Mixed-Site Variance

Ross Tate
Cornell University
ross@cs.cornell.edu

Abstract

Java introduced wildcards years ago. Wildcards were very expressive, and they were integral to updating the existing libraries to make use of generics. Unfortunately, wildcards were also complex and verbose, making them hard and inconvenient for programmers to adopt. Overall, while an impressive feature, wildcards are generally considered to be a failure. As such, many languages adopted a more restricted feature for generics, namely declaration-site variance, because designers believed its simplicity would make it easier for programmers to adopt. Indeed, declaration-site variance has been quite successful. However, it is also completely unhelpful for many designs, including many of those in the Java SDK. So, we have designed *mixed-site* variance, a careful combination of definition-site and use-site variance that avoids the failings of wildcards. We have been working with JetBrains to put this into practice by incorporating it into the design of their upcoming language, Kotlin. Here we exposit our design, our rationale, and our experiences.

1. Introduction

The adoption of (parametric) polymorphism[1] [13] by major industry object-oriented languages has been an interesting progression. C++ was possibly the first widely used object-oriented language to adopt this technology as the feature known as *templates*. Yet C++ templates were lacking in two critical ways. First, a template had no way to express the requirements on the type it was polymorphic over, nor a way to check that the implementation always worked under those assumptions. Thus, the only way to know is to try, making it hard for programmers to plan ahead when devising implementation plans, and making it hard to know who was at fault when a type error arose (especially since the error messages for templates are famously convoluted). Second, templates were not well integrated with a key type feature of object-oriented languages: subtyping. This was not a huge problem for C++ since its types tended to be strongly tied to implementation, but other languages such as Java made heavy use of interfaces and relied on subtyping to make them convenient to use. Thus, when it came time for Java to incorporate polymorphism, Sun had an opportunity to address many of the issues faced by the C++ programmers it was hoping to convert.

And so, Java reintroduced polymorphism to the family of widely adopted object-oriented languages in the form of *generics* [8]. With generics, one could write a `List` interface that could simultaneously work for many types of elements, such as a `List<Integer>` and a `List<String>`, much like with C++ templates. Plus, in order to address the first aforementioned weakness of C++ templates, Java incorporated F-bounded polymorphism [6]. This feature allows programmers to specify which interfaces/classes they can expect/require the type being parameterized to satisfy, and it allows the type checker to ensure that the implementation is indeed safe under only those assumptions. While there still are some limitations occasionally important in practice, overall this feature has been a success.

Unfortunately, Java's solution to C++ templates' poor integration with subtyping is generally considered to be a failure. The feature known as *wildcards* [17] is in practice an encoding of use-site variance [15] using a restrictive form of existential types [4, 5, 9, 16–18]. They are actually quite a clever and powerful feature. However, they are also quite complex and quite verbose. For example regarding complexity, in order to address the challenge of mixing use-site variance with polymorphism, Java introduced a technique known as wildcard capture [17] that introduces a constrained fresh type variable representing the existentially qualified type denoted by a wildcard. Wildcard capture is powerful when type checking succeeds but can lead to some famously convoluted error messages when type checking fails. Wildcard syntax itself, `List<? extends Number>`, reads as "a list of some subtype of numbers", a little too complex to be intuitive. This syntax is also rather verbose, made significantly worse by the fact `? extends` occurs extremely frequently in code with precise usage of use-site variance (especially since Java does not have type inference for local variables). This, in turn, means programmers occasionally have to trade off between readability and precision, since precise types often have multiply nested uses of `? extends` and `? super`. In fact, there is a widespread convention to never use a wildcard in the return type of a method [3].

[1] We will use the term *polymorphism* to refer to parametric rather than subtype polymorphism.

To make matters worse, there are many classes and interfaces where use-site annotations should *always* be present, `Iterator` and `Iterable` being the most significant examples. This observation, along with the widespread ill feelings towards wildcards, lead Microsoft to adopt declaration-site variance when it introduced generics to C# [7]. With declaration-site variance, a programmer can annotate a type parameter for a generic class/interface as being covariant (`out` in C#) or contravariant (`in` in C#). For example, C#'s variant of iterators is declared `IEnumerator<out T>`. This enables one to use an `IEnumerator<String>` as an `IEnumerator<Object>` because `String` is a subtype of `Object`. Another way to think of it is that `IEnumerator` implicitly always has the `? extends` annotation. This makes declaration-site variance much less verbose than wildcards, and programmers tend to find it more intuitive. Scala actually had declaration-site variance from the beginning [11], and so, unlike C#, designed its core libraries around declaration-site variance, which conveniently coincided with Scala's push for immutable data structures in order to embrace concurrency.

Yet, despite its popularity, declaration-site variance is not without limitations. In particular, declaration-site variance is particularly poor for mutable data structures, such as most of the Java collections library. Consider a hypothetical generic class `Array<T>`, which is an array with the usual ability to read from and write to indices. It is unsafe to treat an `Array<Object>` as an `Array<String>`, because then someone reading from the array of strings would expect to get strings but could in fact get arbitrary objects. From the other direction, it is also unsafe to treat an `Array<String>` as an `Array<Object>`, because then someone could add arbitrary objects to the array when it is supposed to contain only strings — note that both Java and C# have this wrong for historical reasons. This means that `Array` is neither covariant nor contravariant; it is an invariant type constructor. However, often one wants to only read from an array or sometimes they only want to write to an array. Interestingly, the read-only aspects of arrays are almost all covariant, and the write-only aspects of arrays are almost all contravariant. Yet, programmers working with declaration-site languages have no direct way to specify they want only the covariant or contravariant portions of arrays and so have to jump through annoying and inefficient hoops to circumvent the limitations of the type system. In fact, Altidor et al. found that 61% of use-site annotations in existing code bases would still be necessary even if Java added declaration-site variance [1]. While techniques employed by Scala and C# library designers may improve applicability of declaration-site variance, there are still many examples that need use-site variance.

