-site variance annotations in the judgment "$\overline{\neg\texttt{vX}} \vdash \texttt{T}$ *mono*" reflects that `T` is assumed to be in a contravariant position. We need to reverse the subtype relationship order for argument types and ranges in method type signatures. Negating the variance annotations for the argument types ensures the argument types are more general supertypes for the subtype.[3]

---

[2]If field assignments were allowed, then field types would be in both co- and contravariant positions, and both $\mathit{ftype}(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>})$ and $\mathit{ftype}(\texttt{f}; \texttt{N'})$ would be subtypes of each other. This is explained further in Section 7.2.

[3]Bounds on class type parameters may make unrestricted use of type parameters by similar reasoning as in [24, p.7]. Once an object is created, they are forgotten.

Negating the range of a method type signature ensures the range is *wider* for the subtype. For code examples motivating why ranges need to be widened for the subtype, see Section 2.4 of [24]. More generally, if (1) `e.<T>m()` typechecks implying the type actual `T` is within the type bounds for `m`'s type argument and (2) *typeof*(`e'`) <: *typeof*(`e`), then `e'.<T>m()` should type check as well even if `m` is overridden in the subclass. Hence, the subtype's version of `m` should accept a superset/wider range of types than accepted by the supertype's version of `m`.

6.4   Subtyping

Subtyping in VarJ is defined similarly to TameFJ. Figure 6.6 contains the subtyping rules. There are three levels of subtyping in VarJ, as in TameFJ. The first level of subtyping in TameFJ, the subclass relation, has been replaced with the definition-site subtyping relation ≺: defined on non-existential types. Def-site subtyping is defined by the `SD-*` rules, which are similar to the subtyping rules from [39]. Like the subtype relation from [39], ≺: is defined by syntax-directed rules[4] and shares the reflexive and transitive properties by similar reasoning as in [39]. The ≺: judgment requires a typing context to check subtyping relationships between pairs of type actuals as done in the SD-Var rule.

The existential subtyping relation ⊏: is defined by the `SE-*` rules and is similar to the "Extended subclasses" relation in TameFJ. The XS-Env rule from TameFJ was renamed to SE-Pack; it is the only subtyping rule that can pack (and actually also unpack) types into existential type variables. The XS-Sub-Class rule was not only renamed to SE-SD but also had its premise updated to use def-site subtyping. SE-SD allows def-site subtyping to be applied to both type variables

---

[4]The syntax-directed nature of these rules does not ensure that an algorithmic test of ≺: is straightforward, because the premise of rule SD-Var appeals to the definition of the full <: relation (hidden inside the `v` shorthand).

in the type context Δ and existential type variables in Δ'. As a result, a type packed into an existential type variable may not be in the range of the variable. For example, if `Iterator` is covariant in its type parameter ($VT(\texttt{Iterator}) = \texttt{+X}$), then the following subtype relationship is derivable: $\exists\emptyset.Iterator < PrettyDog >{<}:$ $\exists\emptyset.Iterator < Dog >{<}: \exists\texttt{Y} \rightarrow \texttt{[Dog-Animal]}.\texttt{Iterator<Y>}$. Subtyping between two types implies the subsumption principle between the types. Since `Iterator<Dog>` can be packed into $\exists\texttt{X} \rightarrow \texttt{[Dog-Animal]}.\texttt{Iterator<X>}$ and `Iterator<PrettyDog>` <: `Iterator<Dog>`, it must be the case that `Iterator<PrettyDog>` can also be packed into $\exists\texttt{X} \rightarrow \texttt{[Dog-Animal]}.\texttt{Iterator<X>}$. This intuition is formalized in Lemma 4, which is similar to Lemma 35 from TameFJ, and establishes a relationship between existential subtyping and def-site subtyping.

Lemma 4 (Existential subtyping to def-site subtyping)**.** If (a) $\Delta \vdash \exists\Delta'.\texttt{R}' \sqsubset:$ $\exists\overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}.\texttt{R}$ and (b) $\emptyset \vdash \Delta \ OK$, then there exists $\overline{\texttt{T}}$ such that: (1) $\Delta, \Delta' \vdash$ $\texttt{R}' \prec: \overline{\texttt{[T/X]}}\texttt{R}$ and (2) $\Delta, \Delta' \vdash \overline{\texttt{[T/X]}\texttt{B}_L <: \texttt{T}}$ and (3) $\Delta, \Delta' \vdash \overline{\texttt{T} <: \texttt{[T/X]}\texttt{B}_U}$ and (4) $fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta')$.

Existential subtyping does not conclude subtype relationships for type variables except for the reflexive case using SE-REFL. The (all) subtyping relation <: allows non-reflexive subtype relationships with type variables by considering their bounds in the typing context. Since `T` or `U` may be type variables in a subtype relationship `T` <: `U`, we want a stronger relationship between the non-variable upper bounds of `T` and `U`. Lemma 5 formalizes this notion and is similar to lemma 17 from TameFJ. The non-variable upper bound of a type `T` is $ubound_\Delta(\texttt{T})$, defined in Figure 6.5.

Lemma 5 (Subtyping to existential subtyping)**.** If (a) $\Delta \vdash \texttt{T} <: \texttt{T}'$ and (b) $\emptyset \vdash \Delta \ OK$ then $\Delta \vdash ubound_\Delta(\texttt{T}) \sqsubset: ubound_\Delta(\texttt{T}')$.

6.5 Typing and Wildcard Capture

The expression typing rules in VarJ are mostly the same as in TameFJ and are given in Figure 6.7. Unlike TameFJ, VarJ allows method signatures to have lower bounds. The *sift* function is needed for safe wildcard capture and is applied in the T-Invk rule for typing method invocations. The definition of *sift* required updating because of interaction with variant types. First, we give a brief overview of expression typing; see [12] for more thorough coverage.

6.5.1 Expression Typing

Consider the Java segment below. It typechecks because the expression `box.elem` is typed as `String`. The type of `box.elem` is the same as the type actual passed to the `Box` type constructor. In this case, the type actual is "`? extends String`", which refers to some unknown subtype of `String`. To type `box.elem` with some known/named type, the most specific named type that can be assigned to `box.elem` is chosen, which is `String`.

```
class Box<E> { E elem; Box(E elem) { this.elem = elem; } }
Box<? extends String> box = ...
box.elem.charAt(0);
```

We explain this type derivation through the formal calculus. Types hidden by wildcards such as "`? extends String`" are "captured" as existential type variables. The type `Box<? extends String>` is modeled in VarJ by $\exists X \to [\bot\text{-}String].Box<X>$. Expression typing judgments have the form $\Delta; \Gamma \vdash e : T \mid \Delta'$. The second type variable environment Δ' is the *guard* of the judgment. It is used to keep track of type variables that have been unpacked from existential types during type checking. Variables in $dom(\Delta')$ may occur free in T and model hidden types. To type an expression without exposed (free) hidden types (existential type variables), the T-Subs rule is applied to find a suitable type without free existential type variables. The example typ-

Expression Typing: $\Delta;\ \Gamma \vdash \mathtt{e}\ :\ \mathtt{T}\mid\Delta$

$$\frac{}{\Delta;\ \Gamma \vdash \mathtt{x}\ :\ \Gamma(\mathtt{x})\mid\emptyset}$$
$$(\text{T-Var})$$

$$\frac{\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\ OK \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}} \qquad \overline{\mathit{ftype}(f, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \mathtt{U}} \qquad \Delta;\ \Gamma \vdash \overline{\mathtt{e}\ :\ \mathtt{U}\mid\emptyset}}{\Delta;\ \Gamma \vdash \mathtt{new}\ \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}(\overline{\mathtt{e}})\ :\ \exists\emptyset.\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\mid\emptyset}$$
$$(\text{T-New})$$

$$\frac{\Delta;\ \Gamma \vdash \mathtt{e}\ :\ \exists\Delta'.\mathtt{N}\mid\emptyset \qquad \mathit{ftype}(\mathtt{f};\mathtt{N}) = \mathtt{T}}{\Delta;\ \Gamma \vdash \mathtt{e.f}\ :\ \mathtt{T}\mid\Delta'}$$
$$(\text{T-Field})$$

$$\frac{\Delta;\ \Gamma \vdash \mathtt{e}\ :\ \mathtt{U}\mid\Delta' \qquad \Delta,\Delta' \vdash \mathtt{U} <: \mathtt{T} \qquad \Delta \vdash \Delta'\ OK \qquad \Delta \vdash \mathtt{T}\ OK}{\Delta;\ \Gamma \vdash \mathtt{e}\ :\ \mathtt{T}\mid\emptyset}$$
$$(\text{T-Subs})$$

$$\frac{
\begin{array}{c}
\Delta;\ \Gamma \vdash \mathtt{e}\ :\ \exists\Delta'.\mathtt{N}\mid\emptyset \qquad \mathit{mtype}(\mathtt{m};\mathtt{N}) = \texttt{<}\overline{\mathtt{Y} \to [\mathtt{B}_L\texttt{-}\mathtt{B}_U]}\texttt{>}\ (\overline{\mathtt{U}}) \to \mathtt{U} \\
\Delta \vdash \overline{\mathtt{P}}\ OK \qquad \Delta;\ \Gamma \vdash \overline{\mathtt{e}\ :\ \exists\Delta.\mathtt{R}\mid\emptyset} \\
\mathit{sift}(\overline{\mathtt{R}};\overline{\mathtt{U}};\overline{\mathtt{Y}}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'}) \qquad \mathit{match}(\overline{\mathtt{R}'};\overline{\mathtt{U}'};\overline{\mathtt{P}};\overline{\mathtt{Y}};\overline{\mathtt{T}}) \\
\Delta'' = \Delta,\Delta',\overline{\Delta} \qquad \Delta'' \vdash \overline{\exists\emptyset.\mathtt{R} <: [\overline{\mathtt{T}/\mathtt{Y}}]\mathtt{U}} \\
\Delta'' \vdash \overline{[\overline{\mathtt{T}/\mathtt{Y}}]\mathtt{B}_L <: \mathtt{T}} \qquad \Delta'' \vdash \overline{\mathtt{T} <: [\overline{\mathtt{T}/\mathtt{Y}}]\mathtt{B}_U}
\end{array}
}{\Delta;\ \Gamma \vdash \mathtt{e.}\texttt{<}\overline{\mathtt{P}}\texttt{>}\mathtt{m}(\overline{\mathtt{e}})\ :\ [\overline{\mathtt{T}/\mathtt{Y}}]\mathtt{U}\mid\Delta',\overline{\Delta}}$$
$$(\text{T-Invk})$$

Match:

$$\frac{
\begin{array}{c}
\forall j,\ \mathtt{P}_j = \star \implies \mathtt{Y}_j \in \mathit{fv}(\overline{\mathtt{R}'}) \qquad \forall i,\ \mathtt{P}_i \neq \star \implies \mathtt{T}_i = \mathtt{P}_i \\
\emptyset \vdash \overline{\mathtt{R} \prec: [\overline{\mathtt{T}/\mathtt{Y}}, \overline{\mathtt{T}'/\mathtt{X}}]\mathtt{R}'} \\
\mathit{dom}(\overline{\Delta}) = \overline{\mathtt{X}} \qquad \mathit{fv}(\overline{\mathtt{T}},\overline{\mathtt{T}'}) \cap \overline{\mathtt{Y}},\overline{\mathtt{X}} = \emptyset
\end{array}
}{\mathit{match}(\overline{\mathtt{R}};\overline{\exists\overline{\Delta}.\mathtt{R}'};\overline{\mathtt{P}};\overline{\mathtt{Y}};\overline{\mathtt{T}})}$$
$$(\text{Match})$$

Sift: $\mathit{sift}(\overline{\mathtt{R}};\overline{\mathtt{U}};\overline{\mathtt{Y}}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'})$

$$\frac{}{\mathit{sift}(\emptyset;\emptyset;\overline{\mathtt{Y}}) = (\emptyset;\emptyset)}$$
$$(\text{Sift-Empty})$$

$$\frac{\overline{\mathtt{Y}} \cap \mathit{fv}(\mathtt{U}) = \overline{\mathtt{X}} \qquad \mathit{var}(\overline{\mathtt{X}};\mathtt{U}) = \overline{o} \qquad \mathit{sift}(\overline{\mathtt{R}};\overline{\mathtt{U}};\overline{\mathtt{Y}}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'})}{\mathit{sift}((\mathtt{R},\overline{\mathtt{R}});(\mathtt{U},\overline{\mathtt{U}});\overline{\mathtt{Y}}) = ((\mathtt{R},\overline{\mathtt{R}'});(\mathtt{U},\overline{\mathtt{U}'}))}$$
$$(\text{Sift-Add})$$

$$\frac{\overline{\mathtt{Y}} \cap \mathit{fv}(\mathtt{U}) = \overline{\mathtt{X}} \qquad \mathit{var}(\mathtt{X}_j;\mathtt{U}) \neq o,\ \text{for some}\ \mathtt{X}_j \in \overline{\mathtt{X}} \qquad \mathit{sift}(\overline{\mathtt{R}};\overline{\mathtt{U}};\overline{\mathtt{Y}}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'})}{\mathit{sift}((\mathtt{R},\overline{\mathtt{R}});(\mathtt{U},\overline{\mathtt{U}});\overline{\mathtt{Y}}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'})}$$
$$(\text{Sift-Skip})$$

Figure 6.7. Expression Typing and Auxiliary Functions For Wildcard Capture

71

ing derivation below illustrates this process on typing the "`box.elem`" expression from the previous code segment, where we assume $\Gamma = \texttt{box} : \exists \texttt{X} \to \texttt{[}\bot\texttt{-String].Box<X>}$.

$$
\frac{\begin{array}{c} \dfrac{\emptyset; \ \Gamma \vdash \texttt{box} \ : \ \exists \texttt{X} \to \texttt{[}\bot\texttt{-String].Box<X>} \mid \emptyset \qquad \textit{ftype}(\texttt{elem}; \texttt{Box<X>}) = \texttt{X}}{\emptyset; \ \Gamma \vdash \texttt{box.elem} \ : \ \texttt{X} \mid \texttt{X} \to \texttt{[}\bot\texttt{-String]}} \ \text{(T-Field)} \qquad \begin{array}{c} \emptyset, \texttt{X} \to \texttt{[}\bot\texttt{-String]} \vdash \texttt{X} <: \texttt{String} \\[4pt] \emptyset \vdash \texttt{X} \to \texttt{[}\bot\texttt{-String]} \ \mathit{OK} \\[4pt] \emptyset \vdash \texttt{String} \ \mathit{OK} \end{array} \end{array}}{\emptyset; \ \Gamma \vdash \texttt{box.elem} \ : \ \texttt{String} \mid \emptyset} \ \text{(T-Subs)}
$$

$$(6.9)$$

6.5.2 Matching for Wildcard Capture

The T-Invk rule typechecks a method invocation and uses *match* to perform wild-card capture. The definition of *match* is updated to use the definition-site subtyping relation ($\prec:$). Ignoring return types, consider a polymorphic method declared with type `<`$\overline{\texttt{Y}}$`>m(`$\overline{\texttt{U}}$`)` and called with types `<`$\overline{\texttt{P}}$`>m(`$\overline{\exists\Delta.\texttt{R}}$`)`. The *match* function is used to infer actual type arguments for method invocations using the actual value arguments. The parameters of $match(\overline{\texttt{R}}; \overline{\texttt{U}}; \overline{\texttt{P}}; \overline{\texttt{Y}}; \overline{\texttt{T}})$ and their expected conditions are:

1. The bodies of the actual value argument types of a method invocation ($\overline{\texttt{R}}$).

2. The formal value argument types of a method ($\overline{\texttt{U}}$).

3. The *specified* type actuals of a method invocation ($\overline{\texttt{P}}$).

4. The formal type arguments of a method ($\overline{\texttt{Y}}$).

5. The *inferred* type actuals of a method invocation ($\overline{\texttt{T}}$).

We briefly explain an example typing derivation where wildcard capture is performed. More detailed coverage is given in [12, Section 3.4]. We type the method invocation "`uselessExchange(list)`" from the code example in Figure 5.1 in Section 5.2. To explain the type derivation using the VarJ calculus, we translate syntactic elements from the code example to their corresponding syntax in VarJ. For example, the local variable declaration "`List<?> list;`" translates to "$\exists$Y.`List<Y> list;`" Method invocation "`uselessExchange(list)`" translates to "`this.<*>uselessExchange(list)`". The marker $\star$ in the position of an actual type argument signals that an actual type argument should be inferred at that position. The inferred type argument will be passed to the formal type argument `X` of method `uselessExchange`. To derive the type of "`this.<*>uselessExchange(list)`", we list the following judgments that rule T-INVK will be applied to. First, let Γ be the expression variable context for the typing of that expression. For example, the domain of Γ includes the `this` reference, so $\Gamma(\text{this}) = \exists\emptyset.\text{Client<>}$. The other map entries in Γ can easily be determined by inspecting the code example. In the judgments below, we highlight the existentially quantified type variable when it occurs free in a type expression.

1. $\emptyset; \Gamma \vdash \text{this} : \exists\emptyset.\text{Client<>} \mid \emptyset$, since $\Gamma(\text{this}) = \exists\emptyset.\text{Client<>}$.

2. $mtype(\text{uselessExchange}; \text{Client<>}) = \text{<X>} (\exists\emptyset.\text{List<X>}) \rightarrow \text{void}$.

3. $\emptyset \vdash \star\ OK$, by W-I in Figure 6.5.

4. $\emptyset; \Gamma \vdash \text{list} : \exists\text{Y}.\text{List<Y>} \mid \emptyset$, since $\Gamma(\text{list}) = \exists\text{Y}.\text{List<Y>}$.

5. $sift(\text{List<}\ \text{Y}\ \text{>}; \exists\emptyset.\text{List<X>}; \text{X}) = (\text{List<}\ \text{Y}\ \text{>}; \exists\emptyset.\text{List<X>})$. The *sift* function is used to filter formal method value argument types that should *not* be instantiated with inferred actual argument types. No such type is in this example. *sift* is explained further in Section 6.5.3.

6. $match(\texttt{List<}\boxed{\texttt{Y}}\texttt{>}; \exists\emptyset.\texttt{List<X>}; \star; \texttt{X}; \boxed{\texttt{Y}})$. This holds because there exists a type $\texttt{T}$ such that replacing the method's formal type argument $\texttt{X}$ with $\texttt{T}$ in the *body* of the formal value argument type $\texttt{List<X>}$ makes it a supertype of the body of the actual value argument type $\texttt{List<Y>}$. In this case, $\texttt{T} = \texttt{Y}$ and $\texttt{List<Y>} \prec:$ $[\texttt{Y}/\texttt{X}] \texttt{List<X>} = \texttt{List<Y>}$.

7. $\emptyset \vdash \exists\emptyset.\texttt{List<}\boxed{\texttt{Y}}\texttt{>} <: [\boxed{\texttt{Y}}/\texttt{X}]\exists\emptyset.\texttt{List<X>} = \exists\emptyset.\texttt{List<}\boxed{\texttt{Y}}\texttt{>}$

Applying rule T-INVK to the judgments listed above gives the following:

$$\emptyset; \; \Gamma \vdash \texttt{this.<}\star\texttt{>uselessExchange(list)} \; : \; \texttt{void} \; \big| \; \boxed{\texttt{Y}} \qquad (6.10)$$

The existentially quantified type variable, $\texttt{Y}$, in the type of $\texttt{list}$ escapes to the guard of typing judgment 6.10 in case $\texttt{Y}$ occurred free in the type of the method invocation. Rule T-SUBS can be applied in similar fashion to that in typing derivation 6.9 to type the method invocation with an empty guard. Typing with an empty guard ensures that no existential type variable occurs free in the type of an expression.

Figure 6.8 contains the reduction rules for performing runtime evaluation. The R-INVK rule also uses *match* to compute inferred type actuals because some of the specified type actuals ($\overline{\texttt{P}}$) may be the type inference marker $\star$. Since each occurrence of the $\star$ marker may refer to different inferred types, *match* is needed to compute the concrete types to substitute for the formal type arguments' ($\overline{\texttt{Y}}$) occurrences in the method body.

6.5.3 Sifting for Wildcard Capture

To preserve the type of a method invocation during execution, we want inferred type arguments to remain the same throughout execution. Type arguments are inferred from actual value arguments. Recall that the dynamic type of a value argument may be a more specific subtype of the argument's static type. Thus, the dynamic

Computation Rules: $e \mapsto e$

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{f}}}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{v}}\texttt{).f}_i \mapsto \texttt{v}_i}$$

$$\left(\text{R-Field}\right)$$

$$\texttt{v = new N(}\overline{\texttt{v}'}\texttt{)} \qquad \overline{\texttt{v = new N(}\overline{\texttt{v}''}\texttt{)}} \qquad \mathit{mbody}(\texttt{m}; \texttt{N}) = \langle \overline{\texttt{x}}.\texttt{e}_0 \rangle$$
$$\mathit{mtype}(\texttt{m}; \texttt{N}) = \texttt{<}\overline{\texttt{Y} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{> (}\overline{\texttt{U}}\texttt{)} \to \texttt{U}$$
$$\frac{\mathit{sift}(\overline{\texttt{N}}; \overline{\texttt{U}}; \overline{\texttt{Y}}) = (\overline{\texttt{N}'}; \overline{\texttt{U}'}) \qquad \mathit{match}(\overline{\texttt{N}'}; \overline{\texttt{U}'}; \overline{\texttt{P}}; \overline{\texttt{Y}}; \overline{\texttt{T}})}{\texttt{v.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{v}}\texttt{)} \mapsto [\overline{\texttt{v}/\texttt{x}}, \texttt{v}/\texttt{this}, \overline{\texttt{T}/\texttt{Y}}]\texttt{e}_0}$$

$$\left(\text{R-Invk}\right)$$

Congruence Rules: $e \mapsto e$

$$\frac{\texttt{e} \mapsto \texttt{e}'}{\texttt{e.f} \mapsto \texttt{e}'\texttt{.f}} \qquad\qquad \frac{\texttt{e}_i \mapsto \texttt{e}'_i}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(..e}_i\texttt{..)} \mapsto \texttt{new C<}\overline{\texttt{T}}\texttt{>(..e}'_i\texttt{..)}}$$
$$\left(\text{RC-Field}\right) \qquad\qquad\qquad \left(\text{RC-New-Arg}\right)$$

$$\frac{\texttt{e} \mapsto \texttt{e}'}{\texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \mapsto \texttt{e}'\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}} \qquad \frac{\texttt{e}_i \mapsto \texttt{e}'_i}{\texttt{e.<}\overline{\texttt{P}}\texttt{>m(..e}_i\texttt{..)} \mapsto \texttt{e.<}\overline{\texttt{P}}\texttt{>m(..e}'_i\texttt{..)}}$$
$$\left(\text{RC-Inv-Recv}\right) \qquad\qquad\qquad \left(\text{RC-Inv-Arg}\right)$$

Figure 6.8. Reduction Rules

type of an argument may cause a different type to be inferred than using the static type of an argument. This occurs when a formal type argument occurs in a position that allows variant subtyping. As a result, not all formal type argument are allowed to be bound to inferred type arguments. In particular, if the variance of a formal type argument Y is greater than invariance in the type of a method's formal value argument, then Y cannot be bound to an inferred type argument.

The *sift* function is used in VarJ and TameFJ to filter inputs passed to *match* (in the T-INVK and R-INVK rules). The goal of *sift* is to only allow type inference for formal type arguments that occur at most invariantly in the types of formal value arguments "fixed" or invariant positions. As a result, inferred type arguments for a method invocation do not change throughout execution. Without applying *sift*, counter examples to the subject reduction (type preservation) theorem can result. First, note that the following two judgments are derivable.

1. $match(\texttt{Dog}; \exists\emptyset.\texttt{Y}; \star; \texttt{Y}; \texttt{Dog})$ because $\texttt{Dog} \prec: [\texttt{Dog}/\texttt{Y}]\texttt{Y} = \texttt{Dog}$. The other premises of *match* are easy to verify.

2. $match(\texttt{Dog}; \exists\emptyset.\texttt{Y}; \star; \texttt{Y}; \texttt{Animal})$ because $\texttt{Dog} \prec: [\texttt{Animal}/\texttt{Y}]\texttt{Y} = \texttt{Animal}$.

Assume List is invariant and consider the following method definition and evaluation step of a method invocation

```
<X> List<X> createList(X arg) { return new List<X>(); }
```

$$\texttt{createList<}\star\texttt{>(new Dog()) : List<Animal>}$$
$$\mapsto \texttt{new List<Dog>() : List<Dog>}$$

The expression `createList<`$\star$`>(new Dog())` can be typed with `List<Animal>` because the actual value argument `new Dog()` has type `Animal`. Hence, the inferred type

actual used for typing the expression can be `Animal`, which implies that the type of `createList<*>(new Dog())` is `List<Animal>`. However, the inferred type used for typing the method invocation is not required to be the same inferred type, computed in the R-INVK rule, that is substituted into the method body. Without *sift*, the above evaluation step is possible, which contradicts the subject reduction theorem, since, by the invariance of `List`, `new List<Dog>()` cannot be typed with `List<Animal>`.

Specifically, we want *match* to be a function over the filtered inputs from *sift* and the following property, similar to lemma 37 from [12]. This lemma states that the ability to perform wildcard capture is preserved as actual value arguments are evaluated to concrete values.

Lemma 6 (Subtyping Preserves *match*ing (arguments)). If (a) $\Delta \vdash \overline{\exists\Delta_1.\mathtt{R}_1 \sqsubseteq: \exists\Delta_2.\mathtt{R}_2}$ and (b) $match(sift(\overline{\mathtt{R}_2};\overline{\mathtt{U}};\overline{\mathtt{Y}});\overline{\mathtt{P}};\overline{\mathtt{Y}};\overline{\mathtt{T}})$ and (c) $\overline{\Delta_2} = \overline{\mathtt{Z} \to [\mathtt{B}_L\texttt{-}\mathtt{B}_U]}$ and (d) $fv(\overline{\mathtt{U}}) \cap \overline{\mathtt{Z}} = \emptyset$ and (e) $\emptyset \vdash \Delta\ OK$ and (f) $\Delta \vdash \overline{\exists\Delta_1.\mathtt{R}_1}\ OK$ and (g) $\Delta \vdash \overline{\mathtt{P}}\ OK$, then there exists $\overline{\mathtt{U}'}$ such that: (1) $match(sift(\overline{\mathtt{R}_1};\overline{\mathtt{U}};\overline{\mathtt{Y}});\overline{\mathtt{P}};\overline{\mathtt{Y}};\overline{[\mathtt{U}'/\mathtt{Z}]\mathtt{T}})$ and (2) $\Delta, \Delta' \vdash \overline{[\mathtt{U}'/\mathtt{Z}]\mathtt{B}_L <: \mathtt{U}'}$ and (3) $\Delta, \Delta' \vdash \overline{\mathtt{U}' <: [\mathtt{U}'/\mathtt{Z}]\mathtt{B}_U}$ and (4) $\overline{\mathtt{R}_1 \prec: [\mathtt{U}'/\mathtt{Z}]\mathtt{R}_2}$ (5) $fv(\overline{\mathtt{U}'}) \subseteq dom(\Delta, \Delta')$.

In TameFJ, *sift* filters out a pair of a type actual body $\mathtt{R}$ and a formal type $\mathtt{U}$, if $\mathtt{U} = \exists\emptyset.\mathtt{X}$ and $\mathtt{X}$ is one of the formal type arguments ($\overline{\mathtt{Y}}$). Due to *sift*, the two *match* judgments above could never be derived in TameFJ. Moreover, TameFJ allows an existential type variable to be passed as parameter for a formal type variable argument only if the formal type variable is used as a type parameter. Since every type constructor in TameFJ is assumed to be invariant, every type variable used for inference is in an invariant position. This no longer holds in VarJ with variant type constructors. If we assume `Iterator` is covariant, a counter example similar to the previous one can be produced with the following method:

```
<X> List<X> createList2(Iterator<X> arg) { return new List<X>(); }
```

Hence, we update the definition of *sift* to use *var* to check if a method type parameter occurs at most invariantly. This restriction ensures that instantiations of the

type parameter from type inference do not vary during execution. We find that prohibiting wildcard capture in variant positions is not practically restrictive. A wildcard type for a variant type typically has an equivalent non-wildcard type. `Iterator<?>` is equivalent to `Iterator<Object>` by covariance of `Iterator`. `BiGeneric<?>` is equivalent to `BiGeneric<T>`, for any `T`, if `BiGeneric` is bivariant. In such cases, the need for wildcard capture is eliminated because the required type actuals to specify in a method call can be named and written by the programmer. The VarJ grammar does not allow the bottom type $\perp$ to be specified as a type actual. However, we have not found any practical need for wildcard capture with contravariant types.

6.6 Type Soundness

We prove type soundness for VarJ by proving the progress and subject reduction theorems below. As in TameFJ, a non-empty guard is required in the statement of the progress theorem when applying the inductive hypothesis in the proof for the case when the T-Subs rule is applied.

Theorem 1 (Progress). For any $\Delta, \mathtt{e}, \mathtt{T}$, if $\emptyset; \emptyset \vdash \mathtt{e} \; : \; \mathtt{T} \mid \Delta$, then either $\mathtt{e} \mapsto \mathtt{e}'$ or there exists a $\mathtt{v}$ such that $\mathtt{e} = \mathtt{v}$.

Theorem 2 (Subject Reduction). For any $\mathtt{e}, \mathtt{e}', \mathtt{T}$, if $\emptyset; \emptyset \vdash \mathtt{e} \; : \; \mathtt{T} \mid \emptyset$ and $\mathtt{e} \mapsto \mathtt{e}'$, then $\emptyset; \emptyset \vdash \mathtt{e}' \; : \; \mathtt{T} \mid \emptyset$.

The key difficulty in proving these theorems can be captured by a small number of key lemmas whose proofs are substantially affected by variance reasoning. Lemma 7 is probably the main one, which relates subtyping and wildcard capture, and is similar to lemma 36 from [12]. It states that the method receiver's ability to perform wildcard capture is preserved in subtypes with respect to the method receiver. It shows that the subsumption principle holds even under interaction with wildcard capture.

Lemma 7 (Subtyping Preserves *match*ing (receiver)). If (a) $\Delta \vdash \exists \Delta_1.N_1 \sqsubseteq: \exists \Delta_2.N_2$ and (b) $mtype(m; N_2) = \langle \overline{Y_2 \rightarrow [B_{2L}-B_{2U}]} \rangle (\overline{U_2}) \rightarrow U_2$ and

(c) $mtype(m; N_1) = \langle \overline{Y_1 \rightarrow [B_{1L}-B_{1U}]} \rangle (\overline{U_1}) \rightarrow U_1$ and (d) $sift(\overline{R}; \overline{U_2}; \overline{Y_2}) = (\overline{R'}; \overline{U_2'})$ and (e)

$match(\overline{R'}; \overline{U_2'}; \overline{P}; \overline{Y_2}; \overline{T})$ and (f) $\emptyset \vdash \Delta$ *OK* and (g) $\Delta, \Delta' \vdash \overline{T}$ *OK* then: (1) $sift(\overline{R}; \overline{U_1}; \overline{Y_1}) =$

$(\overline{R'}; \overline{U_1'})$ and (2) $match(\overline{R'}; \overline{U_1'}; \overline{P}; \overline{Y_1}; \overline{T})$.

6.7 Discussion

This section discusses important practical issues for supporting definition-site variance in Java. Section 6.7.1 discusses a type bound analysis Armed with the VarJ calculus, it also revisits the issue related to F-bounded polymorphism that was discussed in Section 5.3. Section 6.7.2 discusses a soundness issue that occurs in any language that supports definition-site variance and that is compiled with an erasure-based translation.

6.7.1 Boundary Analysis

Definition-site variance can imply that the variance of a type does not depend on all of the type bounds that occur in the type. Chapter 4 presented a definition of $var(X; U)$ that performed a simple *boundary analysis* to compute such irrelevant bounds. The variance returned by $var(X; U)$ might not depend on all type bounds that occurred in U. As discussed in Section 3.2, if generic C<Y> is covariant wrt to Y, then the lower bound of a use-site variant instantiation is ignored, which is sound for the VarLang calculus:

$$var(X; C\text{<-T>}) = (+ \sqcup -) \otimes var(X; T) = * \otimes var(X; T) = *$$

Hence, $var(X; C\text{<-T>}) = *$, even if X occurred in the lower bound, T.

This simple boundary analysis is not safe for languages that can declare F-bounded type variables *within* type expressions. F-bounded type variables cannot be declared

within type expressions in VarLang or in Java. In Java, F-bounded class type parameters can be declared in type definitions but not in type expressions. Hence, we can analyze the variance of F-bounds in Java type definitions to safely constrain definition-site variances. Our early work [1] conservatively assumed upper bounds of class type parameters are in invariant positions. Scala assumes upper bounds of class type parameters are covariant positions even if they are an F-bound [51, Section 4.5]. This make sense since the range of class type variables (between lower and upper bounds) should be narrower in the subtype. As explained in a footnote in Section 6.3, bounds on class type parameters do not constrain definition-site variance in VarJ.

