Taming the Wildcards:
Combining Definition- and Use-Site Variance

John Altidor
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003, USA
jaltidor@cs.umass.edu

Shan Shan Huang
LogicBlox Inc.
Two Midtown Plaza
Atlanta, GA 30309, USA
ssh@logicblox.com

Yannis Smaragdakis
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003, USA
and Department of Informatics,
University of Athens, 15784, Greece
yannis@cs.umass.edu—smaragd@di.uoa.gr

Abstract

Variance allows the safe integration of parametric and subtype polymorphism. Two flavors of variance, definition-site versus use-site variance, have been studied and have had their merits hotly debated. Definition-site variance (as in Scala and C#) offers simple type-instantiation rules, but causes fractured definitions of naturally invariant classes; Use-site variance (as in Java) offers simplicity in class definitions, yet complex type-instantiation rules that elude most programmers.

We present a unifying framework for reasoning about variance. Our framework is quite simple and entirely denotational, that is, it evokes directly the definition of variance with a small core calculus that does not depend on specific type systems. This general framework can have multiple applications to combine the best of both worlds: for instance, it can be used to add use-site variance annotations to the Scala type system. We show one such application in detail: we extend the Java type system with a mechanism that *modularly infers* the definition-site variance of type parameters, while allowing use-site variance annotations on any type-instantiation.

Applying our technique to six Java generic libraries (including the Java core library) shows that 20-58% (depending on the library) of generic definitions are inferred to have single-variance; 8-63% of method signatures can be relaxed through this inference, and up to 91% of existing wildcard annotations are unnecessary and can be elided.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism,Data types and structures,Classes and objects

General Terms Design, Languages

Keywords variance, definition-site variance, use-site variance, wildcards, language extensions

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

1. Introduction

Genericity is one of the most significant programming language advances of the past 40 years. Variance mechanisms are the keystone of safe genericity in modern programming languages, as they attempt to reconcile the two fundamental forms of genericity: parametric and subtype 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 [2, 9, 13]. Depending on how the type parameter `X` is used, `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>`; or it can be *invariant*, meaning that `C<S>` and `C<T>` are never subtype-related for different types `T` and `S`.

An alternative approach has been introduced in the past decade: *Use-site variance* [17, 22] interprets the definition of a generic class `C<X>` as the introduction of 4 distinct types in the type system: the covariant, contravariant, bivariant, and invariant part of `C<X>`. Each of these types only supports the methods compatible with each variance designation. For instance, the covariant part of `C<T>` would only support methods that would be safe to call on an object of type `C<S>` when `S` is a subtype of `T`. In this way, the use-site of a generic class declares what it can accept, with a vocabulary such as "`C`-of-any-subtype-of-`T`" (concretely written `C<? extends T>` in Java), and the type system checks that the use is indeed valid.

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* 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" [4].) 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.

For these pragmatic reasons, newer languages, such as Scala [20] and C# [15], have returned to and refined the tried-and-true approach of definition-site variance. Definition-site variance is hardly free of usability problems, however. 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.)

In this paper, we introduce an approach that mitigates the tension between use-site and definition-site variance. We present a core calculus, *VarLang*, for reasoning about definition-site variance in the presence of use-site variance annotations (wildcards). The calculus is very general and its soundness is based *directly on the definition of variance*, and not on any specific type system features. This calculus can be used as the basis for variance reasoning in any type system, for either inference or checking. For instance, our approach can be directly employed for extending Scala with use-site variance annotations, or for extending Java with definition-site variance.

We explore one such application in detail: we define a type system that adds *inference* of definition-site variance to Java. That is, we infer variance automatically for classes that are purely covariant, purely contravariant, or purely bivariant with respect to a type parameter. Our solution is fully compatible with existing Java syntax and only changes the type system to make it more permissive in cases of purely variant types, so that exhaustive use-site annotation is unnecessary. For instance, with our type system, the Apache Commons-Collections programmer would not need to write `Iterator<? extends Map.Entry<? extends K,V>>` because `Iterator` and `Map.Entry` are inferred to be purely covariant with respect to their (first) parameter, hence the type `Iterator<Map.Entry<K,V>>` is exactly as general.

