Java Wildcards Meet Definition-Site Variance

John Altidor[1], Christoph Reichenbach[1], and Yannis Smaragdakis[1,2]

[1] University of Massachusetts, Amherst
[2] University of Athens, Greece

Abstract. Variance is concerned with the interplay of parametric polymorphism (i.e., templates, generics) and subtyping. The study of variance gives answers to the question of when an instantiation of a generic class can be a subtype of another. In this work, we combine the mechanisms of use-site variance (as in Java) and definition-site variance (as in Scala and C#) in a single type system, based on Java. This allows maximum flexibility in both the specification and use of generic types, thus increasing the reusability of code. Our VarJ calculus achieves a safe synergy of def-site and use-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.

1 Introduction

Consider a generic type `C<X>`. When is a type-instantiation `C<`*Exp1*`>` a subtype of another type instantiation `C<`*Exp2*`>`? This is the question that *variance* mechanisms in modern programming languages try to answer. Variance (specifically, subtype variance with respect to generic type parameters) is a key topic in language design because it develops the exact rules governing the interplay of the two major forms of polymorphism: parametric polymorphism (i.e., generics or templates) and subtype (inclusion) polymorphism.

Languages like C# and Scala 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 instance, `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* [11]: 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`

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 509–534, 2012.

`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.

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 natural combination of the two kinds of variance is complicated especially by the interaction of use-site and definition-site annotations: for example, when does the declared variance of a type variable agree with occurrences of that variable in use-site annotations? We recently proposed a unifying framework for checking and inferring both definition and use-site variance [1]. That proposal was not accompanied by a language operational semantics however—its proof of soundness was 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 was welcome as an intuition about why it makes sense to combine variances in a certain way, but did not establish a firm connection with any real programming language.

This paper investigates combining of definition- and use-site variance with all relevant language constructs using a new formal model, VarJ. VarJ applies novel ideas and integrates techniques from various formalisms: Java wildcards are a form of use-site variance that was proven sound with the TameFJ [4] calculus. VarJ directly extends TameFJ with definition-site variance. VarJ also employs ideas from our VarLang calculus [1], which introduces a variance transform operator. Finally, VarJ integrates definition-site subtyping rules from the work of Kennedy et al. [13,8]. The result is a language with highly expressive genericity. For instance, given an invariant class `List`, our type system allows defining a (definition-site) covariant class `ROStack` that returns covariant (intuitively: read-only) `List`s of members:

```
class ROStack<+X> {
  X pop() { ... }
  List<? extends X> toList() { ... }
}
```

Note the simultaneous use of a definition-site variance annotation (`+`) on `ROStack`, as well as a use-site variance annotation on its `toList` method. The former is not safe without the latter.

Overall our work 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., [10]) 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 wildcard 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.

2 Background

We next offer 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 [14,9] typically require each type parameter to be declared with a variance annotation. For instance, Scala [14] 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

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

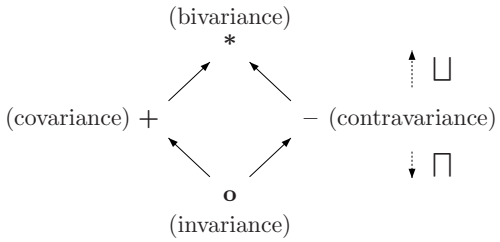

Fig. 1. Standard variance lattice

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 1 presents the variance lattice. Consider the following templates of Scala classes, where v_X, v_Y, and v_Z stand for variance annotations.

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

Intuitively, `RList` is a read-only list: it only supports retrieving objects. The return type of a method indicates this "retrieval" capability. Retrieving objects of type `T` can be safely thought of as retrieving objects of any supertype of `T`. Thus, a read-only list of `T`s (`RList[T]`) can always be safely thought of as a read-only list of some supertype of `T`s (`RList[S]`, where `T<:S`). 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*. Its definition supports this intuition: Objects of type `T` can be written to a write-only list of `T`s (`WList[S]`) and written to a write-only list of `S`s (`WList[S]`), where `T<:S`, because objects of type `T` are also objects of type `S`. Hence, a `WList[S]` can safely be thought of as a `WList[T]`, if `T<:S`.

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.

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, since designating a type parameter as bivariant typically means it does not appear in the generic's type signature. Nevertheless, the concept is useful in our more general treatment.