F-Bounds in Existential Types

The ability to ignore type bounds is present in a disciplined way in our VarJ formalism, although there is no explicit variance joining mechanism in the definition of *var*. For example, if `Iterator` is covariant in its type parameter, we can infer the following, where the notation $T \equiv U$ denotes "$T <: U \wedge U <: T$":

$$\exists X \rightarrow \text{[Dog-Animal].Iterator<X>} \equiv \exists X \rightarrow \text{[}\bot\text{-Animal].Iterator<X>}$$

Clearly, $\exists X \rightarrow \text{[Dog-Animal].Iterator<X>} <: \exists X \rightarrow \text{[}\bot\text{-Animal].Iterator<X>}$ is derivable using rule SE-PACK because the range of the existential type variable is wider in the supertype. However, it is not always safe nor derivable to *narrow* the range of a type variable in the supertype. In the inverse relationship, the range of `X` is squeezed from `[⊥-Animal]` to `[Dog-Animal]` in the supertype:

$$\exists X \rightarrow \text{[}\bot\text{-Animal].Iterator<X>} <: \exists X \rightarrow \text{[Dog-Animal].Iterator<X>} \qquad (6.11)$$

This relationship is derivable by applying a combination of the SE-SD, SE-PACK, and ST-* rules:

$\exists$X $\to$ [$\perp$-Animal].Iterator<X>

<: $\exists$X $\to$ [$\perp$-Animal].Iterator<Animal>

   (by covariance of Iterator and X $\to$ [$\perp$-Animal] $\vdash$ X <: Animal)

<: $\exists\emptyset$.Iterator<Animal>   (since X does not occur in body Iterator<Animal>)

<: $\exists$X $\to$ [Dog-Animal].Iterator<X>   (packed argument Animal to an existential)

As we saw in Section 5.3, narrowing the range of an existential type variable in a supertype is not sound in the presence of F-bounded polymorphism. It is important to realize that this issue is not limited to use of recursive bounds of class type parameters.[5] The counterexample in Section 5.3 used interface Trouble<P extends List<P>> extends Iterator<P> {}. However, even if we restrict our attention to a plain Iterator (or, equivalently, if the class type constraint, extends List<P>, of Trouble is removed) it is still *not* safe to assume the following subtype relation, by reasoning similar to that used in Section 5.3:

$\exists$X $\to$ [YourList-List<X>].Iterator<X> <: $\exists$X $\to$ [MyList-List<X>].Iterator<X>

More generally, the above subtype relationship would violate the subsumption principle. An instance of the latter type can return a $\exists$X $\to$ [MyList-List<X>].List<X> from its next method, but the former type cannot because by the invariance of List,

$\exists$X $\to$ [YourList-List<X>].List<X> $\not<:$ $\exists$X $\to$ [MyList-List<X>].List<X>

---

[5]In our earlier work [1], when we inferred definition-site variance of Java type parameters, it sufficed to be overly conservative at this point: the mere appearance of a class type variable in an upper bound of any type parameter caused us to consider the definition as invariant relative to that class type variable. If a class type parameter occurs in the upper bound of a method type parameter, we no longer restrict the definition-site variance to invariance. Such an occurrence is not a recursive bound on a class type parameter because class and method type parameters are distinct. Upper bounds of method type parameters are in contravariant positions, as explained in Section 6.3.

In contrast to subtype relationship (6.11), above, VarJ does not support the above erroneous subtyping because it cannot establish that the upper bounds of the two instantiations of `Iterator` are related. In particular, we cannot derive that $\exists X \rightarrow$ `[YourList-List<X>].List<X>` is a subtype of some non-existential type, $\exists \emptyset$.`List<T>`, which is, in turn, a subtype of $\exists X \rightarrow$ `[MyList-List<X>].List<X>`.

Contrasting the two examples shows that boundary analysis is complex and can be unintuitive to the programmer. Note, however, that the VarJ calculus merely tells us what is possible to infer correctly. A practical implementation may choose not to perform all possible inferences. A specific scenario is that of separating boundary analysis from type checking. Useless bounds can be "removed" during a preprocessing step performed *before* type checking. This is analogous to general type inference algorithms relative to type checking algorithms: type checking can be performed independently of the type inference performed to compute type annotations that were skipped by programmers [32]. Similarly, our variance-based type checking can be performed independently of the "useless boundary analysis". For example, a boundary preprocessing step could transform input type $\exists X \rightarrow$ `[Dog-Animal].Iterator<X>` to the equivalent type $\exists X \rightarrow$ `[⊥-Animal].Iterator<X>`. This opens the door to many practical instantiations—e.g., an optimistic but possibly unsound bound inference inside an IDE (which interacts with the user, offering immediate feedback and suggesting relaxations of expressions that the user types in) combined with a simple but sound checking inside the compiler.

6.7.2 Definition-Site Variance and Erasure

A practical issue with definition-site variance concerns its use with *erasure* [28, Section 4.6]. In an erasure-based translation, a compiler does not preserve type argu-

ments in byte code. The type `List<String>` is compiled to the type `List`, for example.[6]

Unfortunately, for any language that supports definition-site variance, supports cast expressions, and is compiled using an erasure-based translation, *downcasts* [28, Section 5.1.6] that should fail at runtime instead succeed. A cast expression "`(T)` e" is a downcast if the static type of e is a supertype of `T`. The static type of expression (`(T)` e) is `T`. A cast should "fail" at runtime if the dynamic type of e is not a subtype of `T`. A `ClassCastException` is thrown when a cast expression fails, in Java.

The following code example demonstrates the issue with the combination of casts, definition-site variance, and erasure. Class `A` is covariant and only supports reading data using method `get`. Class `B` is invariant, extends class `A`, and adds the ability to write data using method `set`.

```
class A<+X> {
  private X elem;
  A(X elem) { this.elem = elem; }
  public X get() { return elem; }
}
class B<oX> extends A<X> {
  B(X elem) { super(elem); }
  void set(X elem) { this.elem = elem; }
}
void main() {
  A<Integer> a = new B<Integer>(8);
  A<Object> a2 = a;   // fine by covariance of A
  B<Object> b = (B<Object>) a2; // downcast succeeds with erasure
  b.set("string");
  Integer i = a.get(); // error, a.get() returns a String
}
```

In a language with an expansion-based translation, such as C#, type parameters are preserved in byte code. As a result, the downcast will fail dynamically: an object with dynamic type `B<Integer>` cannot be cast to a `B<Object>`, by the invariance of class `B`.

---

[6]A type such as `List` that uses a generic type without supplying actual type arguments is known as a raw type [28, Section 2.8].

In an erasure-based translation, however, the cast cannot check the type parameter (which has been erased) and will therefore succeed, causing errors further down the road. Specifically, a runtime type error could result in a *non-cast* expression. This violates type soundness. This property requires that runtime type errors only occur in cast expressions. In this code example, however, the runtime error occurs when executing the last line, which does not contain a cast.

This practical consideration affects all type systems that combine definition-site variance, casts, and erasure. For instance, Scala already handles such cases with a static type warning. Effectively, no cast to a subtype with tighter variance is safe. This result is somewhat counter-intuitive because it defies common patterns for safe casting. For instance, the downcast above could have been performed after an "`a2 instanceof B<Object>`" check [28, Section 15.20.2] to establish that `a2` is indeed of type `B<Object>`. In this case the programmer would expect that the cast warning can be ignored, which is not the case. In practice, any deployment of the VarJ type system in an erasure-based setting would have to follow the same policy as Scala regarding cast warnings.

CHAPTER 7

VARIANCE SOUNDNESS

This section provides intuition as to when variant subtyping (subtyping between instantiations of a single type) is safe for program execution (does not result in runtime type errors). A rigorous proof of type soundness of VarJ is given in Appendix B. However, the proof of *variance soundness* is hidden in the vast amount of details. Variance soundness is the property that the variance analysis infers only subtype relationships that are type safe or cannot result in a runtime type error. This section highlights why our variance analysis infers only type safe subtype relationships for VarJ. Although variance soundness will be explained using VarJ, general variance concepts that are independent of a particular language *supporting execution* will be presented to highlight fundamental issues for verifying type soundness with variance.

VarLang helped established language neutral concepts for reasoning about variance. However, *VarLang* is not accompanied by an operational semantics. The proof of its soundness Theorem (3) does *not* show how to prove that particular variant subtyping rules do not cause runtime type errors. We will use VarJ to provide a template for proving the absence of runtime type errors with variant subtyping.

We investigate why type soundness of *VarJ* holds with variant subtyping (rules in Figure 6.6) and the variances assigned to positions in class definitions. Type soundness of VarJ is the conjunction of the preservation Theorem (2) and the progress Theorem (1). Satisfying these two properties implies that a runtime type error does not result during execution of *VarJ* programs. Hence, proving these theorems shows

that the variance analysis is safe in practice or for a programming language where programs written in the language can execute.

The subsumption property is required for type soundness because abstract terms or variables are replaced with concrete terms or values during execution. Although the static type of the variable may differ from the dynamic type of the value, the type checking rules ensure that the dynamic type is a subtype of the static type. If the subtype relation satifies the subsumption principle, the dynamic type of the variable can do everything the static type of the variable can do. Consider the following program written in VarJ with a couple of extra language features from Java (explicit constructors and `static` and `void` methods).

```
class Animal { void speak() { ... } }
class Dog extends Animal {
  void speak() { bark(); }
  void bark() { ... }
}
class Box<+X> {
 public final X elem;
 Box(X elem) { this.elem = elem; }
}
class Client {
 public static Animal unbox(Box<Animal> box) {
  return box.elem;
 }
 public static void main() {
   unbox(new Box<Dog>(new Dog())).speak();
 }
}
```

Because `Box` is declared to be covariant, `Box<Dog> <: Box<Animal>`. The static type of argument `box` in the body of method `unbox` is `Box<Animal>`. When executing the `main` method in class `Client`, the method invocation `unbox(new Box<Dog>(new Dog()))` reduces to `[new Box<Dog>(new Dog())/box]box.elem` = `new Box<Dog>(new Dog()).elem`. The dynamic type of `box` for that method invocation is `Box<Dog>`, so it needs to support the operations of `Box<Animal>` that are requested in the method body. In this case, the method body requires reading an `Animal` from the `elem` field

of a `Box<Animal>`. A `Box<Dog>` also supports this operation, so a runtime type error does not result.

The key properties for showing that the variance analysis only infers safe subtyping are illustrated at a high level in Figure 7.1. The left-hand side of the figure contains abstract descriptions of important variance properties that are circled. The right-hand side contains corresponding concrete properties that are specific to VarJ. The implication arrows represent lemmas that are needed for type soundness. They relate variance of a type to subtyping and then to subsumption or preserving operations in the subtype. The first arrow/lemma states that the variance of a type implies subtype relationships between instantiations of a type. For VarJ, this property is expressed as the Subtype Lifting Lemma 1. The second arrow/lemma states that every subtype relationship satisfies the subsumption principle. Together these two lemmas imply variance soundness; that is, subtyping relationships between two instantiations of a type allowed by variance also satisfy the subsumption principle. As a result, runtime type errors do not result from these additional subtype relations. The VarJ subsumption property in the figure expresses that a field's type becomes more specific for the subtype with variant subtyping.

The remainder of this chapter provides intuition on how variance soundness is achieved for VarJ without going into all of the details of VarJ's type soundness proof. Since intuition of the proof of the Subtype Lifting Lemma 1 (the first arrow) was already given in Section 6.2, we focus on how the subsumption principle is satisfied (the second arrow).

Proving Subsumption in VarJ

In VarJ, only three operations can be peformed with an object:

1. Reading the field of an object.

2. Invoking a method of an object.

Variance Soundness Lemmas in General	Variance Soundness Lemmas in VarJ

Variance of Types

$\mathtt{v} \leq var(\mathtt{X}; \mathtt{N})$
and
$\mathtt{v}(\mathtt{U}; \mathtt{T})$

$\Downarrow$ (Lemma 1)

Variant Subtyping

$[\mathtt{U/X}]\mathtt{N} <: [\mathtt{T/X}]\mathtt{N}$

$\Downarrow$ (Lemma 2)

Subsumption Principle

$ftype(\mathtt{f}; [\mathtt{T/X}]\mathtt{N}) = \mathtt{S}$
$\Longrightarrow$
$ftype(\mathtt{f}; [\mathtt{U/X}]\mathtt{N})$
$<:$
$ftype(\mathtt{f}; [\mathtt{T/X}]\mathtt{N})$
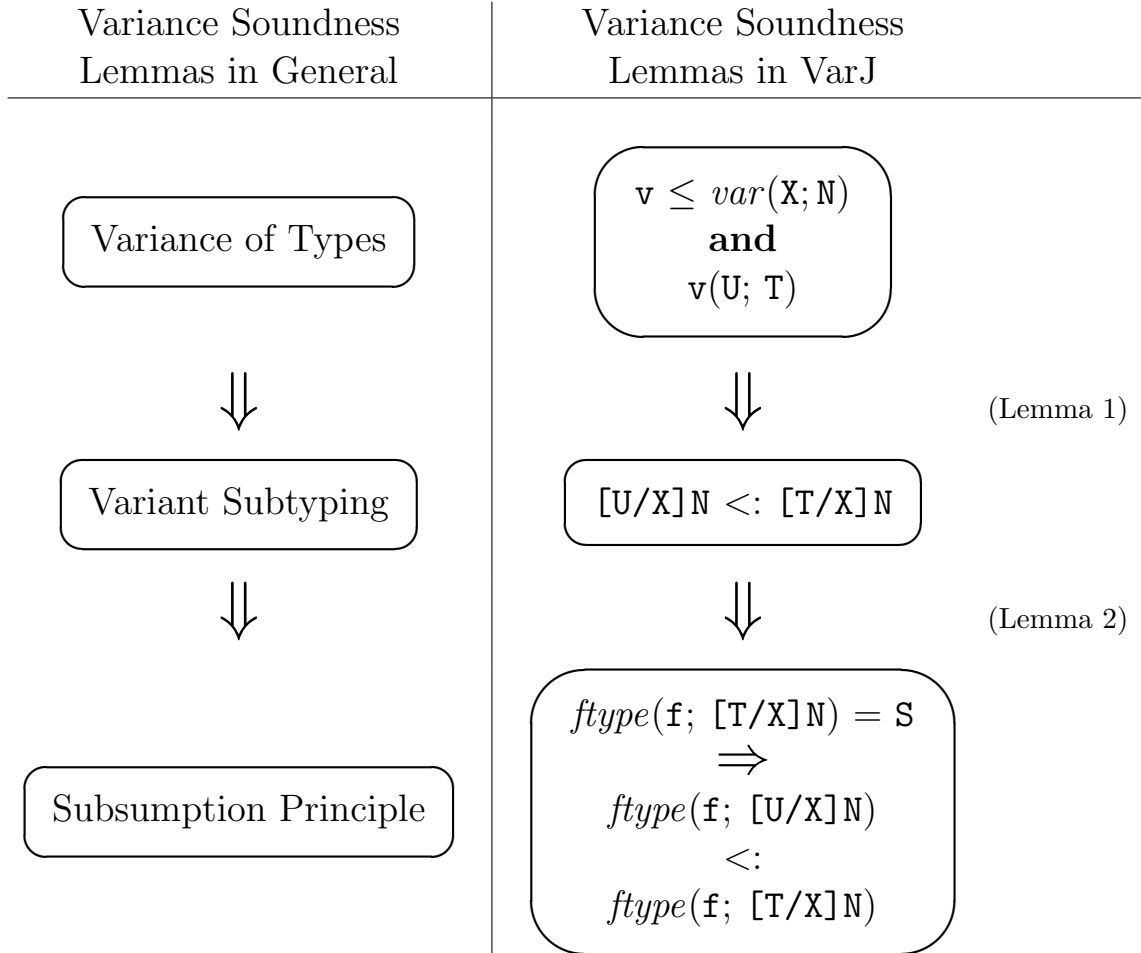
Figure 7.1. Key lemmas for proving variance analysis only infers type safe subtyping. Arrows denote implication. We skip some parameters in the subtyping judgments in this figure such as the type variable context Δ because the exact rules are not the focus of this chapter.

3. Passing an object "directly" as an argument in a call to a method or constructor. For example, variable `obj` is passed directly in method call "`m(obj)`" but not in method call "`m(obj.g())`".

These operations need to be preserved for the subtype in order to satisfy type soundness. The third operation is preserved because the subtype relation is transitive (rule ST-TRAN). For example, in method call "`m(obj)`" the static type of `obj` must be a subtype of the method's `m` formal argument type. Since `obj`'s dynamic type is a subtype of its static type, by transitivity of subtyping, the dynamic type is also a subtype of the formal argument type. Subtyping is still transitive with variant subtyping.

Preserving the first two operations with variance is far less trivial and depends on the variance analysis. In particular, we show how the variance analysis supports lemmas 2 and 3. These two lemmas specify that a subtype preserves the ability to read a field and invoke a method. Satisfying these lemmas depends on the subtype lifting lemma and the variances assigned to type positions in class definitions. Although detailed proofs of these lemmas can be found in Appendix B, we highlight the reasoning related to variance in the proof of Lemma 2. Similar reasoning applies in the proof of Lemma 3, so we skip its high-level proof.

VarJ does not support the ability to modify the values of fields. Section 7.2 also discusses how the variance reasoning would need to be updated if fields could be updated.

7.1 High-Level Proof of Lemma 2

Lemma 8 (Subtyping Specializes Field Type). If (a) $\vdash$ `class C<`$\overline{vX \rightarrow [\ldots]}$`>` $\lhd$ `N`... *OK* and (b) $\Delta \vdash$ `C<`$\overline{\mathtt{T}}$`>` $\prec:$ `N`$'$ and (c) *ftype*(`f`; `N`$'$) = `T`, then $\Delta \vdash$ *ftype*(`f`; `C<`$\overline{\mathtt{T}}$`>`) $<:$ `T`.

Before we embark on the proofs for the particular cases, note that premise (a) implies definition-site variances in all class definitions in the proof of this lemma type check. Also, to simplify the presentation, we only use one subtype relation symbol,

<:, to denote subtyping between types of any syntactic category, rather than the three relations from Section 6.4

This lemma is proved by structural induction on both the derivations of judgments (b) and (c).

Case:

$$\dfrac{VT(\texttt{C}) = \overline{v\texttt{X}} \qquad \Delta \vdash \overline{v(\texttt{T},\texttt{U})}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: \underbrace{\texttt{C<}\overline{\texttt{U}}\texttt{>}}_{N'}} \quad (\textsc{SD-Var})$$

$$\dfrac{\texttt{class C<}\overline{v\texttt{X}} \to \texttt{[...]>} \lhd \texttt{N} \{\ \overline{\texttt{S f};}\ \ \overline{M}\ \}}{\textit{ftype}(\texttt{f}_i; \texttt{C<}\overline{\texttt{U}}\texttt{>}) = [\overline{\texttt{U}/\texttt{X}}]\texttt{S}_i} \quad (\textsc{FT-Class})$$

Proof:

This proof case is the only base case of this inductive proof. This case does not require applying the inductive hypothesis. In this proof case, we use the fact that field types are in covariant positions. Since class $\texttt{C}$ type checks, we can establish the following relationship between the definition-site variance annotations of $\texttt{C}$ and the variances of the type parameters $\overline{\texttt{X}}$ in the field type $\texttt{S}_i$:

1. $\overline{\texttt{v} \leq \textit{var}(\texttt{X}; \texttt{S}_i)}$

Applying the subtype lifting lemma (Lemma 1) to (1) and the assumption $\Delta \vdash \overline{\texttt{v}(\texttt{T}, \texttt{U})}$ gives the following:

$$
\begin{array}{ccc}
\Delta \vdash & [\overline{\texttt{T}/\texttt{X}}]\texttt{S}_i & <: & [\overline{\texttt{U}/\texttt{X}}]\texttt{S}_i \\
& \| & & \| \\
& \textit{ftype}(\texttt{f}_i; \texttt{C<}\overline{\texttt{T}}\texttt{>}) & <: & \textit{ftype}(\texttt{f}_i; \texttt{C<}\overline{\texttt{U}}\texttt{>})
\end{array}
$$

□

Case:

$$\frac{VT(\texttt{C}) = \overline{\texttt{vX}} \qquad \Delta \vdash \overline{\texttt{v(T,U)}}}{\Delta \vdash \texttt{C<\overline{T}>} <: \underbrace{\texttt{C<\overline{U}>}}_{\texttt{N}'}}$$
$$(\text{SD-Var})$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{>} \lhd \texttt{N} \{ \overline{\texttt{S f;}} \ \overline{M} \} \qquad \texttt{f} \notin \overline{\texttt{f}}}{ftype(\texttt{f}; \texttt{C<\overline{U}>}) = ftype(\texttt{f}; \overline{[\texttt{U}/\texttt{X}]}\texttt{N})}$$
$$(\text{FT-Super})$$

Proof:

This proof case uses the fact that parent types are in covariant positions. Since class C type checks, we can establish the following relationship between the definition-site variance annotations of C and the variances of the type parameters $\overline{\texttt{X}}$ in the parent type N:

1. $\overline{\texttt{v} \leq var(\texttt{X}; \texttt{N})}$

Applying the subtype lifting lemma (Lemma 1) to (1) and the assumption $\Delta \vdash \overline{\texttt{v(T,U)}}$ gives:

2. $\Delta \vdash \overline{[\texttt{T}/\texttt{X}]}\texttt{N} <: \overline{[\texttt{U}/\texttt{X}]}\texttt{N}$

3. WLOG, assume $\texttt{N} = \texttt{D<\overline{V}>}$.

4. Class D type checks as explained in the beginning of this proof.

Applying the inductive hypothesis to (4), (2), and the assumption that $ftype(\texttt{f}; \overline{[\texttt{U}/\texttt{X}]}\texttt{N})$ is defined gives the following:

5. $\Delta \vdash ftype(\texttt{f}; \overline{[\texttt{T}/\texttt{X}]}\texttt{N}) <: ftype(\texttt{f}; \overline{[\texttt{U}/\texttt{X}]}\texttt{N})$

Since we assumed rule FT-Super applied, field f is inherited from the parent type N. Hence,

6. $ftype(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = ftype(\texttt{f}; [\overline{\texttt{T}/\texttt{X}}]\texttt{N})$

Therefore, we have the following:

$$\Delta \vdash ftype(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>})$$

$$= ftype(\texttt{f}; [\overline{\texttt{T}/\texttt{X}}]\texttt{N}) \qquad \text{by (6)}$$

$$<: ftype(\texttt{f}; [\overline{\texttt{U}/\texttt{X}}]\texttt{N}) \qquad \text{by (5)}$$

$$= ftype(\texttt{f}; \texttt{C<}\overline{\texttt{N}}\texttt{>}) \qquad \text{by assumption}$$

□

Case:

$$\frac{\texttt{class C<}\overline{\texttt{vX} \to [\ldots]}\texttt{> } \lhd \texttt{ N \{ }\overline{\texttt{S f;}}\texttt{ } \overline{M}\texttt{ \} } \qquad \texttt{C} \neq \texttt{D} \qquad \Delta \vdash \overline{[\overline{\texttt{T}/\texttt{X}}]\texttt{N}} <: \texttt{D<}\overline{\texttt{U}}\texttt{>}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{> } <: \underbrace{\texttt{D<}\overline{\texttt{U}}\texttt{>}}_{\texttt{N}'}}$$

$$(\textsc{SD-Super})$$

Proof:

This proof case does not depend on which rule derived premise (c) $ftype(\texttt{f}; \texttt{D<}\overline{\texttt{U}}\texttt{>}) =$ $\texttt{T}$. Since $\texttt{C} \neq \texttt{D}$, we know that field $\texttt{f}$ is not defined in class $\texttt{C}$ ($\texttt{f} \notin \overline{\texttt{f}}$).[1] The lemma holds for this case because of the inheritance subtyping. This can be seen by noting that the inductive hypothesis applies to the premise $\Delta \vdash \overline{[\overline{\texttt{T}/\texttt{X}}]\texttt{N}} <: \texttt{D<}\overline{\texttt{U}}\texttt{>}$. Furthermore, since $\texttt{f}$ is an inherited field, $ftype(\texttt{f}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = ftype(\texttt{f}; [\overline{\texttt{T}/\texttt{X}}]\texttt{N})$, by rule FT-Super.

---

[1]Although Java allows inherited field names to be redefined in subclasses [28, Section 15.11] with a new type, VarJ does not allow this for simplicity. For example, class definitions "`class A { String f; }`" and "`class B extends A { int f; }`" are allowed in Java. Contrasting with dynamic dispatch of methods, the declaration accessed by a field access expression is determined completely at compile time. [28, Section 15.11.2] presents the syntax used to access a field with the same name in the super class. Hence, that language feature is not related to subtyping or variance.

The proof for this case does not depend on variance reasoning. Since the purpose of this proof is to highlight variance reasoning, we direct the reader to the proof of this lemma in the appendix for further details. This lemma appears as Lemma 18 in Appendix B. $\square$

7.2   Supporting Field Writes

VarJ allows reading values from fields but not modifying the values of fields. This section describes how the variance reasoning in VarJ would be updated if field values could change. To explain the updates to the variance reasoning, we show how the statement and proof of Lemma 8 would be updated. We also state the additional constraint on definition-site variance annotations to satisfy the updated version of Lemma 8.

The only proof case of Lemma 8 that would need to change is the base case. In the other proof cases, applying the inductive hypothesis would allow us to derive the desired properties. In the base case, rules SD-VAR and FT-CLASS were applied to derive premises (b) and (c), respectively.

The base case shows how we get the desired subtype relationship between field types, $ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{T}}\texttt{>}) <: ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{U}}\texttt{>})$, when $\texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{C<}\overline{\texttt{U}}\texttt{>}$. This subtype relationship is desired to support the subsumption principle. In particular, this relationship shows that we can retrieve a value of type $ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{U}}\texttt{>})$ when reading field $\texttt{f}_i$ from an instance of the subtype $\texttt{C<}\overline{\texttt{T}}\texttt{>}$.

If field $\texttt{f}_i$ could be updated, then the inverse subtype relationship is also desired, when $\texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{C<}\overline{\texttt{U}}\texttt{>}$:

$$ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{U}}\texttt{>}) <: ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{T}}\texttt{>}) \tag{7.1}$$

In order to write a term of type $\texttt{S}$ to field $\texttt{f}_i$ of an object of type $\texttt{C<}\overline{\texttt{U}}\texttt{>}$ it must be the case that $\texttt{S} <: ftype(\texttt{f}_i; \texttt{C<}\overline{\texttt{U}}\texttt{>})$. If subtype relationship 7.1 also holds, then

$S <: ftype(f_i; C\texttt{<}\overline{U}\texttt{>}) <: ftype(f_i; C\texttt{<}\overline{T}\texttt{>})$. Hence, the ability to write an instance of type S to field f_i is supported by an instance of the subtype $C\texttt{<}\overline{T}\texttt{>}$, given relationship 7.1. As a result, the statement of Lemma 8 should be updated by adding the conclusion $\Delta \vdash ftype(f; N') <: ftype(f; C\texttt{<}\overline{T}\texttt{>})$.

Subtype relationship 7.1 holds if we assume field f_i's type is *also* in a contravariant position. That is, field f_i's type, S_i, is in both a covariant position and also in a contravariant position. To reflect in the type checking rules that field types are in contravariant positions, the following judgment would need to be added as a premise to rule W-CLS from Figure 6.3 in Section 6.3:

$$\overline{\neg vX} \vdash \overline{T} \; mono$$

Since S_i is in a contravariant position and class C type checks, we can establish the following relationship between the definition-site variance annotations of C and the variances of the type parameters $\overline{X}$ in the field type S_i:

1. $\overline{v \le - \otimes var(X; S_i)}$

Applying Lemma 15 from Appendix B to (1) and the assumption $\Delta \vdash \overline{v(T, U)}$ of rule SD-VAR gives us the desired subtype relationship:

$$
\begin{array}{ccc}
\Delta \vdash & \overline{[U/X]}S_i & <: & \overline{[T/X]}S_i \\[4pt]
& \parallel & & \parallel \\[4pt]
& ftype(f_i; C\texttt{<}\overline{U}\texttt{>}) & <: & ftype(f_i; C\texttt{<}\overline{T}\texttt{>})
\end{array}
$$

Since we have derived subtype relationship 7.1, we have showed that the ability to write values to a field is preserved in the subtype.

CHAPTER 8

AN APPLICATION: DEFINITION-SITE VARIANCE INFERENCE FOR JAVA

To showcase the potential of our unified treatment of use-site and definition-site variance, we implemented a mapping from Java to VarLang and used it to produce a (definition-site) variance *inference* algorithm. This software application applies the formal framework developed in previous chapters to reason about and infer definition-site variance. We evaluated the potential benefit of adding definition-site variance to Java. We analyzed six large Java libraries with generics (including the standard library). Findings from this experiment are given in this chapter.

8.1 Applications

Our mapping from Java to VarLang is straightforward: We produce a VarLang module definition for each Java class or interface, and all Java type expressions are mapped one-to-one on VarLang type expressions with the same name. The module definitions contain variance constraints that correspond to the well-understood variance of different positions (as discussed in Section 6.3): return types are a covariant position, argument types are a contravariant position, types of non-final fields are both covariant and contravariant positions, supertypes are a covariant position.

Our mapping is conservative: Although we handle the entire Java language, we may constrain definition-site variances to less-general variances than required to be safe. For instance, we ignore the potential for more general typing through reasoning about member visibility (i.e., `private`/`protected` access control). Member visibility,

in combination with conditions on self-reference in type signatures, can be used to establish that some fields or methods cannot be accessed from outside a class/package. Nevertheless, our mapping does not try to reason about such cases to produce less restrictive variance constraints. We prefer to infer unquestionably safe variances at the expense of slightly worse numbers (which still fully validate the potential of our approach).

We used this mapping to implement a definition-site variance inference algorithm. That is, we took regular Java code, written with no concept of definition-site variance in mind, and inferred how many generics are purely covariant/contravariant/bivariant. Inferring pure variance for a generic has several practical implications:

- One can use our algorithm to replace the Java type system with a more liberal one that infers definition-site variance and allows subtyping based on the inferred variances. Such a type system would accept all current legal Java programs, yet allow programs that are currently not allowed to type-check, without violating soundness. This would mean that wildcards can be omitted in many cases, freeing the programmer from the burden of always specifying tedious types in order to get generality. For instance, if a generic `C` is found to be covariant, then any occurrence of `C<? extends T>` is unnecessary. (We report such instances as "unnecessary wildcards" in our measurements.) Furthermore, any occurrence of `C<T>` or `C<? super T>` will be immediately considered equivalent to `C<? extends T>` or `C<?>`, respectively, by the type system, resulting in more general code. (We report such instances as "over-specified methods" in our measurements.)

- One can use our algorithm as a programmer's assistant in the context of an IDE or as an off-line tool, to offer suggestions for more general types that are, however, still sound. For instance, for a covariant generic, `C`, every occurrence of type `C<T>` can be replaced by `C<? extends T>` to gain more generality without

any potential for more errors. Just running our algorithm once over a code body will reveal multiple points where a programmer missed an opportunity to specify a more general type. The programmer can then determine whether the specificity was intentional (e.g., in anticipation that the referenced generic will later be augmented with more methods) or accidental.

In practice, our implementation (in Scala) of the optimized constraint solving algorithm described in Section 4.4 takes less than 3 minutes (on a 3.2GHz Intel Core i3 machine w/ 4GB RAM) to analyze the generics of the entire Java standard library. Almost all of the time is spent on loading, parsing, and processing files, with under 30 seconds constraint solving time.

Finally, we need to emphasize that our signature-only based inference algorithm is *modular*. Not only does it reason entirely at the interface level (does not inspect method bodies), but also the variance of a generic depends only on its own definition and the definition of types *it* (transitively) references, and not on types that reference it. This is the same modularity guarantee as with standard separate compilation.

We can also generate constraints from inspecting method bodies. This method-body based analysis is explained later in Section 9.4. When the method-body analysis is performed, we expect improved numbers (since, for instance, an invariant type may be passed as a parameter, but only its covariance-safe methods may be used—e.g., a list argument may only be used for reading). Nevertheless, analyzing the bodies of methods has a cost in modularity: the analysis would still not depend on clients of a method, but it would need to examine subtypes, to analyze all the possible overriding methods.

8.2   Analysis of Impact

To measure the impact of our approach, we ran our inference algorithm over 6 Java libraries, the largest of which is the core Java library from Oracle's JDK 1.6, i.e.,

97

classes and interfaces in the packages of `java.*`. The other libraries are JScience [22], a Java library for scientific computing; Guava [10], a superset of the Google collections library; GNU Trove [25]; Apache Commons-Collection [5]; and JPaul [56], a library supporting program analysis.

The results of our experiment appear in Figures 8.1, 8.2, and 8.3. Definition-site variances were inferred with two types of analyses. The signature-only analysis inferred definition-site variance using only the type signature of members of class/type definitions. The method-body analysis is described in Section 9.4. The method-body analysis may relax the constraints on definition-site variances, thereby allowing more generics to be variant. Statistics computed using the method-body analysis are shaded in the tables. Shaded results are for the method body analysis, unshaded for the signature-only analysis

Together, these libraries define 3,827 classes and interfaces, out of which 1,093 are generics. These generics declare 1,442 type parameters—i.e., some of the generics declare more than one type parameter. Statistics in Figure 8.1 are collapsed per-class: An invariant class is invariant in all of its type parameters, whereas a variant class is variant in at least one of its type parameters. Hence, a class can be counted as, e.g., both covariant and contravariant, if it is covariant in one type parameter and contravariant in another. The "variant" column, however, counts the class only once. The five "invar./variant/cov./contrav./biv/" columns show the percentage of classes and interfaces that are inferred by our algorithm to be invariant versus variant, for all three flavors of variance.