Illustration of approach. 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, ⊗ (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 ⊗, 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.

Contributions. Our paper makes several contributions:

- We present a general approach for reasoning about variance. The approach consists of a core calculus, whose soundness is proved by direct appeal to the definition of variance, i.e., independently of any specific type system. To our knowledge, this is the first approach to allow general reasoning about definition-site variance in the presence of either use-site variance annotations or type recursion.
- We apply the approach in the context of Java to produce a (conservative) inference algorithm for computing the definition-site variance of classes and interfaces. Unlike past work (e.g., [19]), our inference technique is *modular*: the variance of a generic class is purely determined by its own interface definition (and that of the types it uses), *not* by the code that uses the class. Under our updated type system, many uses of variance annotations become redundant. All previously legal Java programs remain legal, although some newly type-correct programs might have been previously rejected.
- We conduct a large study to show how use-site and definition-site variance coexist fruitfully, even in code that was written with only use-site variance in mind. Past literature [12, 17] has largely assumed that single-variant type parameters are seldom used, thus promoting use-site variance for flexibility. We show that this is not the case—over all libraries (including the standard library: all packages in `java.*`) we find that 32% of the generic classes and interfaces have single-variance type parameters. This is strong evidence for the need to combine definition- and use-site variance, since they both occur commonly. Additionally, 13% of all signatures of methods using generics are found to use too-restricted types. (Importantly, this analysis is without concern for what the method actually does—i.e., treating the body as a black box, which can change in the future.) Furthermore, our inference algorithm obviates the need for many uses of wildcards: 37% of existing wildcard uses in these libraries are rendered unnecessary.

2. Type-Checking Variance

The study of variance has a long history [2, 6–9, 13, 16, 17, 23]. It aims to provide safe conditions for subtyping between different instantiations of the same generic class or interface. Consider the following example:

```
class List<X> {
  void set(int i, X x) { ... }
  X    get(int i)      { ... }
}
```

If we naively assume that List<S> <: List<T> for any S <: T (where <: is the subtyping relation), the following ill-typed code could pass compile-time type-checking, and result in runtime errors:

```
List<Integer> intList = new List<Integer>();
List<Number>  numList = intList;
numList.set(0, new Float(0.0f));
            // Writing Float into intList!
```

The key to safe parametric subtyping lies with the concept of the *variance* of type parameter X in the definition of C<X>. If X only appears *covariantly*, then it is always safe to assume that C<S> <: C<T> if S <: T. If X only appears *contravariantly*, it is always safe to assume that C<S> <: C<T> if T <: S. If X appears both co- and contravariantly, C is invariant: C<S> <: C<T> only if S = T. This notion of variance is called *definition-site variance*.

2.1 Definition-site Variance.

Languages supporting definition-site variance [15, 20] typically require each type parameter to be declared with a variance annotation. For instance, Scala [20] requires the annotation + for covariant type parameters, - for contravariant type parameters, and invariance is the default. A well-established set of rules can then be used to verify that the use of the type parameter in the generic[1] is consistent with the annotation. We provide an overview of these rules here, as they form the basis of both use-site variance and our inference.