More recently, the company JetBrains have been designing a new programming language which they call Kotlin, seeking to create a tidier successor to Java by reflecting on the evolution and lessons of object-oriented languages over the last decade, and whom we have been advising and collaborating with particularly on the topic of type systems. In our reflections, we observed that, although wildcards failed, that does not necessarily mean use-site variance has failed. Rather, use-site variance has the ability to address the weaknesses of declaration-site variance, provided we allow programmers to use declaration-site variance when possible as it is the less verbose option. Thus, we have designed what we call mixed-site variance, which is a carefully crafted combination of declaration-site variance and use-site variance that also avoids the complexities of wildcards that we have thoroughly discussed in recent work [14]. In addition to giving us the best of both worlds, mixed-site variance has an intuitive correspondence with Java wildcards, which aids in making Kotlin able to conveniently interact with existing Java code bases. While mixed-site variance is intuitive for the most part, there are some unexpected challenges and interesting subtleties, which we explain here so that others may understand the reasons behind our design as well as understand how and why they might incorporate the feature into their own language designs.

So far we have briefly described the history of our design space. In Section 2 we will discuss and formalize our form of use-site variance. In Section 3 we will add declaration-site variance to our system to address the weaknesses of use-site variance we just detailed, while illustrating our rationale as to precisely how we chose to mix the two together. In Section 4 we will explain how these types fit into a language as a whole, focusing on those features where mixed-site variance has the most impact. Finally, we will conclude with a discussion of forwards compatibility, highlighting some of the more interesting interactions mixed-site variance has with other common language features so that designers can know what other points to take into consideration.

2. Use-Site Variance

We have yet to properly describe what use-site variance is. With use-site variance, every time you use a generic class/interface you furthermore specify whether you want to access it invariantly, covariantly, contravariantly, or independently. For example, `Array<inv Number>` indicates you want invariant access to an array of numbers, which grants you access to all attributes[2] of the array but also restricts the array instances you can be provided. In particular, I could not provide you with an array of objects because you may try to read numbers from the array, and I could not provide you with an array of integers because you may try to write arbitrary numbers into the array. On the other hand, `Array<out Number>` indicates you want covariant access to an array of numbers, which intuitively grants you access to attributes letting you get numbers `out` of the array but not letting you put numbers `in` to the array. This means I can safely give you an array of integers since I do not have

[2] We use the term *attributes* to refer to both fields and methods.

$$\tau \ ::= \ \bot \mid \top \mid v \mid C\texttt{<in } \tau \texttt{ out } \tau\texttt{>}$$

Figure 1. A simplified grammar for use-site variance

to worry about you adding arbitrary integers into the array. Alternatively, `Array<in Number>` indicates you want contravariant access, intuitively granting you access to attributes letting you put numbers `in` to the array but not letting you take them `out`, so that I can safely give you an array of objects. Lastly, `Array<?>` indicates you want access to the type-parameter-independent portion of the array, which includes attributes that tell you its size or clears it of all its contents.

It is important that, while `out` and `in` intuitively mean read only and write only, those are just intuitions. For example, with an `Array<out Number>` I can still safely clear the array of all its contents (assuming we are in a language with nulls, although interestingly Kotlin is not such a language), and can even safely rearrange the contents of the array. And on the other hand, with an `Array<in Number>` I can still safely get the size of the array. Thus, it is important to understand that they actually represent the type-theoretic concepts of covariance and contravariance.

The fact that `Array<out •>` is covariant means that, if τ is a subtype of τ', then `Array<out` τ`>` is a subtype of `Array<out` τ'`>`. For example, `Array<out String>` is a subtype of `Array<out Object>`. In order to be safe, e.g. prevent people from writing objects into an array of strings, the type system projects the covariant portion of the type signature for arrays, typically the read-only attributes, and hides everything else, typically the writing attributes. From the other side, the fact that `Array<in •>` is contravariant means that, if τ is a subtype of τ', then `Array<in` τ'`>` is a subtype of `Array<in` τ`>` — note the reversal. For example, `Array<in Object>` is a subtype of `Array<in String>`. Again, in order to be safe, e.g. prevent people from reading strings from an array of objects, the type system projects the contravariant portion of the type signature for arrays, typically the write-only attributes, and hides everything else, typically the read-only attributes. There are some variations on how this projection and hiding is done, and we will illustrate our version in Section 4.1. For now, though, we focus on the types themselves.

2.1 Types

Figure 1 shows an extremely simplified syntax. To keep from being distracted by details, we assume a bottom type, $\bot$, which intuitively is the empty set and formally is a subtype of every type, and a top type, $\top$, which intuitively is the set of all values and formally is the supertype of every type. In this way, the annotation `in` $\bot$ intuitively means "you can put nothing in" so that the contravariant portion is useless, and the annotation `out` $\top$ intuitively means "you will get arbitrary values out" so that the covariant portion

$$\frac{}{\Gamma \vdash \bot <: \tau} \qquad \frac{}{\Gamma \vdash \tau <: \top} \qquad \frac{}{\Gamma \vdash v <: v}$$

$$\frac{\tau_i <: v <: \tau_o \in \Gamma \quad \Gamma \vdash \tau <: \tau_i}{\Gamma \vdash \tau <: v} \qquad \frac{\tau_i <: v <: \tau_o \in \Gamma \quad \Gamma \vdash \tau_o <: \tau}{\Gamma \vdash v <: \tau}$$

$$\frac{\begin{array}{c} C\texttt{<P>} \text{ is a subclass of } D\texttt{<}\tau_{\texttt{P}}\texttt{>} \\ v \text{ is fresh} \\ \Gamma, \tau_i <: v <: \tau_o \vdash \tau_i' <: \tau_{\texttt{P}}[\texttt{P} \mapsto v] \\ \Gamma, \tau_i <: v <: \tau_o \vdash \tau_{\texttt{P}}[\texttt{P} \mapsto v] <: \tau_o' \end{array}}{\Gamma \vdash C\texttt{<in } \tau_i \texttt{ out } \tau_o\texttt{>} <: D\texttt{<in } \tau_i' \texttt{ out } \tau_o'\texttt{>}}$$