As can be seen, 26% of classes or interfaces are variant in at least one type parameter. (Our "Total" row treats all libraries as if they were one, i.e., sums individual numbers before averaging. This means that the "Total" is influenced more by larger libraries, especially for metrics that apply to all *uses* of generics, which may also occur in non-generic code.) This means that about a 1/4 of the generics defined should

Library		# Types	# Generics	Type Definitions				
				invar.	variant	cov.	contrav.	biv.
Java	classes	1786	156	80%	20%	13%	5%	2%
		1786	156	80%	20%	13%	5%	2%
	interfaces	329	31	55%	45%	39%	6%	0%
		329	31	55%	45%	39%	6%	0%
	total	2115	187	76%	24%	17%	5%	2%
		2115	187	76%	24%	17%	5%	2%
JScience	classes	110	46	54%	46%	7%	0%	39%
		110	46	54%	46%	7%	0%	39%
	interfaces	60	13	62%	38%	8%	15%	15%
		61	14	50%	50%	14%	21%	14%
	total	170	59	56%	44%	7%	3%	34%
		171	60	53%	47%	8%	5%	33%
Apache	classes	370	302	76%	24%	12%	8%	4%
		370	302	76%	24%	12%	8%	4%
	interfaces	29	28	64%	36%	25%	11%	0%
		30	29	62%	38%	24%	14%	0%
	total	399	330	75%	25%	13%	8%	3%
		400	331	75%	25%	13%	8%	3%
Guava	classes	551	289	84%	16%	9%	7%	1%
		551	289	84%	16%	9%	7%	1%
	interfaces	50	36	42%	58%	39%	17%	3%
		50	36	42%	58%	39%	17%	3%
	total	601	325	79%	21%	12%	8%	1%
		601	325	79%	21%	12%	8%	1%
Trove	classes	322	46	78%	22%	15%	7%	0%
		322	46	78%	22%	15%	7%	0%
	interfaces	76	17	0%	100%	6%	94%	0%
		76	17	0%	100%	6%	94%	0%
	total	398	63	57%	43%	13%	30%	0%
		398	63	57%	43%	13%	30%	0%
JPaul	classes	129	114	78%	22%	10%	7%	5%
		129	114	77%	23%	10%	8%	5%
	interfaces	13	13	69%	31%	0%	31%	0%
		13	13	69%	31%	0%	31%	0%
	total	142	127	77%	23%	9%	9%	5%
		142	127	76%	24%	9%	10%	5%
Total	**classes**	**3268**	**953**	**78%**	**22%**	**11%**	**6%**	**4%**
		3268	**953**	**78%**	**22%**	**11%**	**7%**	**4%**
	interfaces	**559**	**140**	**49%**	**51%**	**25%**	**24%**	**2%**
		559	**140**	**47%**	**53%**	**26%**	**25%**	**2%**
	total	**3827**	**1093**	**75%**	**25%**	**13%**	**9%**	**4%**
		3827	**1093**	**74%**	**26%**	**13%**	**9%**	**4%**

Figure 8.1. Definition-Site Variance Inference Statistics by Type Definitions. An invariant class is invariant in all of its type parameters, whereas a variant class is variant in at least one of its type parameters. Shaded results are for the method body analysis, unshaded for the signature-only analysis.

Library		# Types	# Generics	Unnecessary wildcards	Over-specified methods
Java	classes	1786	156	16%	7%
		1786	156	16%	7%
	interfaces	329	31	12%	9%
		329	31	12%	9%
	total	2115	187	16%	7%
		2115	187	16%	7%
JScience	classes	110	46	89%	30%
		110	46	89%	30%
	interfaces	60	13	100%	20%
		61	14	100%	20%
	total	170	59	89%	29%
		171	60	89%	29%
Apache	classes	370	302	59%	16%
		370	302	59%	16%
	interfaces	29	28	9%	0%
		30	29	9%	0%
	total	399	330	58%	16%
		400	331	58%	16%
Guava	classes	551	289	41%	10%
		551	289	41%	10%
	interfaces	50	36	33%	4%
		50	36	33%	4%
	total	601	325	41%	10%
		601	325	41%	10%
Trove	classes	322	46	30%	26%
		322	46	30%	26%
	interfaces	76	17	0%	0%
		76	17	0%	0%
	total	398	63	30%	26%
		398	63	30%	26%
JPaul	classes	129	114	5%	25%
		129	114	5%	25%
	interfaces	13	13	0%	0%
		13	13	0%	0%
	total	142	127	5%	25%
		142	127	5%	25%
Total	**classes**	**3268**	**953**	**39%**	**15%**
		3268	**953**	**39%**	**15%**
	interfaces	**559**	**140**	**19%**	**7%**
		559	**140**	**19%**	**7%**
	total	**3827**	**1093**	**39%**	**15%**
		3827	**1093**	**39%**	**15%**

Figure 8.2. Unnecessary Wildcards and Over-Specified Methods. Shaded results are for the method body analysis, unshaded for the signature-only analysis.

Library		# Type Params	Type Parameters					Recursive variances
			invar.	variant	cov.	contrav.	biv.	
Java	classes	190	84%	16%	11%	4%	2%	18%
		190	84%	16%	11%	4%	2%	18%
	interfaces	36	61%	39%	33%	6%	0%	33%
		36	61%	39%	33%	6%	0%	33%
	total	226	80%	20%	14%	4%	1%	20%
		226	80%	20%	14%	4%	1%	20%
JScience	classes	50	58%	42%	6%	0%	36%	54%
		50	58%	42%	6%	0%	36%	54%
	interfaces	15	67%	33%	7%	13%	13%	7%
		15	53%	47%	13%	20%	13%	7%
	total	65	60%	40%	6%	3%	31%	43%
		65	57%	43%	8%	5%	31%	43%
Apache	classes	399	82%	18%	9%	6%	3%	8%
		399	82%	18%	9%	6%	3%	8%
	interfaces	37	70%	30%	22%	8%	0%	30%
		37	68%	32%	22%	11%	0%	30%
	total	436	81%	19%	10%	6%	3%	10%
		436	81%	19%	10%	6%	3%	10%
Guava	classes	433	88%	12%	7%	5%	0%	9%
		433	88%	12%	7%	5%	0%	9%
	interfaces	53	51%	49%	30%	17%	2%	21%
		53	51%	49%	30%	17%	2%	21%
	total	486	84%	16%	10%	6%	1%	10%
		486	84%	16%	10%	6%	1%	10%
Trove	classes	48	77%	23%	15%	8%	0%	31%
		48	77%	23%	15%	8%	0%	31%
	interfaces	18	0%	100%	6%	94%	0%	0%
		18	0%	100%	6%	94%	0%	0%
	total	66	56%	44%	12%	32%	0%	23%
		66	56%	44%	12%	32%	0%	23%
JPaul	classes	149	83%	17%	8%	5%	4%	21%
		149	82%	18%	8%	6%	4%	21%
	interfaces	14	64%	36%	0%	36%	0%	7%
		14	64%	36%	0%	36%	0%	7%
	total	163	81%	19%	7%	8%	4%	20%
		163	80%	20%	7%	9%	4%	20%
Total	**classes**	**1269**	**83%**	**17%**	**9%**	**5%**	**3%**	**14%**
		1269	**83%**	**17%**	**9%**	**5%**	**3%**	**14%**
	interfaces	**173**	**54%**	**46%**	**22%**	**22%**	**2%**	**21%**
		173	**53%**	**47%**	**23%**	**23%**	**2%**	**21%**
	total	**1442**	**80%**	**20%**	**10%**	**7%**	**3%**	**15%**
		1442	**79%**	**21%**	**10%**	**7%**	**3%**	**15%**

Figure 8.3. Definition-Site Variance Inference Statistics by Type Parameters. Shaded results are for the method body analysis, unshaded for the signature-only analysis.

be allowed to enjoy general variant subtyping without users having to annotate them with wildcards.

Figure 8.2 illustrates the burden of default invariant subtyping in Java, and the benefits of our approach. "Unnecessary Wildcards" shows the percentage of wildcards in method signatures that are unnecessary in our system, based on the inferred definition-site variance their generics. For instance, given that our technique infers interface `java.util.Iterator<E>` to be covariant, all '`? extends`' annotations in instantiations of `Iterator` are unnecessary. This number shows that, using our technique, 39% of the current wildcard annotations can be eliminated without sacrificing either type safety or the generality of types!

The "Over-specified Method" column lists the percentage of method arguments that are overly specific *in the Java type system*, based on the inferred definition-site variance of their generics. For instance, given that the inferred definition-site variance of `Iterator<E>` is covariant, specifying a method argument with type `Iterator<T>`, instead of `Iterator<? extends T>`, is overly specific, since the Java type system would preclude safe invocations of this method with arguments of type `Iterator`-of-some-subtype-of-`T`. Note again that this percentage is *purely based on the inferred definition-site variance of the arguments' types, not on analysis of the arguments' uses in the bodies of methods.* We find that 15% of methods are over-specified. This means that 15% of the methods could be used in a much more liberal, yet still type-safe fashion. It is also interesting that this number is derived from *libraries* and not from client code. We expect that the number of over-specified methods would be much higher in client code, since programmers would be less familiar with wildcards and less confident about the operations supported by variant versions of a type.

Variance statistics per generic type parameter are in Figure 8.3. The last column, "Recursive variances", shows the percentage of type parameters for which their definition-site variances are recursively constrained. Example types of recursively

constrained variances were given in Section 3.3. For example, recursive variance type 1 would result in a constraint of the form $var(\texttt{X};\texttt{C}) \sqsubseteq + \otimes var(\texttt{X};\texttt{C})$, where $\texttt{X}$ is a type parameter of generic $\texttt{C}$. As discussed in that chapter, recursively constrained type parameters can be bivariant even when their type parameter is used in the type definition. We know of no other technique that would infer anything other than *invariance* for recursively constrained type parameters. The JScience library shows the largest impact of inferring general variances with recursive variances. Out of the six libraries, the JScience library has the largest percentage of recursive variances. Perhaps not surprisingly, it also has the largest percentage of bivariant type parameters.

Backward Compatibility and Discussion

As discussed earlier, our variance inference algorithm can be used to replace the Java type system with a more liberal one, or can be used to offer suggestions to programmers in the context of an IDE. Replacing the Java type system with a type system that infers definition-site variance is tempting, but would require a pragmatic language design decision, since there is a cost in backward compatibility: in some cases the programmer may have relied on types being rejected by Java, even though these types can never cause a dynamic type error.

We found one such instance in our experiments. In the reference implementation for JSR 275 (Measures and Units) [23], included with the JScience library [22], a group of 11 classes and interfaces are collectively bivariant in a type parameter, `Q extends Quantity`. In the definition of `Unit<Q extends Quantity>`, for example, the type parameter `Q` appears nowhere other than as the type argument to `Unit<Q>`. Closer inspection of the code shows that `Quantity` is extended by 43 different subinterfaces, such as `Acceleration`, `Mass`, `Torque`, `Volume`, etc. It appears that the authors of the library are actually relying on the invariant subtyping of Java generics, to ensure, e.g., that `Unit<Acceleration>` is never used as `Unit<Mass>`.

Of course, full variance inference is only one option in the design space. Any combination of inference and explicitly stated variance annotations, or just adding explicit definition-site variance to Java, are strictly easier applications from a typing standpoint. The ultimate choice is left with the language designer, yet the potential revealed by our experiments is significant.

CHAPTER 9

REFACTORING BY INFERRING WILDCARDS

Wildcard annotations can improve the generality of Java generic libraries, but require heavy manual effort. This chapter presents 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.

9.1 Contributions Relative to Past Work

This chapter is based on the observation that the straightforward approach of Chapter 8 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,

we have not explained how we can infer more general types by analyzing method bodies. Most importantly, however, the statistics in Chapter 8 showed the potential impact when 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 [21, 41] and assumes no changes to Java (unlike, e.g., Chapter 6).

In outline, this 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. Similar, to work in Chapter 4, our approach infers definition-site

variance based on type signatures for determining if a parameterized type is overly specified. Unlike that work, we also perform an inter-procedural analysis based on how objects are actually used in the program to determine if the parameterized types specified by the programmer are overly restrictive. A method taking in a parameter of type `List<T>`, for instance, may only invoke methods available from a `List<? extends T>`.

- 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. For instance, the user may not want the return type of some method to be changed to a supertype, since the supertype provides fewer methods of a class to a client.

- 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. To measure the benefit of analyzing method bodies, we performed analyses both when taking in method bodies into account and when inferring definition-site variance solely based on a generic's type signature. 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`. Appendix D formally argues why the refactoring algorithm preserves the ability to compile programs.

9.2 Illustration

We next illustrate the impact and intricacies of inferring variance annotations in a pragmatic setting. Figure 9.1 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 21, 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. 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>`. `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

108

```
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> extends WList<Map.Entry<K,V>> {
21   @Override void add(Map.Entry<K, V>  entry )  { }
22  }
```

```
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> extends WList<Map.Entry<K,V>> {
21   @Override void add(Map.Entry<K, V> entry) { }
22  }
```

Figure 9.1. Code comparison. Original code on the top. Refactored code on the bottom. Declarations that were selected for generalization are shaded in the original version.

the limited use of `source` in `addAll` we can 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` field (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

---

[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>`.

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 21. 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* [28, Section 4.6] of the 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.

_____

[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 [51, Sections 5.2 and 4.5]: `elems` is not only private to `WList` but also only accessed from a `this` qualifier on line 8.

(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 9.3.6. 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 [28, Section 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.

9.3 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

113

expression from A to B: generalizing the type of A would require B's type to also be generalized. Subsequent subsections provide further details.

9.3.1 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`:

$$
\begin{array}{lll}
\texttt{InfluenceNode} & ::= & \texttt{MethodDecl} \mid \texttt{Variable} \\
\texttt{Variable} & ::= & \texttt{VariableDeclaration} \\
& \mid & \texttt{FieldDeclaration} \\
& \mid & \texttt{ParameterDeclaration}
\end{array}
$$

`MethodDecl`, `VariableDeclaration`, `FieldDeclaration` and `ParameterDeclaration` are the syntactic entities that their names suggest, as defined in the JLS [28]. Both static and instance fields are instances of `FieldDeclaration`s. `VariableDeclaration`s are local variable declarations, which occur in blocks, method bodies, initialization statements of for-loops, etc. Formal value arguments from methods and constructors are `ParameterDeclaration`s. Arguments in exception-catch declarations are ignored; we found that generalizing their types would not provide significant benefit. `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 [32], 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 9.2. 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 $\overline{A}$ denotes the possibly empty vector $A_1, A_2, \ldots, A_n$. 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 ($\overline{T}$) and a qualifier component ("`e.`"), we allow invocations where both can be skipped.

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

Figure 9.2. FGJ* Syntax

Auxiliary functions are defined in Figure 9.3. 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*(m; $\overline{e}$) returns the declaration of the method being called (statically) by the method invocation `m(`$\overline{e}$`)`.[3] Detailed definitions of look up functions and other auxiliary functions that are omitted here can be found in Chapter 6 and [32].

---

[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)

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

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

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

destinationNode: (representative rules)

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

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

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

Lookup$(\texttt{m};\overline{\texttt{e}})$ = the declaration of the method being called (statically) by the invocation of method $\texttt{m}$ with arguments $\overline{\texttt{e}}$.

returnTypeDependsOnParams$(M) \equiv M$ is a generic method with a type parameter $\texttt{X}$ that syntactically occurs in both the return type and in an argument type.

varDecl$(\texttt{e})$ = the declaration referred to by expression $\texttt{e}$ (e.g. $varDecl(\texttt{x.f})$ = declaration of field $\texttt{f}$).

accessedNodes$(\texttt{e})$ = the set of declarations accessed in expression $\texttt{e}$ (e.g. $accessedNodes(\texttt{x.f}) = \{\texttt{x},\texttt{f}\}$).

enclosingMethod$(\texttt{e})$ = the enclosing method of $\texttt{e}$ (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 $M'\ |\ M' \in hierarchyMethods(M)\}$, where $\texttt{x}$ 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 9.3. Auxiliary Functions

9.3.2 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 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<?>`.

117

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`.

9.3.3 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 9.2, the type of the parameter `source` changes from `List<E>` to `List<? extends E>` in the refactored code. The return 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 9.3 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.`

---

[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.

`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.

9.3.4 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 21) 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.

9.3.5 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 9.3. 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 9.3.2

1: **for each** method call `<`$\overline{\mathtt{T}}$`>m(`$\overline{\mathtt{e}}$`)` $\in P$ **do**

2:    *qualifierDecl* $\leftarrow$ *varDecl*(`e`)

3:    `<`$\overline{\mathtt{Y} \lhd \mathtt{U}}$`> T m(`$\overline{\mathtt{T\ x}}$`) { return ... ; }` $\leftarrow$ *Lookup*(`m`; $\overline{\mathtt{e}}$)

4:    Add edge (*qualifierDecl*, $\mathtt{x}_i$) to G, *for each* $\mathtt{x}_i \in \overline{x}$.

5: **end for**

    // Analysis from Section 9.3.3

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 9.3.4

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 $\mathtt{x} \in$ *formalParams*(M) **do**

13:        **for each** parameter $\mathtt{y} \in$ *hierarchyParams*($\mathtt{x}$) **do**

14:            Add edge $(\mathtt{y}, \mathtt{x})$ to G

15:        **end for**

16:    **end for**

17: **end for**

18: **return** G

9.3.6 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 21 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 an argument type or return type of a non-static and non-final method not to be rewritable if the type is just a type variable.

Discussion: parent types are not rewritable. As mentioned in Section 9.2, 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

---

[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 9.4. Simplified code example from the Apache collections library at the top. Subtyping (interface-implements) relationships at the bottom, if we annotate K with "`? extends`" only in the parent type `OrderedIterator<Map.Entry<K,V>>`.

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 9.4, which is a simplified version of a code segment from the Apache collections library [5].

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 [28, Section 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 9.4 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

---

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

123

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 [39, 61] 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.

9.4   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. For example, a method may take a `List<String>` as an argument, but never invoke a method on the argument that contains a type parameter in its type signature (e.g., it may only call the `size` method in the `List` interface). In this case, the argument could be declared with the more liberal type `List<?>` without

---

[7]Subtyping in the presence of definition-site variance and contravariant type constructors in parent type declarations was shown to be undecidable in [39]. [39, 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. [61] identified a decidable fragment that does not allow "`? super`" in parent types.

causing a type error, and the more liberal type would allow the method to accept more types of list arguments. 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);
  }
}
```

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] In this case, the (implicitly) specified use-site variance annotation, invariance, has been replaced with covariance. 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. [9]

It was easy to manually inspect and determine the most liberal use-site variance required for argument `lx`. The `compareFirst` has few lines of code and the type expressions in the signature of the members accessed by `lx` are simple (i.e. do not contain any parameterized types). This reasoning becomes more difficult and error prone as methods become longer and more parameterized types are involved. Also,

---

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

[9]The covariance of `X` in `List<? extends X>` is transformed by the contravariant method argument position that it occurs in. The other type occurrences of `X` in class `Body` also only constrain $dvar(\texttt{X}; \texttt{Body})$ to contravariance.

changing a use-site annotation may cause a method to no longer override a previously overridden method. Performing a method body analysis to infer use-site annotations, thus, warrants automation. The remainder of this section presents how we generate constraints from uses and we present sets of constraints generated from example classes.

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 4.2. Use-site variances can now vary (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 `C<vT>` and X_i is the ith type parameter of generic `C`, then $uvar(X_i; C; y)$ 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.

$$dvar(\texttt{X}; \texttt{Body}) \sqsubseteq - \otimes var(\texttt{X}; \texttt{List<}o\texttt{X>})$$

$$= - \otimes (dvar(\texttt{E}; \texttt{List}) \sqcup o)$$

$$= - \otimes dvar(\texttt{E}; \texttt{List}) = - \otimes o = o,$$

where `E` is the type parameter of `List`.

$dvar(\texttt{E}; \texttt{List})$ 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 $\texttt{lx}$. In this example, we would infer $uvar(\texttt{E}; \texttt{List}; \texttt{lx}) = +$ and $dvar(\texttt{X}; \texttt{Body}) = -$. Further details of the constraint generation process performed during the method body analysis can be found in Appendix C.

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

$$= -\otimes (o \sqcup o \sqcup uvar(\texttt{E}; \texttt{List}; \texttt{lx}))$$

$$= -\otimes uvar(\texttt{E}; \texttt{List}; \texttt{lx})$$

9.5   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 parameteric type `C<vT>`, where generic `C` is safely (definition-site) variant (covariant, contravariant, or bivariant) in at least one of its type parameters; otherwise, we call `C<vT>` an invariant type. Because `Iterator` is covariant in its type parameter, for exam-

ple, `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 9.3.6, 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 2.1) 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,

128

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 9.4), 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. The tables in Figures 9.5 and 9.7 show an instance of this result.

9.6 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.

Section 8.2 showed that the majority (53%) of interfaces and a large proportion (22%) of classes in popular, large Java generic libraries are variant even though they

129

Library		# P-Decl Total	# Rewritable P-Decl Total	Rewriteable P-Decl %	Rewritten Total	Rewritten Percentage
Java	classes	4900	4284	87%	569	12%
		4900	4193	86%	584	12%
	interfaces	170	153	90%	20	12%
		170	148	87%	34	20%
	total	5070	4437	88%	589	12%
		5070	4341	86%	618	12%
JScience	classes	1553	1042	67%	217	14%
		1553	1017	65%	229	15%
	interfaces	56	53	95%	43	77%
		56	53	95%	44	79%
	total	1609	1095	68%	260	16%
		1609	1070	67%	273	17%
Apache	classes	3357	2567	76%	565	17%
		3357	2491	74%	600	18%
	interfaces	46	38	83%	1	2%
		46	38	83%	16	35%
	total	3403	2605	77%	566	17%
		3403	2529	74%	616	18%
Guava	classes	5794	3973	69%	355	6%
		5794	3690	64%	384	7%
	interfaces	69	57	83%	2	3%
		69	56	81%	2	3%
	total	5863	4030	69%	357	6%
		5863	3746	64%	386	7%
Trove	classes	953	531	56%	127	13%
		953	531	56%	139	15%
	interfaces	0	0	0%	0	0%
		0	0	0%	0	0%
	total	953	531	56%	127	13%
		953	531	56%	139	15%
JPaul	classes	1350	1085	80%	137	10%
		1350	1067	79%	187	14%
	interfaces	11	11	100%	0	0%
		11	11	100%	1	9%
	total	1361	1096	81%	137	10%
		1361	1078	79%	188	14%
Total	**classes**	**17907**	**13482**	**75%**	**1970**	**11%**
		17907	**12989**	**73%**	**2123**	**12%**
	interfaces	**352**	**312**	**89%**	**66**	**19%**
		352	**306**	**87%**	**97**	**28%**
	total	**18259**	**13794**	**76%**	**2036**	**11%**
		18259	**13295**	**73%**	**2220**	**12%**

Figure 9.5. 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		Flowsto Avg. Size	Flowsto-R Avg. Size
Java	classes	61.10	1.23
		61.37	1.16
	interfaces	39.91	2.75
		40.08	2.76
	total	60.39	1.29
		60.66	1.21
JScience	classes	52.04	5.42
		54.59	5.49
	interfaces	10.21	0.66
		10.27	0.66
	total	50.59	5.19
		53.05	5.25
Apache	classes	119.81	0.69
		122.31	0.72
	interfaces	84.61	0.61
		84.63	0.61
	total	119.33	0.69
		121.80	0.71
Guava	classes	289.27	0.97
		313.20	0.93
	interfaces	134.59	3.70
		154.91	3.73
	total	287.45	1.01
		311.34	0.97
Trove	classes	13.93	0.26
		13.95	0.28
	interfaces	N/A	N/A
		N/A	N/A
	total	13.93	0.26
		13.95	0.28
JPaul	classes	15.60	0.50
		15.98	0.53
	interfaces	0.73	0.73
		0.73	0.73
	total	15.48	0.50
		15.86	0.53
Total	**classes**	**139.21**	**1.28**
		147.74	**1.26**
	interfaces	**58.36**	**2.23**
		62.44	**2.24**
	total	**137.65**	**1.30**
		146.10	**1.28**

Figure 9.6. The flows-to set of a declaration D is the set of all declarations that are reachable from/influenced by D in the influence graph. Flowsto Avg. Size is the average size of the flows-to set for all declarations in the influence graph. Flowsto-R Avg. Size is the average size of the flows-to set for all *rewritable* declarations in the influence graph.

Library		# V-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 9.7. 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.

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 [22], a Java library for scientific computing; Guava [10], a superset of the Google collections library; GNU Trove [25]; Apache Commons-Collection [5]; and JPaul [56], a library supporting program analysis.

The results of our experiment appear in Figures 9.5, 9.6, and 9.7. 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 9.7'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 9.5) 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 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.

9.7   Comparison to Related Work

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

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

$$\Downarrow$$

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

$$\Downarrow$$

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

Figure 9.8. Refactoring resulting from applying Kiezun et al.'s [41] and then our refactoring tool

Kiezun et al. [41] 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 9.8. 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. [15,21] 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: `List<[`T_L`,`T_U`]>` $<:$ `List<[`S_L`,`S_U`]>`, if $S_L <: T_L$ and $T_U <: S_U$. Types
`List<[T,T]>`, `List<[`$\bot$`,T]>`, and `List<[T,`$\top$`]>` are abbreviated as `List<`$\odot$`T>`, `List<`$\oplus$`T>`,
and `List<`$\ominus$`T>`, where $\bot$ and $\top$ are subtypes and supertypes of every type, respec-
tively. 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 parameteric 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,
`List<`$\ominus$`T>`, should be able to read the `comp` field as an instance of `Comparator<`$\ominus$`T>`.
However, Craciun et al.'s type promotion technique [15, Section 4.2], would only be
able to read a `Comparator<`$\oplus$`T>` from the `comp` field of a `List<`$\ominus$`T>`. A `Comparator<`$\oplus$`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<`$\ominus$`T> c =`
`x.comp;`", the use-site annotation in the type of `x` inferred by the approach would be
invariance. Our tool, however, would rewrite the type of `x` with a greater (contravari-
ant) use-site annotation; more generally, our approach infers more liberal use-site
variances when fields are of parametric types.

CHAPTER 10

RELATED WORK

This chapter presents relevant related research and comparisons to our work. We also discuss how this research fits in the broader field of programming languages by explaining how this work relates to and can be applied to other subfields of programming languages.

10.1   Related Research on Variance

Definition-site variance was first investigated in the late 80's [4, 11, 18], when parametric types were incorporated into object-oriented languages. For example, [18] presents a proposal for making Eiffel type safe, where type attributes are used to declare definition-site variances of generic type parameters. Definition-site variance has experienced a resurgence in recent years, as newer languages such as Scala [51] and C# [31] chose it as means to support variant subtyping. Perhaps surprisingly, with such a long history, it has only recently been formalized and proven sound in a non-toy setting [24].

Previous work focuses on the safety characteristics of definition-site variance, not on how such variance can be inferred (other than for basic, non-recursive types). There has never been a study of how to determine definition-site variance in the face of recursive type definitions—which are ubiquitous in object-oriented programs. This work is the first, to our knowledge, to determine safe definition-site variances with recursive type definitions. Our implementation solves the problem fully, not just for the four example cases presented in Section 3.3. Any recursive constraint on

definition-site variances can be encoded in the variance constraint language presented in Section 4.2. Our constraint-solving algorithm (presented in Section 4.2) computes the most general definition-site variances that satisfy a system of the constraint inequalities.

Use-site variance was introduced as *structural virtual types* by Thorup and Torgersen [62] in response to the rigidity in class definitions imposed by definition-site variance. A language with virtual types supports *virtual type members*, also known as abstract type members in Scala [51, Section 4.3], where, in addition to fields and methods, a type can be declared as a member of a class. Virtual type members can be overridden or instantiated in subclasses. For example, a `List` class may have a virtual type member `ElemType` that represents the type of the elements stored in the list. A class `IntegerList` may extend `List` and override `ElemType` by setting it to the type `IntegerList`. A possible implementation of these two classes in a Java-like language supporting virtual types is below:

```
class List {
 type ElemType <: Object
 void add(ElemType e) { ... }
 ElemType get(int index)  { ... }
}
class IntegerList extends List {
  type ElemType == Integer;
}
```

Structural virtual types avoid the need to create a subclass in order to create an instantiation of a type. They allow bindings to type members to be specified in type expressions. The `IntegerList` class above can be emulated with the type expression `List[ElemType==Integer]`. Use-site variance is supported with type expressions such as `List[ElemType<:Object]`, where `List[ElemType==Integer]` <: `List[ElemType<:Object]`.

Java does not support type members but does facilitate abstract types using generic type parameters. The concept of use-site variance from structural virtual

types was later applied to Java generics by Igarashi and Viroli [33, 34]. This work formalized use-site variance by developing a formal language model that extended Featherweight GJ with *variant parametric types* (VPTs). They also generalized the notion of use-site variance to also support contravariant and bivariant use-site annotations; the approach in [62] only supported covariant use-site variance.

The elegance and flexibility of the approach evoked a great deal of enthusiasm. As a result, use-site variance was quickly introduced into Java by extending it with wildcards [63]. Java wildcards facilitate use-site variance and support capabilities that are not provided by VPTs. Unlike VPTs, Java wildcards do not rely on read- and write-only semantics. For example, invoking the `get()` method to read an element on an instance of the VPT `List<-Number>` is prohibited because this type returns a write-only version of `List`. Invoking `get()` on the contravariant wildcard type `List<? super Number>`, however, does type check. That method invocation would be typed with the top type, `Object`, since any supertype of `Number` is still a subtype of `Object`. Also, a `List<+Number>` does not allow calling `add()` to add `null` to itself. A `List<? extends Number>` facilitates this operation.

Java wildcards support further capabilities inspired by existential types such as capture conversion [28, Section 5.1.10] and wildcard capture [28, Section 15.12.2.7], where types hidden by wildcards can be opened in invocations to polymorphic methods. These further capabilities raise practical issues not addressed by VPTs. We explained these issues in Section 5.2.

The flexibility of wildcards has also proved challenging to both researchers and practitioners. The soundness of wildcards in Java has only recently been proven [12], and the implementation of wildcards has been mired in issues [15,59,61]. Decidability of subtyping with wildcards is still an open problem [29]. Subtyping with definition-site variance was shown to be undecidable [39]. Since definition-site variance can be emulated with use-site variance [3], subtyping with use-variance is likely also unde-

cidable. Greenman et al. [29] have identified a fragment of Java with wildcards for which their algorithm for deciding subtyping is sound and complete. After surveying 13.5 millions lines of open-source Java code, they found that all surveyed code was within that decidable fragment. This suggests that subtyping with use-site variance is decidable for most practical Java programs.

[17] discusses the complex relationship between type-erasure and wildcards and the difficulty of providing runtime information about generic type instantiations in that context. [35] presents variant subtyping between *path types* that describe *exact* and *inexact* qualifications/paths to nested type definitions. Let `T` be a path type and `C` and `D` be class names. The exact qualification `T@C` accesses a class `C` that is a member of the type definition for `T`. The inexact qualification `T.C` accesses a class `C` that is defined at any level of nesting within `T`'s type definition. Exact qualifications are invariant type constructors: `T@C` is a subtype of `T@D` only when `C = D`. Inexact qualifications are covariant type constructors: `T.C` is a subtype of `T.D` when class `C` extends class `D` and both classes are inside the definition of type `T`.

Operations Available to a Variant Type

The work of Viroli and Rimassa [65] attempts to clarify when variance is to be used, introducing concepts of produce/consume, which are an improvement over the read/write view. Under this view, expression variables of parameteric types (`C<T>`) that only produce elements of the type argument (`T`) should be annotated with '`?` `extends`' (`C<? extends T>`). Similarly, parameteric types of variables that only consume should be annotated with '`?` `super`'. Using that perspective, that work presents access restrictions rules to compute the signature of members of generic definitions given a parametric type. The producer/consumer view does not further clarify the signature of members that include parametric types `C<T>` that include a class type parameter. Our approach offers a generalization and a high-level way to reason soundly

about the variance of arbitrary nested type expressions. The variance of member type signatures can be used for determining which members are available to a variant version of a type. We demonstrate this benefit with the code example in Figure 10.1. We will explain how our approach can be used to determine which non-static methods are available to an instance of `SimpleGen<? super T>`, where `T` is some type expression. In this case, available methods are methods that can be called with a non-`null` value. If calling a method with any non-`null` value causes a compiler (a type checking) error, then that method is considered *not* to be available.

A non-null value can clearly be passed to `meth1` when invoking that method on an instance of `SimpleGen<? super T>`. The argument type of `meth1` is just the class type parameter `E`, so it is easy to determine that `meth1` is available to consumers. Static method `foo1` demonstrates that `meth1` is available because the method call `sg.meth1(str)` in its body type checks.

The argument types in the remaining methods are parameteric types. Determining which of the remaining methods are available to a `SimpleGen<? super T>` is not clear with the producer/consumer view. The only other available method is `meth3`. Using our formalism, it is easy to determine that method `meth3` is available but methods `meth2` and `meth4` are not. Our approach can easily determine that class type parameter `E` appears contravariantly in the type signatures of `meth1` and `meth3`. Hence, these methods are available to a contravariant projection of `SimpleGen`. Method `foo3` type checks, which confirms the availability of `meth3`. `E` appears invariantly in `meth2` and covariantly in `meth4`; so these two methods are not available to a `SimpleGen<? super T>`.

The compiler error messages that result from compiling methods `meth2` and meth4 are complex. They do not provide the essence of why they do not compile because Java does not have the notion of the variance of a position. For example, the error message resulting from trying to compile `meth4` with `javac` version `1.6.0_65` is below.