2.2 Use-Site Variance

An alternative approach to variance is *use-site variance* [11,18,4]. 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 considered 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, `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. (The `get` method returns `Object` for a `C<? 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: `C<?>`. Using this notation, `C<S>`<:`C<?>`, 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 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. 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>>)
```

3 Combining Definition- and Use-Site Variance

Our formalism supports combining definition- and use-site variance in the context of Java. We next see informally some of its main insights and complications.

3.1 Insights for Combining Variances

High-level elements of our approach are inherited from our earlier work [1], in which we presented rules for combining definition- and use-site variance in a type system. These rules are a significant generalization over what has been explored in the past (mainly in Scala) in two ways:[2]

- The variance of an arbitrary type expression with respect to a type variable is defined for all cases with the help of a "transform" operator, $\otimes$. The operator determines how variances compose. Given two generic types `A<X>` and `B<X>` with declared variances v_A and v_B for their parameters (i.e., declared as `A<`v_A `X>` and `B<`v_B `X>`), we can compute the variance of type `A<B<X>>` as $v = v_A \otimes v_B$. Variances take values from the lattice of Figure 1.

  Figure 2 summarizes the behavior of the transform operator: invariance and bivariance override other variances, while covariance preserves and contravariance reverses variance (with invariance and bivariance being their own reverses, respectively). To sample why the definition of the transform operator makes sense, we derive one case relating the inputs and output. (Remaining cases are derived similarly.)

  - **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 two types T_1 and T_2:

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

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

Definition of variance transformation: $\otimes$

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

Fig. 2. Variance transform operator

- The interaction of use-site and definition-site variance is expressed as a join operation on the same variance lattice of Figure 1. In the VarLang calculus [1], types have the form `C<`$\overline{v}T$`>`, where $\overline{\mathtt{v}}$ are use-site annotations. Considering the `List` generic from Section 2.2, for example, `List<+T>` passes type `T` to a *covariant*

[2] There is also a third way: our earlier framework allowed reasoning about unknown variances, represented as variables in recursive constraints, thus enabling variance *inference* instead of *checking*. This capability is not relevant here, however.

version of the `List`, where the `add` method was "removed"[3] because it contains the type parameter in a contravariant position. If generic `C<X>` has definition-site variance v_1 with respect to `X`, then the type expression `C<`v_2`X>` has variance $v_1 \sqcup v_2$ with respect to `X`. Consider a covariant class `RList`. If we request a contravariant instantiation, we end up with a bivariant type expression ($+ \sqcup - = *$). That is, a method "`void foo(RList<? super Animal> l) {...}` can really accept any `RList`: the method is guaranteed to never use its argument in a way that reveals anything about the type of element in the list. (In practice, this means that the method may only take the size of the list, or only treats its elements as being of the general type `Object`, etc.)

In practice, the ability to combine definition- and use-site variance gives the programmer maximum flexibility. The variance of a generic class does not need to be anticipated at its definition site. Consider the usual invariant `List` class (from Section 2.2). This `List` supports both reading and writing of data, hence it includes both kinds of methods, instead of being split into two types (as, for instance, is common in the Scala libraries). The methods that use `List` can still be made fully general, however, 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.

At the same time, allowing definition-site variance eliminates much of the need for extensive use-site variance annotations and the risk of too-restricted types: purely variant types can be declared up-front without burdening the programmer at the use point.

Of course, combining definition- and use-site variance means more than just using each kind separately, when applicable. It also includes using one kind of annotation when reasoning about the other. For instance, consider the example of a read-only stack type that we briefly saw in the Introduction. The stack refers to the invariant class `List` (defined earlier):

```
class ROStack<+X> {
  X pop() { ... }
  List<? extends X> toList() { ... }
}
```

Note the use of a definition-site variance annotation (`+`) on `ROStack`, as well as a use-site variance annotation on its `toList` method. The example would not have been safe if the return type of `toList` were merely `List<X>`. With such an invariant use of type parameter `X`, we could use a (dynamic) `List<Dog>` as a (static) `List<Animal>` and thus dynamically add a `Cat` (which is fine to add to a `List<Animal>`) to a `List<Dog>`.

3.2 Realistic Complications

The main contribution of our present work consists of formalizing and proving sound the combination of definition- and use-site variance in the context of Java.

[3] Again, method `add` can only be called with `null` for its second argument.

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.

One complication is that Java wildcards are not merely use-site variance, but also include mechanisms inspired by existential typing mechanisms. *Wildcard capture* is the process of passing an unknown type, hidden by a wildcard, as a type parameter in a method invocation. Consider the following method, which swaps the order the two elements at the top of a stack.

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

Although a programmer may want to pass an object of type `Stack<?>` as a value argument to the `swapLastTwo` method, the type parameter to pass for `E` cannot be manually specified because the type hidden by `?` cannot be named by the programmer. However, passing a `Stack<?>` type checks because Java allows the compiler to automatically generate a name for the unknown type (capture conversion) and use this name in a method invocation. Our formalism has to model wildcard capture and its interaction with definition-site variance. This handling comprises some of the more significant changes of our formalism relative to TameFJ [4].

Another major complication concerns F-bounded polymorphism. 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). For 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):[4]