Figure 2. Algorithmic subtyping rules for use-site variance

is useless. Thus we can encode the more user-friendly use-site annotations described above in terms of a single all-encompassing `in out` annotation:

$$\begin{array}{lcl} C\texttt{<inv } \tau\texttt{>} & \mapsto & C\texttt{<in } \tau \texttt{ out } \tau\texttt{>} \\ C\texttt{<out } \tau\texttt{>} & \mapsto & C\texttt{<in } \bot \texttt{ out } \tau\texttt{>} \\ C\texttt{<in } \tau\texttt{>} & \mapsto & C\texttt{<in } \tau \texttt{ out } \top\texttt{>} \\ C\texttt{<?>} & \mapsto & C\texttt{<in } \bot \texttt{ out } \top\texttt{>} \end{array}$$

In practice, Kotlin users never actually write both an `in` and `out` annotation, and unannotated uses are invariant by default (unless the class/interface specifies otherwise). Nonetheless, even if a user never explicitly writes an `in out` annotation, they can arise intermediately as a result of the approximation algorithm presented in Section 4.1, though we will discuss how this can be avoided through restrictions should a language designer prefer to do so. Conceptually, the type C`<in` τ `out` τ'`>` can be thought of as instances of C whose actual type argument is bounded below by τ and above by τ'. Note that a language does not actually need a $\bot$ or $\top$ type, and of course classes/interfaces can have an arbitrary number of type parameters, but we elide those details as they are not relevant to the intuitions, challenges, or algorithms discussed in this paper.

2.2 Subtyping

An advantage of simplifying the grammar is that we can present the subtyping rules for use-site variance concisely, as shown in Figure 2. A context Γ indicates the lower and upper bound on each type variable v, effectively using $\bot$ to indicate no lower bound and $\top$ to indicate no upper bound. Note that the rules are algorithmic; in particular, there is no rule for transitivity, rather transitivity is a property that can be proven about the system when the context is consistent. Soundness is near trivial given soundness of existing systems as this system can easily be translated to both declarations-site variance with multiple inheritance and to bounded existential types.

The one rule to focus attention on is that for subtyping two classes. First, the clause "C`<P>` is a subclass of D`<`$\tau_{\texttt{P}}$`>`" indicates that, according to the inheritance hierarchy, the generic class C<P> is declared to be a subclass of D<$\tau_{\texttt{P}}$>, where P is the type parameter of C and subclassing is reflexive and transitive — note that P and $\tau_{\texttt{P}}$ are direct type arguments without any use-site annotations. More importantly, let us explain the intuition behind the fresh variable v. The idea is that any instance of C is allocated with some type argument, call it v. For that instance to belong to C<in τ_i out τ_o>, v must be a supertype of τ_i so that it is safe to write τ_is to it, and v must be a subtype of τ_o so that it is safe to read τ_os from it. So, if we can prove the rest of the subtyping via inheritance and use-site variance using just these constraints on v, which is what the remaining clauses do, then we know that instance can safely be treated as a D<in τ_i' out τ_o'>.

Now, there are issues with termination for this algorithm. In general, since declaration-site variance can easily be encoded into use-site variance, Kennedy and Pierce's undecidability result applies here as well [10]. Thankfully, though, our formulation of use-site variance can be encoded as declaration-site variance with multiple inheritance, so their decidability results also apply here. In particular, unlike with wildcards [14], disallowing expansive inheritance [10] ensures termination with one caveat: as you recurse you need to look up the stack to see if the same argument pair has ever been recursed upon before, in which case you accept the subtyping (although C# unnecessarily rejects in this situation). Interestingly, this algorithm can also be encoded into bounded existential types, so we can apply the decidability results from there here as well [14]. In particular, disallowing in annotations in declarations of superclasses/interfaces ensures termination without the above caveat, though one has to optimize inv annotations to use unification rather than both an upper- and lower-bound check, which can only be done if the type system can ensure that if there is an explict in τ_i out τ_o annotation then τ_i is a subtype of τ_o. Regardless of the algorithms and restriction used, with our rules we have one significant advantage over wildcards and existential types: two types are equivalent in our system (meaning subtypes of each other) if and only if they are syntactically identical. This observation leads us to more thoroughly contrast our system with wildcards.

2.3 Wildcards

The use-site variance system we propose here is admittedly very similar to wildcards. Yet the small differences have been carefully tuned to account for our own type-theoretic issues with wildcards [14] and the opinions expressed by various Java developers whom we have discussed the matter with. The first small difference is the syntax. One issue is that the syntax `? extends` does not convey any intuition about how to use the type; it simply conveys the existential formalization behind the type system. Scala's `_<:` suffers the same criticism. While not entirely accurate, out informs the programmer that intuitively this type is something you can only get values out of. This is particularly relevant to beginning programmers; initially they work primarily with collections where this annotation accurately represents the usage pattern, and so they can afford to learn the details only later when they have become more experienced with the language. On the other hand, when first exposed to wildcards, beginning programmers often misread ? to be a special name, like $\top$ and $\bot$, and so think all the ?s represent the same type. Thus, we believe our small syntactic change can help with adoption of the feature and the language.