```
public class SimpleGen<E>
{
  E elem;
  SimpleGen(E elem) { this.elem = elem; }

  // available to SimpleGen<? super E>
  public void meth1(E arg) { }

  // not available to SimpleGen<? super E>
  public void meth2(Vector<E> arg) { }

  // available to SimpleGen<? super E>
  public void meth3(Iterator<? extends E> arg) { }

  // not available to SimpleGen<? super E>
  public void meth4(Comparable<? super E> arg) { }

  public static void foo1(SimpleGen<? super String> sg,
                          String str)
  {
    sg.meth1(str); // OK, type checks
  }
  public static void foo2(SimpleGen<? super String> sg,
                          Vector<String> vstr)
  {
    sg.meth2(vstr); // error, does not type check
  }
  public static void foo3(SimpleGen<? super String> sg,
                          Iterator<? extends String> itr)
  {
    sg.meth3(itr); // OK, type checks
  }
  public static void foo4(SimpleGen<? super String> sg,
                          Comparable<? super String> cstr)
  {
    sg.meth4(cstr); // error, does not type check
  }
}
```

Figure 10.1. Code example for investigating which non-static methods are available to an instance of `SimpleGen<? super T>`, where `T` is some type expression. In this example, available methods are methods that can be called with a non-`null` value. If calling a method with any non-`null` value causes a compiler (a type checking) error, then that method is considered *not* to be available.

```
meth4(java.lang.Comparable<? super capture#537
  of ? super java.lang.String>)
in SimpleGen<capture#537 of ? super
  java.lang.String>
cannot be applied to
  (java.lang.Comparable<capture#407
    of ? super java.lang.String>)
```

10.2 Variance and Programming Language Research

This section relates the work of this dissertation to the broader field of programming languages. Specifically, we discuss other subfields of programming languages and how they relate to the approach presented in this dissertation.

10.2.1 Nominal Subtyping and Structural Subtyping

Nominal subtyping [53, Section 19.3] is the ability to *declare* one type to be a subtype of another type. Object-oriented languages facilitate nominal subtyping by allowing one class to declare that it is extending another class or implementing an interface. When one type declares that it is a subtype of another, the semantics of the language should ensure that the subsumption principle is satisfied for declared subtype relationships. In the case of Java, when class `C` is declared to extend another class `D`, `C` is a subtype of `D`. Also, class `C` inherits all of the operations from class `D`. Hence, nominal subtyping in Java satisfies the subsumption principle.

Structural subtyping [14] is defined using the structure of types. Languages with structural subtyping support *type constructors*, also known as type operators [53, Chapter 29]. A type constructor is a generalization of a generic and is a function that returns types. The `ref` type constructor in ML [47], for example, takes in a type `T` and returns a type representing a reference cell storing values of type `T`.

Structural subtyping does not require programmers to declare one type to be a subtype of another. A type system supporting structural subtyping will assign vari-

ances to arguments of type constructors that are defined in a language specification. Type constructors defined in the language are operators for aggregating simpler structures into more complex structures. Only those type constructors will support variant subutyping. For example, a product type [53, Section 11.6], $T_1 \times T_2$, represents an immutable pair of elements (x, y), where the first element x is of type T_1 and the second element y is of type T_2. Since pairs are immutable, products are covariant in their two element types. For example, if $\texttt{int} <: \texttt{real}$, then $\texttt{int} \times \texttt{int} <: \texttt{real} \times \texttt{real}$.

Nominal and structural subtyping each have their own advantages and disadvantages. Structural subtype relationships are not required to be declared by programmers. This supports unanticipated use of one type as another. However, structural subtyping only supports variant subtyping for a fixed set of type constructors that are defined in a language specification. Structural subtyping rules do not establish variant subtyping with user-defined type constructors.

This dissertation investigates subtyping with user-defined type constructors, such as generics. It focuses on integrating nominal subtyping and parametric polymorphism. In addition, Section 6.3 provides insights that are useful for determining how to assign variances to arguments of type constructors in general.

Nominal subtyping allows programmers to extend software without modifying existing code by declaring that a newly created type is a subtype of an existing type. Software extension with new data types is common in practice [55]. Variance allows code with parameterized types to be applied to new data types. Supporting software extension without modification is discussed further in Section 10.2.2.

Currently, most programming languages do not support both nominal and structural subtyping. Malayeri et al. [42] investigated supporting both nominal and structural subtyping in a single language called Unity. However, Unity does not support generics. That work did not investigate variant subtyping between user-defined type constructors. We briefly discuss what this combination looks like in Section 11.2.

10.2.2 Nominal Subtyping and Software Extension

Nominal subtyping is a key mechanism in object-oriented languages to designing software entities that follow the *open-closed principle* [46]. This principle states that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". In other words, software should be able to be extended with new features without modifying existing code. Nominal subtyping supports software extension without modifying existing code because instances of new types can be used where instances of existing types are expected. For example in Java, a new class may implement an interface that existing methods expect as arguments. More generally, nominal subtyping makes it easy to extend software with new classes or *datatypes* without modifying existing code.

However, adding *new operations* that vary by type is difficult in pure object-oriented languages. Type-varying operations are implemented as instance methods in classes. A method is overridden in a subclass to implement the appropriate behavior for that subclass. Adding a new type-varying operation requires modifying existing classes to add a new instance method that implements the operation.

In summary, object-oriented languages support adding new datatypes without modifying existing code because of nominal subtyping. Adding type-varying operations requires modifying existing code in pure object-oriented languages.

10.2.3 Nominal Subtyping and Functional Languages

Nominal subtyping is not supported in most pure functional languages [20] such as Haskell [38, 43]. In pure functional languages, type-varying functions operate over *algebraic datatypes* [38, Section 4.2.1] also known as *variants* [53, Section 11.10]. Terms of an algebraic datatype are generated by a finite set of *constructors* or functions that return terms of the datatype. For example, the following line of Haskell code declares

an algebraic datatype `Tree` to represent a binary tree of integers. `Tree` is declared with two constructors, `Empty` and `Node`.

```
data Tree = Empty | Node Int Tree Tree
```

This line states that a `Tree` can be either of the following:

1. An empty tree represented by the constructor `Empty`.

2. A node that stores an integer (an `Int`) and has two subtrees. The constructor `Node` can be thought of as a function that returns a new `Tree` given an `Int` and two instances of `Tree`.

Functions on algebraic datatypes are implemented using *pattern matching* [38, Section 3.17], which is a case analysis on the structure of the term. The structure of a term is determined by the constructor used to create the term. The following function computes the height of a `Tree` using case analysis on the structure of the input `Tree`.

```
height :: Tree -> Int
height Empty = 0
height (Node num left right) = 1 + max (height left) (height right)
```

The first line declares the type of function `height`. The second line states that if an instance of `Tree` was generated using constructor `Empty`, then `height` returns 0. The last line states that the height of a `Tree` of the form `(Node num left right)` is one plus the larger of the two heights of the subtrees `left` and `right`.

As shown in the example above, type-varying functions are not members of a type, as in object-oriented languages. Unlike the case in object-oriented languages, new type-varying functions can be added without modifying existing code.

Programs in purely-functional languages depend on a closed-world assumption of datatypes. Functions that take in an algebraic datatype expect terms to be in one of a finite set of forms. This conflicts with the ability to apply existing code to new data

147

types, which is a goal of nominal subtyping. Adding a new constructor/data type to an algebraic datatype requires updating every function that takes in that type.

In summary, software can be extended either by adding new data types or new operations. Object-oriented languages support adding new data types without modifying existing code, but adding new operations requires modifying existing code. Conversely, functional languages only support adding new operations without modifying existing code. Supporting the ability to add both data types and operations without modifying existing code is known as the expression problem [19, 54, 66].

Robbes et al. [55] investigated which type of software extension, new datatypes or new operations, is more common. This work analyzed software projects from the Squeaksource repository [60] that were implemented in the purely object-oriented language, Smalltalk [27]. More than half a billion lines of code, distributed over 2,505 projects, and 111,071 commits were analyzed. They found that both kinds of extensions occur with roughly the same frequency. Hence, subtyping enables a common kind of software extension without modifying existing code. They also found that larger class hierarchies over time tend to need more operation extensions than datatype extensions.

10.2.4 Generalized Constraints with Existential Types

Emir et al. [24] presented a calculus modeling C# with definition-site variance and *generalized constraints*. Generalized constraints are lists of subtyping constraints between arbitrarily complex types. For each subtyping constraint `T <: U`, `T` and `U` are called the lower bound and the upper bound of the constraint, respectively. Also, for each constraint, neither the lower bound nor the upper bound are required to be a type variable. This section discusses our initial investigation of how our approach to reasoning about variance may provide a general framework for reasoning about generalized constraints. For example, we describe in Section 10.2.4.2 how to reason

about generalized constraints with existential types, which are not supported in the formalism of [24].

A method's generalized constraints can be used to add a *locally* assumed bound on a class type parameter such as in the code example below, taken from [24, Section 1.2]. A subtype constraint $T <: U$ is expressed in C#'s syntax as `T : U`.

```
interface ICollection<X> {
  ...
  void Sort() where X : IComparable<X>;
  bool Contains(X item) where X : IEquatable<X>;
  ...
}
```

These additional constraints are assumed to hold only for the method that declares them. If method `Sort()` had a method body, then, within that method body, methods available to instances of `IComparable<X>` can also be invoked on instances of `X`.

Calling a method with additional constraints requires that actual type arguments satisfy the constraints. For example, method `Sort()` can be invoked on an instance of `ICollection<T>` only if $T <: $ `IComparable<T>`.

10.2.4.1 Deconstructing Generalized Constraints

Deconstructing generalized constraints is the process of inferring subtype relationships between subterms of types occurring in the generalized constraints. If the subtype relationship $T <: U$ holds, deconstructing this relationship investigates which subtype relationships must have been derived in order to derive $T <: U$. We demonstrate this process using the code segment in Figure 10.2. All class type parameters in this example are implicitly declared to be invariant.

Type parameters of generic methods are declared in generalized constraints. Any type variable in a method's generalized constraint that is not a class type parameter is a type parameter of the method. For example, type variables `C` and `D` are type parameters of method `EqTuple` in class `Exp`.

149

```
abstract class Exp<X> {
  public abstract X Eval();
  public abstract bool Eq(Exp<X> that);

  /** This is a generic method with type parameters
    * C and D introduced in the generalized constraint.
    */
  public abstract bool EqTuple<C,D>(Tuple<C,D> that)
    where Tuple<C,D> : Exp<X>;
}
class Tuple<A,B> : Exp<Pair<A,B>> // class Tuple extends Exp
{
  public Exp<A> e1;
  public Exp<B> e2;
  public Tuple(Exp<A> e1, Exp<B> e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public override Pair<A,B> Eval() {
    return new Pair<A,B>(e1.Eval(), e2.Eval());
  }

  public override bool Eq(Exp<Pair<A,B>> that) {
    // Note that Tuple<A,B> <: Exp<Pair<A,B>>,
    // which satisfies constraint of EqTuple.
    return that.EqTuple<A,B>(this);
  }