Each typing position in a generic's signature has an associated variance. For instance, method return and exception types, supertypes, and upper bounds of type parameters are covariant positions; method argument types and 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.* Consider the following templates of Scala classes, where v_X, v_Y, and v_Z stand for variance annotations.

```
abstract class RList[$v_X$X] { def get(i:Int):X }
abstract class WList[$v_Y$Y] { def set(i:Int, y:Y):Unit }
abstract class IList[$v_Z$Z] { 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 type checks 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 type checks only if $v_Z = o$ because Z occurs in a covariant and a contravariant position.

Intuitively, RList is a read-only list: it only supports retrieving objects. 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. Thus, RList is covariant in X.[2] Similarly, WList is a write-only list, and is intuitively contravariant. Its definition supports this intuition: Objects of type T can be written to a write-only list of Ts and to a write-only list of WList[S], where T <: S, because objects of type T are also objects of type S. Hence, a WList[S] can be safely 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 was 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 variance is *use-site variance* [7, 17, 23]. 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 those methods and fields of C in which X appears covariantly. 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 considered a covariant position, and the lower bound of a wildcard a contravariant position.

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

[2] We use interchangeably the wordings "*C* is *v*-variant in *X*", "*C* is *v*-variant/has variance *v* with respect to *X*", and "*X*'s variance in *C* is *v*". For brevity, if it is clear from the context, we do not specify which type parameter a definition-site variance is with respect to.

For example, `List<? extends T>`, only has access to method "`X get(int i)`", but not method "`void set(int i, X x)`". (More precisely, method `set` can only be called with `null` for its second argument. We elide such fine distinctions in this section.)

Similarly, `C<? super T>` is the contravariant version of `C`, and is a supertype of any `C<S>` and `C<? super S>`, where `T <: S`. Of course, `C<? super T>` has access only to methods and fields in which `X` appears contravariantly or not at all.

Use-site variance also allows the representation of the *bivariant* version of a generic. In Java, this is accomplished through the unbounded wildcard: `C<?>`. Using this notation, `C<S> <: C<?>`, for any `S`. The bivariant type, however, only has 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 instantiation rules and subtyping relationships are clear. However, the class or interface designer must pay for such simplicity by splitting the definitions of data 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. However, the users of these generics must pay the price, by carefully considering 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 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 instance, 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 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 above example of the long signatures in the two type-instantiations of `Iterator`. 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: (at least) 8% 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 5.2.

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 different 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.

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 [17]): "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 as 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>>`.[3] This 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 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,[4] then the variance of `X` in type expression `C<`E`>` is v_3.

The behavior of the transform operator is simple: invariance transforms everything into invariance, bivariance transforms everything into bivariance, covariance transforming a variance leaves it