Now, from a more type-theoretic perspective, there is a much more important difference between wildcards and our use-site variance. Our subtyping rules for use-site variance do *not* use *implicit* constraints [14]. In Java, if we declared a class `Sorted<T extends Comparable<T>>`, which does happen in practice even though `Comparable` should always have a `? super` argument, then `Sorted<?>` is a subtype of `Sorted<? extends Comparable<?>>` and even of `Sorted<? extends Comparable<? extends Comparable<?>>>` and so on forever. This is because, when Java introduces a fresh variable v much like we do in subtyping, it also knows that v must satisfy the constraints imposed upon the type parameter, in this case v <: Comparable<v>, and so Java adds those to the context as well. These implicit constraints are perfectly legitimate and in a way quite clever, but they are also the source of many of the issues exposed by our earlier work [14]. For one, they impact type equivalence, since the above subtypings mean `Sorted<?>` is actually equivalent to `Sorted<? extends Comparable<?>>`. Thus, an `Array<Sorted<? extends Comparable<?>>` should be assignable to `Array<Sorted<?>>` and vice versa even though `Array` is invariant but the two arguments are not syntactically identical. Hopefully this sampling offers a glimpse of all the algorithmic challenges we avoid with our simpler design.

Thus so far our use-site variance design addresses both the syntactic and algorithmic weaknesses of wildcards. But we have not yet addressed what is possibly the worst aspect of wildcards: frequency. For this we, like C# and Scala, look to declaration-site variance. Yet, unlike C# and Scala, we do not abandon use-site variance as an integral component of the type system. Instead, we ensure these two forms of variance collaborate with each other in a manner both intuitive and efficient.

3. Adding Declaration-Site Variance

In use-site variance, every type *argument* is annotated with its intended variance, limiting how that specific type can be used. In declaration-site variance, every type *parameter* is annotated with its intended variance, limiting how that type parameter can be used. The latter has the advantage that there are significantly fewer type parameters than type arguments, making it less burdensome. This is why it has become the widespread standard. However, it also has the disadvantage that some type parameters do not naturally fall within the restricted uses of just covariance or just contravariance. Hence we believe there is room for use-site variance to make an improvement.

First, let us restate how declaration-site variance works. In a signature for a generic class/interface, there are invariant, covariant, and contravariant uses of a type parameter. For example, having the return type of a method be a type parameter is a covariant use, whereas having a parameter type of a method be a type parameter is a contravariant use. Having a mutable field whose type is a type parameter is an invariant use, supposing that field is visible publicly or even just to other instances of the same class. With declaration-site variance, if one annotates a type parameter as `in` or `out` (`-` or `+` in Scala), the type checker ensures that the signature of the class/interface contains respectively only contravariant or covariant uses of that type parameter. In other words, the type checker ensures that, for that class C, the type C`<inv` τ`>` is no more useful than C`<in` τ`>` or C`<out` τ`>` respectively so that there is no point in distinguishing the invariant case from the appropriate variant case. However, below we will offer some slight variations on how a declaration-site variance can be interpreted.

Surprisingly, there are multiple ways to combine use-site and declaration-site variance. First, we explain the solutions we did not take and why. Then, we explain our own solution and its advantages. For sake of discussion, we will use `In`, `Out`, and `Invariant` to represent generics with type parameters that have been declared to be contravariant, covariant, and invariant respectively.

3.1 Defaulting

One approach to combining use-site and declaration-site variance is to make the declared variance simply a default variance that can be overridden by a use-site annotation. This way a programmer can indicate that typically a class will be used, say, covariantly, but occasionally fellow programmers will want more powerful access and use the `inv` annotation to indicate so (or even the `in` annotation if they only want to access the less commonly used write-only attributes). This means that a programmer can declare a type parameter to be covariant even if the class/interface's signature contains invariant or contravariant uses of that parameter. So `Out<P>` could actually have a method `void add(P elem)`, which by default people do not have access to so that they can conveniently use covariance, but which people can explicitly ask for by specifying `Out<in String>` or `Out<inv String>`.

This sounds nice and flexible at first as it lets the library designer, who has first-hand knowledge of what the usage patterns are expected to be, specify what the most convient variance would be for those expected usage patterns. However, some of the consequences of this strategy are unintuitive. For example, an `In<Integer>` would *not* be a subtype of `In<out Number>` even though `Integer` is a subtype of `Number` and it looks like you have *weakened* your use of `In` to be covariant. The reason is that `In<Integer>` is just a shorthand for `In<in Integer>`, so it could actually be an instance of `In<inv Object>`, which is not an `In<out Number>`. We felt such examples behaved contrary to our own expectations, let alone those of beginning programmers, so we decided against this strategy.

3.2 Layering

Interestingly, two separate groups have proposed layering use-site variance on top of declaration-site variance. Scala uses existential types on top of its declaration-site variance in order to simulate use-site variance and be compatible with Java wildcards [11], and Altidor et al. provide a calculus VarJ of existential types over nominal declaration-site types as a means to add declaration-site variance to Java [2]. Unfortunately, layering both leads to unexpected subtypings and prevents capturing of implicit constraints. These have some algorithmic implications, so we suspect neither Scala nor VarJ have decidable subtyping for reasons described below.

To illustrate, consider the type `In<out String>`. With layering, this is a subtype of `In<out Integer>` even though `String` is not a subtype of `Integer` nor even the other way around. The reason is that `In<out String>` is considered to represent the existential type $\exists v <:$ `String.` `In<`v`>`, meaning there exists some type v that is a subtype of `String` such that this is an `In<`v`>` with `In` being contravariant as usual. Now, due to contravariance, $\exists v <:$ `String.` `In<`v`>` is a subtype of $\exists v <:$ `String.` `In<`$\bot$`>`, since $\bot$ is a subtype of v regardless of what v is. At this point, the body of the existential makes no reference to v, so we can just disregard v entirely, indicating that $\exists v <:$ `String.` `In<`$\bot$`>` is a subtype of simply `In<`$\bot$`>`. So, at this point we have that `In<out String>` is a subtype of `In<`$\bot$`>`. Next, clearly $\bot$ is a subtype of `Integer`, so `In<`$\bot$`>` satisfies the existential type $\exists v <:$ `Integer.` `In<`v`>`, which is the long form of `In<out Integer>`. Thus, `In<`$\bot$`>` is a subtype of `In<out Integer>`. Finally, by transitivity using `In<`$\bot$`>` as the middle type, we get that `In<out String>` is a subtype of `In<out Integer>`.