```
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()); }
Trouble<YourList> preitr = ...;
foo(preitr);
```

Function `foo` type checks 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`).

The problem arises in the last line. 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 `YourLists`. Thus, the joining of definition- and use-site variances needs to be carefully restricted in the presence of F-bounded polymorphism. We revisit the above example more formally in Section 5.

4 VarJ

We investigate extending Java with definition-site variance by developing the VarJ calculus. VarJ's syntax found in Figure 3 is a slight extension of the TameFJ syntax. A program that type checks in TameFJ also type checks 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, $\bot$, 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 $\exists\emptyset.\texttt{X}$; substitution is performed as usual except $[\texttt{T}/\texttt{X}]\exists\emptyset.\texttt{X} = \texttt{T}$; $\star$ is a syntactic marker designating that a method type parameter (i.e., for a polymorphic method) should be inferred. Class type parameters ($\overline{\texttt{X}}$) now have definition-site variance annotations ($\overline{\texttt{v}}$) and lower bounds ($\overline{\texttt{B}_L}$). Method type variables now have lower bounds as well. The remainder of this section focuses on semantic differences between VarJ and TameFJ and new concepts from adding definition-site variance. Sections 4.1 and 4.2 formally present notions of the variance of type expression and the variance of a type position. Section 4.3 covers subtyping with definition-site and use-site variance in VarJ. Section 4.4 discusses the updates made to allow safe interaction between wildcard capture and variant types.

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

Syntax:

$$
\begin{array}{lll}
\texttt{e} & ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} & \textit{expressions} \\
\texttt{s} & ::= \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{s}}\texttt{)} & \textit{values} \\
\texttt{v} & ::= + \mid - \mid * \mid o & \textit{variance} \\
\\
Q & ::= \texttt{class C<}\overline{\texttt{v X} \rightarrow \texttt{[}\texttt{B}_L\texttt{-B}_U\texttt{]}}\texttt{>} \lhd \texttt{N} \{\ \overline{\texttt{T f;}}\ \ \overline{M}\ \} & \textit{class declarations} \\
M & ::= \texttt{<}\overline{\texttt{X} \rightarrow \texttt{[}\texttt{B}_L\texttt{-B}_U\texttt{]}}\texttt{>}\ \texttt{T m(}\overline{\texttt{T x}}\texttt{)}\ \{\ \texttt{return e;}\ \} & \textit{method declarations} \\
\\
\texttt{N} & ::= \texttt{C<}\overline{\texttt{T}}\texttt{>} & \textit{non-variable types} \\
\texttt{R} & ::= \texttt{N} \mid \texttt{X} & \textit{non-existential types} \\
\texttt{T} & ::= \exists \Delta.\texttt{N} \mid \exists \emptyset.\texttt{X} & \textit{existential types} \\
\texttt{B} & ::= \texttt{T} \mid \bot & \textit{type bounds} \\
\texttt{P} & ::= \texttt{T} \mid \star & \textit{method type parameter} \\
\\
\Delta & ::= \overline{\texttt{X} \rightarrow \texttt{[}\texttt{B}_L\texttt{-B}_U\texttt{]}} & \textit{type ranges} \\
\Gamma & ::= \overline{\texttt{x : T}} & \textit{var environments} \\
\texttt{X} & ::= \ldots & \textit{type vars} \\
\texttt{x} & ::= \ldots & \textit{expr vars} \\
\texttt{C} & ::= \ldots & \textit{class names}
\end{array}
$$

Lookup Functions:
Shared premise for lookup rules except F-Obj:

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

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

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

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

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

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

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

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

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

Fig. 3. Syntax and Lookup Functions

4.1 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 4.3. When is a type instantiation `C<`$Exp1$`>` a subtype of another instantiation `C<`$Exp2$`>`? We answer a more general question by defining a general predicate $var(\texttt{X};\texttt{T})$, where `X` is a type variable and `T` is a type expression. The goal of *var* is to determine: 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 $\texttt{[U/X]T} <: \texttt{[U'/X]T}$, if $\texttt{U} <: \texttt{U}'$.

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

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

Note that, by the variance lattice (in Figure 1), we have

$$\texttt{v} \leq \texttt{w} \implies \Big[\texttt{v}(\texttt{T};\texttt{T}') \implies \texttt{w}(\texttt{T};\texttt{T}')\Big] \tag{1}$$

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 [8]:

$$var(\texttt{X};\texttt{T}) = \texttt{v} \implies \Big[\texttt{v}(\texttt{U};\texttt{U}') \implies \texttt{[U/X]T} <: \texttt{[U'/X]T}\Big] \tag{2}$$

By (1), (2) entails a more general implication:

$$\texttt{v} \leq var(\texttt{X};\texttt{T}) \implies \Big[\texttt{v}(\texttt{U};\texttt{U}') \implies \texttt{[U/X]T} <: \texttt{[U'/X]T}\Big] \tag{3}$$

We assume there is a usual class table CT that maps class identifiers `C` to their definition (i.e. $CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \rightarrow \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\quad \ldots\ \}$). 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 i^{th} def-site variance annotation; e.g, if $VT(\texttt{C}) = \overline{\texttt{vX}}$, then $VT(\texttt{C}, i) = \texttt{v}_i$.

The expression $var(\texttt{X};\texttt{B})$ computes the variance of type variable `X` in type expression `B`. Figure 4 contains *var*'s definition. *var*'s type input is overloaded for non-existential types (`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 a technical report available at `http://people.cs.umass.edu/~jaltidor/ecoop12tr.pdf`.)

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

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