  /** The constraint inherited from the overridden
    * method specializes to
    * "where Tuple<C,D> <: Exp<Pair<A,B>>"
    * is inherited from the overridden method.
    */
  public override bool EqTuple<C,D>(Tuple<C,D> that) {
    return e1.Eq(that.e1) && e2.Eq(that.e2);
  }
}
```

Figure 10.2. Example C# program with generalized constraints. This example is based on an example from [24, Section 2.5].

The above program type checks according to the typing rules in [24]. In particular, generic method `Tuple.EqTuple` (method `EqTuple` in the class `Tuple`) type checks because of deconstructing generalized constraints. Consider the body of generic method `Tuple.EqTuple`. Note the following four expression typings:

1. `e1` has type `Exp<A>`.

2. `e2` has type `Exp<B>`.

3. `that.e1` has type `Exp<C>`.

4. `that.e2` has type `Exp<D>`.

In order for the body of `Tuple.EqTuple` to type check, it must be the case that `Exp<C>` <: `Exp<A>`; otherwise, the method invocation `e1.Eq(that.e1)` would not type check. Similarly, `Exp<D>` <: `Exp<B>` because of method invocation `e2.Eq(that.e2)`. These two subtype relationships can be inferred from the generalized constraint "`Tuple<C,D>` : `Exp<Pair<A,B>>`" inherited from the parent class. In order to derive `Tuple<C,D>` <: `Exp<Pair<A,B>>`, those two subtype relationships must have been derived. [24, Section 2.5] explains in detail how those two subtype relationships are derived from the generalized constraint. The derivation relied on the fact that classes `Exp` and `Tuple` are declared to be invariant in all of their type parameters.

10.2.4.2 Deconstructing Existential Subtyping

Rules for deconstructing subtyping relationships between non-existential types are presented in [24, Section 3]. If `C<X>` is covariant or invariant and `C<T>` <: `C<U>`, then rule `DECON`[+] would conclude `T` <: `U`. Similarly, if `C<X>` is contravariant or invariant, rule `DECON`[−] gives the implication `C<T>` <: `C<U>` $\implies$ `U` <: `T`. We present a generalization of these rules for deconstructing both existential and non-existential types using the predicate *var*, our novel notion of a variance of a type defined in Figure 6.2.

Suppose `List` is invariant and consider the following subtype relationship:

$$\texttt{List<? extends T>} <: \texttt{List<? extends U>} \tag{10.1}$$

Although `List` is invariant, we should be able to deduce `T` $<:$ `U` because of the "`? extends`" wildcard annotation. This is because both `T` and `U` occur in the same covariant position. We apply *var* to provide rules Decon-B and Decon-R below. They are powerful yet simple rules for deconstructing subtyping relations. They deconstruct existential subtyping and non-existential subtyping, respectively. The syntax of terms and the definition of judgments used in these are in Chapter 6.

$$
\frac{\overline{\texttt{v}} \geq var(\overline{\texttt{X}};\texttt{B}) \qquad \Delta \vdash \overline{[\texttt{U}/\texttt{X}]\texttt{B}} <: \overline{[\texttt{U}'/\texttt{X}]\texttt{B}}}{\Delta \vdash \overline{\texttt{v}(\texttt{U},\texttt{U}')}}
$$

$$(\textsc{Decon-B})$$

$$
\frac{\overline{\texttt{v}} \geq var(\overline{\texttt{X}};\texttt{R}) \qquad \Delta \vdash \overline{[\texttt{U}/\texttt{X}]\texttt{R}} <: \overline{[\texttt{U}'/\texttt{X}]\texttt{R}}}{\Delta \vdash \overline{\texttt{v}(\texttt{U},\texttt{U}')}}
$$

$$(\textsc{Decon-R})$$

The following example applies Decon-B to subtyping relationship (10.1). The wildcard types in that relationship were translated to their existential versions (e.g., `List<? extends T>` is translated to $\exists \texttt{Y} \to \texttt{[}\bot\texttt{-T]}.\texttt{List<Y>}$).

$$
\frac{+ \geq var(\texttt{X}; \exists \texttt{Y} \to \texttt{[}\bot\texttt{-X]}.\texttt{List<Y>}) = + \qquad \Delta \vdash \texttt{[T/X]}\exists \texttt{Y} \to \texttt{[}\bot\texttt{-X]}.\texttt{List<Y>} <: \texttt{[U/X]}\exists \texttt{Y} \to \texttt{[}\bot\texttt{-X]}.\texttt{List<Y>}}{\Delta \vdash +(\texttt{T},\texttt{U}) \equiv \texttt{T} <: \texttt{U}} \textsc{Decon-B}
$$

10.2.4.3 Boundary Analysis and Deconstructing Constraints

Unfortunately, the boundary analysis described in Section 6.7.1. could no longer be performed as a preprocessing step with deconstructing generalized constraints. For example, assuming that `Itr` is covariant and that *no bounds* are ignored, then `Itr<?`

`super T> <: Itr<? super U>` $\implies$ `U <: T`. However, if the subtype relation can ignore useless bounds, then we cannot make safe assumptions about types in useless bounds.

The previous implication would no longer be safe to assume. If the subtype relation ignores useless bounds, then by the joining of use-site and definition-site variances, lower bounds in instantiations of covariant generics are ignored. For example, `Itr<? super T>` $\equiv$ `Itr<?>` because applying the contravariant use-site annotation '`?` `super`' to a covariant generic `Itr` returns a bivariant version of the generic. Recall that, `Itr<? super T>` $\equiv$ `Itr<?>` denotes `Itr<? super T> <: Itr<?>` $\wedge$ `Itr<?> <: Itr<? super T>`. Since `T` was arbitrary, we could establish `Itr<? super String> <: Itr<?> <: Itr<? super Dog>`. Applying rule DECON-B to this relationship would derive `String <: Dog`, which is not safe.

In order for the addition of the DECON rules to be safe, *var* must not be *conservative*. *var* must compute the greatest possible variance safe for the subtype relation. For example, suppose the subtype relation ignores useless bounds and the following safe relationship is derivable:

$$\exists Y \to \texttt{[String-Object].Itr<Y>} <: \exists Y \to \texttt{[Dog-Object].Itr<Y>}$$

Then it must *not* be the case that $var(\texttt{X}; \exists Y \to \texttt{[X-Object].Itr<Y>}) < *$, since `String` and `Dog` are not subtype related. The judgment $*(\texttt{T}, \texttt{U})$ holds for any two types `T` and `U`. Therefore, deriving $*(\texttt{String}, \texttt{Dog})$ does not add any additional subtype assumptions.

In order for the deconstruction rules to be safe, the following converse of the subtype lifting lemma must hold for the subtype relation without the DECON rules.

$$\Delta \vdash \overline{[\texttt{U}/\texttt{X}]}\,\texttt{T} <: \overline{[\texttt{U}'/\texttt{X}]}\,\texttt{T} \textbf{ and } \overline{\texttt{v}} \leq var(\overline{\texttt{X}}; \texttt{T}) \implies \Delta \vdash \overline{\texttt{v}(\texttt{U}, \texttt{U}')} \qquad (10.2)$$

This implication does not hold for the VarJ calculus by the following counter example. $var(\texttt{X}; \exists Y \to \texttt{[X-Object].Itr<Y>}) = -$, and $-(\texttt{String}, \texttt{Dog})$, which is equivalent

to Dog <: String, is not derivable. The following subtype relationship is derivable in VarJ:

$$[\texttt{String/X}]\big(\exists \texttt{Y} \to [\texttt{X-Object}].\texttt{Itr<Y>}\big)$$

$$= \exists \texttt{Y} \to [\texttt{String-Object}].\texttt{Itr<Y>}$$

$$<: \exists \texttt{Y} \to [\bot\texttt{-Object}].\texttt{Itr<Object>}$$

$$<: \exists \emptyset.\texttt{Itr<Object>}$$

$$<: \exists \texttt{Y} \to [\texttt{Dog-Object}].\texttt{Itr<Y>}$$

$$= [\texttt{Dog/X}]\big(\exists \texttt{Y} \to [\texttt{X-Object}].\texttt{Itr<Y>}\big)$$

The judgment above violates implication (10.2).

Given the above discussion, we describe two possible language design options for supporting generalized constraints with both definition- and use-site variance.

1. The more complex option is to have a subtype relation that ignores useless bounds, which also requires a more complex definition of *var* that ignores useless bounds. Although ignoring useless bounds allows more subtyping, performing boundary analysis and deconstructing generalized constraints would impose the burden of understanding boundary analysis on the programmer. For example, in order to determine if the compiler will deduce U <: T from the constraint, $\exists \texttt{Y} \to [\texttt{T-Object}].\texttt{Itr<Y>} <: \exists \texttt{Y} \to [\texttt{U-Object}].\texttt{Itr<Y>}$, the programmer would need to determine if T and U are useless bounds.

2. The simpler option is to have a subtype relation that never allows narrowing the range of an existential type variable in the supertype. This does not support as much subtyping as the first option. We expect programmers to find such a subtype relation easier to understand. Furthermore, since type bounds are specified by programmers, a programmer may expect the compiler to draw conclusions about the specified bounds.

10.2.5 Proofs of Language Properties

Formal language definitions are detailed enough to support rigorous proofs of language properties. Chapter 6 presented VarJ, a formal language modeling a subset of Java with both definition-site variance and wildcards. Appendix B contains the type soundness proof of VarJ.

Proofs of language properties are typically more detailed than mathematical proofs in other subjects. This occurs because type systems are defined in a *intuitionistic logic*, which is more generally known as *constructive logic* [44]. In this logic, a proposition P is true *if and only if* there exists a proof of P. Unlike classical logic, we cannot assume the law of the excluded middle. That is, we cannot assume $P \vee \neg P$. Constructive logic requires a proof of P or a proof of $\neg P$ in order to prove $P \vee \neg P$.

The set of true propositions in constructive logic is closed under a finite set of axioms. Since a proposition is true *iff* there is a proof of it, a proposition is true only if it can be derived from a finite set of axioms. This corresponds to how most compilers semantically analyze programs. For example, a compiler will assign a type to an expression only if there are rules from the language specification to support that claim.

Language properties can be proved by structural induction on the derivation of judgments. For example, the subtype lifting lemma in VarJ (Lemma 16 in Appendix B) is proved by structural induction on the derivation of $var(\mathtt{X}; \phi)$, where $\phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta$. For each rule that can derive the judgment, this type of proof shows that the lemma is true when that rule is applied. If the lemma is proved for all rules, then the lemma holds in general because a proposition can only be derived using only the specified rules.

Mechanized Proofs

The appendices contains rigorous proofs of many language properties. These proofs typically are long, tedious, and involve many cases. VarJ's type soundness proof in Appendix B is over 30 pages. As a result, there are many opportunities for errors in proofs. The length and detail of the these proofs makes it difficult for humans to find errors.

Proof assistants such as Twelf [57], Coq [45], and Isabelle [50] are software tools that enable one to encode theory and prove properties. Mechanized proofs are proofs that are written in a language understood by a proof assistant. Proof assistants are designed to find errors in proofs. When a mechanized proof is verified by a proof assistant, there is high confidence that the proof is indeed correct.

Proof assistants are a hybrid of proof checkers and automated theorem provers. A proof checker is software that verifies proofs written in a computer language. An automated theorem prover finds proofs of properties on their own. No automated theorem prover can automatically finds proofs of all valid theorems, by Gödel's incompleteness theorems [26]. A goal of using a proof assistant is therefore to have "easy" or routine steps of a proof be found automatically by the proof assistant. Only "difficult steps" should require the assistance of a human.

However, proof assistants are very complex and using them remains an esoteric skill. Although proof assistants provide some automation, mechanized proofs typically require specifying a lot more details than manually-written proofs in natural languages [16]. These additional details usually do not contribute significant conceptual gain. For example, Aydemir et al. [6] showed encoding variable binding in the Coq proof assistant requires a substantial amount of boilerplate code and requisite theorems for reasoning about substitution.

Rather than provide a longer proof than already given in Appendix B, we chose to provide a manually written but detailed proof that focuses on important reasons for why a theorem is true.

10.2.6 Barendregt's Variable Convention

The Barendregt variable convention [64] is followed in the definition of VarJ. Barendregt's statement [7, Page 26] of the covention is below:

> If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

In the context of the rule induction, the requirements for following this convention are the following:

1. All bound variables in a rule are distinct from all other variables.

2. No bound variable occurs free in a term in the conclusion.

Urban et al. [64] showed that following this convention ensures that predicates or relations defined in the type system are *equivariant*: Changing the name of any binder (bound variable) to a fresh name should not invalidate any derived judgment. They also showed that following this convention implies it is safe to assume in a proof that bound variables are distinct from any other variable occurring in the proof.

Urban et al. also showed that if the variable convention is not followed, properties resulting from the convention are not guaranteed. [64, Section 1] provides an example of an inductived-defined relation, where its definition violates the rules of the convention. In that work, a lemma over that relation is accompanied by a faulty proof. A counter example to the lemma is also provided. The mistake in the proof is that it assumes that a bound variable is distinct from all other variable names in the proof; that assumption is not valid for the relation in question.

The soundness proof of VarJ requires that this convention is followed. For example, judgment (2) of the proof of Lemma 9, $dom(\Delta) \cap \overline{\mathtt{X}} = \emptyset$, is derived using a consequence of Barendregt's variable convention. In that proof, type variables $\overline{\mathtt{X}}$ are binders. Because VarJ's type system rules follow Barendregt's variable convention, it is safe to assume that $\overline{\mathtt{X}}$ are distinct from other variable names such as names in $dom(\Delta)$.

In rule VAR-T from Figure 6.2, $dom(\Delta)$ are binders. The premise $\mathtt{X} \notin dom(\Delta)$ is explicitly stated and is needed to follow Barendregt's variable convention. As discussed in Section 6.2, this premise ensures that var is an equivariant relation. However, conditions for following variable conventions usually are not explicitly expressed in inference rules to make important premises more overt. Such side conditions are typically implicitly assumed in rules.

We illustrate the amount of detail added to rules by explicitly stating such side conditions. We use the standard simple `let` expression, `let` x `be` e_1 `in` e_2, as an example to show that even simple rules require significantly more detail. This `let` expression is evaluated by substituting e_1 for occurrences of variable x in the body expression e_2. The expression `let` x `be` e_1 `in` e_2 is written in higher-order abstract syntax [52] as `let`$(e_1,\ x.e_2)$. The term $x.e_2$ makes it clear that x is a bound variable and that the scope of x is e_2. The following typing and reduction (evaluation) rules below are given in many texts and do not state the side conditions for Barendregt's variable convention:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(e_1,\ x.e_2) : \tau_2} \ \text{T-LET}$$

$$\frac{}{\mathtt{let}(e_1,\ x.e_2) \mapsto [e_1/x]e_2} \ \text{R-LET}$$

The following rules are new versions of the previous ones with judgments (side conditions) that ensure that Barendregt's variable convention is followed. By convention, $fv(t)$ is the set of free variables in term t.

$$\frac{x \notin \mathit{fv}(\Gamma) \quad x \notin \mathit{fv}(e_1) \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}(e_1, \ x.e_2) : \tau_2} \quad \text{T-LET*}$$

$$\frac{x \notin \mathit{fv}(e_1)}{\mathtt{let}(e_1, \ x.e_2) \mapsto [e_1/x]e_2} \quad \text{R-LET*}$$

The starred rules explicitly specify side conditions that ensure that Barendregt's variable convention is followed. They also show that stating such conditions in an already complex type system such as VarJ would add a lot of technical details without much conceptual gain.

CHAPTER 11

CONCLUSION

We conclude this dissertation by summarizing the contributions of this work and discussing future work.

11.1 Summary of Contributions

The goal of this dissertation is to improve support for variance in programming languages. Variance is a programming language feature that enables programmers to safely write one piece of code that applies to multiple instantiations of a generic. This dissertation addresses the shortcomings of current variance mechanisms by providing theoretical and practical foundations for combining definition-site and use-site variance in a single language. This approach allows simpler type expressions than in languages with only use-site variance. Unlike languages only supporting definition-site variance, this approach does not require creating redundant types to facilitate variant versions of a type.

Variance is one of the least understood programming language features. Joshua Bloch, the architect of the Java collections library and author of a popular book [9] on how to use advanced features in Java, has heavily criticized Java wildcards for their complexity [8]. Although definition-site variance is arguably simpler than use-site variance, it was purposely avoided in the Dart programming language [36, Chapter 18].

This dissertation not only generalizes but also clarifies all previous related work. The transform operator $\otimes$ (Section 3.1) provides a clear way to reason about the

variance of nested types. The variance predicate *var* (Section 6.2) gives a high-level way to reason soundly about the variance of a type. Section 6.3 explains how to assign variances to positions in type definitions. Section 10.1 describes how the concepts developed in this dissertation simplify determining which operations are available to a variant type.

This dissertation also contributes practical foundations for supporting both definition- and use-site variance. Section 4.4 presents a constraint-solving algorithm that computes the most-general definition site variances that satisfy a system of constraint inequalities. The algorithm is efficient and runs in polynomial time with respect to the number of constraints. This work is the first to determine the most general definition-site variances in the face of recursive type definitions, which are ubiquitous in object-oriented programs. It also allows bivariant definition-site annotations, which are not supported in C#, Scala, or any other mainstream language. Section 3.3.6 showed that bivariant annotations would allow more generics to be variant.

Because we provide clear and practical foundations for understanding, implementing, and applying variance mechanism, there are plans to adopt ideas from this dissertation to mainstream languages. Mozilla is implementing the constraint-solving algorithm from 4.4 to support definition-site variance inference [48] in the Rust [49] programming language. A proposal to add definition-site variance to Java using ideas from this dissertation [58] was recently put forward in the Oracle Java language development forums.

Given these plans, we expect variance mechanisms to improve in mainstream languages. Better and easier-to-use variance mechanism should increase usage of variance in software designs. As a result, software libraries will be developed with interfaces that support greater reuse.

11.2 Future Work

This section discusses how the ideas of this dissertation can be further utilized.

Some directions for future work based on our results were already presented throughout this dissertation. Section 10.2.4 described how our approach to reasoning about variance may provide a general framework for reasoning about generalized constraints. Generalized constraints are lists of subtyping constraints between arbitrarily complex types. Emir et al. [24] showed how to support generalized constraints with limited type expressiveness. For example, that work did not support parametric types with use-site variance annotations. Section 10.2.4 discussed how generalized constraints with existential types could be supported. Moreover, that section described at a high-level how generalized constraints with types of any syntactic structure can be supported. Our approach supports this generalization because our notion of a variance of a type can applied to types of any structure.

Section 10.2.1 described and compared nominal subtyping and structural subtyping. Nominal and structural subtyping each have their own advantages and disadvantages. Currently, most programming languages do not support both nominal and structural subtyping. Malayeri et al. [42] investigated supporting both nominal and structural subtyping in a single language called Unity. However, Unity does not support generics. That work did not investigate variant subtyping between user-defined type constructors, i.e., functions that can take in types and return types.

Higher-order polymorphism [53, Chapter 30] refers to treating types as first class values, i.e., the ability to write a function that takes in types as inputs and returns a type. Hence, languages with higher-order polymorphism support user-defined type constructors. The standard type system $F^{\omega}_{<:}$ [53, Chapter 31] (called "F-omega-sub")

supports structural subtyping and higher-order polymorphism.[1] However, different applications of functions defined by users are never subtype-related.

Variance annotations could be supported in a language with higher-order polymorphism [53, Chapter 30]. For example, one could define a function `f` that takes in a type `T` and returns the product type `T` × `T`. Applying function `f` to type `T` in $F^\omega_{<:}$ is written as (`f T`). As discussed in Section 10.2.1, product types are covariant in their element types. Hence, it is safe to assume that `f` is covariant in its single argument. In this case, if `U` <: `T`, then, by the covariance of `f`, it is safe to assume (`f U`) <: (`f T`).

A definition-site variance annotation on the argument of function `f` could inform the type system and clients of the function that `f` is covariant in its argument type. The type system would need to ensure that the covariant definition-site variance annotation is safe according to the definition of `f`.

Another function `g` could use `f` and other user-defined type constructors in its definition. For example, suppose `h` is a type constructor that is declared to be contravariant in its argument. Function `g` can be defined to take in a type `T` as input and return type (`f (h (f T))`). Section 3.1 described how to reason about the variance of nested applications of type constructors using the transform operator $\otimes$. The definition-site variance of a type constructor transforms the variance of the type argument. Using this reasoning, it is easy to determine that the variance of `g`'s argument is contravariance: $\underbrace{+}_{f} \otimes (\underbrace{-}_{h} \otimes \underbrace{+}_{f}) = -$.

Hence, a language supporting both structural subtyping and variant subtyping on user-defined type constructors can be investigated using ideas from this dissertation. One approach to this investigation would be to extend $F^\omega_{<:}$ with definition-site variance annotations. These annotations label type arguments of functions.

---

[1] Functions that return types in $F^\omega_{<:}$ are also called *type abstractions*.

A language supporting nominal subtyping, structural subtyping, and variant subtyping on user-defined type constructors can be studied, for example, by extending the Unity calculus by Malayeri et al. [42] with generics and variant subtyping. Definition-site variances can be declared or inferred for generic type parameters in that language extension using ideas from this dissertation. Type expressions with use-site variance annotations, wildcards, or existential types can also be added to this language combination using ideas from this dissertation.

APPENDIX A

VARLANG SOUNDNESS

We prove the soundness of our treatment of *VarLang* sentences denotationally—
that is, by direct appeal to the original definition and axioms of use-site variance [33],
and not relative to a specific type system. At the same time, we want our proof to
apply to actual type systems. For this purpose, we try to state clearly the assumptions
we make for a type system to be able to use our approach.

Specifically, our proof is based on the following meaning of use-site variance, over
some subtyping lattice (i.e., subtyping is the partial order of the lattice):

- `C<+T>` $= \bigsqcup_{T' <: T}$ `C<T'>`

- `C<-T>` $= \bigsqcup_{T <: T'}$ `C<T'>`

- `C<*>` $= \bigsqcup_{T'}$ `C<T'>`

(The rules are presented for the case of a single-argument generic—for multiple ar-
guments the rules should be read to apply to the same position.) This meaning of
use-site variance is consistent with a types-as-sets treatment [33], hence the reader
can be assigning meaning using set operations in any universe of types that stand for
sets of values. The definitions essentially say that a variant type is equivalent to a
type encompassing all possible values of the appropriate invariant types. There are
two elements worth noting: First, our treatment assumes the existence of a bottom
element (since it is on a lattice). Second, the above treatment does not take into
account type identity: Use-site variant types are really pairs of a unique identifier
and their above denotation, i.e., two different occurrences of `C<+T>` are incompatible,

although they map to the same element of the lattice. We omit the identity aspect since it only makes the discussion more tedious without affecting our argument.

The above meaning of use-site variance yields the common variance properties:

- `C<T>` $<:$ `C<+T>` $<:$ `C<*>`

- `C<T>` $<:$ `C<-T>` $<:$ `C<*>`

- `T` $<:$ `T`$'$ $\Rightarrow$ `C<+T>` $<:$ `C<+T'>` $\wedge$ `C<-T'>` $<:$ `C<-T>`

(Note that every type expression `C<*T'>` refers to the same type, which we write `C<*>` when more convenient.)

The denotation of use-site variance is only half the story, however. The other important part of the semantic domain is the meaning of the variance of a position, i.e., a formalization of the concepts "type `T` appears covariantly/contravariantly/bi-variantly/invariantly in the definition of class `C`". In other words, using the definition of variance we can assign meaning to *VarLang* expressions of the form `C<v̄T>` but what about *VarLang* expressions of the form $\overline{Tv}$? To give such a definition we first introduce notation allowing variance annotations, `v`, to also define binary predicates on types:

- $+(T_1, T_2) = T_1 <: T_2$

- $-(T_1, T_2) = T_2 <: T_1$

- $o(T_1, T_2) = \textit{false}$

- $*(T_1, T_2) = \textit{true}$

Then, to have a type `T` appear in a position with variance `v` at the top level of the definition of class `C` means that if we were to replace the occurrence of `T` with a `T`$'$, such that `v(T`$'$`, T)` we would be defining a safe subtype of `C`. For instance, the reason it is safe to consider the return type of a class's method to be a covariant position is that

166

replacing the return type with a subtype produces a class definition that can safely be a subtype of the original class. This "meaning" of the variance of a position is a fairly common understanding, but we need to bind its components (e.g., subtyping, considering a module to represent a type definition in a real language, etc.) to specific languages and type systems.

Consider a real programming language JSCW (for "Java-Scala-C#-Whatever") to which we want to apply our typing framework. The JSCW language needs to have use-site variance annotations in the type vocabulary, but may or may not already support some use-site-variance-based reasoning—after all, this is what our approach adds. We first introduce the idea of a variance oracle.

Definition 1. A *variance oracle*, O, for a program P is a finite set of *oracular assertions* (or just *assertions*) of the form $var(\texttt{X})\texttt{T} = \texttt{v}$ or $var(\texttt{X})\texttt{C} = \texttt{v}$ (for a type $\texttt{T}$ or class name $\texttt{C}$), such that:

- There is a single assertion per type expression, $\texttt{T}$, and per class name, $\texttt{C}$.

- The oracle is closed: if it contains an assertion for a type expression, it also contains assertions for all its constituent type expressions and class names appearing in them. If the oracle contains an assertion for a class name, it also contains assertions for all type expressions appearing in the body of the class definition.

Definition 2. We say that a set of oracular assertions is *consistent* with program P and language JSCW iff taking the assertions as facts does not violate the soundness of JSCW. That is, the type system of JSCW can consult the assertions and infer subtyping accordingly: if $var(\texttt{X})\texttt{T} = \texttt{v}$, (resp. $var(\texttt{X})\texttt{C} = \texttt{v}$) then for any types $\texttt{T}_1$ and $\texttt{T}_2$ for which $\texttt{v}(\texttt{T}_1, \texttt{T}_2)$ (recall our treatment of variance annotations as binary predicates), the type system can infer that $\texttt{T}[\texttt{T}_1/\texttt{X}] <: \texttt{T}[\texttt{T}_2/\texttt{X}]$ (resp. $\texttt{C}[\texttt{T}_1/\texttt{X}] <: \texttt{C}[\texttt{T}_2/\texttt{X}]$). ($\texttt{T}[\texttt{T}'/\texttt{X}]$ is defined as the type expression produced by substituting $\texttt{T}'$ for $\texttt{X}$ in $\texttt{T}$. $\texttt{C}[\texttt{T}_1/\texttt{X}]$ is the

name of a class with the same definition (body) as C, after substituting T′ for X in the body.) If the JSCW type system enhanced with the set of oracular assertions in this way still respects all its soundness properties for program P (e.g., that if P is well-typed it will cause no semantic violation), then the set of assertions is consistent with P and JSCW.

Next, we can state more precisely the mapping between a program in JSCW and its corresponding sentence in *VarLang*.

Definition 3. A *VarLang* sentence S *models* a program P of language JSCW (possibly enhanced with oracle O) iff:

- Modules in S and classes in P are in one-to-one correspondence. (We assume a module has the same name as the corresponding class.)

- For every class C<X> in P and every type expression, T, containing X in the definition of C, if the occurrence of the type expression may affect the variance of C, then sentence S contains a corresponding member T inside `module C<X̄>` { ... Tv... }. Conversely, every member T of the definition of a module C in S has a corresponding syntactic source in the definition of C in P. That is, C in program P can be defined as a JSCW syntax tree with a hole, *Cdef*[∘], where ∘ appears once in the syntax tree *Cdef*[∘] and *Cdef*[T] (i.e., replacing ∘ with T in the tree) is equal to the definition of C in program P.

- The suffix variance annotations in S (i.e., the descriptions of variance positions) are consistent with P's semantics under the subtyping relation and semantics of language JSCW. That is, if S contains a `module C<X̄>` { ... Tv... } and in program P it is v(T′, T) for some type T′ (possibly inferred with the help of oracle O) then it would not violate the soundness of the type system of JSCW to have *Cdef*[T′] <: *Cdef*[T].

This definition captures the obligations of language JSCW and its mapping to *VarLang*: the mapping has to always produce *VarLang* sentences that fully and accurately describe the variance information of the JSCW program (i.e., correctly describe the variance of each position inside a class definition).

Now we can state our soundness theorem, essentially as a meta-theorem, conditional on a sentence S modeling a program P.

Theorem 3. Consider a *VarLang* sentence S that models a JSCW program P, and a set of variance assertions of the form $var(\mathtt{X})\mathtt{T} = \mathtt{v}$ and $var(\mathtt{X})\mathtt{C} = \mathtt{v}$ that satisfy all constraints generated by the translation of sentence S. The following properties hold:

- The set of variance assignments forms an oracle O.

- The module assertions, $var(\mathtt{X})\mathtt{C} = \mathtt{v}$, are consistent with P and JSCW.

- The type expression assertions, $var(\mathtt{X})\mathtt{T} = \mathtt{v}$, are consistent with each other and with the module assertions, under the definition of use-site variance. That is, any subtyping that can be inferred by consulting the oracle's type expression assertions is also inferrable directly from the definition of use-site variance.

In other words, the *soundness* claim of our approach is that it only computes variances that are permitted under the definition of use-site variance. The expectation is that the type system of the unknown language JSCW will remain sound under such sound-in-principle oracular assertions.

Proof. Solving the constraints from the translation of a *VarLang* sentence S results in an oracle O, since the translation of S assigns exactly one variance value to every type expression and module name in sentence S (as can be seen by comparing the translation rules with the grammar of S).

We next prove that the assertions of O are consistent with the definition of use-site variance. We do this by considering what subtypings can be inferred by consulting the oracle's assertions.

The theorem trivially holds for rules 4.2 and 4.3. Since X does not occur in the type expression of the rule, any substitution of X by two types related by variance v will result in a subtype (the original type expression itself). Thus, we can assign any variance to $var(\texttt{X})\texttt{C<>}$ and to $var(\texttt{X})\texttt{Y}$, exactly as these rules prescribe.