Note that transitivity is key to this deduction, which is why we suspect such a layered subtyping system to be undecidable, considering all decidability results on similar (actually easier) subtyping systems work by first finding a deduction system without transitivity [10, 14]. In particular, the above strategy only works if there is an appropriate intermediate type without use-site annotations, which does not actually always exist due to constraints on type parameters. Although not formally documented, Scala's implementation seems to skirt that issue by permitting the intermediate type to be an invalid type, meaning a type that the type checker would reject had the programmer written it explicitly. This is most likely sound under a set-theoretic model, but these inconsistencies and messy details are part of why we opted against this strategy.

Another issue with layering is that it is not compatible with the concept of wildcard capture. Recall the interface `Sorted<N extends Comparable<N>>`, and suppose we had an instance of `Sorted<? extends Number>`. We know that `?` stands for some type, so wildcard capture names this type with a fresh variable v and knows that v is a subtype of `Number` *and* a subtype of `Comparable<`v`>`. This reasoning is integrated throughout Java's type system, so we would like `Sorted<out Number>` to be compatible with it. Unfortunately, with the layering system, if we furthermore declared `Sorted` to be contravariant, wildcard capture would no longer be sound. The reasoning above can be used to show that `Sorted<String>` is a subtype of `Sorted<out Number>` if `Sorted` is contravariant. Clearly `String` is not a subtype of `Number`, so assuming the aforementioned variable v is a subtype of `Number` would be unsound. Rather, all we can assume is that v is a subtype of `Comparable<`v`>` and a supertype of some unknown type v' that is a subtype of `Number`. Not only is the capture behavior unexpected, but this cannot be expressed with wildcards, so it is not compatible with Java. This issue is even more important with the next variation of combining use-site variance and declaration-site variance.

3.3 Joining

In an earlier work, Altidor et al. [1] proposed joining use-site variances with declaration-site variances in a lattice-theoretic sense. For example, `Out<in String>` is equivalent to `Out<?>`, regardless of what constraints `Out` imposes on its type parameter, because the join of the declaration-site annotation `out` and the use-site annotation `in` is `?` in the annotation lattice. In fact, Scala's actual implementation seems to take this approach. The issue is, despite the name of their publication, Altidor's proposal is unsound for Java wildcards. Again, this is due to wildcard capture with implicit constraints, which Scala does not do and so avoids this issue.

To see the issue, consider the following covariant class:[3]

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

This seems benign, but we can use this aptly named class to exploit the combination of wildcard capture and joining variances to produce an unsafe memory access. For that, we will need the following two classes:

```
class AList extends ArrayList<AList> {}
class BList extends ArrayList<BList> {
    int i = 0;
    public boolean add(BList bs) {
        System.out.println(bs.i);
        return super.add(bs);
    }
}
```

These are designed to be two different implementations satisfying the requirements of `Trouble`'s type parameter, one implementation accessing a field the other does not have.

Next, we need code that takes advantage of wildcard capture:

```
void sneaky(Trouble<? super AList> trouble) {
    trouble.next().add(new AList())
}
```

Due to wildcard capture, the expression `trouble.next()` returns a v where v is a fresh type variable known to be a subtype of `List<`v`>`, due to the implicit constraint generated from `Trouble`'s contraint on its type parameter, and to be a supertype of `AList`, due to the explicit constraint `? super AList`. The implicit constraint tells us there is an `add` method that accepts a v. Then, since the explicit constraint tells us `AList` is a subtype of v, we can supply that `add` method an `AList`, which is why `sneaky` type checks.

Lastly, for sake of concision suppose `troubleMaker` is a generic method that constructs a `Trouble` by simply wrapping an `Iterator` of an appropriate type. Then the following code results in an unsafe memory access if executed:

```
List<BList> bs = new ArrayList<BList>();
bs.add(new BList());
Iterator<BList> itr = bs.iterator();
Trouble<BList> trouble = troubleMaker(itr);
sneaky(trouble);
```

The body of `sneaky(trouble)` adds a new `AList` to the `BList` in `bs`, which accesses a non-existent field `i` because it expects a `BList` rather than a `AList`. Thankfully this does not type check in Java, because `Trouble<BList>` is not a subtype of `Trouble<? super AList>`. However, it is a subtype of `Trouble<?>`, which in Altidor et al.'s system would actually be equivalent to `Trouble<? super AList>` since `Trouble` is covariant, so with the joining strategy we would erroneously accept this invalid code. Thus, we cannot adopt the joining strategy if we want to be compatible with Java or consistent with capture.

3.4 Mixed-Site Variance

Now that we have conveyed the options and our careful consideration of their pros and cons, we finally present our solution. One observation with the above variants is that they all have reasonable intuitions, which means that a (beginning) programmer could look at a type and expect any of the above behaviors without even recognizing the other possibilities,

[3] While Scala and C# would not actually allow `Trouble` to be declared covariant because the constraint on its type parameter is not covariant, that restriction is actually unnecessary as we will discuss in Section 4.3.

leading them to make mistakes. Interestingly, though, these variants only disagree when the declaration-site variance is opposite the use-site variance (i.e. one is `in` while the other is `out`), and furthermore the problems we discussed only arise in this opposing situation. Thus, our version is quite simple: all use-site variances must be weaker than their corresponding declaration-site variance. So one can use arbitrary use-site variances on invariant classes, but only `out` and `?` on covariant classes and `in` and `?` on contravariant classes. Thus if a programmer writes a type whose interpretation is unclear, we reject that type. In practice, we have yet to see a programmer purposely try to have a use-site variance opposing the declaration-site variance, so we view this restriction more as a formalization of realistic usage behavior than as a limitation of expressiveness. Furthermore, this restriction keeps our type system compatible with wildcards and capture via the obvious translation, so it meets Kotlin's goal of simple and familiar correspondence with Java.