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

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

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

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

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

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

Fig. 4. Variance of types and ranges

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

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 $+(\mathtt{T}; \mathtt{U})$, which is equivalent to $\mathtt{T} <: \mathtt{U}$, then $\mathtt{[T/X]X} = \mathtt{T} <: \mathtt{U} = \mathtt{[U/X]X}$. 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. 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 $\overline{[\mathtt{B}_L - \mathtt{B}_U]}$ is a *subrange* of range $\overline{[\mathtt{A}_L - \mathtt{A}_U]}$ if $\overline{\mathtt{A}_L <: \mathtt{B}_L}$ and $\overline{\mathtt{B}_U <: \mathtt{A}_U}$. 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.[5] The subtype lifting lemma can be used to entail subrange relationships:

$$var(\mathtt{X}; \overline{\mathtt{Y} \to \mathtt{[B}_L\mathtt{-B}_U\mathtt{]}}) = \mathtt{v} \textbf{ and } \mathtt{v(T;U)}$$
$$\implies \overline{\mathtt{[U/X]B}_L <: \mathtt{[T/X]B}_L} \textbf{ and } \overline{\mathtt{[T/X]B}_U <: \mathtt{[U/X]B}_U}$$

The variance of an existential type variable is just the meet of the variances of its range (Δ) and its body ($\mathtt{R}$). The VAR-T rule has the premise "$\mathtt{X} \notin dom(\Delta)$" to follow *Barendregt's variable convention* [19], as in the TameFJ formalism. (*var* is undefined when this premise is not satisfied.) The variable convention substantially reduces the number of places requiring alpha-conversion to be applied and allows for more elegant proofs. The rules of the convention basically are: (1) relations in the type system are equivariant (respect alpha-renaming), and (2) no binder (declared variable) in a rule occurs free in the conclusion. For example, condition (1) holds for $var(\mathtt{X}; \mathtt{T})$ because we can rename binders

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

to fresh names in existential types in T without changing the variance of X in T. Without the premise of rule VAR-T, this property would no longer hold. So that the important premises are more apparent, 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.

4.2 Variance of a Position

To see how *var* is used in our type system, we need to consider the variance of positions in a class definition. For example, return types are assumed to be in covariant positions while argument types are assumed to be in contravariant positions. These assumptions are used to type check class and method definitions and their def-site variance annotations.

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

$$\left[\overline{\mathtt{vX}} \vdash \mathtt{B}\ \mathit{mono}\right] \equiv \left[\overline{\mathtt{v}} \leq \mathit{var}(\overline{\mathtt{X}}; \mathtt{B})\right] \tag{4}$$

$$\left[\neg \mathtt{v}\right] = \left[- \otimes \mathtt{v}\right] \tag{5}$$

A "monotonicity" judgment of the syntactic form "$\overline{\mathtt{vX}} \vdash \mathtt{T}\ \mathit{mono}$" appears originally in Emir et al.'s definition-site variance treatment [8] and later in Kennedy and Pierce [13] as "$\overline{\mathtt{vX}} \vdash \mathtt{T}\ \mathit{ok}$". The semantics of these judgments in the aforementioned sources are similar to its definition here but differs in that they had no function similar to *var* nor a variance lattice. The negation operator $\neg$ also appears in [8] and [13] and is used to transform a variance by contravariance. Using the implications in Section 4.1, it is easy to show the following properties, which are important for type checking class definitions:

$$\mathtt{w} = \neg \mathtt{v} \implies \left[\mathtt{v}(\mathtt{B}, \mathtt{B}') \iff \mathtt{w}(\mathtt{B}', \mathtt{B})\right] \tag{6}$$

$$\overline{\mathtt{vX}} \vdash \mathtt{B}\ \mathit{mono} \implies \left[\overline{\mathtt{v}(\mathtt{T}, \mathtt{U})} \implies \mathtt{[}\overline{\mathtt{T}/\mathtt{X}}\mathtt{]B} <: \mathtt{[}\overline{\mathtt{U}/\mathtt{X}}\mathtt{]B}\right] \tag{7}$$

$$\overline{\neg \mathtt{vX}} \vdash \mathtt{B}\ \mathit{mono} \implies \left[\overline{\mathtt{v}(\mathtt{T}, \mathtt{U})} \implies \mathtt{[}\overline{\mathtt{U}/\mathtt{X}}\mathtt{]B} <: \mathtt{[}\overline{\mathtt{T}/\mathtt{X}}\mathtt{]B}\right] \tag{8}$$

Figure 5 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 are used to compute the types of members (fields and methods) in class definitions. Their definitions are in Figure 3. These lookup functions take in non-variable types (N) instead of existential types. In the expression typing rules (in Figure 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 4.4 has a brief overview of this process and an example type derivation.

Class and Method Typing:

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

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

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

Wellformed Ranges: $\Delta \vdash \Delta\ \mathit{OK}$

$$\frac{}{\Delta \vdash \emptyset\ \mathit{OK}} \text{ (W-Rng-empty)} \qquad \frac{\texttt{X} \notin \mathit{dom}(\Delta) \qquad \Delta, \texttt{X} \to \texttt{[}\texttt{B}_L\texttt{-}\texttt{B}_U\texttt{]}, \Delta' \vdash \texttt{B}_L, \texttt{B}_U\ \mathit{OK} \qquad \Delta \vdash \mathit{ubound}_\Delta(\texttt{B}_L) \sqsubset: \mathit{ubound}_\Delta(\texttt{B}_U) \qquad \Delta \vdash \texttt{B}_L <: \texttt{B}_U \qquad \Delta, \texttt{X} \to \texttt{[}\texttt{B}_L\texttt{-}\texttt{B}_U\texttt{]} \vdash \Delta'\ \mathit{OK}}{\Delta \vdash \texttt{X} \to \texttt{[}\texttt{B}_L\texttt{-}\texttt{B}_U\texttt{]}, \Delta'\ \mathit{OK}} \text{ (W-Rng)}$$

Non-Variable Upper Bound: $\mathit{ubound}_\Delta(\texttt{B})$

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

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