Rule 4.4 is similarly easy. The rule says that oracle O can assign $var(\texttt{X})\texttt{X}$ to at most $+$, i.e., to either $+$ or o. Assigning o clearly produces a consistent oracle (since the antecedent of the definition of consistency is not satisfied: $o(\texttt{T}_1, \texttt{T}_2) = \textit{false}$ for every $\texttt{T}_1$ and $\texttt{T}_2$). So, we only need to consider $+$, which results in making the definition of consistency a tautology: $+(\texttt{T}_1, \texttt{T}_2) = \texttt{T}_1 <: \texttt{T}_2$, and if the type system can prove this subtyping then substituting the two types for X will clearly result in sound subtyping.

We covered the above trivial cases in detail so that the flow of the argument becomes clear. The real issue, however, is to prove the theorem for rule 4.5. This rule is the essence of our variance reasoning. It effectively says that the transform operator correctly builds the variance of composite type expressions from that of component type expressions, and that use-site variance annotations are tantamount to a join in the lattice.

The rule introduces N constraints, where N is the number of type parameters of C. We reason about each constraint separately. If X does not occur in the i-th type $\texttt{T}_i$ that parameterizes C then no unsoundness is introduced by any type assignment consistent with the rule, by the same reasoning as for rules 4.2 and 4.3, above. Therefore we only consider the case where X occurs in $\texttt{T}_i$, and, thus, a substitution of X by a subtype may introduce unsoundness. To simplify the presentation, we subsequently write all type expressions for the case of a single-type-parameter generic. In the case of multiple type-parameters, the argument should be understood to apply to the same parameter position of any two expressions.

We consider all cases for the three variables on the right hand side of the constraint: v, $var(\texttt{Y})\texttt{C}$, $var(\texttt{X})\texttt{T}$:

- $v = *$: the r.h.s. of the constraint is $*$, hence O may assign any variance to $var(\mathtt{X})\mathtt{C}\texttt{<}*\texttt{>}$. Thus, the type system enhanced with oracle O can infer subtyping between type expressions substituting $\mathtt{X}$ with any two types. This is sound, since $\mathtt{C}\texttt{<}*\mathtt{T}[\mathtt{T}_1/\mathtt{X}]\texttt{>} <: \mathtt{C}\texttt{<}*\mathtt{T}[\mathtt{T}_2/\mathtt{X}]\texttt{>}$, by the standard variance properties.

- $var(\mathtt{Y})\mathtt{C} = *$: similar argument as above. The right hand side of the rule is $*$, but any variance is safe for a composite type expression on $\mathtt{C}$ if the variance of $\mathtt{C}$ is $*$.

- $var(\mathtt{X})\mathtt{T} = o$ and either $v = +, var(\mathtt{Y})\mathtt{C} = +$, or $v = +, var(\mathtt{Y})\mathtt{C} = o$, or $v = -, var(\mathtt{Y})\mathtt{C} = -$, or $v = -, var(\mathtt{Y})\mathtt{C} = o$, or $v = o, var(\mathtt{Y})\mathtt{C} = +$, or $v = o, var(\mathtt{Y})\mathtt{C} = -$: the r.h.s. of the constraint is o, thus oracle O cannot introduce unsoundness, since no subtyping substitution is allowed by the assertion.

- $v = o$ and $var(\mathtt{Y})\mathtt{C} = o$: always safe since constraint has a r.h.s. of o.

- $var(\mathtt{X})\mathtt{T} = *$, $v \neq *$, and either $var(\mathtt{Y})\mathtt{C} = +$ or $var(\mathtt{Y})\mathtt{C} = -$: the r.h.s. of the constraint is $*$ and oracle O can assign any variance to $var(\mathtt{X})\mathtt{C}\texttt{<}v\mathtt{T}\texttt{>}$. $v \neq *$ and $var(\mathtt{Y})\mathtt{C} = +$ or $var(\mathtt{Y})\mathtt{C} = -$ means that $v \sqcup var(\mathtt{Y})\mathtt{C} = +$ or $v \sqcup var(\mathtt{Y})\mathtt{C} = -$. But for any two types $\mathtt{T}_1$ and $\mathtt{T}_2$, $\mathtt{T}[\mathtt{T}_1/\mathtt{X}] <: \mathtt{T}[\mathtt{T}_2/\mathtt{X}]$ (and vice versa), since $var(\mathtt{X})\mathtt{T} = *$. Hence we have $\mathtt{C}\texttt{<}\mathtt{T}[\mathtt{T}_1/\mathtt{X}]\texttt{>} <: \mathtt{C}\texttt{<}\mathtt{T}[\mathtt{T}_2/\mathtt{X}]\texttt{>}$ (because of the co-variance or contravariance of $var(\mathtt{Y})\mathtt{C}$ taken in either direction) and finally both $\mathtt{C}\texttt{<-}\mathtt{T}[\mathtt{T}_1/\mathtt{X}]\texttt{>} <: \mathtt{C}\texttt{<-}\mathtt{T}[\mathtt{T}_2/\mathtt{X}]\texttt{>}$ and $\mathtt{C}\texttt{<+}\mathtt{T}[\mathtt{T}_1/\mathtt{X}]\texttt{>} <: \mathtt{C}\texttt{<+}\mathtt{T}[\mathtt{T}_2/\mathtt{X}]\texttt{>}$ (by the standard variance properties). Therefore, the assertion of O follows from the variance definition.

- $var(\mathtt{X})\mathtt{T} = *$, $var(\mathtt{Y})\mathtt{C} \neq o$, and either $v = +$ or $v = -$: the r.h.s. of the constraint is $*$ and oracle O can assign any variance to $var(\mathtt{X})\mathtt{C}\texttt{<}v\mathtt{T}\texttt{>}$. As in the previous case, $v \sqcup var(\mathtt{Y})\mathtt{C} = +$ or $v \sqcup var(\mathtt{Y})\mathtt{C} = -$. But for any two

types T_1 and T_2, $T[T_1/X] <: T[T_2/X]$ (and vice versa), since $var(X)T = *$, hence
`C<-T[`T_1`/X]>` $<:$ `C<-T[`T_2`/X]>` and `C<+T[`T_1`/X]>` $<:$ `C<+T[`T_2`/X]>` (by the standard variance properties takein in either direction). Therefore, the assertion of O follows from the variance definition.

We will use the same argument structure for all the individual cases below, which contain the core (i.e., the hardest cases) of the proof, but due to the length of the reasoning we will not belabor the inference steps, assuming that the reader understands the argument flow from the earlier cases.

- `v` $= +$, $var(Y)$`C` $= +$, $var(X)$`T` $= +$: constraint r.h.s is $+$. Soundness preserved since:

$$T_1 <: T_2 \Rightarrow \qquad \text{(by variance of T)}$$
$$T[T_1/X] <: T[T_2/X] \Rightarrow \qquad \text{(by variance properties)}$$
$$\texttt{C<+T[}T_1\texttt{/X]>} <: \texttt{C<+T[}T_2\texttt{/X]>}$$

- `v` $= +$, $var(Y)$`C` $= +$, $var(X)$`T` $= -$: constraint r.h.s is $-$. Soundness preserved since:

$$T_1 <: T_2 \Rightarrow \qquad \text{(by variance of T)}$$
$$T[T_2/X] <: T[T_1/X] \Rightarrow \qquad \text{(by variance properties)}$$
$$\texttt{C<+T[}T_2\texttt{/X]>} <: \texttt{C<+T[}T_1\texttt{/X]>}$$

- `v` $= +$, $var(Y)$`C` $= -$: constraint r.h.s is $*$. Soundness preserved since:

$$\bot <: \texttt{T} \Rightarrow \qquad \text{(by variance properties)}$$
$$\texttt{C<+}\bot\texttt{>} <: \texttt{C<+T>}$$

172

But also, for any type T':

$$\bot <: \mathtt{T'} \Rightarrow \qquad\qquad (\mathtt{C}\text{ contravariant})$$

$$\mathtt{C<T'>} <: \mathtt{C<\bot>}$$

Therefore:

$$\mathtt{C<+T>} = \qquad\qquad (\text{by variance def})$$

$$\bigsqcup_{\mathtt{T'}<:\mathtt{T}} \mathtt{C<T'>} <: \qquad\qquad (\text{by above})$$

$$\mathtt{C<\bot>} <: \qquad\qquad (\text{by variance properties})$$

$$\mathtt{C<+\bot>}$$

Hence, all `C<+T>` are always subtype-related, i.e., have type `C<*>`.

- $\mathtt{v} = +, var(\mathtt{Y})\mathtt{C} = o, var(\mathtt{X})\mathtt{T} = +$: constraint r.h.s is $+$. Soundness preserved since:

$$\mathtt{T_1} <: \mathtt{T_2} \Rightarrow \qquad\qquad (\text{by variance of }\mathtt{T})$$

$$\mathtt{T[T_1/X]} <: \mathtt{T[T_2/X]} \Rightarrow \qquad (\text{by variance properties})$$

$$\mathtt{C<+T[T_1/X]>} <: \mathtt{C<+T[T_2/X]>}$$

- $\mathtt{v} = +, var(\mathtt{Y})\mathtt{C} = o, var(\mathtt{X})\mathtt{T} = -$: constraint r.h.s is $-$. Soundness preserved since:

$$\mathtt{T_1} <: \mathtt{T_2} \Rightarrow \qquad\qquad (\text{by variance of }\mathtt{T})$$

$$\mathtt{T[T_2/X]} <: \mathtt{T[T_1/X]} \Rightarrow \qquad (\text{by variance properties})$$

$$\mathtt{C<+T[T_2/X]>} <: \mathtt{C<+T[T_1/X]>}$$

173

- $v = -, var(\texttt{Y})\texttt{C} = +$: constraint r.h.s is $*$. Soundness preserved since:

$$\texttt{T} <: \top \Rightarrow \qquad \text{(by variance properties)}$$

$$\texttt{C<-}\top\texttt{>} <: \texttt{C<-T>}$$

But also, for any type $\texttt{T}$':

$$\texttt{T}' <: \top \Rightarrow \qquad \text{(\texttt{C} covariant)}$$

$$\texttt{C<T'>} <: \texttt{C<}\top\texttt{>}$$

Therefore:

$$\texttt{C<-T>} = \qquad \text{(by variance def)}$$

$$\bigsqcup_{\texttt{T}<:\texttt{T}'} \texttt{C<T'>} <: \qquad \text{(by above)}$$

$$\texttt{C<}\top\texttt{>} <: \qquad \text{(by variance properties)}$$

$$\texttt{C<-}\top\texttt{>}$$

Hence, all $\texttt{C<-T>}$ are the same type, i.e., $\texttt{C<*>}$.

- $v = -, var(\texttt{Y})\texttt{C} = -, var(\texttt{X})\texttt{T} = +$: constraint r.h.s is $-$. Soundness preserved since:

$$\texttt{T}_1 <: \texttt{T}_2 \Rightarrow \qquad \text{(by variance of \texttt{T})}$$

$$\texttt{T}[\texttt{T}_1/\texttt{X}] <: \texttt{T}[\texttt{T}_2/\texttt{X}] \Rightarrow \qquad \text{(by variance properties)}$$

$$\texttt{C<-T[T}_2\texttt{/X]>} <: \texttt{C<-T[T}_1\texttt{/X]>}$$

- $v = -, var(\texttt{Y})\texttt{C} = -, var(\texttt{X})\texttt{T} = -$: constraint r.h.s is $+$. Soundness preserved since:

$$\texttt{T}_1 <: \texttt{T}_2 \Rightarrow \qquad \qquad \text{(by variance of \texttt{T})}$$

$$\texttt{T}[\texttt{T}_2/\texttt{X}] <: \texttt{T}[\texttt{T}_1/\texttt{X}] \Rightarrow \qquad \text{(by variance properties)}$$

$$\texttt{C<-T[T}_1\texttt{/X]>} <: \texttt{C<-T[T}_2\texttt{/X]>}$$

- $v = -, var(\texttt{Y})\texttt{C} = o, var(\texttt{X})\texttt{T} = +$: constraint r.h.s is $-$. Soundness preserved since:

$$\texttt{T}_1 <: \texttt{T}_2 \Rightarrow \qquad \qquad \text{(by variance of \texttt{T})}$$

$$\texttt{T}[\texttt{T}_1/\texttt{X}] <: \texttt{T}[\texttt{T}_2/\texttt{X}] \Rightarrow \qquad \text{(by variance properties)}$$

$$\texttt{C<-T[T}_2\texttt{/X]>} <: \texttt{C<-T[T}_1\texttt{/X]>}$$

- $v = -, var(\texttt{Y})\texttt{C} = o, var(\texttt{X})\texttt{T} = -$: constraint r.h.s is $+$. Soundness preserved since:

$$\texttt{T}_1 <: \texttt{T}_2 \Rightarrow \qquad \qquad \text{(by variance of \texttt{T})}$$

$$\texttt{T}[\texttt{T}_2/\texttt{X}] <: \texttt{T}[\texttt{T}_1/\texttt{X}] \Rightarrow \qquad \text{(by variance properties)}$$

$$\texttt{C<-T[T}_1\texttt{/X]>} <: \texttt{C<-T[T}_2\texttt{/X]>}$$

- $v = o, var(\texttt{Y})\texttt{C} = +, var(\texttt{X})\texttt{T} = +$: constraint r.h.s is $+$. Soundness preserved since:

$$\texttt{T}_1 <: \texttt{T}_2 \Rightarrow \qquad \qquad \text{(by variance of \texttt{T})}$$

$$\texttt{T}[\texttt{T}_1/\texttt{X}] <: \texttt{T}[\texttt{T}_2/\texttt{X}] \Rightarrow \qquad \text{(by variance of \texttt{C})}$$

$$\texttt{C<T[T}_1\texttt{/X]>} <: \texttt{C<T[T}_2\texttt{/X]>}$$

175

- $\mathtt{v} = o, var(\mathtt{Y})\mathtt{C} = +, var(\mathtt{X})\mathtt{T} = -$: constraint r.h.s is $-$. Soundness preserved since:

$$\mathtt{T}_1 <: \mathtt{T}_2 \Rightarrow \qquad \text{(by variance of } \mathtt{T})$$

$$\mathtt{T}[\mathtt{T}_2/\mathtt{X}] <: \mathtt{T}[\mathtt{T}_1/\mathtt{X}] \Rightarrow \qquad \text{(by variance of } \mathtt{C})$$

$$\mathtt{C\texttt{<}T[T_2/X]\texttt{>}} <: \mathtt{C\texttt{<}T[T_1/X]\texttt{>}}$$

- $\mathtt{v} = o, var(\mathtt{Y})\mathtt{C} = -, var(\mathtt{X})\mathtt{T} = +$: constraint r.h.s is $-$. Soundness preserved since:

$$\mathtt{T}_1 <: \mathtt{T}_2 \Rightarrow \qquad \text{(by variance of } \mathtt{T})$$

$$\mathtt{T}[\mathtt{T}_1/\mathtt{X}] <: \mathtt{T}[\mathtt{T}_2/\mathtt{X}] \Rightarrow \qquad \text{(by variance of } \mathtt{C})$$

$$\mathtt{C\texttt{<}T[T_2/X]\texttt{>}} <: \mathtt{C\texttt{<}T[T_1/X]\texttt{>}}$$

- $\mathtt{v} = o, var(\mathtt{Y})\mathtt{C} = -, var(\mathtt{X})\mathtt{T} = -$: constraint r.h.s is $+$. Soundness preserved since:

$$\mathtt{T}_1 <: \mathtt{T}_2 \Rightarrow \qquad \text{(by variance of } \mathtt{T})$$

$$\mathtt{T}[\mathtt{T}_2/\mathtt{X}] <: \mathtt{T}[\mathtt{T}_1/\mathtt{X}] \Rightarrow \qquad \text{(by variance of } \mathtt{C})$$

$$\mathtt{C\texttt{<}T[T_1/X]\texttt{>}} <: \mathtt{C\texttt{<}T[T_2/X]\texttt{>}}$$

This concludes the proof for rule 4.5, which establishes that all oracle assertions for *type expressions* are consistent with the definition of use-site variance. The last piece is to prove that all module assertions, $var(\mathtt{X})\mathtt{C} = \mathtt{v}$, are consistent with program P and JSCW. Such assertions are solutions of constraints generated by rule 4.1, which combines the constraints on type expressions with the variance of their position, to

derive the variance of an entire module definition. The consistency of the assertions is based on the theorem's assumption that sentence S models program P.

We want to show the consistency of a module assertion with P and JSCW, which means that treating the assertion as a fact will not cause an unsound inference. Since constraints generated by rule 4.1 have the variance of an entire module on their left-hand-side, consider two substitutions $\texttt{C}[\texttt{T}_1/\texttt{X}]$ and $\texttt{C}[\texttt{T}_2/\texttt{X}]$. These substitutions consist of replacing $\texttt{X}$ wherever it occurs in the definition of $\texttt{C}$. Consider one such occurrence in a type expression$\texttt{T}$, contained in $\texttt{module C<}\overline{\texttt{X}}\texttt{> \{ ... Tv... \}}$. Define $Cdef[\circ]$, to be the syntax tree of the definition of class $\texttt{C}$ with a hole in place of $\texttt{Tv}$. We again consider all possible cases for the variables $\texttt{v}, var(\texttt{X})\texttt{T}$ on the right hand side of the corresponding constraint. The proof is case-by-case analogous to that for rule 4.5, above, specialized for $var(\texttt{Y})\texttt{C} = o$. We discuss one sample case, $\texttt{v} = +, var(\texttt{X})\texttt{T} = -$, which results in a constraint with an upper bound of $-$. For the rest of the cases, one only needs to adapt the argument below based on the corresponding case of rule 4.5. We have:

$$\texttt{T}_1 <: \texttt{T}_2 \Rightarrow \qquad\qquad\qquad \text{(by variance of } \texttt{T})$$

$$\texttt{T}[\texttt{T}_2/\texttt{X}] <: \texttt{T}[\texttt{T}_1/\texttt{X}] \Rightarrow \qquad \text{(since } S \text{ models } P \text{ and } \texttt{v} = +)$$

$$Cdef[\texttt{T}_2] <: Cdef[\texttt{T}_1]$$

By covering all cases and getting the same result for all constraints generated by rule 4.1, we can combine the results for the individual $Cdef[\circ]$s and show that $\texttt{C}[\texttt{T}_1/\texttt{X}]$ and $\texttt{C}[\texttt{T}_2/\texttt{X}]$ are related in the appropriate way (since the variance of $\texttt{C}$ has to satisfy all constraints). Hence, module assertions in O are consistent with P and JSCW, and this concludes the proof. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

APPENDIX B

PROOF OF VARJ SOUNDNESS

The numbering of the lemmas in this appendix does not correspond to that in the main text. Each lemma also appearing in the main text is clearly labeled with both numbers.

Lemma 9 (Widening Range Gives Supertype)**.** If

    a. $\Delta \vdash \overline{A_L <: B_L}$

    b. $\Delta \vdash \overline{B_U <: A_U}$

   Then

    1. $\Delta \vdash \exists \overline{X \to [B_L\text{--}B_U]}.N <: \exists \overline{X \to [A_L\text{--}A_U]}.N$

Proof.   1. $\overline{X}$ are binders.

    2. $dom(\Delta) \cap \overline{X} = \emptyset$, by Barendregt.

    3. $\Delta, \overline{X \to [B_L\text{--}B_U]} \vdash \overline{A_L <: B_L}$, applied weakening lemma to (2) and (a).

    4. $\Delta, \overline{X \to [B_L\text{--}B_U]} \vdash \overline{B_U <: A_U}$, applied weakening lemma to (2) and (b).

    5. $\Delta, \overline{X \to [B_L\text{--}B_U]} \vdash \overline{B_L <: \exists \emptyset.X}$, by ST-LBOUND.

    6. $\Delta, \overline{X \to [B_L\text{--}B_U]} \vdash \overline{\exists \emptyset.X <: B_U}$, by ST-UBOUND.

    7. $\Delta, \overline{X \to [B_L\text{--}B_U]} \vdash \overline{A_L <: \exists \emptyset.X}$, applied ST-TRAN to (3) and (5).

8. $\Delta, \overline{X \rightarrow [B_L\text{-}B_U]} \vdash \overline{\exists\emptyset.X <: A_U}$, applied ST-Tran to (6) and (4).

9. $fv(\exists\overline{X \rightarrow [A_L\text{-}A_U]}.N) \cap \overline{X} = \emptyset$

10. $\overline{X} \subseteq \Delta, \overline{X \rightarrow [B_L\text{-}B_U]}$

11. $\Delta, \overline{X \rightarrow [B_L\text{-}B_U]} \vdash \overline{[\overline{X}/X]A_L <: \exists\emptyset.X}$, by (7).

12. $\Delta, \overline{X \rightarrow [B_L\text{-}B_U]} \vdash \overline{\exists\emptyset.X <: [\overline{X}/X]A_U}$, by (8).

13. $\Delta \vdash \exists\overline{X \rightarrow [B_L\text{-}B_U]}.N <: \exists\overline{X \rightarrow [A_L\text{-}A_U]}.N$, applied ST-Pack to (9)–(12).

□   □

Definition 4.

$$\overline{X \rightarrow [B_L\text{-}B_U]} <: \overline{X \rightarrow [A_L\text{-}A_U]} \equiv \overline{A_L <: B_L} \textbf{ and } \overline{B_U <: A_U}$$

Lemma 10 (Corollary of Lemma 9)**.** If

   a. $\Delta_1 <: \Delta_2$

  Then

   1. $\Delta \vdash \exists\Delta_1.R <: \exists\Delta_2.R$

Proof. Trivial by applying Lemma 9 for the case $R = N$ and applying ST-Refl for the case $R = X$, since $R = X \implies \Delta_1 = \emptyset = \Delta_2$.   □   □

Lemma 11 (Variance Ordering Implies Subtyping)**.** Let v and w be variance annotations. If

   a. $v \leq w$

   b. $\Delta \vdash v(B; B')$

Then

1. $\Delta \vdash \mathtt{w}(\mathtt{B}, \mathtt{B}')$

Proof. Trivial by definition of $\mathtt{v}(\mathtt{B}; \mathtt{B}')$. □   □

Lemma 12 (Negation reverses subtyping). Let $\mathtt{v}$ and $\mathtt{w}$ be variance annotations. If

a. $\mathtt{w} = \neg v = - \otimes \mathtt{v}$

Then

1. $\mathtt{v}(\mathtt{B}, \mathtt{B}') \equiv \mathtt{w}(\mathtt{B}', \mathtt{B})$

Proof. This lemma is proved by inspection of the following table and seeing that for each data row, the two rightmost entries contain equivalent formulas.

$\mathtt{v}$	$\mathtt{w} = - \otimes \mathtt{v}$	$\mathtt{v}(\mathtt{B}, \mathtt{B}')$	$\mathtt{w}(\mathtt{B}', \mathtt{B})$
$+$	$-$	$+(\mathtt{B}, \mathtt{B}') \equiv \mathtt{B} <: \mathtt{B}'$	$-(\mathtt{B}', \mathtt{B}) \equiv \mathtt{B} <: \mathtt{B}'$
$-$	$+$	$-(\mathtt{B}, \mathtt{B}') \equiv \mathtt{B}' <: \mathtt{B}$	$+(\mathtt{B}', \mathtt{B}) \equiv \mathtt{B}' <: \mathtt{B}$
o	o	$o(\mathtt{B}, \mathtt{B}') \equiv \mathtt{B} <: \mathtt{B}' \wedge \mathtt{B}' <: \mathtt{B}$	$o(\mathtt{B}', \mathtt{B}) \equiv \mathtt{B}' <: \mathtt{B} \wedge \mathtt{B} <: \mathtt{B}'$
$*$	$*$	$*(\mathtt{B}, \mathtt{B}') \equiv true$	$o(\mathtt{B}', \mathtt{B}) \equiv true$

(□ □ appears to the right of the $-$/$+$ row)

Lemma 13 (Negation preserves inequality). Let $\mathtt{v}$ and $\mathtt{w}$ be variance annotations.

$$\mathtt{v} \leq - \otimes \mathtt{w} \iff - \otimes \mathtt{v} \leq \mathtt{w}$$

Proof. This lemma can be verified by inspection of a variance table that enumerates all possible variance value assignments for $\mathtt{v}$ and $\mathtt{w}$. □   □

Lemma 14 (Monotonicity of $\otimes$). If

a. $\mathtt{v}_1 \leq \mathtt{v}_2$

b. $\mathtt{v}_3 \leq \mathtt{v}_4$

180

Then

1. $v_1 \otimes v_3 \leq v_2 \otimes v_4$

Proof. This lemma can be verified by inspection of a variance table that enumerates all possible variance value assignments for $v_1, \ldots, v_4$. □ □

The next three lemmas are mutually dependent on each other. As a result, in the proofs of these lemmas, we can only apply these lemmas as "inductive hypotheses" so that the proofs "terminate". That is, we only apply these lemmas to *subterms* of terms in the premises of lemmas. More specifically, the only cases where Lemma 16 depends on Lemma 15 are in proof cases VAR-N and VAR-R, and in both cases they apply Lemma 15 to strict subterms of the relevant type term. Since Lemma 15's call to Lemma 17 does not alter the size of the type term, the proof terminates. (The anchor cases are then the other cases in Lemma 16.) The lemmas could have been combined into a single lemma, but this division makes the important reasoning more apparent and the proof clearer.

Lemma 15 (Subtype Lifting – Transform)**.** If

a. $\bar{v} \leq w \otimes var(\bar{X}; B)$

b. $\Delta \vdash \overline{v(U, U')}$

Then

1. $\Delta \vdash w(\overline{[U/X]}B, \overline{[U'/X]}B)$

Proof. Proof by case analysis on w:

Case 1: $w = *$.

1.1. $\Delta \vdash *(\overline{[U/X]}B, \overline{[U'/X]}B) = true$

□ for case 1

Case 2: $w = +$.

2.1. $\overline{\mathtt{v}} \leq + \otimes var(\overline{\mathtt{X}}; \mathtt{B}) = var(\overline{\mathtt{X}}; \mathtt{B})$

2.2. $\Delta \vdash \overline{[\mathtt{U}/\mathtt{X}]}\mathtt{B} <: \overline{[\mathtt{U}'/\mathtt{X}]}\mathtt{B}$, applied Lemma 17 to (2.1) and (b) gives.

2.3. $\Delta \vdash +(\overline{[\mathtt{U}/\mathtt{X}]}\mathtt{B}, \overline{[\mathtt{U}'/\mathtt{X}]}\mathtt{B})$, by (2.2).

□ for case 2

   Case 3: $\mathtt{w}=$ -.

3.1. $\overline{\mathtt{v}} \leq - \otimes var(\overline{\mathtt{X}}; \mathtt{B})$

3.2. $- \otimes \overline{\mathtt{v}} \leq var(\overline{\mathtt{X}}; \mathtt{B})$, applied Lemma 13 to (3.1).

3.3. Let $\overline{\mathtt{v}'} = - \otimes \overline{\mathtt{v}}$.

3.4. $\overline{\mathtt{v}'} \leq var(\overline{\mathtt{X}}; \mathtt{B})$, by (3.2) and (3.3).

3.5. $\Delta \vdash \overline{\mathtt{v}'(\mathtt{U}', \mathtt{U})}$, applied Lemma 12 to (b) and (3.3).

3.6. $\Delta \vdash \overline{[\mathtt{U}'/\mathtt{X}]}\mathtt{B} <: \overline{[\mathtt{U}/\mathtt{X}]}\mathtt{B}$, applied Lemma 17 to (3.4) and (3.5).

3.7. $\Delta \vdash -(\overline{[\mathtt{U}/\mathtt{X}]}\mathtt{B}, \overline{[\mathtt{U}'/\mathtt{X}]}\mathtt{B})$, by (3.6).

□ for case 3

   Case 4: $\mathtt{w}=$ o.

4.1. $\overline{\mathtt{v}} \leq o \otimes var(\overline{\mathtt{X}}; \mathtt{B})$

By Lemma 14, (4.1) implies:

4.2. $\overline{\mathtt{v}} \leq o \otimes var(\overline{\mathtt{X}}; \mathtt{B}) \leq + \otimes var(\overline{\mathtt{X}}; \mathtt{B})$, since $o \leq +$.

4.3. $\overline{\mathtt{v}} \leq o \otimes var(\overline{\mathtt{X}}; \mathtt{B}) \leq - \otimes var(\overline{\mathtt{X}}; \mathtt{B})$, since $o \leq -$.

Similar reasoning in cases 2 and 3 still apply.

4.4. $\Delta \vdash +(\overline{[\texttt{U/X}]\texttt{B}}, \overline{[\texttt{U}'/\texttt{X}]\texttt{B}})$, applied case 2 to (4.2).

4.5. $\Delta \vdash -(\overline{[\texttt{U/X}]\texttt{B}}, \overline{[\texttt{U}'/\texttt{X}]\texttt{B}})$, applied case 3 to (4.3).

4.6. $\Delta \vdash o(\overline{[\texttt{U/X}]\texttt{B}}, \overline{[\texttt{U}'/\texttt{X}]\texttt{B}})$, by (4.4) and (4.5).

□ for case 4

All cases covered. □ □

Lemma 16 (Subtype Lifting – Single Variable). If

  a. $\Delta \vdash v(\texttt{U}, \texttt{U}')$

and if

  b. $v \le var(\texttt{X}; \texttt{B})$

then

  1. $\Delta \vdash [\texttt{U/X}]\texttt{B} <: [\texttt{U}'/\texttt{X}]\texttt{B}$

and if

  c. $v \le var(\texttt{X}; \texttt{R})$

then

  2. $\Delta \vdash [\texttt{U/X}]\texttt{R} \prec: [\texttt{U}'/\texttt{X}]\texttt{R}$

and if

  d. $v \le var(\texttt{X}; \Delta')$

then

  3. $\Delta \vdash [\texttt{U/X}]\Delta' <: [\texttt{U}'/\texttt{X}]\Delta'$

Proof by induction on the derivation of $var(\texttt{X}; \phi)$, where $\phi ::= \texttt{B} \mid \texttt{R} \mid \Delta$.

Case: VAR-XY.

Proof. Trivial. □   □

   Case: VAR-B.

Proof. Trivial. □   □

   Case: VAR-XX.

Proof.   1. $\mathtt{v} \leq var(\mathtt{X};\mathtt{X}) = +$, by VAR-XX.

   2. $\Delta \vdash +(\mathtt{U},\mathtt{U}')$, applied Lemma 11 to (1) and (a).

   3. $\Delta \vdash \mathtt{U} <: \mathtt{U}'$, by (2).

   4. $\Delta \vdash [\mathtt{U}/\mathtt{X}]\mathtt{X} <: [\mathtt{U}'/\mathtt{X}]\mathtt{X}$, by (3).

   □   □

   Case: VAR-N.

Proof.   1. $VT(\mathtt{C}) = \overline{\mathtt{w}\mathtt{Y}}$, premise of VAR-N.

   2. $\mathtt{v} \leq var(\mathtt{X};\mathtt{C}{<}\overline{\mathtt{T}}{>}) = \prod_{i=1}^{n}\big(\mathtt{w}_i \otimes var(\mathtt{X};\mathtt{T}_i)\big)$, by VAR-N.

   3. $\overline{\mathtt{v} \leq \mathtt{w} \otimes var(\mathtt{X};\mathtt{T})}$, by (2).

   4. $\Delta \vdash \overline{\mathtt{w}([\mathtt{U}/\mathtt{X}]\mathtt{T}, [\mathtt{U}'/\mathtt{X}]\mathtt{T})}$, applied Lemma 15 to (3) and (a).

   5. $\Delta \vdash \mathtt{C}{<}\overline{[\mathtt{U}/\mathtt{X}]\mathtt{T}}{>} \prec: \mathtt{C}{<}\overline{[\mathtt{U}'/\mathtt{X}]\mathtt{T}}{>}$, applied SD-VAR to (1) and (4).

   □   □

   Case: VAR-R.

Proof. 1. $\mathtt{v} \le var(\mathtt{X}; \overline{\mathtt{Y} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}) = \prod_{j=1}^{n} \left[\left(- \otimes var(\mathtt{X}; \mathtt{B}_{Lj}) \right) \sqcap \left(+ \otimes var(\mathtt{X}; \mathtt{B}_{Uj}) \right) \right]$,

by VAR-R.

We prove this case by showing that for an arbitrary $i \in |\overline{\mathtt{Y} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}|$, we have both

$$\Delta \vdash [\mathtt{U}'/\mathtt{X}]\mathtt{B}_{Li} <: [\mathtt{U}/\mathtt{X}]\mathtt{B}_{Li}$$

and

$$\Delta \vdash [\mathtt{U}/\mathtt{X}]\mathtt{B}_{Ui} <: [\mathtt{U}'/\mathtt{X}]\mathtt{B}_{Ui}$$

2. $\mathtt{v} \le - \otimes var(\mathtt{X}; \mathtt{B}_{Li})$, by (1).

3. $\mathtt{v} \le + \otimes var(\mathtt{X}; \mathtt{B}_{Ui})$, by (1).

4. $\Delta \vdash -([\mathtt{U}/\mathtt{X}]\mathtt{B}_{Li}; [\mathtt{U}'/\mathtt{X}]\mathtt{B}_{Li}) \equiv [\mathtt{U}'/\mathtt{X}]\mathtt{B}_{Li} <: [\mathtt{U}/\mathtt{X}]\mathtt{B}_{Li}$, applied Lemma 15 to (2) and (a).

5. $\Delta \vdash +([\mathtt{U}/\mathtt{X}]\mathtt{B}_{Li}; [\mathtt{U}'/\mathtt{X}]\mathtt{B}_{Li}) \equiv [\mathtt{U}/\mathtt{X}]\mathtt{B}_{Li} <: [\mathtt{U}'/\mathtt{X}]\mathtt{B}_{Li}$, applied Lemma 15 to (3) and (a).

6. $\Delta \vdash \overline{[\mathtt{U}'/\mathtt{X}]\mathtt{B}_L <: [\mathtt{U}/\mathtt{X}]\mathtt{B}_L}$, since i was arbitrary.

7. $\Delta \vdash \overline{[\mathtt{U}/\mathtt{X}]\mathtt{B}_U <: [\mathtt{U}'/\mathtt{X}]\mathtt{B}_U}$, since i was arbitrary.

8. $\Delta \vdash [\mathtt{U}/\mathtt{X}]\overline{\mathtt{Y} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]} <: [\mathtt{U}'/\mathtt{X}]\overline{\mathtt{Y} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}$, by (6) and (7).

□   □

Case: VAR-T.

Proof. 1. $\mathtt{v} \le var(\mathtt{X}; \exists \Delta'.\mathtt{R}) = var(\mathtt{X}; \Delta') \sqcap var(\mathtt{X}; \mathtt{R})$, by VAR-T.

2. $\mathtt{v} \le var(\mathtt{X}; \Delta')$, by (1).

3. $\mathtt{v} \le var(\mathtt{X}; \mathtt{R})$, by (1).

4. $\Delta \vdash$ [U/X]Δ' <: [U'/X]Δ', applied inductive hypothesis to (a) and (2).

5. $\Delta \vdash$ [U/X]R $\prec:$ [U'/X]R, applied inductive hypothesis to (a) and (3).

6. $\Delta \vdash \exists$[U/X]Δ'.[U/X]R <: $\exists$[U'/X]Δ'.[U/X]R, applied Lemma 10 to (4).

7. $dom(\Delta')$ are binders, by (1).

8. $dom(\Delta') \cap dom(\Delta) = \emptyset$, by Barendregt.

9. Δ, [U'/X]$\Delta' \vdash$ [U/X]R $\prec:$ [U'/X]R, applied weakening lemma to (8) and (5).

10. $\Delta \vdash \exists$[U'/X]Δ'.[U/X]R $\sqsubset:$ $\exists$[U'/X]Δ'.[U'/X]R, applied SE-SD to (9).

11. $\Delta \vdash \exists$[U'/X]Δ'.[U/X]R <: $\exists$[U'/X]Δ'.[U'/X]R, applied ST-SE to (10).

12. $\Delta \vdash$ [U/X]$\exists\Delta'$.R <: $\exists$[U'/X]Δ'.[U/X]R, by (6) and X $\notin$ $dom(\Delta')$, premise of VAR-T case assumption.

13. $\Delta \vdash \exists$[U'/X]Δ'.[U/X]R <: [U'/X]$\exists\Delta'$.R, by (11) and X $\notin$ $dom(\Delta')$, premise of VAR-T case assumption.

14. $\Delta \vdash$ [U/X]$\exists\Delta'$.R <: [U'/X]$\exists\Delta'$.R, applied ST-TRAN.

□   □

All cases covered for Lemma 16.   □

Lemma 17 (Lemma 1 from Main Text – Subtype Lifting)**.** Let $\phi ::= $ B $|$ R $|$ Δ.   If

a. $\overline{v} \leq var(\overline{X}; \phi)$

b. $\Delta \vdash \overline{v(U; U')}$

Then

1. $\Delta \vdash \overline{[U/X]}\phi <: \overline{[U'/X]}\phi$

186

Proof. For an arbitrary $i \in \overline{\text{X}}$, we have $\Delta \vdash [\text{U}_i/\text{X}_i]\phi <: [\text{U}'_i/\text{X}_i]\phi$ by applying Lemma 16. Since i was arbitrary, we have $\Delta \vdash \overline{[\text{U}/\text{X}]}\phi <: \overline{[\text{U}'/\text{X}]}\phi$. $\quad\square$ $\qquad\qquad\square$

Lemma 18 (Lemma 2 from Main Text – Subtyping Specializes Field Type)**.** If

    a. $\vdash$ `class C<`$\overline{\text{vX} \rightarrow [...]}$`>` $\lhd$ `N`... *OK* and

    b. $\Delta \vdash$ `C<`$\overline{\text{T}}$`>` $\prec:$ `N`$'$ and

    c. *ftype*$(\text{f}; \text{N}') = \text{U}$

  Then

    1. *ftype*$(\text{f}; \text{C<}\overline{\text{T}}\text{>}) = \text{T}$

    2. $\Delta \vdash \text{T} <: \text{U}$

  Proof by induction on (b) and (c).

  Case (SD-VAR, FT-SUPER):

Proof.

$$
\frac{VT(\text{C}) = \overline{\text{vX}} \qquad \Delta \vdash \overline{\text{v}(\text{T},\text{U})}}{\Delta \vdash \text{C<}\overline{\text{T}}\text{>} \prec: \underbrace{\text{C<}\overline{\text{U}}\text{>}}_{\text{N}'}}
\quad (\text{SD-VAR})
$$

$$
\frac{CT(\text{C}) = \text{class C<}\overline{\text{vX} \rightarrow [\text{B}_L\text{-B}_U]}\text{>} \lhd \text{N } \{\ \overline{\text{S f;}}\ \ \overline{M}\ \} \qquad \text{f} \notin \overline{\text{f}}}{\textit{ftype}(\text{f}; \text{C<}\overline{\text{U}}\text{>}) = \textit{ftype}(\text{f}; \overline{[\text{U}/\text{X}]}\text{N})}
\quad (\text{FT-SUPER})
$$

    1. $VT(\text{C}) = \overline{\text{vX}}$, premise of SD-VAR.

    2. $\Delta \vdash \overline{\text{v}(\text{T},\text{U})}$, premise of SD-VAR.

    3. $\Delta \vdash \text{C<}\overline{\text{T}}\text{>} \prec: \text{C<}\overline{\text{U}}\text{>}$, conclusion of SD-VAR.

    4. `class C<`$\overline{\text{vX} \rightarrow [...]}$`>` $\lhd$ `N`..., premise of FT-SUPER.

5. $\mathtt{f} \notin \overline{\mathtt{f}}$, premise of FT-SUPER.

6. $\mathtt{U} = \mathit{ftype}(\mathtt{f}; \mathtt{C}\mathtt{<}\overline{\mathtt{U}}\mathtt{>}) = \mathit{ftype}(\mathtt{f}; [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{N})$, by conclusion of FT-SUPER and (c).

7. Let Δ' by the class type parameter environment $\overline{\mathtt{X} \to [\dots]}$.

8. $\Delta' \vdash \mathtt{N}$ *OK*, applied inversion of W-CLS to (a).

9. $\overline{v\mathtt{X}} \vdash \mathtt{N}$ *mono*, applied inversion of W-CLS to (a).

10. $\Delta \vdash [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N} \prec: [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{N}$, applied Lemma 17 to (9) and (2).

11. WLOG, assume $\mathtt{N} = \mathtt{D}\mathtt{<}\overline{\mathtt{V}}\mathtt{>}$

12. $CT(\mathtt{D}) = \mathtt{class\ D}\mathtt{<}\overline{\mathtt{wY} \to [\dots]}\mathtt{>} \lhd \mathtt{N}'' \dots$, applied inversion of W-N on (8).

13. $\vdash \mathtt{class\ D}\mathtt{<}\overline{\mathtt{wY} \to [\dots]}\mathtt{>} \lhd \mathtt{N}'' \dots$ *OK*, every class in CT is wellformed.

   Applying inductive hypothesis to (13), (10), and (6) gives the next two judgments:

14. $\mathit{ftype}(\mathtt{f}; [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N}) = \mathtt{T}$

15. $\Delta \vdash \mathtt{T} <: \mathtt{U}$

16. $\mathit{ftype}(\mathtt{f}; \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{ftype}(\mathtt{f}; [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N}) = \mathtt{T}$, applied FT-SUPER to (4) and (5).

17. $\quad \square$, by (16) and (15).

$\hfill \square$

   Case (SD-VAR, FT-CLASS):

Proof.

$$\frac{VT(\mathtt{C}) = \overline{v\mathtt{X}} \qquad \Delta \vdash \overline{v(\mathtt{T},\mathtt{U})}}{\Delta \vdash \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>} \prec: \underbrace{\mathtt{C}\mathtt{<}\overline{\mathtt{U}}\mathtt{>}}_{\mathtt{N}'}} \text{(SD-VAR)} \qquad \frac{\mathtt{class\ C}\mathtt{<}\overline{v\mathtt{X} \to [\dots]}\mathtt{>} \lhd \mathtt{N} \,\{\, \overline{\mathtt{S\ f};}\ \overline{M}\, \}}{\mathit{ftype}(\mathtt{f}_i; \mathtt{C}\mathtt{<}\overline{\mathtt{U}}\mathtt{>}) = [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{S}_i} \text{(FT-CLASS)}$$

1. $f = f_i$

2. $\Delta \vdash \overline{v(T,U)}$, premise of SD-VAR.

3. `class C<`$\overline{vX \rightarrow [\ldots]}$`>` $\lhd$ `N {` $\overline{S\ f}$`;` $\overline{M}$ `}`, premise of FT-CLASS.

4. $U = \mathit{ftype}(f_i; C<\overline{U}>) = \overline{[U/X]}S_i$, conclusion of FT-CLASS.

5. $\overline{vX} \vdash S_i$ *mono*, applied inversion of W-CLS to (a).

6. $\Delta \vdash \overline{[T/X]}S_i <: \overline{[U/X]}S_i$, applied Lemma 17 to (5) and (2).

7. $\mathit{ftype}(f_i; C<\overline{T}>) = \overline{[T/X]}S_i$, applied FT-CLASS to (3).

8. $\mathit{ftype}(f_i; C<\overline{T}>) = \mathit{ftype}(f_i; C<\overline{T}>) = \overline{[T/X]}S_i$, by (1) and (7).

9. $\quad \Box$, by (8), (6) and (4).

$\Box$

Case (SD-SUPER, *):

Proof.

$$\frac{\texttt{class C<}\overline{vX \rightarrow [\ldots]}\texttt{>} \lhd \texttt{N \{ } \overline{S\ f}\texttt{; } \overline{M} \texttt{ \}} \qquad C \neq D \qquad \Delta \vdash \overline{[T/X]}N \prec: D<\overline{U}>}{\Delta \vdash C<\overline{T}> \prec: \underbrace{D<\overline{U}>}_{N'}} \text{(SD-SUPER)}$$

1. `class C<`$\overline{vX \rightarrow [\ldots]}$`>` $\lhd$ `N {` $\overline{S\ f}$`;` $\overline{M}$ `}`, premise of SD-SUPER.

2. $\Delta \vdash \overline{[T/X]}N \prec: D<\overline{U}>$, premise of SD-SUPER.

3. $\Delta \vdash C<\overline{T}> \prec: D<\overline{U}>$, conclusion of SD-SUPER.

4. Let Δ' be the type parameter environment $\overline{X \rightarrow [\ldots]}$ of class C.

5. $\Delta' \vdash$ N OK, applied inversion of W-CLS to (a).

6. WLOG, assume N= K<...>

7. K $\in dom(CT)$, applied inversion of W-N on (5).

8. $\vdash CT(\text{K})$ OK, every class in CT is wellformed.

   Applying inductive hypothesis to (8), (2) and (c) gives the next two judgments.

9. $ftype(\text{f}; \overline{[\text{T}/\text{X}]}\text{N}) = \text{T}$

10. $\Delta \vdash \text{T} <: \text{U}$

11. $\text{f} \in fields(\text{N})$, by (9).

12. $\text{f} \notin \overline{\text{f}}$, applied distinctness of fieldnames to (11) and (1).

13. $ftype(\text{f}; \text{C<}\overline{\text{T}}\text{>}) = ftype(\text{f}; \overline{[\text{T}/\text{X}]}\text{N}) = \text{T}$, applied FT-SUPER to (1) and (12).

14. $\Box$, by (13) and (10).

$\Box$

$\Box$ for all cases of Lemma 18.

Lemma 19 (Subclassing Preserves Method Type)**.** If

a. $\vdash$ class C<$\overline{\text{vX} \rightarrow [\dots]}$> $\lhd$ N { ... $\overline{M}$ } OK and

b. $mtype(\text{m}; \overline{[\text{T}/\text{X}]}\text{N}) = \text{<}\overline{\text{Y} \rightarrow [\text{B}_L\text{--B}_U]}\text{>}$ $(\overline{\text{U}}) \rightarrow$ U

Then

1. $mtype(\text{m}; \text{C<}\overline{\text{T}}\text{>}) = mtype(\text{m}; \overline{[\text{T}/\text{X}]}\text{N})$

Proof by case analysis on $\mathtt{m} \in \overline{M}$.

Case 1: $\mathtt{m} \notin \overline{M}$.

Proof. Applying MT-SUPER to (a) and the case assumption gives $mtype(\mathtt{m};\ \mathtt{C<\overline{T}>}) = mtype(\mathtt{m};\ [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N})$. □