Now that we have a clean way of mixing use-site and declaration-site variance, we can take advantage of use-site variance to add flexibility that would normally need to be disallowed by declaration-site variance due to inexpressiveness. In particular, with just declaration-site variance one must ensure that a co/contravariant class/interface has an entirely co/contravariant signature. However, with mixed-site variance we can offer some convenience. In particular, the signature can be whatever the programmer desires, but only the co/contravariant portion of the signature can be accessed. This would be completely useless, except we make one exception: `this` can access the entire signature of itself (and itself only). The particular advantage of doing this is that it allows a co/contravariant class with type parameter `P` to have mutable fields of type containing `P`.

The following shows how this might be used in practice:

```
class Lazy<out V>(Function<Null,V> comp) {
    private mutable V|Null value;
    public V get() {
        if (value != null) {
            return value;
        } else {
            V val = comp.invoke(null);
            value = val;
            return val;
        }
    }
}
```

Here `comp` represents an expensive computation to be evaluated only if its result is ever actually needed. However, since it is expensive, we want to evaluate it at most once. Thus, `Lazy` needs a mutable `V|Null` field to remember what that value is after it has been computed. This signature is clearly covariant, yet a typical declaration-site language like C# would have to reject the `out` annotation on `V` due to the mutable field of type `V|Null`. The reason is that, if there were a method that accessed the fields of *another* instance of `Lazy<V>`, due to variance the run-time type of that other instance could use some strict subtype of `V` so that writing just a `V` to that instance's `value` field would be unsound. To avoid this issue, whenever the implementation of a co/contravariant class accesses *other* instances of that class, we restrict those accesses to only the co/contravariant portion of the signature for those instances. The existing standard restrictions would instead force the programmer to separate out the interface and then have their class implement that interface.

Of course, a language designer might choose to opt out of this nuanced feature, mainly since beginning programmers might be confused by not having access to operations that *appear* to be there, and even advanced programmers would appreciate the type checker informing them when their designed signature does not have the variance they expect it to have. Right now Kotlin accommodates this issue by allowing only private attributes to violate a declaration-site variance, so that library designers can ensure that users are getting the public interface they expect. Scala accomodates this issue by having a `private[this]` annotation that prevents other instances from accessing such attributes at all, even blocking the appropriate co/contravariant aspects of those attributes. Nonetheless, we felt it prudent to mention the option, especially since our mixed-site variance has been designed so that the variance-restriction process can be done nicely from both programmer and type-theoretic perspectives, which brings us to discussing how mixed-site variance interacts with the type system as a whole.

4. Type Checking

Now that we have the basis of how we have chosen to mix use-site and declaration-site variance, there are a few key aspects of a type system most impacted by this mix. First we will discuss how to type attribute accesses given restrictions imposed by use-site annotations. Then we will discuss how to propogate use-site annotations through polymorphic functions. Lastly we will discuss how to treat constraints on type parameters with variance.

4.1 Approximation

We mentioned that a use-site annotation like `out Number` restricts what access the user has to various attributes. But we have not described precisely what that restriction is. One solution is to simply hide all attributes whose type is not already covariant. In fact, this is the solution a language designer needs to take if they want to avoid having unnamed type variables or `in out` annotations arise intermediately during type checking (e.g. in order to express the type returned by a method call). Yet such a solution is rather harsh. Consider something like the `subList` method of Java's `List<E>` that returns another `List<E>`. The intent is to provide a (mutable) view of the larger list. So,

$$\frac{}{\Gamma \vdash_v \bot \mapsfrom \bot \mapsto \bot} \qquad \frac{}{\Gamma \vdash_v \top \mapsfrom \top \mapsto \top} \qquad \frac{\tau_i <: v <: \tau_o \in \Gamma}{\Gamma \vdash_v \tau_i \mapsfrom v \mapsto \tau_o}$$

$$\frac{\Gamma \vdash_v \tau_i^{\downarrow} \mapsfrom \tau_i \mapsto \tau_i^{\uparrow} \qquad \Gamma \vdash_v \tau_o^{\downarrow} \mapsfrom \tau_o \mapsto \tau_o^{\uparrow}}{\Gamma \vdash_v C\texttt{<in } \tau_i^{\uparrow} \texttt{ out } \tau_o^{\downarrow}\texttt{>} \mapsfrom C\texttt{<in } \tau_i \texttt{ out } \tau_o\texttt{>} \mapsto C\texttt{<in } \tau_i^{\downarrow} \texttt{ out } \tau_o^{\uparrow}\texttt{>}}$$

Figure 3. Algorithm for tightly under/overapproximating a type to remove a non-recursively constrained type variable v

if you had a (read-only) `List<out Number>`, you would reasonably expect `subList` to return a (read-only) view of type `List<out Number>`. Yet the aforementioned solution would simply block all access to the `subList` method. So we look for more powerful solutions.

Java addresses this by using wildcard capture. Java will introduce a fresh type variable with both the implicit and explicit constraints and then type the attribute using that type variable as the type argument. While quite clever and powerful, if a type check fails then the capture approach leads to type errors containing widely despised `capture#`n `of ? extends Number` types, which are especially confusing when implicit constraints get thrown into the mix. From the other direction, the capture approach is so powerful that it will occasionally accept code that programmers often expect to be rejected, such as our earlier `sneaky` function; more on that later.

Now, in formalizing our subtyping algorithm, we also used fresh type variables with constraints, much like Java's approach. However, we only did this for simplicity and efficiency, not out of necessity. In particular, recall that, unlike Java, we opted *not* to use implicit constraints. We did this not only because our change more accurately reflects practical use of wildcards and reduces algorithmic complications, but also because it ensures that constraints on fresh variables never reference fresh variables but rather are always types directly expressible and readable by the programmer. Because of that, it is actually possible to *optimally* approximate any such type both above and below by completely programmer-expressible types.