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

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{vX} \to \texttt{[}\texttt{B}_L\texttt{-}\texttt{B}_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\ \ldots\ \} \\ \Delta \vdash \overline{\texttt{[T/X]B}_L <: \texttt{T}} \qquad \Delta \vdash \texttt{T} <: \overline{\texttt{[T/X]B}_U} \\ \Delta \vdash \overline{\texttt{T}}\ \mathit{OK}\end{array}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \mathit{OK}} \text{ (W-N)} \qquad \frac{\Delta \vdash \Delta'\ \mathit{OK} \qquad \Delta, \Delta' \vdash \texttt{R}\ \mathit{OK}}{\Delta \vdash \exists\Delta'.\texttt{R}\ \mathit{OK}} \text{ (W-T)}$$

Wellformed Expression Variable Environments $\Delta \vdash \Gamma\ \mathit{OK}$

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

Fig. 5. Wellformedness Judgments

The definition-site subtyping relation judgment $\Delta \vdash \texttt{N} \prec: \texttt{N}'$ is defined over non-variable types and considers definition-site annotations when concluding subtype relationships. For example, $VT(\texttt{C}) = +\texttt{X} \implies \Delta \vdash \texttt{C<}\exists\emptyset\texttt{.Dog>} \prec: \texttt{C<}\exists\emptyset\texttt{.Animal>}$, assuming $\Delta \vdash \exists\emptyset\texttt{.Dog} <: \exists\emptyset\texttt{.Animal}$. This relation is defined in Figure 6.

Definition-Site Subtyping: $\texttt{R} \prec: \texttt{R}$

$$\frac{\texttt{class C<}\overline{\texttt{vX} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{>} \lhd \texttt{N}\ \{\ \ldots\ \} \quad \texttt{C} \neq \texttt{D} \quad \Delta \vdash \texttt{[}\overline{\texttt{T/X}}\texttt{]N} \prec: \texttt{D<}\overline{\texttt{U}}\texttt{>}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{D<}\overline{\texttt{U}}\texttt{>}}\ \text{(SD-Super)} \qquad \frac{VT(\texttt{C}) = \overline{\texttt{vX}} \quad \Delta \vdash \overline{\texttt{v(T,U)}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \prec: \texttt{C<}\overline{\texttt{U}}\texttt{>}}\ \text{(SD-Var)} \qquad \frac{}{\Delta \vdash \texttt{X} \prec: \texttt{X}}\ \text{(SD-X)}$$

Existential Subtyping: $\Delta \vdash \texttt{B} \sqsubset: \texttt{B}$

$$\frac{\Delta, \Delta' \vdash \texttt{N} \prec: \texttt{N}'}{\Delta \vdash \exists\Delta'\texttt{.N} \sqsubset: \exists\Delta'\texttt{.N}'}\ \text{(SE-SD)} \qquad \frac{}{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}}\ \text{(SE-Refl)} \qquad \frac{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}' \quad \Delta \vdash \texttt{B}' \sqsubset: \texttt{B}''}{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}''}\ \text{(SE-Tran)}$$

$$\frac{}{\Delta \vdash \bot \sqsubset: \texttt{B}}\ \text{(SE-Bot)} \qquad \frac{dom(\Delta') \cap fv(\exists\overline{\texttt{X} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{.N}) = \emptyset \quad fv(\overline{\texttt{T}}) \subseteq dom(\Delta, \Delta') \quad \Delta, \Delta' \vdash \overline{\texttt{[T/X]}}\texttt{B}_L <: \texttt{T} \quad \Delta, \Delta' \vdash \texttt{T} <: \overline{\texttt{[T/X]}}\texttt{B}_U}{\Delta \vdash \exists\Delta'\texttt{.[}\overline{\texttt{T/X}}\texttt{]N} \sqsubset: \exists\overline{\texttt{X} \to \texttt{[B}_L\texttt{-B}_U\texttt{]}}\texttt{.N}}\ \text{(SE-Pack)}$$

Subtyping: $\Delta \vdash \texttt{B} <: \texttt{B}$

$$\frac{\Delta \vdash \texttt{B} \sqsubset: \texttt{B}'}{\Delta \vdash \texttt{B} <: \texttt{B}'}\ \text{(ST-SE)} \qquad \frac{\Delta \vdash \texttt{B} <: \texttt{B}' \quad \Delta \vdash \texttt{B}' <: \texttt{B}''}{\Delta \vdash \texttt{B} <: \texttt{B}''}\ \text{(ST-Tran)} \qquad \frac{\Delta(\texttt{X}) = [\texttt{B}_L - \texttt{B}_U]}{\Delta \vdash \texttt{B}_L <: \exists\emptyset\texttt{.X}}\ \text{(ST-Lbound)} \quad \frac{}{\Delta \vdash \exists\emptyset\texttt{.X} <: \texttt{B}_U}\ \text{(ST-Ubound)}$$

Fig. 6. Subtyping Relations

The motivation for the assumed variances of positions is to ensure the subsumption principle holds for the subtyping hierarchy. Informally, if $\texttt{T} <: \texttt{U}$, then a value of type T may be provided whenever a value of type U is required. In the case of VarJ, the 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 f of the supertype $\texttt{N}'$ become a more specific type for the subtype 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<`$\overline{\texttt{vX} \rightarrow \texttt{[...]}}$`> ◁ N...` *OK* and (b) $\Delta \vdash$ `C<`$\overline{\texttt{T}}$`>` $\prec:$ `N′` and (c) *ftype*(`f`; `N′`) = `T`, then $\Delta \vdash$ *ftype*(`f`; `C<`$\overline{\texttt{T}}$`>`) $<:$ `T`.[6]

Lemma 3 (Subtyping Specializes Method Type).
If (a) $\vdash$ `class C<`$\overline{\texttt{vX} \rightarrow \texttt{[...]}}$`> ◁ N...` *OK* and (b) $\Delta \vdash$ `C<`$\overline{\texttt{T}}$`>` $\prec:$ `N′` and (c) *mtype*(`m`; `N′`) = `<`$\overline{\texttt{Y} \rightarrow \texttt{[}B_L\texttt{-}B_U\texttt{]}}$`>` $\overline{\texttt{U}} \rightarrow$ `U`, then: (1) *mtype*(`m`; `C<`$\overline{\texttt{T}}$`>`) = `<`$\overline{\texttt{Y} \rightarrow \texttt{[}A_L\texttt{-}A_U\texttt{]}}$`>` $\overline{\texttt{V}} \rightarrow$ `V`, (2) $\Delta \vdash$ `V` $<:$ `U`, (3) $\Delta \vdash \overline{\texttt{U} <: \texttt{V}}$, (4) $\Delta \vdash \overline{A_L <: B_L}$, (5) $\Delta \vdash \overline{B_U <: A_U}$, and (6) $var(\overline{\texttt{Y}}; \texttt{U}) = var(\overline{\texttt{Y}}; \texttt{V})$.

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 (7), *not* negating the def-site variance annotations, $\overline{\texttt{v}}$, in the judgment "$\overline{\texttt{vX}} \vdash$ `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 (8), negating the def-site variance annotations in the judgment "$\overline{\neg\texttt{vX}} \vdash$ `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.[7]

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 [8]. More generally, if (1) `e.<T>m()` type checks 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`.

4.3 Subtyping