[3] 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 covariant in `X` iff $T_1 <: T_2 \implies E[T_1/\mathtt{X}] <: E[T_2/\mathtt{X}]$. (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.)

[4] 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.

Definition of variance transformation: ⊗

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

Figure 1. Variance transform operator.

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:

$$\begin{aligned} T_1 \;<:\; T_2 &\implies && \text{(by contravariance of } E\text{)}\\ E[T_2/\mathtt{X}] \;<:\; E[T_1/\mathtt{X}] &\implies && \text{(by covariance of } \mathtt{C}\text{)}\\ \mathtt{C<}E[T_2/\mathtt{X}]\mathtt{>} \;<:\; \mathtt{C<}E[T_1/\mathtt{X}]\mathtt{>} &\implies &&\\ \mathtt{C<}E\mathtt{>}[T_2/\mathtt{X}] \;<:\; \mathtt{C<}E\mathtt{>}[T_1/\mathtt{X}] & && \end{aligned}$$

  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:

$$\begin{aligned} \text{for any types } S \text{ and } T &\implies && \text{(by bivariance of } \mathtt{C}\text{)}\\ \mathtt{C<}E[S/\mathtt{X}]\mathtt{>} \;<:\; \mathtt{C<}E[T/\mathtt{X}]\mathtt{>} &\implies &&\\ \mathtt{C<}E\mathtt{>}[S/\mathtt{X}] \;<:\; \mathtt{C<}E\mathtt{>}[T/\mathtt{X}] & && \end{aligned}$$

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

As can be seen by inspection of all cases in Figure 1, operator ⊗ 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 ⊗ 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 $o\otimes*=o$:

$$\begin{aligned} \mathtt{C<}E\mathtt{>}[S/\mathtt{X}] \;<:\; \mathtt{C<}E\mathtt{>}[T/\mathtt{X}] &\implies &&\\ \mathtt{C<}E[S/\mathtt{X}]\mathtt{>} \;<:\; \mathtt{C<}E[T/\mathtt{X}]\mathtt{>} &\implies && \text{(by invariance of } \mathtt{C}\text{)}\\ E[S/\mathtt{X}] = E[T/\mathtt{X}] &\implies && \text{(assuming } \mathtt{X} \text{ occurs in } E\text{)}\\ S = T & && \end{aligned}$$

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, $E[S/\mathtt{X}] \;<:\; E[T/\mathtt{X}]$ and $E[T/\mathtt{X}] \;<:\; E[S/\mathtt{X}]$. Hence, $E[S/\mathtt{X}]$ is equivalent to $E[T/\mathtt{X}]$ 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 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. Section 4.5 of [20] 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 ⊗ 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 [17] showed how to reason about use-site variance for Java. Emir et al [13] formalized definition-site variance as it occurs in of C#.[5] 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 type checks 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). 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.

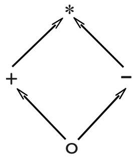

Figure 2. Usual variance lattice.

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

[5] Their calculus is an extension of C# minor [18].

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 $\bigsqcup_{\mathtt{T'} <: \mathtt{T}}$ C<T'>. This definition yields the standard variance theorems $\mathtt{T} <: \mathtt{T'} \Rightarrow$ 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.) We have:

$$\begin{aligned} \bot <: \mathtt{T} \Rightarrow &\qquad \text{(by first theorem above)} \\ \mathtt{C<+}\bot\mathtt{>} <: \mathtt{C<+T>} &\qquad (1) \end{aligned}$$

But also, for any type T':

$$\begin{aligned} \bot <: \mathtt{T'} \Rightarrow &\qquad \text{(C contravariant)} \\ \mathtt{C<T'>} <: \mathtt{C<}\bot\mathtt{>} &\qquad (2) \end{aligned}$$

Therefore:

$$\begin{aligned} \mathtt{C<+T>} = &\qquad \text{(by variance def)} \\ \bigsqcup_{\mathtt{T'} <: \mathtt{T}} \mathtt{C<T'>} <: &\qquad \text{(by (2), above)} \\ \mathtt{C<}\bot\mathtt{>} <: &\qquad \text{(by second theorem above)} \\ \mathtt{C<+}\bot\mathtt{>} &\qquad (3) \end{aligned}$$

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 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.

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 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".

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 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.

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.

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.

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 and requires some form of search. 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.

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_XX] { def foo:C1[X] }