Case 2: $\mathtt{m} \in \overline{M}$.

Proof.   1. $\mathtt{<\Delta>\ S\ m(\overline{S\ x})\ \{\ return\ e;\ \}} \in \overline{M}$, by case assumption.

2. $\vdash \overline{M}\ OK$ in $\mathtt{C}$, applied inversion of W-CLS to (a).

3. $\vdash \mathtt{<\Delta>\ S\ m(\overline{S\ x})\ \{\ return\ e;\ \}}\ OK$ in $\mathtt{C}$, by (1) and (2).

4. $override(\mathtt{m};\mathtt{N};\mathtt{<\Delta>}\ (\overline{\mathtt{S}}) \to \mathtt{S})$, applied inversion of W-METH to (3).

5. $mtype(\mathtt{m};\mathtt{N})$ is defined, by (b).

6. $mtype(\mathtt{m};\mathtt{N}) = \mathtt{<\Delta>}\ (\overline{\mathtt{S}}) \to \mathtt{S}$, by (4), (5), and definition *override* (OVER-DEF).

7. $mtype(\mathtt{m};\mathtt{C<\overline{X}>}) = \mathtt{<\Delta>}\ (\overline{\mathtt{S}}) \to \mathtt{S}$, applied MT-CLASS to (a) and (1).

8. $mtype(\mathtt{m};\mathtt{C<\overline{X}>}) = mtype(\mathtt{m};\mathtt{N})$, by (6) and (7).

9. $[\overline{\mathtt{T}/\mathtt{X}}]\,mtype(\mathtt{m};\mathtt{C<\overline{X}>}) = [\overline{\mathtt{T}/\mathtt{X}}]\,mtype(\mathtt{m};\mathtt{N})$, by (8).

10.

$$mtype(\mathtt{m};\mathtt{C<\overline{T}>}) = [\overline{\mathtt{T}/\mathtt{X}}]\,mtype(\mathtt{m};\mathtt{C<\overline{X}>})$$
$$= [\overline{\mathtt{T}/\mathtt{X}}]\,mtype(\mathtt{m};\mathtt{N})$$
$$= mtype(\mathtt{m};\ [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N}),$$

by (9)

☐ for case. ☐

☐ for all cases of Lemma 19.

Lemma 20 (Lemma 3 from Main Text – Subtyping Specializes Method Type)**.** If

a. $\vdash$ `class C<`$\overline{\text{vX} \rightarrow [\ldots]}$`>` $\lhd$ `N {` ... $\overline{M}$ `}` *OK* and

b. $\Delta \vdash$ `C<`$\overline{\text{T}}$`>` $\prec:$ `N`$'$ and

c. $mtype(\text{m}; \text{N}') = $ `<`$\overline{\text{Y} \rightarrow [\text{B}_L\text{-B}_U]}$`> (`$\overline{\text{U}}$`)` $\rightarrow$ `U`

Then

1. $mtype(\text{m}; \text{C<}\overline{\text{T}}\text{>}) = $ `<`$\overline{\text{Y} \rightarrow [\text{A}_L\text{-A}_U]}$`> (`$\overline{\text{V}}$`)` $\rightarrow$ `V`,

2. $\Delta \vdash$ `V` $<:$ `U`,

3. $\Delta \vdash \overline{\text{U} <: \text{V}}$,

4. $\Delta \vdash \overline{\text{A}_L <: \text{B}_L}$,

5. $\Delta \vdash \overline{\text{B}_U <: \text{A}_U}$, and

6. $\overline{var(\overline{\text{Y}}; \text{U}) = var(\overline{\text{Y}}; \text{V})}$.

Proof. Conclusions 1–5 can be derived by similar reasoning in the proof of Lemma 18. So instead we focus on deriving conclusion (6). As in the proof of Lemma 18, we derive conclusion (6) by induction on premises (b) and (c). ☐

Case (SD-VAR, MT-CLASS):

Proof.

$$
\cfrac{VT(\text{C}) = \overline{\text{vX}} \qquad \Delta \vdash \overline{v(\text{T}, \text{T}')}}{\Delta \vdash \text{C<}\overline{\text{T}}\text{>} \prec: \underbrace{\text{C<}\overline{\text{T}'}\text{>}}_{\text{N}'}} \text{(SD-VAR)}
$$

$$
\cfrac{\text{class C<}\overline{\text{vX} \rightarrow [\ldots]}\text{>} \lhd \text{N \{ } \ldots \text{ } \overline{M} \text{ \}} \qquad \overline{\text{<Y} \rightarrow [\text{B}'_L\text{-B}'_U]}\text{> S m(}\overline{\text{S x}}\text{) \{ return e; \}} \in \overline{M}}{mtype(\text{m}; \text{C<}\overline{\text{T}'}\text{>}) = [\overline{\text{T}'/\text{X}}]\overline{\text{<Y} \rightarrow [\text{B}'_L\text{-B}'_U]}\text{> (}\overline{\text{S}}\text{)} \rightarrow \text{S}} \text{(MT-CLASS)}
$$

192

1. $VT(\text{C}) = \overline{\text{vX}}$, premise of SD-VAR.

2. $\Delta \vdash \overline{\text{v}(\text{T}, \text{T}')}$, premise of SD-VAR.

3. $\Delta \vdash \text{C<}\overline{\text{T}}\text{>} \prec: \text{C<}\overline{\text{T}'}\text{>}$, conclusion of SD-VAR.

4. `class C<`$\overline{\text{vX} \to [\ldots]}$`> ` $\lhd$ ` N {` $\ldots$ $\overline{M}$ `}`, premise of MT-CLASS.

5. `<`$\overline{\text{Y} \to [\text{B}'_L\text{-B}'_U]}$`> S m(`$\overline{\text{S x}}$`) { return e; }` $\in \overline{M}$, premise of MT-CLASS.

6. $mtype(\text{m}; \text{C<}\overline{\text{T}'}\text{>}) = [\overline{\text{T}'/\text{X}}]\text{<}\overline{\text{Y} \to [\text{B}'_L\text{-B}'_U]}\text{> } (\overline{\text{S}}) \to \text{S}$, conclusion of MT-CLASS.

7. $\overline{\text{U} = \overline{[\text{T}'/\text{X}]\text{S}}}$, by (6) and (c).

8. $mtype(\text{m}; \text{C<}\overline{\text{T}}\text{>}) = [\overline{\text{T}/\text{X}}]\text{<}\overline{\text{Y} \to [\text{B}'_L\text{-B}'_U]}\text{> } (\overline{\text{S}}) \to \text{S}$, applied MT-CLASS to (4) and (5).

9. Let $\overline{\text{V} = \overline{[\text{T}/\text{X}]\text{S}}}$.

10. $\overline{\text{Y}}$ are binders by (c).

11. $\overline{\text{Y}} \cap fv(\text{N}') = \emptyset$, by Barendregt premise of (c) and (10).

12. $\overline{\text{Y}} \cap fv(\text{C<}\overline{\text{T}'}\text{>}) = \emptyset$, by (11) since $\text{N}' = \text{C<}\overline{\text{T}'}\text{>}$.

13. $\overline{\text{Y}} \cap fv(\overline{\text{X}}) = \emptyset$, by Barendregt and (10).

14. $\overline{var(\overline{\text{Y}}; \text{S}) = var(\overline{\text{Y}}; [\overline{\text{T}'/\text{X}}]\text{S})}$, by (12) and (13).

15. $\overline{\text{Y}} \cap fv(\overline{\text{T}}) = \emptyset$, by Barendregt and (10).

16. $\overline{var(\overline{\text{Y}}; \text{S}) = var(\overline{\text{Y}}; [\overline{\text{T}/\text{X}}]\text{S})}$, by (15) and (12).

17. $\overline{var(\overline{\text{Y}}; \underbrace{[\overline{\text{T}'/\text{X}}]\text{S}}_{\text{U}}) = var(\overline{\text{Y}}; \text{S}) = var(\overline{\text{Y}}; \underbrace{[\overline{\text{T}/\text{X}}]\text{S}}_{\text{V}})}$, by (15) and (13).

$\square$ for case $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\square$

Case (SD-Var, MT-Super):

Proof.

$$
\frac{VT(\texttt{C}) = \overline{\texttt{vX}} \qquad \Delta \vdash \overline{\texttt{v}(\texttt{T}, \texttt{T}')}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \underbrace{\texttt{C<}\overline{\texttt{T}'}\texttt{>}}_{\texttt{N}'}}
\quad (\textsc{SD-Var})
\qquad\qquad
\frac{\texttt{class C<}\overline{\texttt{vX} \to [\dots]}\texttt{>} \lhd \texttt{N \{ } \dots \ \overline{M} \texttt{ \}} \qquad \texttt{m} \notin \overline{M}}{mtype(\texttt{m}; \texttt{C<}\overline{\texttt{T}'}\texttt{>}) = mtype(\texttt{m}; [\overline{\texttt{T}'/\texttt{X}}]\texttt{N})}
\quad (\textsc{MT-Super})
$$

1. $VT(\texttt{C}) = \overline{\texttt{vX}}$, premise of SD-Var.

2. $\Delta \vdash \overline{\texttt{v}(\texttt{T}, \texttt{T}')}$, premise of SD-Var.

3. $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{C<}\overline{\texttt{T}'}\texttt{>}$, conclusion of SD-Var.

4. $\texttt{class C<}\overline{\texttt{vX} \to [\dots]}\texttt{>} \lhd \texttt{N \{ } \dots \ \overline{M} \texttt{ \}}$, premise of MT-Super.

5. $\texttt{m} \notin \overline{M}$, premise of MT-Super.

6. $mtype(\texttt{m}; \texttt{C<}\overline{\texttt{T}'}\texttt{>}) = mtype(\texttt{m}; [\overline{\texttt{T}'/\texttt{X}}]\texttt{N})$, conclusion of MT-Super.

7. $mtype(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mtype(\texttt{m}; [\overline{\texttt{T}/\texttt{X}}]\texttt{N})$, applied MT-Super to (4) and (5).

8. Let Δ' be the type parameter environment $\overline{\texttt{X} \to [\dots]}$ of class $\texttt{C}$.

9. $\Delta' \vdash \texttt{N} \ OK$, applied inversion of W-Cls to (a).

10. $\overline{\texttt{vX}} \vdash \texttt{N} \ mono$, applied inversion of W-Cls to (a).

11. $\Delta \vdash [\overline{\texttt{T}/\texttt{X}}]\texttt{N} \prec: [\overline{\texttt{T}'/\texttt{X}}]\texttt{N}$, applied Lemma 17 to (10) and (2).

12. WLOG, assume $\texttt{N} = \texttt{K<}\dots\texttt{>}$.

13. $CT(\texttt{K}) = \texttt{class K<}\dots\texttt{>} \lhd \texttt{N}'' \dots$, by (12) and applied inversion of W-N on (9).

14. $\vdash$ `class K<...> ◁ N″`... *OK*, by (13) and every class in *CT* is wellformed.

Applying the inductive hypothesis to (14), (11), and (c) gives the following two judgments.

15. $mtype(\texttt{m}; \overline{[\texttt{T}/\texttt{X}]\texttt{N}}) = \texttt{<}\overline{\texttt{Y} \to [\texttt{A}_L\texttt{-}\texttt{A}_U]}\texttt{>} (\overline{\texttt{V}}) \to \texttt{V}$

16. $\overline{var(\overline{\texttt{Y}}; \texttt{U}) = var(\overline{\texttt{Y}}; \texttt{V})}$

17. $\Box$, by (7), (15), and (16).

$\Box$

Case (SD-SUPER, *):

Proof.

$$\frac{\texttt{class C<}\overline{\texttt{vX} \to [\ldots]}\texttt{>} ◁ \texttt{N} \{ \overline{\texttt{S f};}\ \overline{M} \} \qquad \texttt{C} \neq \texttt{D} \qquad \Delta \vdash \overline{[\texttt{T}/\texttt{X}]\texttt{N}} \prec: \texttt{D<}\overline{\texttt{T}'}\texttt{>}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \underbrace{\texttt{D<}\overline{\texttt{T}'}\texttt{>}}_{\texttt{N}'}}$$

$$(\text{SD-SUPER})$$

1. `class C<`$\overline{\texttt{vX} \to [\ldots]}$`> ◁ N {` $\overline{\texttt{S f};}$ $\overline{M}$ `}`, premise of SD-SUPER.

2. $\texttt{C} \neq \texttt{D}$, premise of SD-SUPER.

3. $\Delta \vdash \overline{[\texttt{T}/\texttt{X}]\texttt{N}} \prec: \texttt{D<}\overline{\texttt{T}'}\texttt{>}$, premise of SD-SUPER.

4. $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{D<}\overline{\texttt{T}'}\texttt{>}$, conclusion of SD-SUPER.

5. $mtype(\texttt{m}; \texttt{D<}\overline{\texttt{T}'}\texttt{>}) = \texttt{<}\overline{\texttt{Y} \to [\texttt{B}_L\texttt{-}\texttt{B}_U]}\texttt{>} (\overline{\texttt{U}}) \to \texttt{U}$, by (c), since $\texttt{D<}\overline{\texttt{T}'}\texttt{>} = \texttt{N}'$.

6. Let Δ' be the type parameter environment $\overline{\texttt{X} \to [\ldots]}$ of class C.

7. $\Delta' \vdash \texttt{N}$ *OK*, applied inversion of W-CLS to (a).

195

8. WLOG, assume $N = K<\ldots>$.

9. $CT(K) = \texttt{class K<}\ldots\texttt{>} \lhd \texttt{N}''\ldots$, by (8) and applied inversion of W-N to (7).

10. $\vdash \texttt{class K<}\ldots\texttt{>} \lhd \texttt{N}''\ldots$ *OK*, by (9) and every class in CT is wellformed.

Applying the inductive hypothesis to (10), (3), and (c) gives the following two judgments.

11. $mtype\big(\texttt{m}; \overline{[\texttt{T}/\texttt{X}]}\texttt{N}\big) = \texttt{<}\overline{\texttt{Y} \to \texttt{[A}_L\texttt{-A}_U\texttt{]}}\texttt{>} \ (\overline{\texttt{V}}) \to \texttt{V}$

12. $\overline{var(\overline{\texttt{Y}}; \texttt{U})} = var(\overline{\texttt{Y}}; \texttt{V})$

13. $mtype(\texttt{m}; \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mtype\big(\texttt{m}; \overline{[\texttt{T}/\texttt{X}]}\texttt{N}\big)$, applied Lemma 19 to (a) and (11).

14. $\quad \Box$, by (13), (11), and (12).

$\Box$

$\Box$ for all cases of Lemma 20.

Lemma 21 (Reflexivity of Definition-Site Subtyping)**.**

$$\Delta \vdash \texttt{R} \prec: \texttt{R}$$

Proof. Easy, since by ST-REFL, for any type $\texttt{T}$ and any variance $\texttt{v}$, we have $\Delta \vdash \texttt{v}(\texttt{T}, \texttt{T})$.
$\Box$ $\Box$

Lemma 22 (Transitivity of Definition-Site Subtyping)**.** If

a. $\Delta \vdash \texttt{R} \prec: \texttt{R}'$

b. $\Delta \vdash \texttt{R}' \prec: \texttt{R}''$

Then

1. $\Delta \vdash \texttt{R} \prec: \texttt{R}''$

Proof. Proof by similar reasoning found in Appendix B of work by Kennedy and Pierce [13]. □ □

Lemma 23 (Lemma 4 from Main Text – Existential subtyping to def-site subtyping)**.** If

a. $\Delta \vdash \exists \Delta'.\texttt{R}' \sqsubseteq: \exists \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}.\texttt{R}$

b. $\emptyset \vdash \Delta$

Then there exists $\overline{\texttt{T}}$ such that

1. $\Delta, \Delta' \vdash \texttt{R}' \prec: [\overline{\texttt{T}/\texttt{X}}]\texttt{R}$

2. $\Delta, \Delta' \vdash \overline{[\overline{\texttt{T}/\texttt{X}}]\texttt{B}_L <: \texttt{T}}$

3. $\Delta, \Delta' \vdash \overline{\texttt{T} <: [\overline{\texttt{T}/\texttt{X}}]\texttt{B}_U}$

4. $fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta')$

Proof. Proof by similar reasoning found in proof of Lemma 35 from TameFJ, with one new induction case on (a):

Case: SE-SD.
$$\frac{\Delta, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]} \vdash \texttt{N} \prec: \texttt{N}'}{\Delta \vdash \exists \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}.\texttt{N} \sqsubseteq: \exists \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}.\texttt{N}'}$$

1. Choose $\overline{\texttt{T}} = \overline{\texttt{X}}$.

2. $\Delta, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]} \vdash \texttt{N} \prec: \texttt{N}'$, premise of SE-SD.

3. $\Delta, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]} \vdash \overline{\texttt{B}_L <: \exists \emptyset.\texttt{X}}$, by ST-LBOUND.

4. $\Delta, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]} \vdash \overline{\exists \emptyset.\texttt{X} <: \texttt{B}_U}$, by ST-UBOUND.

5. $\overline{\texttt{X}} \subseteq dom(\Delta, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]})$

$\square$, by (1-5). $\hspace{6cm}$ $\square$

Lemma 24 (Lemma 5 from Main Text – Subtyping to existential subtyping). If

    a. $\Delta \vdash \mathtt{T} <: \mathtt{T}'$

    b. $\emptyset \vdash \Delta \ OK$

  Then

    1. $\Delta \vdash ubound_\Delta(\mathtt{T}) \sqsubseteq: ubound_\Delta(\mathtt{T}')$

Proof. Proof by similar reasoning found in proof of Lemma 17 from TameFJ. $\hspace{1cm}$ $\square$

Lemma 25 (Subtitution Preserves Subtyping). If

    a. $\Delta = \Delta_1, \overline{\mathtt{X} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}, \Delta_2$

    b. $\Delta' = \overline{[\mathtt{T}/\mathtt{X}]}(\Delta_1, \Delta_2)$

    c. $fv(\overline{\mathtt{T}}) \subseteq dom(\Delta')$

    d. $\Delta' \vdash \overline{\mathtt{T} <: \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{B}_U}$

    e. $\Delta' \vdash \overline{\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{B}_L <: \mathtt{T}}$

and if

    f. $\Delta \vdash \mathtt{R} \prec: \mathtt{R}'$

then

    1. $\Delta' \vdash \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{R} \prec: \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{R}'$

and if

    g. $\Delta \vdash \mathtt{B} \sqsubseteq: \mathtt{B}'$

then

2. $\Delta' \vdash \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{B} \sqsubset: \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{B}'$

and if

h. $\Delta \vdash \mathtt{B} <: \mathtt{B}'$

then

4. $\Delta' \vdash \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{B} <: \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{B}'$

Proof by induction on $\Delta \vdash \phi <<: \phi'$, where "$\phi <<: \phi'$" $\equiv (\phi \prec: \phi'$ **OR** $\phi \sqsubset: \phi'$ **OR** $\phi <: \phi')$ and $\phi ::= \mathtt{R} \mid \mathtt{B}$.

Case: SD-SUPER

Proof.

$$\frac{\mathtt{class\ C}<\overline{\mathtt{vY} \to [\ldots]}> \lhd \mathtt{N}\ \{\ \ldots\ \ \} \qquad \mathtt{C} \neq \mathtt{D} \qquad \Delta \vdash \overline{[\mathtt{U}/\mathtt{Y}]}\mathtt{N} \prec: \mathtt{D}<\overline{\mathtt{V}}>}{\Delta \vdash \mathtt{C}<\overline{\mathtt{U}}> \prec: \mathtt{D}<\overline{\mathtt{V}}>}$$

$$(\text{SD-SUPER})$$

1. class $\mathtt{C}<\overline{\mathtt{vY} \to [\ldots]}> \lhd \mathtt{N}\ \{\ \ldots\ \ \}$, premise of SD-SUPER.

2. $\Delta \vdash \overline{[\mathtt{U}/\mathtt{Y}]}\mathtt{N} \prec: \mathtt{D}<\overline{\mathtt{V}}>$, premise of SD-SUPER.

3. $\Delta \vdash \mathtt{C}<\overline{\mathtt{U}}> \prec: \mathtt{D}<\overline{\mathtt{V}}>$, conclusion of SD-SUPER.

4. $\Delta' \vdash \overline{[\mathtt{T}/\mathtt{X}]}\,\overline{[\mathtt{U}/\mathtt{Y}]}\mathtt{N} \prec: \mathtt{D}<\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{V}>$, applied inductive hypothesis to (2) and (a-e).

5. $\overline{\mathtt{Y}}$ are binders by (1).

6. $\overline{\mathtt{Y}} \cap (fv(\overline{\mathtt{T}}) \cup \overline{\mathtt{X}}) = \emptyset$, applied Barendregt to (5).

7. $\vdash$ class $\mathtt{C}<\overline{\mathtt{vY} \to [\ldots]}> \lhd \mathtt{N}\ \{\ \ldots\ \ \}$ *OK*, applied every class is wellformed by to (1).

8. $Y \rightarrow [\ldots] \vdash N$ *OK*, applied inversion of W-Cls to (7).

9. $fv(N) \subseteq \overline{Y}$, by (8).

10. $fv(N) \cap \overline{X} = \emptyset$, by (9) and (6).

11. $\overline{[T/X]}\,\overline{[U/Y]}N = \overline{[\overline{[T/X]}U/Y]}N$, by (10) and (6).

12. $\Delta' \vdash \overline{[\overline{[T/X]}U/Y]}N \prec: D<\overline{[T/X]}V>$, (4) and (11).

13. $C \neq D$, premise of SD-Super case assumption.

14. $\Delta' \vdash C<\overline{[T/X]}U> \prec: D<\overline{[T/X]}V>$, applied SD-Super to (1), (13), and (12).

□ for case   □

  Case SD-Var:

Proof.

$$\frac{VT(C) = \overline{vY} \qquad \Delta \vdash \overline{v(U, V)}}{\Delta \vdash C<\overline{U}> \prec: C<\overline{V}>}$$

$$(\text{SD-Var})$$

1. $VT(C) = \overline{vY}$, premise of SD-Var.

2. $\Delta \vdash \overline{v(U, V)}$, premise of SD-Var.

3. $\Delta \vdash C<\overline{U}> \prec: C<\overline{V}>$, conclusion of SD-Var.

4. $\Delta' \vdash \overline{v(\overline{[T/X]}U, \overline{[T/X]}V)}$, applied inductive hypothesis to (2) and (a-e).

5. $\Delta \vdash C<\overline{[T/X]}U> \prec: C<\overline{[T/X]}V>$, applied SD-Var to (1) and (4).

□ for case   □

  Case SE-SD:

Proof.

$$\frac{\Delta_1, \overline{\mathtt{X} \to \mathtt{[B}_L\text{-}\mathtt{B}_U\mathtt{]}}, \Delta_2, \Delta_3 \vdash \mathtt{N} \prec: \mathtt{N}'}{\Delta_1, \overline{\mathtt{X} \to \mathtt{[B}_L\text{-}\mathtt{B}_U\mathtt{]}}, \Delta_2 \vdash \exists\Delta_3.\mathtt{N} \sqsubset: \exists\Delta_3.\mathtt{N}'}$$

$$(\text{SE-SD})$$

1. $\Delta_1, \overline{\mathtt{X} \to \mathtt{[B}_L\text{-}\mathtt{B}_U\mathtt{]}}, \Delta_2, \Delta_3 \vdash \mathtt{N} \prec: \mathtt{N}'$, premise of SE-SD.

2. $\Delta_1, \overline{\mathtt{X} \to \mathtt{[B}_L\text{-}\mathtt{B}_U\mathtt{]}}, \Delta_2 \vdash \exists\Delta_3.\mathtt{N} \sqsubset: \exists\Delta_3.\mathtt{N}'$, conclusion of SE-SD.

3. $fv(\overline{\mathtt{T}}) \subseteq dom(\Delta') \subseteq dom(\Delta', \Delta_3)$, by (c).

4. $dom(\Delta_3) \cap dom(\Delta_1, \overline{\mathtt{X} \to \mathtt{[B}_L\text{-}\mathtt{B}_U\mathtt{]}}, \Delta_2) = \emptyset$, by Barendregt premise of SE-SD.

5. $dom(\Delta_3) \cap dom(\overline{\mathtt{[T/X]}}\Delta_1, \overline{\mathtt{[T/X]}}\Delta_2) = \emptyset$, by (4).

6. $dom(\Delta_3) \cap \overline{\mathtt{X}} = \emptyset$, by (4).

7. $\overline{\mathtt{[T/X]}}\Delta_3 = \Delta_3$, by (6).

8. Let $\Delta'' = \Delta_1, \overline{\mathtt{X} \to \mathtt{[B}_L\text{-}\mathtt{B}_U\mathtt{]}}, \Delta_2, \Delta_3$.

9. Let $\Delta''' = \overline{\mathtt{[T/X]}}\Delta_1, \overline{\mathtt{[T/X]}}\Delta_2, \overline{\mathtt{[T/X]}}\Delta_3$.

10. $fv(\overline{\mathtt{T}}) \subseteq dom(\Delta') \subseteq dom(\Delta', \Delta_3) = dom(\Delta''')$, by (9) and (7).

11. $dom(\Delta_3) \cap dom(\Delta') = \emptyset$, by (5).

12. $\underbrace{\Delta', \Delta_3}_{\Delta'''} \vdash \overline{\mathtt{T} <: \overline{\mathtt{[T/X]}}\mathtt{B}_U}$, applied weakening lemma to (11) and (d).

13. $\Delta''' \vdash \overline{\overline{\mathtt{[T/X]}}\mathtt{B}_L <: \mathtt{T}}$, applied weakening lemma to (11) and (e).

14. $\overline{\mathtt{[T/X]}}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{\mathtt{[T/X]}}\mathtt{N} \prec: \overline{\mathtt{[T/X]}}\mathtt{N}'$, applied inductive hypothesis to (9), (10), (12), (13), and (1).

15. $\overline{\mathtt{[T/X]}}(\Delta_1, \Delta_2) \vdash \exists\overline{\mathtt{[T/X]}}\Delta_3.\overline{\mathtt{[T/X]}}\mathtt{N} \sqsubset: \exists\overline{\mathtt{[T/X]}}\Delta_3.\overline{\mathtt{[T/X]}}\mathtt{N}'$, applied SE-SD to (14).

16. $\overline{[\texttt{T}/\texttt{X}]}(\Delta_1,\Delta_2) \vdash \overline{[\texttt{T}/\texttt{X}]}\exists\Delta_3.\texttt{N} \sqsubset: \overline{[\texttt{T}/\texttt{X}]}\exists\Delta_3.\texttt{N}'$, by Barendregt since $dom(\Delta_3)$ are binders.

□ for case □

Case ST-LBOUND:

Proof.

$$\frac{(\Delta_1, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}, \Delta_2)(\texttt{Y}) = [\texttt{B}_L - \texttt{B}_U]}{\Delta_1, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}, \Delta_2 \vdash \texttt{B}_L <: \exists\emptyset.\texttt{Y}}$$

$$(\text{ST-LBOUND})$$

1. $(\Delta_1, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}, \Delta_2)(\texttt{Y}) = [\texttt{B}_L - \texttt{B}_U]$, premise of ST-LBOUND.

2. $\Delta_1, \overline{\texttt{X} \to [\texttt{B}_L\texttt{-B}_U]}, \Delta_2 \vdash \texttt{B}_L <: \exists\emptyset.\texttt{Y}$, conclusion of ST-LBOUND.

Completing proof by case analysis on $\texttt{Y} \in \overline{\texttt{X}}$.

Case 1: $\texttt{Y} \in \overline{\texttt{X}}$.

1.1. $\texttt{Y} = \texttt{X}_i$, for some i, by case assumption

1.2. $\overline{[\texttt{T}/\texttt{X}]}\exists\emptyset.\texttt{Y} = \overline{[\texttt{T}/\texttt{X}]}\exists\emptyset.\texttt{X}_i = \texttt{T}_i$, by (1.1).

1.3. $\texttt{Y} \to [\texttt{B}_L\texttt{-B}_U] = \texttt{X}_i \to [\texttt{B}_{L_i}\texttt{-B}_{U_i}]$, by (1.1) and (1).

1.4. $\Delta' \vdash \overline{[\texttt{T}/\texttt{X}]}\texttt{B}_{L_i} <: \texttt{T}_i$, by (e).

1.5. $\Delta' \vdash \overline{[\texttt{T}/\texttt{X}]}\texttt{B}_L <: \overline{[\texttt{T}/\texttt{X}]}\exists\emptyset.\texttt{Y}$, by (1.4), (1.3), and (1.2).

□ for case 1

Case 2: $\texttt{Y} \notin \overline{\texttt{X}}$ **and** $\texttt{Y} \in dom(\Delta_1,\Delta_2)$.

2.1. $(\Delta_1,\Delta_2)(\texttt{Y}) = [\texttt{B}_L - \texttt{B}_U]$, by (1) and case assumption.

2.2. $\overline{[\texttt{T}/\texttt{X}]}(\Delta_1,\Delta_2)(\texttt{Y}) = [\overline{[\texttt{T}/\texttt{X}]}\texttt{B}_L - \overline{[\texttt{T}/\texttt{X}]}\texttt{B}_U]$, by (2.1).

2.3. $[\overline{\texttt{T/X}}](\Delta_1, \Delta_2) \vdash [\overline{\texttt{T/X}}]\texttt{B}_L <: \exists\emptyset.\texttt{Y}$, applied ST-LBound to (2.2).

2.4. $[\overline{\texttt{T/X}}]\exists\emptyset.\texttt{Y} = \exists\emptyset.\texttt{Y}$, by case assumption.

2.5. $[\overline{\texttt{T/X}}](\Delta_1, \Delta_2) \vdash [\overline{\texttt{T/X}}]\texttt{B}_L <: [\overline{\texttt{T/X}}]\exists\emptyset.\texttt{Y}$, by (2.3) and (2.4).

$\square$ for case 2

Case 3: $\texttt{Y} \notin \overline{\texttt{X}}$ **and** $\texttt{Y} \notin dom(\Delta_1, \Delta_2)$.

3.1. $\texttt{Y} \notin dom(\Delta_1, \overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2)$, by case assumption.

3.2. **contradiction**, by (3.1) and (1).

$\square$ for case 3

We completed the case analysis on $\texttt{Y} \in \overline{\texttt{X}}$ and completed the proof for the ST-LBound case of this lemma. $\square$ $\square$

Case ST-UBound:

Proof. Similar reasoning in the proof for the ST-LBound case proves the lemma for this case. $\square$ for case $\square$

Case SE-Pack

Proof.

$$\dfrac{\begin{array}{c} dom(\Delta_3) \cap fv(\exists\overline{\texttt{Y} \rightarrow \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\texttt{N}) = \emptyset \qquad fv(\overline{\texttt{U}}) \subseteq dom(\Delta_1, \overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2, \Delta_3) \\ \Delta_1, \overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2, \Delta_3 \vdash \overline{[\overline{\texttt{U/Y}}]\texttt{A}_L <: \texttt{U}} \\ \Delta_1, \overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2, \Delta_3 \vdash \overline{\texttt{U} <: [\overline{\texttt{U/Y}}]\texttt{A}_U} \end{array}}{\Delta_1, \overline{\texttt{X} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2 \vdash \exists\Delta_3.[\overline{\texttt{U/Y}}]\texttt{N} \sqsubseteq: \exists\overline{\texttt{Y} \rightarrow \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\texttt{N}}$$

$$(\text{SE-Pack})$$

1. $dom(\Delta_3) \cap fv(\exists\overline{\texttt{Y} \rightarrow \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\texttt{N}) = \emptyset$, premise of SE-Pack.

2. $fv(\overline{\mathtt{U}}) \subseteq dom(\Delta_1, \overline{\mathtt{X} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2, \Delta_3)$, premise of SE-Pack.

3. $\Delta_1, \overline{\mathtt{X} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2, \Delta_3 \vdash \overline{\texttt{[U/Y]}\texttt{A}_L} <: \mathtt{U}$, premise of SE-Pack.

4. $\Delta_1, \overline{\mathtt{X} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2, \Delta_3 \vdash \mathtt{U} <: \overline{\texttt{[U/Y]}\texttt{A}_U}$, premise of SE-Pack.

5. $\Delta_1, \overline{\mathtt{X} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}, \Delta_2 \vdash \exists\Delta_3.\overline{\texttt{[U/Y]}}\mathtt{N} \sqsubset: \exists\overline{\mathtt{Y} \to \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\mathtt{N}$, conclusion of SE-Pack.

6. $fv(\overline{\mathtt{T}}) \subseteq dom(\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2)) \subseteq dom(\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2, \Delta_3))$, by (c).

7. $dom(\overline{\texttt{[T/X]}}\Delta_3) \cap dom(\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2)) = \emptyset$, by Barendrengt premise of SE-Pack.

8. $\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{\mathtt{T}} <: \overline{\texttt{[T/X]}\texttt{B}_U}$, applied weakening lemma to (7) and (d).

9. $\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{\texttt{[T/X]}\texttt{B}_L} <: \overline{\mathtt{T}}$, applied weakening lemma to (7) and (e).

10. $\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{\texttt{[T/X]} \texttt{[U/Y]}\texttt{A}_L} <: \overline{\texttt{[T/X]}\mathtt{U}}$, applied inductive hypothesis to (6-9) and (3).

11. $\overline{\texttt{[T/X]}}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{\texttt{[T/X]}\mathtt{U}} <: \overline{\texttt{[T/X]} \texttt{[U/Y]}\texttt{A}_U}$, applied inductive hypothesis to (6-9) and (4).

12. $dom(\Delta_3) = dom(\overline{\texttt{[T/X]}}\Delta_3)$

13. $\overline{\mathtt{Y}} \cap \overline{\mathtt{X}} = \emptyset$, by Barendregt, since $\overline{\mathtt{Y}}$ are binders.

14.

$$fv(\exists\overline{\mathtt{Y} \to \texttt{[}\overline{\texttt{[T/X]}\texttt{A}_L}\texttt{-}\overline{\texttt{[T/X]}\texttt{A}_U}\texttt{]}}.\overline{\texttt{[T/X]}}\mathtt{N})$$

$$= fv(\overline{\texttt{[T/X]}}\exists\overline{\mathtt{Y} \to \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\mathtt{N})$$

$$= \overline{\texttt{[T/X]}}fv(\exists\overline{\mathtt{Y} \to \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\mathtt{N})$$

$$\subseteq fv(\overline{\mathtt{T}}) \cup fv(\exists\overline{\mathtt{Y} \to \texttt{[A}_L\texttt{-A}_U\texttt{]}}.\mathtt{N})$$

15.

$$fv(\overline{\mathtt{T}}) \cap dom(\overline{[\mathtt{T}/\mathtt{X}]}\Delta_3)$$

$$= fv(\overline{\mathtt{T}}) \cap dom(\Delta_3) \qquad \text{by (12)}$$

$$\subseteq dom(\Delta_1, \Delta_2) \cap dom(\Delta_3) \qquad \text{by (c)}$$

$$= \emptyset \qquad \text{by Barendrengt premise of SE-Pack}$$

16.

$$dom(\overline{[\mathtt{T}/\mathtt{X}]}\Delta_3) \cap fv(\exists \overline{\mathtt{Y} \to [\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{A}_L\text{-}\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{A}_U]}.\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{N})$$

$$\subseteq dom(\Delta_3) \cap \left[fv(\overline{\mathtt{T}}) \cup fv(\exists \overline{\mathtt{Y} \to [\mathtt{A}_L\text{-}\mathtt{A}_U]}.\mathtt{N})\right] \qquad \text{by (9) and (10)}$$

$$= \left[dom(\Delta_3) \cap fv(\overline{\mathtt{T}})\right] \cup \left[dom(\Delta_3) \cap fv(\exists \overline{\mathtt{Y} \to [\mathtt{A}_L\text{-}\mathtt{A}_U]}.\mathtt{N})\right]$$

$$= \emptyset \cup \emptyset \qquad \text{by (15) and (1)}$$

17.

$$fv(\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{U}) \subseteq fv(\overline{\mathtt{T}}) \cup \left[fv(\overline{\mathtt{U}}) - \overline{\mathtt{X}}\right]$$

$$\subseteq dom(\Delta_1, \Delta_2) \cup \left[fv(\overline{\mathtt{U}}) - \overline{\mathtt{X}}\right] \qquad \text{by (c)}$$

$$\subseteq dom(\Delta_1, \Delta_2) \cup \left[dom(\Delta_1, \overline{\mathtt{X} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}\Delta_2, \Delta_3) - \overline{\mathtt{X}}\right] \qquad \text{by (2)}$$

$$= dom(\Delta_1, \Delta_2, \Delta_3) = dom((\overline{[\mathtt{T}/\mathtt{X}]})\Delta_1, \Delta_2, \Delta_3)$$

18. $(fv(\overline{\mathtt{T}}) \cup \overline{\mathtt{X}}) \cap \overline{\mathtt{Y}} = \emptyset$, by Barendregt, since $\overline{\mathtt{Y}}$ are binders.

19. $\overline{[\mathtt{T}/\mathtt{X}]}\,\overline{[\mathtt{U}/\mathtt{Y}]}\mathtt{A}_L = \overline{[\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{U}/\mathtt{Y}]}\,\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{A}_L$, by (18).

20. $\overline{[\mathtt{T}/\mathtt{X}]}\,\overline{[\mathtt{U}/\mathtt{Y}]}\mathtt{A}_U = \overline{[\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{U}/\mathtt{Y}]}\,\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{A}_U$, by (18).

21. $\overline{[\mathtt{T}/\mathtt{X}]}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{[\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{U}/\mathtt{Y}]}\,\overline{[\mathtt{T}/\mathtt{X}]}\mathtt{A}_L <: \overline{[\mathtt{T}/\mathtt{X}]}\mathtt{U}$, by (19) and (10).

22. $\overline{[\text{T}/\text{X}]}(\Delta_1, \Delta_2, \Delta_3) \vdash \overline{[\text{T}/\text{X}]\text{U}} <: \overline{[[\overline{\text{T}/\text{X}]\text{U}/\text{Y}}]\overline{[\text{T}/\text{X}]}\text{A}_U}$, by (20) and (11).

23. $\overline{[\text{T}/\text{X}]}(\Delta_1, \Delta_2) \vdash \exists\overline{[\text{T}/\text{X}]}\Delta_3.[\overline{[\overline{\text{T}/\text{X}]}\text{U}/\text{Y}}]\text{N} \sqsubset: \exists\overline{\text{Y}} \to [[\overline{\text{T}/\text{X}]}\text{A}_L\text{-}\overline{[\text{T}/\text{X}]}\text{A}_U].[\overline{\text{T}/\text{X}]}\text{N}$,

applied SE-PACK to (16), (17), (21), and (22).

24. $\overline{[\text{T}/\text{X}]}(\Delta_1, \Delta_2) \vdash \overline{[\text{T}/\text{X}]}\exists\Delta_3.[\overline{\text{U}/\text{Y}]}\text{N} \sqsubset: \overline{[\text{T}/\text{X}]}\exists\overline{\text{Y}} \to \overline{[\text{A}_L\text{-}\text{A}_U]}.\text{N}$, by (23) and (18).

$\square$ for case.

Proofs for remaining cases are all trivial.

$\square$ for lemma. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

Lemma 26 (Invariance Fixes Type Instantiations). If

    a. $var(\text{Y}; \text{R}) = o$

    b. $\Delta \vdash \text{R}_1 \prec: [\text{T}/\text{Y}]\text{R}$

    c. $\Delta \vdash \text{R}_2 \prec: [\text{U}/\text{Y}]\text{R}$

  Then

    1. $\Delta \vdash o(\text{T}, \text{U})$

Proof. By standard induction on (b) and (c). $\quad\square$ $\qquad\qquad\qquad\qquad\qquad\square$

Lemma 27 (Invariance Fixes Type Unification Position). If

    a. $var(\text{Y}; \text{R}_3) = o$

    b. $\Delta \vdash \text{R}_2 \prec: [\text{U}/\text{Y}]\text{R}_3$

    c. $\Delta \vdash \text{R}_1 \prec: [\text{T}/\text{X}]\text{R}_2$

    d. $\Delta \vdash \text{R}_1 \prec: [\text{V}/\text{Y}]\text{R}_3$

    e. $(fv(\text{T}) \cup \text{X}) \cap \text{Y} = \emptyset$

f. $fv(\text{R}_3) \cap \text{X} = \emptyset$

Then

1. $\Delta \vdash o(\text{V}, \, \text{[U/X]U})$

Proof.   1. $\Delta \vdash \text{[T/X]R}_2 \prec: \text{[T/X][U/Y]R}_3$, applied Lemma 25 to (b).

2. $\text{[T/X][U/Y]R}_3 = \text{[[T/X]U/Y]R}_3$, by (e) and (f).

3. $\Delta \vdash \text{[T/X]R}_2 \prec: \text{[[T/X]U/Y]R}_3$, by (1) and (2).

4. $\Delta \vdash \text{R}_1 \prec: \text{[[T/X]U/Y]R}_3$, applied Lemma 22 to (c) and (3).

5. $\Delta \vdash o(\text{V}, \, \text{[T/X]U})$, applied Lemma 26 to (a), (d), and (4).

□   □

Lemma 28 (Lemma 7 from Main Text – Subtyping Preserves *match*ing (receiver))**.**
If

a. $\Delta \vdash \exists\Delta_1.\text{N}_1 \sqsubseteq: \exists\Delta_2.\text{N}_2$

b. $mtype(\text{m}; \text{N}_2) = \overline{\texttt{<}\text{Y}_2 \rightarrow \text{[B}_{2L}\text{--B}_{2U}\text{]>}}\,(\overline{\text{U}_2}) \rightarrow \text{U}_2$

c. $mtype(\text{m}; \text{N}_1) = \overline{\texttt{<}\text{Y}_1 \rightarrow \text{[B}_{1L}\text{--B}_{1U}\text{]>}}\,(\overline{\text{U}_1}) \rightarrow \text{U}_1$

d. $sift(\overline{\text{R}}; \overline{\text{U}_2}; \overline{\text{Y}_2}) = (\overline{\text{R}'}; \overline{\text{U}'_2})$

e. $match(\overline{\text{R}'}; \overline{\text{U}'_2}; \overline{\text{P}}; \overline{\text{Y}_2}; \overline{\text{T}})$

f. $\emptyset \vdash \Delta\ OK$

g. $\Delta, \Delta' \vdash \overline{\text{T}}\ OK$

Then

1. $sift(\overline{\text{R}}; \overline{\text{U}_1}; \overline{\text{Y}_1}) = (\overline{\text{R}'}; \overline{\text{U}'_1})$

2. $match(\overline{\mathtt{R}'}; \overline{\mathtt{U}'_1}; \overline{\mathtt{P}}; \overline{\mathtt{Y}_1}; \overline{\mathtt{T}})$

Proof. Proof by similar reasoning as in the proof of lemma 36 of TameFJ exception noting that $\mathtt{R}'$ is returned by both $sift(\overline{\mathtt{R}}; \overline{\mathtt{U}_2}; \overline{\mathtt{Y}_2})$ and $sift(\overline{\mathtt{R}}; \overline{\mathtt{U}_1}; \overline{\mathtt{Y}_1})$ because of conclusion (6) of Lemma 20. $\square$ $\square$

Lemma 29 (Subtyping Preserves *match*ing (arguments)). If

    a. $\Delta \vdash \overline{\exists\Delta_1.\mathtt{R}_1 \sqsubseteq: \exists\Delta_2.\mathtt{R}_2}$

    b. $match(sift(\overline{\mathtt{R}_2}; \overline{\mathtt{U}}; \overline{\mathtt{Y}}); \overline{\mathtt{P}}; \overline{\mathtt{Y}}; \overline{\mathtt{T}})$

    c. $\overline{\Delta_2} = \overline{\mathtt{Z} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}$

    d. $fv(\overline{\mathtt{U}}) \cap \overline{\mathtt{Z}} = \emptyset$

    e. $\emptyset \vdash \Delta \ OK$

    f. $\Delta \vdash \overline{\exists\Delta_1.\mathtt{R}_1} \ OK$

    g. $\Delta \vdash \overline{\mathtt{P}} \ OK$

    Then there exists $\overline{\overline{\mathtt{U}'}}$ such that: Then

    1. $match(sift(\overline{\mathtt{R}_1}; \overline{\mathtt{U}}; \overline{\mathtt{Y}}); \overline{\mathtt{P}}; \overline{\mathtt{Y}}; \overline{[\overline{\mathtt{U}'}/\mathtt{Z}]\mathtt{T}})$

    2. $\Delta, \Delta' \vdash \overline{[\overline{\mathtt{U}'}/\mathtt{Z}]\mathtt{B}_L <: \mathtt{U}'}$

    3. $\Delta, \Delta' \vdash \overline{\mathtt{U}' <: [\overline{\mathtt{U}'}/\mathtt{Z}]\mathtt{B}_U}$

    4. $\overline{\mathtt{R}_1 \prec: [\overline{\mathtt{U}'}/\mathtt{Z}]\mathtt{R}_2}$

    5. $fv(\overline{\mathtt{U}'}) \subseteq dom(\Delta, \Delta')$

Proof. Proof by similar reasoning as in the proof of lemma 37 of TameFJ. $\square$ $\square$

Theorem 4 (Progress). For any $\Delta, \mathtt{e}, \mathtt{T}$, if $\emptyset; \emptyset \vdash \mathtt{e} \ : \ \mathtt{T} \mid \Delta$, then either $\mathtt{e} \mapsto \mathtt{e}'$ or there exists a $\mathtt{v}$ such that $\mathtt{e} = \mathtt{v}$.