Subtyping in VarJ is defined similarly to TameFJ. Figure 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 $\prec:$ defined on non-existential types. Def-site

[6] If field assignments were allowed, then field types would be in both co- and contravariant positions, and both *ftype*(`f`; `C<`$\overline{\texttt{T}}$`>`) and *ftype*(`f`; `N′`) would be subtypes of each other.

[7] Bounds on class type parameters may make unrestricted use of type parameters by similar reasoning as in [8, p.7]. Once an object is created, they are forgotten.

subtyping is defined by the SD-* rules, which are similar to the subtyping rules from [13]. Like the subtype relation from [13], $\prec:$ is defined by syntax-directed rules[8] and shares the reflexive and transitive properties by similar reasoning as in [13]. The $\prec:$ 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 $\sqsubset:$ 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 its premise was updated to use def-site subtyping. SE-SD allows def-site subtyping to be applied to both type variables in the type context Δ and existential type variables in Δ'. As a result, a type packed to 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>` $<:$ $\exists$`Y` $\rightarrow$ `[Dog-Animal].Iterator<Y>`. Subtyping between two types implies the subsumption principle between the types. Since `Iterator<Dog>` can be packed to $\exists$`X` $\rightarrow$ `[Dog-Animal].Iterator<X>` and `Iterator<PrettyDog>` $<:$ `Iterator<Dog>`, it must be the case that `Iterator<PrettyDog>` can also be packed to $\exists$`X` $\rightarrow$ `[Dog-Animal].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) $\Delta, \Delta' \vdash \texttt{R}' \prec: \texttt{[}\overline{\texttt{T/X}}\texttt{]R}$ and (2) $\Delta, \Delta' \vdash \texttt{[}\overline{\texttt{T/X}}\texttt{]B}_L <: \texttt{T}$ and (3) $\Delta, \Delta' \vdash \texttt{T} <: \texttt{[}\overline{\texttt{T/X}}\texttt{]B}_U$ 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 5.

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

4.4 Typing and Wildcard Capture

The expression typing rules in VarJ are mostly the same as in TameFJ and are given in Figure 7. Unlike TameFJ, VarJ allows method signatures to have lower

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

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 [4] for more thorough coverage.

Expression Typing. Consider the Java segment below. It type checks 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 $\exists \mathtt{X} \rightarrow$ `[⊥-String].Box<X>`. Expression typing judgments have the form $\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$. 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 typing derivation below illustrates this process on typing the "`box.elem`" expression from the previous code segment, where we assume $\Gamma = \mathtt{box} : \exists \mathtt{X} \rightarrow$ `[⊥-String].Box<X>`.

$$\dfrac{\dfrac{\emptyset;\ \Gamma \vdash \mathtt{box} : \exists \mathtt{X} \rightarrow \mathtt{[\bot\text{-}String].Box<X>} \mid \emptyset \qquad \mathit{ftype}(\mathtt{elem}; \mathtt{Box<X>}) = \mathtt{X}}{\emptyset;\ \Gamma \vdash \mathtt{box.elem} : \mathtt{X} \mid \mathtt{X} \rightarrow \mathtt{[\bot\text{-}String]}}\ (\text{T-Field}) \qquad \dfrac{\emptyset, \mathtt{X} \rightarrow \mathtt{[\bot\text{-}String]} \vdash \mathtt{X} <: \mathtt{String} \qquad \emptyset \vdash \mathtt{X} \rightarrow \mathtt{[\bot\text{-}String]}\ OK \qquad \emptyset \vdash \mathtt{String}\ OK}{}}{\emptyset;\ \Gamma \vdash \mathtt{box.elem} : \mathtt{String} \mid \emptyset}\ (\text{T-Subs})$$

Matching for Wildcard Capture. The T-Invk rule type checks a method invocation and uses *match* to perform wildcard 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 `<`$\overline{\mathtt{Y}}$`>m(`$\overline{\mathtt{U}}$`)` and called with types `<`$\overline{\mathtt{P}}$`>m(`$\overline{\exists \Delta.\mathtt{R}}$`)`. The parameters of $match(\overline{\mathtt{R}}; \overline{\mathtt{U}}; \overline{\mathtt{P}}; \overline{\mathtt{Y}}; \overline{\mathtt{T}})$ and their expected conditions are:

1. The bodies of the actual value argument types of a method invocation ($\overline{\mathtt{R}}$).
2. The formal argument value types of a method ($\overline{\mathtt{U}}$).
3. The *specified* type actuals of a method invocation ($\overline{\mathtt{P}}$).
4. The formal type arguments of a method ($\overline{\mathtt{Y}}$).
5. The *inferred* type actuals of a method invocation ($\overline{\mathtt{T}}$).

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

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

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

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

Match:

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

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

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

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

Fig. 7. Expression Typing and Auxiliary Functions For Wildcard Capture

Figure 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 types, *match* is needed to compute the concrete types to substitute for the formal type arguments' ($\overline{\texttt{Y}}$) occurrences in the method body.

Computation Rules: $\texttt{e} \mapsto \texttt{e}$

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

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

Congruence Rules: $\texttt{e} \mapsto \texttt{e}$

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

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

Fig. 8. Reduction Rules

Sifting for Wildcard Capture. 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 inference from types that are in "fixed" or invariant positions. Without applying *sift*, counter examples to the subject reduction (type preservation) theorem can result. First, note that the following two judgments are derivable.

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

Assume `List` is invariant and consider the following.

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