trait C2[v_YY] extends C1[Y] { def bar(arg:Y):Unit }
trait C3[v_ZZ] 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.

$v_X = -$	$v_Y = -$	$v_Z = o$
$v_X = +$	$v_Y = o$	$v_Z = +$
$v_X = o$	$v_Y = o$	$v_Z = o$

4. Putting It All Together: 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*, 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 3.

$$\mathtt{M} \in \mathit{Module} ::= \mathtt{module}\ \mathtt{C}\text{<}\overline{\mathtt{X}}\text{>}\ \{\ \overline{\mathtt{Tv}}\ \}$$

$$\mathtt{T} \in \mathit{Type} ::= \mathtt{X} \mid \mathtt{C}\text{<}\overline{\mathtt{vT}}\text{>}$$

$$\mathtt{v} \in \mathit{Variance} ::= + \mid - \mid * \mid o$$

$$\mathtt{C} \in \mathit{ModuleNames} \text{ is a set of module names}$$

$$\mathtt{X} \in \mathit{VariableNames} \text{ is a set of variable names}$$

Figure 3. 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 the Introduction. 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). The constraints are "below"-inequalities and contain variables of the form $\mathit{var}(\mathtt{X},\ \mathtt{T})$ and $\mathit{var}(\mathtt{X},\ \mathtt{C})$, 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.)

$$\begin{gathered}\mathit{var}(\mathtt{X},\ \mathtt{C}) \sqsubseteq \mathtt{v}_i \otimes \mathit{var}(\mathtt{X},\ \mathtt{T}_i), \forall i,\\ \text{where } \mathtt{module\ \ C<}\overline{\mathtt{X}}\mathtt{>\ \{\ }\overline{\mathtt{Tv}}\mathtt{\ \}} \in S\end{gathered} \tag{1}$$

$$\mathit{var}(\mathtt{X},\ \mathtt{C<>}) \sqsubseteq * \tag{2}$$

$$\mathit{var}(\mathtt{X},\ \mathtt{Y}) \sqsubseteq *, \text{ where } \mathtt{X} \neq \mathtt{Y} \tag{3}$$

$$\mathit{var}(\mathtt{X},\ \mathtt{X}) \sqsubseteq + \tag{4}$$

$$\begin{gathered}\mathit{var}(\mathtt{X},\ \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}) \sqsubseteq (\mathtt{v}_i \sqcup \mathit{var}(\mathtt{Y},\ \mathtt{C})) \otimes \mathit{var}(\mathtt{X},\ \mathtt{T}_i), \forall i,\\ \text{where } \mathtt{Y} \text{ is the } i\text{-th type variable in the definition of } \mathtt{C}.\end{gathered} \tag{5}$$

Rule 1 specifies that for each type $\mathtt{T}_i$ in module `C`, the variance of the type variable `X` in `C` must be below the variance of `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 2 and 3 specify that the `X` can have any variance in a type expression for which it does not occur in. Rule 4 constrains the initial variance of a type variable to be at most covariance.

Rule 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 [17]. The proof can be found in the extended version of this paper [1, Appendix I].

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:

$$\begin{array}{rll}
\mathit{var}(\mathtt{X},\ \mathtt{C}) & \sqsubseteq + \otimes \mathit{var}(\mathtt{X},\ \mathtt{X}) & \text{(rule 1)}\\
\mathit{var}(\mathtt{X},\ \mathtt{X}) & \sqsubseteq + & \text{(rule 4)}\\
\mathit{var}(\mathtt{X},\ \mathtt{C}) & \sqsubseteq - \otimes \mathit{var}(\mathtt{X},\ \mathtt{C<-X>}) & \text{(rule 1)}\\
\mathit{var}(\mathtt{X},\ \mathtt{C<-X>}) & \sqsubseteq (- \sqcup \mathit{var}(\mathtt{X},\ \mathtt{C})) \otimes \mathit{var}(\mathtt{X},\ \mathtt{X}) & \text{(rule 5)}\\
\mathit{var}(\mathtt{X},\ \mathtt{C}) & \sqsubseteq + \otimes \mathit{var}(\mathtt{X},\ \mathtt{void}) & \text{(rule 1)}\\
\mathit{var}(\mathtt{X},\ \mathtt{void}) & \sqsubseteq * & \text{(rule 2)}\\
\mathit{var}(\mathtt{X},\ \mathtt{C}) & \sqsubseteq - \otimes \mathit{var}(\mathtt{X},\ \mathtt{D<+X>}) & \text{(rule 1)}\\
\mathit{var}(\mathtt{X},\ \mathtt{D<+X>}) & \sqsubseteq (+ \sqcup \mathit{var}(\mathtt{Y},\ \mathtt{D})) \otimes \mathit{var}(\mathtt{X},\ \mathtt{X}) & \text{(rule 5)}\\
\mathit{var}(\mathtt{Y},\ \mathtt{D}) & \sqsubseteq + \otimes \mathit{var}(\mathtt{Y},\ \mathtt{void}) & \text{(rule 1)}\\
\mathit{var}(\mathtt{Y},\ \mathtt{void}) & \sqsubseteq * & \text{(rule 2)}\\
\mathit{var}(\mathtt{Y},\ \mathtt{D}) & \sqsubseteq - \otimes \mathit{var}(\mathtt{Y},\ \mathtt{C<oY>}) & \text{(rule 1)}\\
\mathit{var}(\mathtt{Y},\ \mathtt{C<oY>}) & \sqsubseteq (o \sqcup \mathit{var}(\mathtt{X},\ \mathtt{C})) \otimes \mathit{var}(\mathtt{Y},\ \mathtt{Y}) & \text{(rule 5)}\\
\mathit{var}(\mathtt{Y},\ \mathtt{Y}) & \sqsubseteq + & \text{(rule 3)}
\end{array}$$