Proof. Proof by similar reasoning as in the proof of the Progress Theorem of TameFJ.

□   □

Theorem 5 (Subject Reduction). For any e, e', T, if $\emptyset; \emptyset \vdash e : T \mid \emptyset$ and $e \mapsto e'$, then $\emptyset; \emptyset \vdash e' : T \mid \emptyset$.

Proof. The proof of this theorem is similar to the proof of the Subject Reduction Theorem of TameFJ except for a technicality with *match* that is related to the issue discussed in Section 6.5.3. The TameFJ soundness proof section did not specify that the *match* relation is a function. However, on p. 73 of the TameFJ paper (full version), consider judgment 54 of the TameFJ Subject Reduction Proof for the case the `R-INVK` rule was applied; judgment 54 is repeated below for quick lookup:

$$\overline{T = \overline{[U_s/X_s]}T''}$$

A brief explanation of the derivation of judgment 54:

1. A premise of `R-INVK` is

$$match(sift(\overline{N}; \overline{U}; \overline{Y}); \overline{P}; \overline{Y}; \overline{T})$$

2. The authors of TameFJ applied lemma 37 to derive

$$match(sift(\overline{N}; \overline{U}; \overline{Y}); \overline{P}; \overline{Y}; \overline{\overline{[U_s/X_s]}T''})$$

3. They assumed the *match* relation is a function, so that the above two judgments imply

$$\overline{T = \overline{[U_s/X_s]}T''}$$

To understand the importance of judgment 54, the judgment

$$match(\mathit{sift}(\overline{\mathtt{N}};\overline{\mathtt{U}};\overline{\mathtt{Y}});\overline{\mathtt{P}};\overline{\mathtt{Y}};\;\overline{[\overline{\mathtt{U}_s/\mathtt{X}_s}]\mathtt{T}''})$$

was used to type the method invocation expression:

$$\mathtt{v.\texttt{<}\overline{P}\texttt{>}m(\overline{v})}$$

However, the evaluation step

$$\mathtt{v.\texttt{<}\overline{P}\texttt{>}m(\overline{v})} \mapsto [\overline{\mathtt{v}/\mathtt{x}}, \mathtt{v}/\mathtt{this}, \overline{\mathtt{T}/\mathtt{Y}}]\mathtt{e}_0$$

was performed under the premise

$$match(\mathit{sift}(\overline{\mathtt{N}};\overline{\mathtt{U}};\overline{\mathtt{Y}});\overline{\mathtt{P}};\overline{\mathtt{Y}};\overline{\mathtt{T}})$$

In order, for the subject reduction theorem to hold for TameFJ, it must be the case that the inferred type actuals, $\overline{[\overline{\mathtt{U}_s/\mathtt{X}_s}]\mathtt{T}''}$, used to assign a type to the method invocation expression, must be the same as or *equivalent* to the inferred type actuals, $\overline{\mathtt{T}}$, used to evaluate (perform on a reduction step on) the method invocation. Two types are equivalent if they are subtypes of each other. Although our type system allows variant type constructors, Lemmas 26 and 27 guarantee that our definition of *match* ensures the set of output types for given inputs are equivalent. $\square$ $\square$

APPENDIX C

METHOD BODY ANALYSIS: CONSTRAINTS ON USE-SITE ANNOTATIONS

The heart of method body analysis is the production of constraints bounding use-site variances, $uvar(\mathtt{X}_i;\ \mathtt{C};\ \mathtt{y})$. These bounds ensure that the inferred use-site annotation supports the limited use of $\mathtt{y}$ 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 $\mathtt{y}$ can be more general because it needs to be constrained by the variance of only those members accessed by $\mathtt{y}$ in the method body.

Consider the argument `source` of method `addAll` from line 7 in the motivating example (Figure 9.1 in Section 9.2). 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: $dvar(\mathtt{E};\mathtt{List}) \sqcup o \sqcup uvar(\mathtt{E};\mathtt{List};\mathtt{source})$. The definition-site variance and specified use-site variance were further relaxed by $uvar(\mathtt{E};\ \mathtt{List};\ \mathtt{source})$ 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 $uvar(\mathtt{E};\mathtt{List};\mathtt{source}) = +$. In this case, $uvar(\mathtt{E};\mathtt{List};\mathtt{source})$ 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

211

$uvar(\texttt{E}; \texttt{List}; \texttt{source})$ is $var(\texttt{E}; \texttt{Iterator<E>}) = +.$[1] This is the same upper bound on $dvar(\texttt{E}; \texttt{List})$ that results from $\texttt{List.iterator}$ alone, but $dvar(\texttt{E}; \texttt{List})$ also needs to respect other constraints.

Figure C.1 contains the constraint generation rules for $uvars$ and auxiliary functions. The first three (MB) rules constrain $uvar(\texttt{Y}; \texttt{C}; \texttt{x})$ by the variance of $\texttt{Y}$ in the (non-static) members of $\texttt{C}$ accessed by $\texttt{x}$ in its enclosing method body. A field read is only a covariant use of its type $\texttt{T}$; $var(\texttt{Y}; \texttt{T})$ 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 Figure 6.3 for details.

Constraints with $uvars$ 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 $uvars$ are only generated for top-level use-site annotations. We chose not to generalize use-site annotations in nested types for simplicity.

Section 9.4 did not need to mention $localvar$ to describe the method body analysis at a high level. However, the actual upper bound generated for $dvar(\texttt{X}; \texttt{Body})$ is $-\otimes$ $localvar(\texttt{X}; \texttt{List<}o\texttt{E>}; \texttt{lx}) = -\otimes\big(dvar(\texttt{E}; \texttt{List}) \sqcup o \sqcup uvar(\texttt{E}; \texttt{List}; \texttt{lx})\big)$, so the initial upper bound expression simplifies to the expression previously stated.

$localvar$ returns an expression with a $uvar$ for a method argument $\texttt{x}$ if it is declared with a parametric type and if it satisfies $hierarchyMaybeGeneralized(\texttt{x})$. $hierarchyMaybeGeneralized(\texttt{x})$ 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 $\texttt{x}$ if one of its corresponding

---

[1]Return types are in a covariant position.

uvar constraint generation: (representative rules)

$$\frac{\begin{array}{c} M = enclosingMethod(\mathtt{x}) \\ \text{``}\mathtt{x.f}\text{''} \in M \qquad LookupType(\mathtt{f}) = \mathtt{T} \\ \neg isWriteTarget(\mathtt{x.f}) \end{array}}{uvar(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq var(\mathtt{Y};\mathtt{T})}$$
(MB-FIELDREAD)

$$\frac{\begin{array}{c} M = enclosingMethod(\mathtt{x}) \\ \text{``}\mathtt{x.f = e}\text{''} \in M \\ LookupType(\mathtt{f}) = \mathtt{T} \end{array}}{uvar(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq -\otimes var(\mathtt{Y};\mathtt{T})}$$
(MB-FIELDWRITE)

$$\frac{\begin{array}{c} M = enclosingMethod(\mathtt{x}) \\ \text{``}\mathtt{x.<\overline{S}>m(\overline{e})}\text{''} \in M \\ Lookup(\mathtt{m};\overline{\mathtt{e}}) = \mathtt{<Y \lhd U>\ T\ m(\overline{T}\ x)\ \{\ return\ s;\ \}} \end{array}}{uvar(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \prod_{i=1}^{|\overline{\mathtt{U}}|}\big(-\otimes var(\mathtt{Y};\mathtt{U})\big) \sqcap var(\mathtt{Y};\mathtt{T})\prod_{i=1}^{|\overline{\mathtt{T}}|}\big(-\otimes \boxed{localvar(\mathtt{Y};\mathtt{T}_i;\ \mathtt{x}_i)}\big)}$$
(MB-METHODCALL)

$$\frac{\begin{array}{c} \mathtt{y} \in hierarchyParams(\mathtt{x}) \\ LookupType(\mathtt{y}) = \mathtt{C<\overline{vT}>} \end{array}}{uvar(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \boxed{localvar(\mathtt{Y};\mathtt{C<\overline{vT}>};\ \mathtt{y})}}$$
(MB-OVERRIDE)

$$\frac{\begin{array}{c} M = enclosingMethod(\mathtt{x}) \\ \text{``}\mathtt{y = x}\text{''} \in M \qquad LookupType(\mathtt{y}) = \mathtt{C<\overline{vT}>} \\ \mathtt{Y} = i^{\text{th}} \text{ type parameter of } \mathtt{C} \end{array}}{uvar(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq \boxed{inferredUseSite(\mathtt{y};\ \overline{\mathtt{v}};\ \mathtt{C};\ i)}}$$
(MB-ASSIGNTOGENERIC-SAME)

$$\frac{\begin{array}{c} M = enclosingMethod(\mathtt{x}) \\ \text{``}\mathtt{y = x}\text{''} \in M \\ LookupType(\mathtt{y}) = \mathtt{D<\overline{vT}>} \qquad \mathtt{C} \neq \mathtt{D} \end{array}}{uvar(\mathtt{Y};\mathtt{C};\mathtt{x}) \sqsubseteq dvar(\mathtt{Y};\mathtt{C})}$$
(MB-ASSIGNTOGENERIC-BASE)

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

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

$\boldsymbol{hierarchyMaybeGeneralized}(\mathtt{x}) \equiv \forall \mathtt{y} \in hierarchyParams(\mathtt{x}), \mathtt{y}$ is declared in available source $\land\ sameGeneric(\mathtt{y};\ \mathtt{x})$.

$\boldsymbol{sameGeneric}(\mathtt{x};\ \mathtt{y}) \equiv LookupType(\mathtt{x}) = \mathtt{C<\overline{vT}>} \land LookupType(\mathtt{y}) = \mathtt{C<\overline{wU}>} \land |\overline{\mathtt{v}}| = |\overline{\mathtt{w}}|$.

$\boldsymbol{LookupType}(\mathtt{x}) = $ the declared type of variable $\mathtt{x}$.

$\boldsymbol{isWriteTarget}(\mathtt{e}) \equiv \mathtt{e}$ is the target (left-hand side) of an assignment.

Figure C.1. Constraint Generation from Method Bodies. Shaded parts show where *uvar* constraints differ from the corresponding *dvar* constraint of the signature-only analysis.

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 21 from Section 9.2. 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*(`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 9.3 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 9.3) 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<wU̅>` than that used

in the type of x, C<$\overline{\texttt{vT}}$>, where C $\neq$ D. This can occur when C<$\overline{\texttt{vT}}$> <: D<$\overline{\texttt{wU}}$>. This subtyping relationship may be derived when there exists another instantiation of the base type, D<$\overline{\texttt{v'S'}}$>, such that (1) C<$\overline{\texttt{vT}}$> <: D<$\overline{\texttt{v'S'}}$> holds not because of variance but instead by the class hierarchy and (2) C<$\overline{\texttt{vT}}$> <: D<$\overline{\texttt{v'S'}}$> <: D<$\overline{\texttt{wU}}$>. 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.

APPENDIX D

SOUNDNESS OF REFACTORING

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 35, which is stated later in this section.

To answer the first question, we apply many properties proven in Appendices A and B, adapted to our refactoring setting. The essence of the proofs can be found in those appendices, 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 Appendix A. 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 (Chapter 6) modeled faithfully a subset of Java that supports language features with complex interactions with variance, such as F-bounded polymorphism [13] 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, and then emulating it with use-site variance in a more complete setting, borrowed from VarJ.

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 Appendices A and B 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 [28] does not conclude subtype relationships using inferred definition-site variances. To relate our subtype relation with that of the JLS, we define another subtype relation $<:_{\mathrm{JLS}}$ that is the same as $<:$ except $<:_{\mathrm{JLS}}$ does not support a definition-site subtyping rule. For example, although `Iterator` is inferred to be covariant in its type parameter, `Iterator<Dog>` $\not<:_{\mathrm{JLS}}$ `Iterator<Animal>`. The following lemma establishes that definition-site variance can be simulated with use-site variance.

Lemma 30 (Use-site can simulate def-site). If $T <: C<\overline{vT}>$, then $T <:_{\mathrm{JLS}} C<\overline{wT}>$, where $w_i = dvar(X_i; C) \sqcup v_i$, for each $i \in |\overline{w}|$.

This lemma establishes that any additional subtype relationships that hold for $<:$ but do not hold for $<:_{\mathrm{JLS}}$ 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 $T <:$ `Iterator<Animal>`. The refactoring tool may change the type of `y` to a greater supertype T'. Since the refactored type is always a subtype of the original type according to $<:$, $T' <: T <:$ `Iterator<Animal>`. Lemma 30 implies that $T' <:_{\mathrm{JLS}}$ `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 $T' =$ `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 of Theorem 3 guarantees that `C<(`$v_u \sqcup v_d$`)T>` `<:` `C<`v_u`T>` 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 31 (Def-site joining does not further generalize)**.** Let `C` be a generic class such that `CT(C)` $=$ "`class C<`$\overline{\text{X}}$`> ...`". Then `C<(`$\overline{\text{w} \sqcup \text{v})\text{T}}$`>` `<:` `C<`$\overline{\text{vT}}$`>`, where $\text{w}_i = dvar(\text{X}_i; \text{C})$, for each $i \in |\overline{\text{w}}|$.

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 *ftype*(`f`; `C<`$\overline{\text{vT}}$`>`) such that given the name `f` of a field that exists in generic class `C` and an instantiation of the generic `C<`$\overline{\text{vT}}$`>`, *ftype*(`f`; `C<`$\overline{\text{vT}}$`>`) returns the type of the field for that instantiation. Also, we assume that there is a (partial) function *mtype*(`m`; `C<`$\overline{\text{vT}}$`>`) such that given the name `m` of a method that exists in generic class `C` and an instantiation of the generic `C<`$\overline{\text{vT}}$`>`, *mtype*(`m`; `C<`$\overline{\text{vT}}$`>`) returns the type signature of the method for that instantiation. Formal definitions of *ftype* and *mtype* can be found in Figure 6.4.

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 Lemmas 2 and 3. The proofs of those lemmas rely on intricate reasoning involving the variances of positions in class definitions, and the subtype lifting lemma (Lemma 1), which establishes the

key relationship between variance and subtyping. We restate those lemmas here since properties in this appendix are specified using the FGJ* syntax (given in Figure 9.2).

Lemma 32 (Subtyping Specializes Field Type). If $T' <: T$ and $ftype(\mathtt{f}; \mathtt{T}) = \mathtt{U}$, then $ftype(\mathtt{f}; \mathtt{T'}) <: \mathtt{U}$.

If Lemma 32 were not true, the subtyping relation would violate the subsumption principle; in that case, the supertype $\mathtt{T}$ could return a $\mathtt{U}$ from its field $\mathtt{f}$ 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 33 states this precisely.

Lemma 33 (Subtyping Specializes Method Type). If $\mathtt{T'} <: \mathtt{T}$ and $mtype(\mathtt{m}; \mathtt{T}) = $ `<`$\overline{\mathtt{Z} \lhd \mathtt{S}}$`>` $(\overline{\mathtt{U}}) \to \mathtt{V}$, then $mtype(\mathtt{m}; \mathtt{T'}) = $ `<`$\overline{\mathtt{Z} \lhd \mathtt{S'}}$`>` $(\overline{\mathtt{U'}}) \to \mathtt{V'}$ 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 $(\mathtt{CT}, \mathtt{e})$ of a class table $\mathtt{CT}$ that maps class names to class definitions and an expression $\mathtt{e}$ 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 $(R(\mathtt{CT}), \mathtt{e})$. R is not applied to $\mathtt{e}$ 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 $\mathtt{e}$ has type $\mathtt{T}$ given class table $\mathtt{CT}$.[1] The following key theorem is satisfied by the refactoring tool and establishes that the refactoring preserves compilation.

---

[1]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.

Theorem 6 (Refactored Types Are Safe). Suppose $\mathtt{CT} \vdash \mathtt{e} : \mathtt{C}\mathtt{<}\overline{\mathtt{vT}}\mathtt{>}$, where $\mathtt{CT}(\mathtt{C}) =$ "$\mathtt{class\ C\mathtt{<}\overline{X}\mathtt{>}\ \ldots}$". Then $R(\mathtt{CT}) \vdash \mathtt{e} : \mathtt{C}\mathtt{<}\overline{\mathtt{v'T}}\mathtt{>}$, where for each $i \in |\overline{\mathtt{T}}|$,

$$
v_i' \sqsubseteq
\begin{cases}
dvar(\mathtt{X}_i; \mathtt{C}) \sqcup \mathtt{v}_i \sqcup uvar(\mathtt{X}_i; \mathtt{C}; \mathtt{y}), \\[2mm]
\quad \text{if } \mathtt{e} = \mathtt{y} \text{ and } \mathtt{y} \text{ is a method argument} \\[2mm]
dvar(\mathtt{X}_i; \mathtt{C}) \sqcup \mathtt{v}_i, \text{ otherwise.}
\end{cases}
$$

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. 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 $\mathtt{C}\mathtt{<}\overline{\mathtt{v'T}}\mathtt{>}$ <: $\mathtt{C}\mathtt{<}\overline{\mathtt{vT}}\mathtt{>}$ 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 [12] 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: $R(\texttt{C<}\overline{\texttt{wT}}\texttt{>}) = \texttt{C<}\overline{\texttt{vT}}\texttt{>}$, where $\texttt{w}_i = dvar(\texttt{X}_i; \texttt{C}) \sqcup \texttt{v}_i$ and $\texttt{X}_i$ is the ith type parameter of $\texttt{C}$. Also, $R(\texttt{X}) = \texttt{X}$, where $\texttt{X}$ is a type variable. Given this definition, we state two important properties:

Lemma 34 (Refactored Type is Subtype of Original). $R(\texttt{T}) <: \texttt{T}$.

Lemma 35 (Refactoring Preserves Subtyping). If $\texttt{T} <: \texttt{U}$, then $R(\texttt{T}) <:_{\text{JLS}} R(\texttt{U})$.

Lemma 34 states that the refactored type is a subtype of the original, and it holds because of Lemma 31, Lemma 35 establishes that the refactoring preserves subtype relations from the original program; it is easy to show using Lemmas 31 and 30. We restate Theorem 6 with the signature-only based refactoring:

Theorem 7 (Refactored Types Are Safe for Sig-Only). If $\texttt{CT} \vdash \texttt{e} : \texttt{T}$, then $R(\texttt{CT}) \vdash \texttt{e} : R(\texttt{T})$. We prove this by structural induction on expression $\texttt{e}$.

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

Case: $\texttt{e} = \texttt{new N(}\overline{\texttt{e}}\texttt{)}$. **Proof**: In the original program, $\texttt{e}$ has type $\texttt{N}$. It also has type $\texttt{N}$ in the refactored program, if it type checks. Furthermore, $\texttt{e}$ would have type $R(\texttt{N})$, by the subsumption typing rule, since $\texttt{N} <:_{\text{JLS}} R(\texttt{N})$. So we only need to show that this expression still type checks in the refactored program. Since $\texttt{new N(}\overline{\texttt{e}}\texttt{)}$ has a type, by inversion (similar to [12, Lemma 31]), we know that for each $i \in |\overline{\texttt{e}}|$, the actual argument $\texttt{e}_i$ also has a type that is a subtype of the type of the ith formal argument of the constructor for $\texttt{N}$. We show that this is also the case in the refactored program.

Let $\texttt{T}_i$ be the type of $\texttt{e}_i$ and $\texttt{U}_i$ by the type of the ith formal argument of the constructor for $\texttt{N}$. By the inductive hypothesis, $R(\texttt{CT}) \vdash \texttt{e}_i : R(\texttt{T}_i)$. As discussed

above, it must be the case that $\mathtt{T}_i <:_{\mathrm{JLS}} \mathtt{U}_i$. By Lemma 35, this implies $R(\mathtt{T}_i) <:_{\mathrm{JLS}}$ $R(\mathtt{U}_i)$. Since i was arbitrary in $|\bar{\mathtt{e}}|$, $\mathtt{new\ N(\bar{e})}$ type checks in the refactored program. Therefore, $R(\mathtt{CT}) \vdash \mathtt{new\ N(\bar{e})} : \mathtt{N}$.   □

Case: $\mathtt{e = e'.f}$. **Proof:**   Since $\mathtt{CT} \vdash \mathtt{e'.f} : \mathtt{T}$, 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 34.

5. $\mathit{ftype}(\mathtt{f}; R(\mathtt{U})) <: \mathit{ftype}(\mathtt{f}; \mathtt{U}) <: \mathtt{T}$, by applying Lemma 32 to (4) and (2).

6. $\mathit{ftype}(\mathtt{f}; R(\mathtt{U})) <:_{\mathrm{JLS}} R(\mathtt{T})$, by applying Lemma 30 to (5).

7. $R(\mathtt{CT}) \vdash \mathtt{e'.f} : R(\mathtt{T})$, by (3), (6), and the subsumption typing rule.   □

Case: $\mathtt{e = e'.<\bar{S}>m(\bar{e})}$. **Proof:**   Since $\mathtt{CT} \vdash \mathtt{e'.<\bar{S}>m(\bar{e})} : \mathtt{T}$, 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. $\mathit{mtype}(\mathtt{m}; \mathtt{U}) = \mathtt{<\overline{Z \lhd V}> (\bar{A})} \to \mathtt{B}$

    4. $\overline{\mathtt{U} <: \mathtt{[S/Z]A}}$.   5. $\overline{\mathtt{S} <: \mathtt{[S/Z]V}}$.   6. $\overline{\mathtt{[S/Z]}}\mathtt{B} <: \mathtt{T}$.

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 34.

Applying Lemma 33 to (3) and (8) gives the following four judgments:

9. $mtype(\mathtt{m}; R(\mathtt{U})) = \mathtt{<}\overline{\mathtt{Z} \lhd \mathtt{V'}}\mathtt{>} (\overline{\mathtt{A'}}) \to \mathtt{B'}$

10. $\overline{\mathtt{A} \mathrel{<:} \mathtt{A'}}$.   11. $\overline{\mathtt{V} \mathrel{<:} \mathtt{V'}}$.   12. $\mathtt{B'} \mathrel{<:} \mathtt{B}$.

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

Lemma 36. If $\mathtt{T} \mathrel{<:} \mathtt{T'}$, then $[\overline{\mathtt{U}/\mathtt{X}}]\mathtt{T} \mathrel{<:} [\overline{\mathtt{U}/\mathtt{X}}]\mathtt{T'}$.

Applying Lemma 36 to (10–12) gives the next three judgments:

13. $\overline{[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A} \mathrel{<:} [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A'}}$

14. $\overline{[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V} \mathrel{<:} [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V'}}$

15. $[\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B'} \mathrel{<:} [\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B}$

16. $R(\mathtt{CT}) \vdash \overline{\mathtt{e} : R(\mathtt{U})}$, by applying the inductive hypothesis to (1).

17. $\overline{R(\mathtt{U}) \mathrel{<:} \mathtt{U}}$, by Lemma 34.

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

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

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

21. $\overline{R(\mathtt{U}) \mathrel{<:_{\mathrm{JLS}}} R([\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{A'})}$, by applying Lemma 30 to (18).

22. $\overline{\mathtt{S} \mathrel{<:_{\mathrm{JLS}}} R([\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{V'})}$, by applying Lemma 30 to (19).

23. $R([\overline{\mathtt{S}/\mathtt{Z}}]\mathtt{B'}) \mathrel{<:_{\mathrm{JLS}}} R(\mathtt{T})$, by applying Lemma 35 to (20).

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

Proof of Theorem 6 with Method Body Analysis.

We next sketch the proof of a theorem analogous to Theorem 7 (which specializes Theorem 6 specifically for the signature-only analysis), for the case of the method body analysis. Although the proof of Theorem 7 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 now reason with a function $R_y(\text{T})$ that refactors the type T with the method body analysis assuming that it is the declared type of method argument y:

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

We define another function R^m to model applying the refactoring tool to the entire class table. $R^m(\text{CT})$ is the same as $R(\text{CT})$ except that when R^m is applied to the type T of method argument y, R^m returns $R_y(\text{T})$ instead of $R(\text{T})$. Theorem 6 is restated with the equivalent implication below:

Theorem 8 (Refactored Types Are Safe for Body Analysis)**.** If $\text{CT} \vdash \text{e} : \text{T}$, then:

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

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 $R^m(T) <:_{JLS} R^m(U)$. 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 9.3.3. 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 $R_y(T) <: T$. However, for the *limited use* of y in the method body, it is safe to assume that $R_y(T) <: T$. We define a new subtype relation where $T' <:_y T$ 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 32 and 33 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 $R_y(T)$ and those lemmas would still hold for the particular members accessed by y. Considering method argument `source` from line 7 of Figure 9.1, for example, even though `List` is invariant, we have $R_{source}(\texttt{List<E>}) = \texttt{List<? extends E>} <:_{source}$ `List<E>`. The only member accessed from `source` in the method body is `iterator()`. Lemma 33 holds with the instantiations $T' = \texttt{List<? extends E>}$, $T = \texttt{List<E>}$, and $m = \texttt{iterator}$.

Contrasting with Lemma 35, in general we cannot establish the implication $T <: U \implies R^m(T) <:_{JLS} R^m(U)$ if T is the type of a method argument y. However, if

$T <:_{\mathrm{JLS}} U$ 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 C.1 guarantee $R^m(T) <:_{\mathrm{JLS}} R^m(U)$. For example, if $R^m(T) = $ `C<`$\overline{\mathtt{vT}}$`>` and $R^m(U) = $ `C<`$\overline{\mathtt{wT}}$`>`, 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 $T <:_{\mathrm{JLS}} U \implies R^m(T) <:_{\mathrm{JLS}} R^m(U)$ 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 7 to prove Theorem 8. For example, in the proof of Theorem 7, for the case when `e` = `new N(`$\overline{\mathtt{e}}$`)`, for each $i \in |\overline{e}|$, we have $T_i <:_{\mathrm{JLS}} U_i$, where T_i is the type of the actual argument $\mathtt{e}_i$ and 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, $R^m(T_i) <:_{\mathrm{JLS}} R^m(U_i)$. Hence, the proof of this theorem for the case when `e` = `new N(`$\overline{\mathtt{e}}$`)` still holds. Augmenting the remainder of the proof is similarly straightforward.

BIBLIOGRAPHY

[1] Altidor, John, Huang, Shan Shan, and Smaragdakis, Yannis. Taming the wild-cards: Combining definition- and use-site variance. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 602–613.

[2] Altidor, John, Reichenbach, Christoph, and Smaragdakis, Yannis. Java wildcards meet definition-site variance. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2012), ECOOP'12, Springer-Verlag, pp. 509–534.

[3] Altidor, John, and Smaragdakis, Yannis. Refactoring java generics by inferring wildcards, in practice. In *Proceedings of Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Portland, OR, Oct. 2014), ACM.

[4] America, Pierre, and van der Linden, Frank. A parallel object-oriented language with inheritance and subtyping. In *European Conf. on Object-Oriented Programming and Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP)* (New York, NY, USA, 1990), ACM, pp. 161–168.

[5] Apache Software Foundation. Apache commons-collections library. `http://larvalabs.com/collections/`. Version 4.01.

[6] Aydemir, Brian, Charguéraud, Arthur, Pierce, Benjamin C., Pollack, Randy, and Weirich, Stephanie. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008), POPL '08, ACM, pp. 3–15.

[7] Barendregt, Henk P. *The Lambda Calculus: Its Syntax and Semantics*, vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, Amsterdam, 1984.

[8] Bloch, Joshua. The closures controversy. `http://www.javac.info/bloch-closures-controversy.ppt`. Accessed Dec. 2013.

[9] Bloch, Joshua. *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[10] Boumillion, Kevin, and Levy, Jared. Guava: Google core libraries for Java 1.5+. `http://code.google.com/p/guava-libraries/`. Release 8.

[11] Bracha, Gilad, and Griswold, David. Strongtalk: typechecking smalltalk in a production environment. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 1993), ACM, pp. 215–230.

[12] Cameron, Nicholas, Drossopoulou, Sophia, and Ernst, Erik. A model for Java with wildcards. In *European Conf. on Object-Oriented Programming (ECOOP)* (2008), Springer, pp. 2–26.

[13] Canning, Peter, Cook, William, Hill, Walter, Olthoff, Walter, and Mitchell, John C. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1989), FPCA '89, ACM, pp. 273–280.

[14] Cardelli, L. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 70–79.

[15] Chin, Wei-Ngan, Craciun, Florin, Khoo, Siau-Cheng, and Popeea, Corneliu. A flow-based approach for variant parametric types. In *Proceedings of Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 273–290.

[16] Chlipala, Adam. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.

[17] Cimadamore, Maurizio, and Viroli, Mirko. Reifying wildcards in java using the ego approach. In *Proceedings of the 2007 ACM Symposium on Applied Computing* (New York, NY, USA, 2007), SAC '07, ACM, pp. 1315–1322.

[18] Cook, William. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming (ECOOP)* (Cambridge, UK, 1989), Cambridge University Press, pp. 57–70.

[19] Cook, William R. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages* (London, UK, UK, 1991), Springer-Verlag, pp. 151–178.

[20] Cousineau, Guy, and Mauny, Michel. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[21] Craciun, Florin, Chin, Wei-Ngan, He, Guanhua, and Qin, Shengchao. An interval-based inference of variant parametric types. In *Proceedings of the 18th European Symposium on Programming (ESOP)* (Berlin, Heidelberg, 2009), ESOP '09, Springer-Verlag, pp. 112–127.

[22] Dautelle, Jean-Marie, et al. Jscience. `http://jscience.org/`. Version 4.3.

[23] Dautelle, Jean-Marie, and Keil, Werner. Jsr-275: Measures and units. `http://www.jcp.org/en/jsr/detail?id=275`. Accessed Nov. 2010.

[24] Emir, Burak, Kennedy, Andrew, Russo, Claudio, and Yu, Dachuan. Variance and generalized constraints for c# generics. In *Proceedings of the 20th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2006), ECOOP'06, Springer-Verlag, pp. 279–303.

[25] Friedman, Eric, and Eden, Rob. Gnu Trove: High-performance collections library for Java. `http://trove4j.sourceforge.net/`. Version 2.1.0.

[26] Gödel, Kurt. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems.* Dover Books on Mathematics. Dover Publications, 2012.

[27] Goldberg, Adele, and Robson, David. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[28] Gosling, James, Joy, Bill, Steele, Guy, Bracha, Gilad, and Buckley, Alex. *The Java Language Specification*, 7th ed. California, USA, February 2012.

[29] Greenman, Ben, Muehlboeck, Fabian, and Tate, Ross. Getting f-bounded polymorphism into shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 89–99.

[30] Harper, Robert. *Practical Foundations for Programming Languages.* Cambridge University Press, Dec. 2012.

[31] Hejlsberg, Anders, Wiltamuth, Scott, and Golde, Peter. *C# Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.

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

[33] Igarashi, Atsushi, and Viroli, Mirko. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming* (London, UK, UK, 2002), ECOOP '02, Springer-Verlag, pp. 441–469.

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

[35] Igarashi, Atsushi, and Viroli, Mirko. Variant path types for scalable extensibility. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (New York, NY, USA, 2007), OOPSLA '07, ACM, pp. 113–132.

[36] International, Ecma. *Dart Programming Language Specification Version 1.3*, 1st ed. Ecma International, Mar. 2014.

[37] ISO Standards Committee. ISO/IEC standard 14882: Programming languages – C++, 1998.

[38] Jones, Simon P. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003.

[39] Kennedy, Andrew, and Pierce, Benjamin. On decidability of nominal subtyping with variance. *FOOL/WOOD* (2007).

[40] Kennedy, Andrew, and Syme, Don. Transposing f to c#: expressivity of parametric polymorphism in an object-oriented language: Research articles. *Concurr. Comput. : Pract. Exper. 16* (June 2004), 707–733.

[41] Kiezun, Adam, Ernst, Michael D., Tip, Frank, and Fuhrer, Robert M. Refactoring for parameterizing Java classes. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 437–446.

[42] Malayeri, Donna, and Aldrich, Jonathan. Integrating nominal and structural subtyping. In *Proceedings of the 22nd European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), ECOOP '08, Springer-Verlag, pp. 260–284.

[43] Marlow, Simon. Haskell 2010 language report, 2010.

[44] Martin-Löf, P. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages* (Upper Saddle River, NJ, USA, 1985), Prentice-Hall, Inc., pp. 167–184.

[45] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[46] Meyer, Bertrand. *Object-Oriented Software Construction*, 1st ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[47] Milner, Robin, Tofte, Mads, and Macqueen, David. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[48] Mozilla. Definition-site variance inference in rust. `http://static.rust-lang.org/doc/0.9/rustc/middle/typeck/variance/index.html`. Accessed Aug. 2014.

[49] Mozilla. The rust reference manual. `http://doc.rust-lang.org/rust.html`.

[50] Nipkow, Tobias, Wenzel, Markus, and Paulson, Lawrence C. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

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

[52] Pfenning, F., and Elliot, C. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), PLDI '88, ACM, pp. 199–208.

[53] Pierce, Benjamin C. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[54] Reynolds, John C. User-defined types and procedural data structures as complementary approaches to data abstraction. In *Conference on New Directions on Algorithmic Languages* (Munich, Aug. 1975), Stephen A. Schuman, IFIP WP 2.1.

[55] Robbes, Romain, Röthlisberger, David, and Tanter, Éric. Extensions during software evolution: Do objects meet their promise? In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2012), ECOOP'12, Springer-Verlag, pp. 28–52.

[56] Salcianu, Alex. Java program analysis utilities library. `http://jpaul.sourceforge.net/`. Version 2.5.1.

[57] Schürmann, Carsten. The twelf proof assistant. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), TPHOLs '09, Springer-Verlag, pp. 79–83.

[58] Smith, Daniel. Jep draft: Improved variance for generic classes and interfaces. `http://openjdk.java.net/jeps/8043488`.

[59] Smith, Daniel, and Cartwright, Robert. Java type inference is broken: can we fix it? In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (New York, NY, USA, 2008), ACM, pp. 505–524.

[60] Squeaksource. Squeaksource repository. `http://www.squeaksource.com/`.

[61] Tate, Ross, Leung, Alan, and Lerner, Sorin. Taming wildcards in java's type system. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 614–627.

[62] Thorup, Kresten Krab, and Torgersen, Mads. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *European Conf. on Object-Oriented Programming (ECOOP)* (London, UK, 1999), Springer-Verlag, pp. 186–204.

[63] Torgersen, Mads, Hansen, Christian Plesner, Ernst, Erik, von der Ahe, Peter, Bracha, Gilad, and Gafter, Neal. Adding wildcards to the Java programming language. In *SAC '04: Proc. of the 2004 Symposium on Applied Computing* (Nicosia, Cyprus, 2004), ACM Press, pp. 1289–1296.

[64] Urban, Christian, Berghofer, Stefan, and Norrish, Michael. Barendregt's variable convention in rule inductions. In *In Proc. of the 21th International Conference on Automated Deduction (CADE), volume 4603 of LNAI* (2007), Springer, pp. 35–50.

[65] Viroli, Mirko, and Rimassa, Giovanni. On access restriction with Java wildcards. *Journal of Object Technology 4*, 10 (Dec. 2005), 117–139.

[66] Wadler, Philip. The expression problem. Email, Nov. 1998. Discussion on the Java Genericity mailing list.