```
createList<*>(new Dog()) : List<Animal>
↦ new List<Dog>() : List<Dog>
```

The expression `createList<*>(new Dog())` can be typed with `List<Animal>` because `new Dog()` : `Animal`, and the inferred type actual used for typing the expression can be `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, 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>`.

In TameFJ, *sift* filters out a pair of a type actual body `R` and a formal type `U`, if $\texttt{U} = \exists\emptyset.\texttt{X}$ and `X` is one of the formal type arguments ($\overline{\texttt{Y}}$). 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. We find the restriction of not allowing wildcard capture in variant positions not to be 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. The VarJ grammar does not allow the bottom type $\bot$ to be specified as a type actual. However, we have not found any practical need for wildcard capture with contravariant types.

4.5 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 $\Delta, \texttt{e}, \texttt{T}$, if $\emptyset;\ \emptyset \vdash \texttt{e} : \texttt{T} \mid \Delta$, then either $\texttt{e} \mapsto \texttt{e}'$ or there exists a `v` such that $\texttt{e} = \texttt{v}$.

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

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 6 is probably the main one, which relates subtyping and wildcard capture, and is similar to lemma 36 from [4]. It states that the method receiver's ability to perform wildcard capture is preserved in subtypes with respect to the

method receiver. (A similar lemma holds for method arguments.) It shows that the subsumption principle holds even under interaction with wildcard capture.

Lemma 6 (Subtyping Preserves *match*ing (receiver)). If (a) $\Delta \vdash \exists\Delta_1.\mathtt{N}_1 \sqsubset: \exists\Delta_2.\mathtt{N}_2$ and (b) $mtype(\mathtt{m};\mathtt{N}_2) = \overline{\texttt{<Y}_2 \to \texttt{[B}_{2L}\texttt{-B}_{2U}\texttt{]>}}\ \overline{\mathtt{U}_2} \to \mathtt{U}_2$ and (c) $mtype(\mathtt{m};\mathtt{N}_1) = \texttt{<Y}_1 \to \texttt{[B}_{1L}\texttt{-B}_{1U}\texttt{]>}\ \overline{\mathtt{U}_1} \to \mathtt{U}_1$ and (d) $sift(\overline{\mathtt{R}};\overline{\mathtt{U}_2};\overline{\mathtt{Y}_2}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'_2})$ and (e) $match(\overline{\mathtt{R}'};\overline{\mathtt{U}'_2};\overline{\mathtt{P}};\overline{\mathtt{Y}_2};\overline{\mathtt{T}})$ and (f) $\emptyset \vdash \Delta\ OK$ and (g) $\Delta, \Delta' \vdash \overline{\mathtt{T}}\ OK$ then: (1) $sift(\overline{\mathtt{R}};\overline{\mathtt{U}_1};\overline{\mathtt{Y}_1}) = (\overline{\mathtt{R}'};\overline{\mathtt{U}'_1})$ and (2) $match(\overline{\mathtt{R}'};\overline{\mathtt{U}'_1};\overline{\mathtt{P}};\overline{\mathtt{Y}_1};\overline{\mathtt{T}})$.

5 Discussion

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. Our earlier work [1] presented a definition of $var(\mathtt{X};\mathtt{U})$ that performed a simple *boundary analysis* to compute such irrelevant bounds. As discussed in Section 3.1, 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 [1]: $var(\mathtt{X};\texttt{C<-T>}) = (+ \sqcup -) \otimes var(\mathtt{X};\mathtt{T}) = * \otimes var(\mathtt{X};\mathtt{T}) = *$. Hence, $var(\mathtt{X};\texttt{C<-T>}) = *$, even if `X` occurred in the lower bound, `T`.

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 $\mathtt{T} \equiv \mathtt{U}$ denotes $\mathtt{T} <: \mathtt{U} \wedge \mathtt{U} <: \mathtt{T}$): $\exists\mathtt{X} \to$ `[Dog-Animal].Iterator<X>` $\equiv \exists\mathtt{X} \to$ `[⊥-Animal].Iterator<X>`. Clearly, $\exists\mathtt{X} \to$ `[Dog-Animal].Iterator<X>` $<: \exists\mathtt{X} \to$ `[⊥-Animal].Iterator<X>` because the range of the type variable is wider in the supertype. The inverse is derivable by applying a combination of the SE-SD, SE-Pack, and ST-* rules:

$$\exists\mathtt{X} \to \texttt{[⊥-Animal].Iterator<X>} <: \exists\mathtt{X} \to \texttt{[⊥-Animal].Iterator<Animal>}$$
$$<: \exists\emptyset.\texttt{Iterator<Animal>} <: \exists\mathtt{X} \to \texttt{[Dog-Animal].Iterator<X>}$$

As we saw in Section 3.2, this reasoning is not sound in the presence of F-bounded polymorphism. It is important to realize that the issue with recursive bounds is not specific to the use of bounds in type definitions.[9] The counterexample of Section 3.2 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 bound, `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 3.2:
$\exists\mathtt{X} \to$ `[YourList-List<X>].Iterator<X>` $<: \exists\mathtt{X} \to$ `[MyList-List<X>].Iterator<X>`.
The above subtype relationship would violate the subsumption principle. The

[9] In our earlier work [1], when we performed an application to Java it sufficed to be overly conservative at this point: the mere appearance of a type variable in the upper bound of a type definition caused us to consider the definition as invariant relative to this type variable. For VarJ, which is richer in terms of where bounds can appear, even this kind of conservatism is not sufficient.

latter type can return a $\exists$`X` $\rightarrow$ `[MyList-List<X>].List<X>` from its `next` method, but the former type cannot because
$\exists$`X` $\rightarrow$ `[YourList-List<X>].List<X>` $\not<:$ $\exists$`X` $\rightarrow$ `[MyList-List<X>].List<X>`, by the invariance of `List`. In contrast to the earlier, correct subtyping, VarJ does not support the above erroneous subtyping because it cannot establish that the upper bounds of the two instatiations of `Iterator` are related: 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 skipped by programmers. 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` $\rightarrow$ `[`$\bot$`-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 the user types in) combined with a simple but sound checking inside the compiler.

Definition-Site Variance and Erasure. A practical issue with definition-site variance concerns its use with an erasure-based translation. Consider a covariant `class A<+X> {...}` and an invariant subtype `class B<oX> extends A<X> {...}`. We can then have:

```
A<Integer> a = new B<Integer>;
A<Object> a2 = a;  // fine by covariance of A
B<Object> b = (B<Object>) a2;
```

In a language with an expansion-based translation, such as C#, the last line will fail dynamically: an object with dynamic type `B<Integer>` cannot be cast to a `B<Object>`. 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. (In this case, a runtime error could result from a non-cast expression, thus violating type soundness.) This practical consideration affects all type systems that combine 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. The result is somewhat counter-intuitive in that it defies common patterns for safe casting. For instance, the cast could have been performed after an "`a2 instanceof B<Object>`" check to establish that `a2` is indeed of type `B<Object>`. In this case the programmer would think that the cast warning can be ignored, which is not the case. In practice, any deployment

of our type system in an erasure-based setting would have to follow the same policy as Scala regarding cast warnings.

6 Related Work

Definition-site variance was first investigated in the late 80's [7,2,3] when parametric types were incorporated into object-oriented languages. It has recently experienced a resurgence as newer languages such as Scala [14] and C# [9] 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 [8].

Use-site variance was introduced by Thorup and Torgersen [17] in response to the rigidity in class definitions imposed by definition-site variance. The concept was later generalized and formalized by Igarashi and Viroli [10]. The elegance and flexibility of the approach evoked a great deal of enthusiasm, and was quickly introduced into Java [18]. The same flexibility also proved challenging to both researchers and practitioners. The soundness of wildcards in Java has only recently been proven [4], and the implementation of wildcards has been mired in issues [5,15,16].

The work of Viroli and Rimassa [20] attempts to clarify when variance is to be used, introducing concepts of produce/consume, which are an improvement over the write/read view. Our approach offers a generalization and a high-level way to reason soundly about the variance of a type. Other recent work discusses the complex relationship between type-erasure and wildcards [6], as well as the concept of variance at the level of tuning access to a path type in tree-like class definitions [12].

7 Conclusion

This paper presented VarJ, the first formal model for Java with definition-site variance, wildcards, and intricate features such as wildcard capture. VarJ gives a framework for reasoning about the variance of various types (e.g., bounded existential types). We presented theory underlying the assumed variances of the positions that types can occur in (e.g., the upper bound of a method type parameter is contravariant). Thus, our calculus resolves questions that are central in the design of any language involving parametric polymorphism and subtyping.

Acknowledgments. We thank the anonymous ECOOP reviewers for their feedback, Ross Tate for providing examples on the complications of wildcards, Nicholas Cameron for discussions on TameFJ and on performing type inference at runtime, Andrew Kennedy for discussions about the C# formalism with definition-site variance, and Christian Urban for clarifying Barendregt's variable convention. This work was funded by the National Science Foundation under grants CCF-0917774 and CCF-0934631.

References

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

2. America, P., van der Linden, F.: 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 (1990)
3. Bracha, G., Griswold, D.: Strongtalk: typechecking smalltalk in a production environment. In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (1993)
4. Cameron, N., Drossopoulou, S., Ernst, E.: A Model for Java with Wildcards. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 2–26. Springer, Heidelberg (2008)
5. Chin, W.-N., Craciun, F., Khoo, S.-C., Popeea, C.: A flow-based approach for variant parametric types. In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2006)
6. Cimadamore, M., Viroli, M.: Reifying wildcards in Java using the EGO approach. In: SAC 2007: Proceedings of the 2007 ACM Symposium on Applied Computing (2007)
7. Cook, W.: A proposal for making Eiffel type-safe. In: European Conf. on Object-Oriented Programming, ECOOP (1989)
8. Emir, B., Kennedy, A., Russo, C.V., Yu, D.: Variance and Generalized Constraints for C# Generics. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 279–303. Springer, Heidelberg (2006)
9. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
10. Igarashi, A., Viroli, M.: On Variance-Based Subtyping for Parametric Types. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 441–469. Springer, Heidelberg (2002)
11. Igarashi, A., Viroli, M.: Variant parametric types: A flexible subtyping scheme for generics. ACM Trans. Program. Lang. Syst. 28(5), 795–847 (2006)
12. Igarashi, A., Viroli, M.: Variant path types for scalable extensibility. In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2007)
13. Kennedy, A.J., Pierce, B.C.: On decidability of nominal subtyping with variance, 2006. In: FOOL-WOOD 2007 (2007)
14. Odersky, M.: The Scala Language Specification v 2.8 (2010)
15. Smith, D., Cartwright, R.: Java type inference is broken: can we fix it? In: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (2008)
16. Tate, R., Leung, A., Lerner, S.: Taming wildcards in Java's type system. In: Programming Language Design and Implementation, PLDI (2011)
17. Thorup, K.K., Torgersen, M.: Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 186–204. Springer, Heidelberg (1999)
18. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahe, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: SAC 2004: Proc. of the 2004 Symposium on Applied Computing (2004)
19. Urban, C., Berghofer, S., Norrish, M.: Barendregt's Variable Convention in Rule Inductions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 35–50. Springer, Heidelberg (2007)
20. Viroli, M., Rimassa, G.: On access restriction with Java wildcards. Journal of Object Technology 4(10), 117–139 (2005)