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. 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 $\mathit{var}(\mathtt{X},\ \mathtt{C})$. 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 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 $var(\mathtt{X},\ \mathtt{C})$ but not $var(\mathtt{X},\ \mathtt{T})$. We can expand $var(\mathtt{X},\ \mathtt{T})$ terms with their upper bounds containing only unknowns of the form $var(\mathtt{X},\ \mathtt{C})$ Consider the constraint generated from `foo`'s argument type: $var(\mathtt{X},\ \mathtt{C}) \sqsubseteq - \otimes var(\mathtt{X},\ \mathtt{C{<}{-}X{>}})$. Because we are computing a maximal solution and because of the monotonicity of $\sqcup$ and $\otimes$, we can replace $var(\mathtt{X},\ \mathtt{C{<}{-}X{>}})$ and $var(\mathtt{X},\ \mathtt{X})$ by their upper bounds, rewriting the constraint as:

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

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. $var(\mathtt{X},\ \mathtt{void}) \sqsubseteq *$). This leads to the following constraints generated for the example two Java classes:

$$\begin{array}{rcl}
\texttt{foo}\text{ return type} & \Longrightarrow & var(\mathtt{X},\ \mathtt{C}) \sqsubseteq + \otimes \underbrace{\ +\ }_{var(\mathtt{X},\ \mathtt{X})} \\
\texttt{foo}\text{ arg type} & \Longrightarrow & var(\mathtt{X},\ \mathtt{C}) \sqsubseteq - \otimes \underbrace{(var(\mathtt{X},\ \mathtt{C}) \sqcup -)}_{var(\mathtt{X},\ \mathtt{C{<}{-}X{>}})} \\
\texttt{bar}\text{ arg type} & \Longrightarrow & var(\mathtt{X},\ \mathtt{C}) \sqsubseteq - \otimes \underbrace{(var(\mathtt{Y},\ \mathtt{D}) \sqcup +)}_{var(\mathtt{X},\ \mathtt{D{<}{+}X{>}})} \\
\texttt{baz}\text{ arg type} & \Longrightarrow & var(\mathtt{Y},\ \mathtt{D}) \sqsubseteq - \otimes \underbrace{(var(\mathtt{X},\ \mathtt{C}) \sqcup o)}_{var(\mathtt{Y},\ \mathtt{C{<}oY{>}})}
\end{array}$$

Letting c denote $var(\mathtt{X},\ \mathtt{C})$ and d denote $var(\mathtt{Y},\ \mathtt{D})$, the above constraints correspond to the four constraints presented in the Introduction.

For each expanded constraint $r \sqsubseteq l$ in a constraint system, r is a $var(\mathtt{X},\ \mathtt{C})$ term and l is an expression where the only unknowns are $var(\mathtt{X},\ \mathtt{C})$ terms. The greatest fixed-point of a constraint system is solved for by, first, assigning every $var(\mathtt{X},\ \mathtt{C})$ 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 $var(\mathtt{X},\ \mathtt{C})$ terms no longer change, which results in computing the greatest fixed-point. Finally, computing the greatest fixed-point runs in at most $O(2n)$ 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).

5. 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*, used it to produce a (definition-site) variance *inference* algorithm, and evaluated its impact on large Java libraries with generics (including the standard library).

5.1 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 2): 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 translate some "tricky" aspects of the language into an invariance constraint, potentially making our algorithm infer less general variances than is occasionally possible. For instance, we do 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 consider this to be an instance of an invariant position. Another source of conservatism is that 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.

The reason for our conservatism is that we do not want to reason about 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 [7]. We prefer to use only positions of unquestionable variance 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

Library		# Type	# Generic	Type Definitions					Recursive	Unnecess.	Over-specif.
		defs	defs	invar.	variant	cov.	contrav.	biv.	variances	wildcards	methods
java.*	classes	5550	99	69%	31%	20%	17%	4%	10%	12%	7%
	interfaces	1710	44	43%	57%	41%	25%	0%	22%	12%	16%
	total	7260	143	61%	39%	27%	20%	3%	14%	12%	8%
JScience	classes	70	25	76%	24%	0%	0%	24%	76%	90%	19%
	interfaces	51	11	55%	45%	0%	9%	36%	7%	100%	11%
	total	121	36	69%	31%	0%	3%	28%	53%	91%	19%
Apache Collec.	classes	226	187	66%	34%	14%	21%	2%	4%	63%	17%
	interfaces	23	22	55%	45%	32%	18%	0%	24%	20%	0%
	total	249	209	65%	35%	16%	21%	1%	6%	62%	17%
Guava	classes	204	101	90%	10%	9%	1%	0%	8%	51%	18%
	interfaces	35	26	46%	54%	38%	19%	4%	18%	38%	5%