The algorithm for this process is presented in Figure 3. The guarantee is that, provided the types bounding variables in Γ do not contain v, $\Gamma \vdash_v \tau^{\downarrow} \mapsfrom \tau \mapsto \tau^{\uparrow}$ holds if and only if every type not containing v that is a sub/supertype of τ is also respectively a subtype of $\tau^{\downarrow}$ or supertype of $\tau^{\uparrow}$. That is, $\tau^{\downarrow}$ is the greatest subtype of τ not containing v, and $\tau^{\uparrow}$ is the least supertype of τ not containing v. Thus, the subtyping rule for inheritance in Figure 2 could equivalently be defined by simultaneously substituting and overapproximating rather than introducing a constrained fresh type variable.

This approximation technique can easily be extended to signatures rather than just types. For example, with a method we underapproximate the parameter types and overapproximate the return types. So if `List<E>` had a method with signature `List<E> sort(Comparator<E>)`, where `Comparator` has been declared to be contravariant, then a `List<in Integer out Number>` would be seen as having a `List<in Integer out Number> sort(Comparator<Number>)` method — no matter what the actual run-time type argument of the list is, given the `in Integer out Number` we know a `Comparator<Number>` can always be safely supplied to `sort` and a `List<in Integer out Number>` can be safely expected from `sort`. Similarly, with a field we underapproximate its writeable type and overapproximate its readable type. So if a generic class `Foo<T>` had a field `bars` of type `List<Out<T>>`, then a read from the `bars` field of a `Foo<out Number>` would have type `List<in Out<`$\bot$`> out Out<Number>>`. We present this complicated example to illustrate that an `in out` annotation can arise from types without `in out` annotations. Indeed both sides to the annotation are useful in this case, since the `in Iterable<`$\bot$`>` annotation lets us add the immutable empty iterable to the list, and the `out Iterable<Number>` annotation lets us get iterables of numbers from the list.

Consistency While we are on the topic of approximation, we should consider the type `List<in `$\top$` out `$\bot$`>`, which essentially is a list I can both put anything I want into and take anything I want out of. One might think this impossible: how can I choose to put a `String` in and then choose to get an `Integer` out? The answer is: throw an exception either whenever anyone puts anything into the list or whenever anyone gets anything from the list. Java's `java.util.Collections.EMPTY_LIST` does the former. These observations exemplify the two practical perspectives on the type `List<in `$\top$` out `$\bot$`>`.

Let us consider the latter perspective: we can use the type `List<in `$\top$` out `$\bot$`>` to express the concept $\forall\alpha$.`List<`α`>`, meaning a list of whatever type we want, since after all `List<in `$\top$` out `$\bot$`>` is a subtype of `List<`α`>` for any α. The shorthand for this type could be `List<*>` as opposed to `List<?>`. In Java, raw types (e.g. the type `List` used for `EMPTY_LIST`) intended primarily for backwards compatibility are often also used for this special case of a universal instance since Java's generic type system is incapable of expressing it. Thus, one could design a language extension enabling programmers to write universal instances of generics, using `List<in `$\top$` out `$\bot$`>` to plug them into the type system. One would have to take care in designing a type system to ensure these instances are actually universal. Also, this feature would be incompatible with our existential model, so soundness may be an issue especially if one uses the capture feature in Section 4.2. These concerns are out of the scope of this paper though; we just mention the opportunity in order to offer a complete picture of the options at hand.

With that aside, suppose on the other hand we want to exploit our existential model to know that such types are uninhabitable (without unsound features such as raw types). After all, an instance of `List<in` $\top$ `out` $\bot$`>` would mean there exists some v that is a supertype of $\top$ and subtype of $\bot$, which is clearly impossible since that would imply $\top$ is a subtype of $\bot$ and so everything would be nothing. In fact, having types C`<in` τ_i `out` τ_o`>` where τ_o is a strict subtype of τ_i is problematic algorithmically as well, since then C`<in` τ_i `out` τ_o`>` can be a subtype of C`<inv` τ`>` without all of τ_i, τ_o, and τ being equivalent, a property that prevents the use of unification when deciding subtyping, which our prior work heavily relies on for decidability [14]. Furthermore, these types are simply unintuitive from a user perspective. So, for these reasons, we offer a variation on the type system.

We consider a type C`<in` τ_i `out` τ_o`>` to be *consistent* only if τ_i is a subtype of τ_o and both are consistent, with $\bot$, $\top$, and v being consistent unconditionally. The challenge is that various operations using such types may introduce types that violate this restriction. For example, if `Foo<T>` had a field `bars` of type `List<List<T>>`, then a read from the `bars` field of a `Foo<?>` has the inconsistent type `List<in List<in` $\top$ `out` $\bot$`> out List<?>>`. In particular, the type contains `List<in` $\top$ `out` $\bot$`>` even though all the original types involved were consistent. Note, though, that this type occurs at a contravariant location. This means that replacing it with $\bot$ would result in the less precise type `List<out List<?>>`. Interestingly, though, this supertype is in fact the most precise *consistent* supertype.

This observation leads us to our fortunate result on this matter. Suppose one were to change the recursive case of the approximation algorithm in Figure 3 so that it checks whether $\tau_i^{\uparrow}$ is a subtype of $\tau_o^{\downarrow}$ and instead underapproximates with $\bot$ if that check fails. Then, supposing τ were valid under our new restrictions, both approximations would be valid *and* they would be the tightest consistent approximations. Thus, since all other typing operations could be adapted to use consistent approximations instead, one could have a type system where all types are consistent and all operations cleanly preserve consistency. This is what the designers of Kotlin have opted to do.

4.2 Capture

The tight approximation feature of our design is key to its usability. For example, it improves the error messages of the language since every type used internally can also be expressed by the programmer; no more `capture#`n `of ?` types. In fact, one can use the more efficient fresh-variable technique intermediately and only approximate when reporting a type error or otherwise leaving the scope of that variable. Also, since we do not use fresh variables formally, any well typed expression can be broken into its parts each on a separate line. That is not true for Java. For example, there is no way to separate any subexpression of `sneaky` onto another line and still get `sneaky` to type check. Scala has the same problem with the same example, though one has to write an explicitly recursively bound existential type. Unfortunately, there is no way to extend our approximation techniques to Java (or Scala), since the recursive nature of implicit constraints from wildcards prevent tight approximations. But since we have designed our subtyping system so that we can have these approximations, we want to propagate the technology through other features of type systems.