	total	239	127	81%	19%	15%	5%	1%	10%	50%	17%
GNU Trove	classes	25	8	62%	38%	12%	25%	0%	33%	0%	62%
	interfaces	8	4	0%	100%	25%	100%	0%	0%	0%	0%
	total	33	12	42%	58%	17%	50%	0%	20%	0%	62%
JPaul	classes	77	65	75%	25%	12%	12%	5%	18%	86%	23%
	interfaces	9	9	67%	33%	11%	33%	0%	9%	0%	0%
	total	86	74	74%	26%	12%	15%	4%	17%	86%	23%
Total	**classes**	**6152**	**485**	**73%**	**27%**	**13%**	**14%**	**3%**	**11%**	**42%**	**13%**
	interfaces	**1836**	**116**	**47%**	**53%**	**32%**	**24%**	**4%**	**18%**	**19%**	**15%**
	total	**7988**	**601**	**68%**	**32%**	**17%**	**16%**	**3%**	**13%**	**39%**	**13%**

Figure 4. Class/Interface Inference Results

Library		# Type	Type Parameters				
		Parameters	inv.	variant total	cov.	contrav.	biv.
java.*	classes	128	67%	33%	16%	14%	3%
	interfaces	54	46%	54%	33%	20%	0%
	total	182	61%	39%	21%	16%	2%
JScience	classes	29	79%	21%	0%	0%	21%
	interfaces	14	64%	36%	0%	7%	29%
	total	43	74%	26%	0%	2%	23%
Apache Collec.	classes	254	72%	28%	11%	16%	1%
	interfaces	33	64%	36%	24%	12%	0%
	total	287	71%	29%	13%	15%	1%
Guava	classes	150	93%	7%	7%	1%	0%
	interfaces	39	54%	46%	26%	18%	3%
	total	189	85%	15%	11%	4%	1%
GNU Trove	classes	9	67%	33%	11%	22%	0%
	interfaces	6	0%	100%	17%	83%	0%
	total	15	40%	60%	13%	47%	0%
JPaul	classes	93	76%	24%	10%	9%	5%
	interfaces	11	64%	36%	9%	27%	0%
	total	104	75%	25%	10%	11%	5%
Total	**classes**	**663**	**77%**	**23%**	**10%**	**10%**	**3%**
	interfaces	**157**	**53%**	**47%**	**24%**	**20%**	**3%**
	total	**820**	**72%**	**28%**	**13%**	**12%**	**3%**

Figure 5. Type Parameter Inference Results

of the time is spent on loading, parsing, and processing files, with under 30sec constraint solving time.

Finally, we need to emphasize that our 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. If we were to allow our algorithm to generate constraints after inspecting method bodies, we would get 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 would have 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. This is yet another way in which our numbers are conservative and only compute a lower bound of the possible impact of integrating definition-site variance inference in Java.

5.2 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 Sun's JDK 1.6, i.e., classes and interfaces in the packages of `java.*`. The other libraries are JScience [10], a Java library for scientific computing; Guava [5], a superset of the Google collections library; GNU Trove [14]; Apache Commons-Collection [3]; and JPaul [21], a library supporting program analysis.

The results of our experiment appear in Figure 4. Together, these libraries define 7,988 classes and interfaces, out of which 601 are generics. 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. The statistics 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. Statistics per type parameter are in included in Figure 5.

As can be seen, 32% 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 1/3 of the generics defined should be allowed to enjoy general variant subtyping without users having to annotate them with wildcards.

The next column shows how many generics have a recursive variance constraint. One can see that these numbers are usually low, especially considering that they include direct self-recursion (e.g., a trivial constraint $var(\texttt{X},\ \texttt{C}) \sqsubseteq + \otimes var(\texttt{X},\ \texttt{C})$).

The last two columns illustrate 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 instantiations of `Iterator<? extends T>`, for any `T`, are unnecessary. This number shows that, using our technique, 37% 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 13% of methods are over-specified. This means that 13% 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.

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) [11], included with the JScience library [10], 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.

6. Conclusions

In this work we showed that the need for the flexibility of use-site variance is somewhat overrated, and the rigidity of definition-site variance is unwarranted. We introduce the first unified framework for reasoning about variance, allowing type system designs that combine the benefits of both definition- and use-site variance. Thus, our approach resolves questions that are central in the design of any language involving parametric polymorphism and subtyping. Our work is the first to fully study how definition-site variance interacts with use-site variance annotations and type recursion. Our calculus allows to reason about complex constraints independently of specific type systems on variances involved in recursive class definitions. As a specific application, we introduce a type system for Java that *modularly infers* definition-site variance, while allowing use-site variance annotations where necessary. We show that combining definition-site and use-site variance allows us to infer more general types than either approach alone. Our Java implementation demonstrates the benefits and practicality of our approach, and our algorithm can also be used in other contexts, such as to aid a programmer via an IDE plugin to choose types that generalize method signatures as much as possible.

Acknowledgments

We would like to thank Jens Palsberg (especially for valuable insights on the constraint solving algorithm) as well as Christoph Reichenbach and several anonymous reviewers for their suggestions that helped improve the paper. This work was funded by the National Science Foundation under grants CCF-0917774, CCF-0934631, and IIP-0838747 to the University of Massachusetts.

References

[1] J. Altidor, S. S. Huang, and Y. Smaragdakis. Taming the wildcards: Combining definition- and use-site variance (extended version). `http://www.cs.umass.edu/~yannis/variance-extended2011.pdf`.

[2] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *OOPSLA/ECOOP '90: Proc. of the European Conf. on object-oriented programming on Object-oriented programming systems, languages, and applications*, 1990.

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

[4] J. Bloch. The closures controversy. http://www.javac.info/bloch-closures-controversy.ppt. Accessed Nov. 2010.

[5] K. Boumillion and J. Levy. Guava: Google core libraries for Java 1.5+. http://code.google.com/p/guava-libraries/. Accessed Nov. 2010.

[6] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proc. of the Conf. on Object-oriented programming systems, languages, and applications*, 1993.

[7] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *ECOOP '08: Proc. of the 22nd European Conf. on Object-Oriented Programming*, 2008.

[8] R. Cartwright and J. Guy L. Steele. Compatible genericity with run-time types for the Java programming language. In *OOPSLA '98: Proc. of the 13th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, 1998.

[9] W. Cook. A proposal for making eiffel type-safe. In *ECOOP '89: Proc. of the 3rd European Conf. on Object-Oriented Programming*, 1989.

[10] J.-M. Dautelle et al. Jscience. http://jscience.org/. Accessed Nov. 2010.

[11] J.-M. Dautelle and W. Keil. Jsr-275: Measures and units. http://www.jcp.org/en/jsr/detail?id=275. Accessed Nov. 2010.

[12] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: constraining parametric polymorphism. In *OOPSLA '95: Proc. of the tenth annual Conf. on Object-oriented programming systems, languages, and applications*, 1995.

[13] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In *ECOOP '06: Proc. of the European Conf. on Object-Oriented Programming*, 2006.

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

[15] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.

[16] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development*, 2007.

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

[18] A. Kennedy and D. Syme. Transposing f to c#: expressivity of parametric polymorphism in an object-oriented language: Research articles. *Concurr. Comput. : Pract. Exper.*, 16:707–733, 2004.

[19] A. Kiezun, M. Ernst, F. Tip, and R. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE '07: Proc. of the 29th Intl. Conf. on Software Engineering*, 2007.

[20] M. Odersky. *The Scala Language Specification v 2.8*. 2010.

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

[22] K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *ECOOP '99: Proc. of the European Conf. on Object-Oriented Programming*, 1999.

[23] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahe, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *SAC '04: Proc. of the 2004 Symposium on Applied Computing*, 2004.