We have used approximation to address how use-site annotations restrict access to attributes. Now we will use approximation to address how use-site annotations are propagated through generic methods. For example, suppose we had `List<T> reverseView<T>(List<T>)`. Should we call `reverseView` with a `List<out Number>`, we would expect the returned value to be a `List<out Number>`. However, we cannot naively substitute `T` with `out Number`. If we passed `List<List<T>> singletonsView(List<T>)` a `List<out Number>`, it would be unsound to have the return type be `List<List<out Number>>`. This would allow us to add a `List<Integer>` to the view, consequently adding `Integer`s to the original list, even if the original list were actually a `List<Double>`. In fact, the return type should be `List<in List<in Number out` $\bot$`> out List<out Number>>`, or `List<out List<out Number>>` if we use only consistent types.

Our basic strategy is simple: reuse the capture concept we stole from wildcards but without implicit constraints, just like we did with subtyping. So if an argument type has a top-level use-site annotation, such as the argument type `List<out Number>` but not the argument type `List<List<out Number>>`, replace the annotated type argument with an appropriately constrained fresh type variable, producing say `List<`v`>` with v constrainted to be a subtype of `Number`. Now none of the invocation's arguments' types have use-site variance, so proceed with generic-method invocation as normal. This will result in a return type containing fresh type variables, so remove those by tightly overapproximating the return type. This basic strategy has one major issue though.

Often a generic method will place constraints on its type parameters. For example, it may require a type parameter to be comparable to itself. When such a method is invoked, the type system needs to check that the type arguments satisfy those constraints. With capture, though, the type arguments may contain capture variables. This means failures on constraint checks could result in the exact kind of unfriendly error messages we have tried so hard to avoid. The alternative, then, is to use approximations before doing constraint checks. Unfortunately, this is lossy since both sides of a recursive constraint would reference capture variables, plus the mismatch between the approximations and the original constraints still make error messages confusing.

We found this behavior unsatisfactory, especially compared to the cleanliness of the rest of the type system. Thankfully, by considering the details of the application, we found a solution remarkably consistent with the rest of the type system. The key observation we made is that the only practical applications of capture occur when a type parameter for the generic method occurs as a top-level invariant argument to one of the method's parameters' types, such as `T` in the parameter type `List<T>` for both `reverseView` and `singletonsView`. The first reason is that a type parameter *must* occur as a top-level (not necessarily invariant) argument to one of the method's parameters' types in order for its corresponding type argument to contain a capture variable, and the second reason is that, except in contrived cases, when it was not an invariant argument then an approximation could be used instead. Thus, capture is only useful in practice when we essentially have a generic method with a context object of sorts as a parameter. So, we realized we should just type this much like we would a method of a generic class.

With that, we modified our basic strategy to emulate method invocation with additional context objects. For each argument to the generic method whose corresponding parameter type is context-object-like as described above, do the following. First, apply inheritance to the type of the argument so that it has the same class/interface name as the corresponding parameter type, rejecting if this is not possible. Second, introduce appropriately constrained capture variables for the use-site annotations except `in out` annotations where the `in` type is equivalent to the `out` type. After doing this for each argument type, determine the type arguments using standard techniques. Note that, since those type parameters whose type arguments can benefit from capture occur as a top-level invariant argument to some parameter type, they are uniquely determined and so always inferable. Other type arguments may need to be specified explicitly by the programmer though. If a constraint fails to be satisfiable, report the error with approximations; due to applying inheritance before introducing capture variables, type arguments can only contain a capture variable if they are exactly a capture variable, and consequently all constraints can be shown to hold if and only if they hold for the approximations. There may still be an apparent mismatch between the approximations and the original description of the constraint, but unfortunately this seems unavoidable by any solution to this challenge of propagating use-site variances through generic methods. Lastly, use the type arguments to construct the return type and then overapproximate to remove the capture variables.

That is complicated, so let us demonstrate with an example. Suppose `Jagged<E>` inherits `List<List<E>>` and that `jagged` has type `Jagged<in Integer out Number>`. We want to determine the type of `reverseView(jagged)`. `reverseView` has a context-object-like parameter because its type is `List<T>` where `List` is invariant and `T` is a type parameter for `reverseView`. Thus, to handle `jagged`'s type's use-site annotation, we will first use inheritance (with approximation) to phrase `jagged`'s type as a `List`. For simplicity, we will use only consistent types. Thus we get the type `List<out List<in Integer out Number>>`. Next we introduce a fresh type variable v, constrain it to be a subtype of `List<in Integer out Number>`, and then treat the type of `jagged` as `List<`v`>`. We then proceed with type-argument inference as usual, which in this case results in `T` being matched with v. We then check constraints on type parameters, which in this case is trivial since there are no constraints. Finally, we substitute type parameters with corresponding type arguments to produce the return type `List<`v`>`. Lastly we overapproximate this to remove the capture variable v, finally resulting in the type `List<out List<in Integer out Number>>` for `reverseView(jagged)`.

This may seem like a lot of effort for one feature, but we view it to be a very important feature. In particular, we need this feature to make generic methods written by others about a generic class/interface to be just as expressive as a method written as an attribute of that generic class/interface. This way a user of a `List` library can write a method like `reverseView` and use it just as conveniently (albeit with slightly different syntax) as if it were part of the `List` interface itself.

Now there is one last alternative that is less convenient but more powerful. In particular, we could enable the programmer to name capture variables. For example, suppose `nums` were a `List<out Number>`. Then we could add a syntax `List<?N> top = nums`, where the top-level argument `?N` indicates that the new type variable `N` should be assigned the capture variable for `List<out Number>`, inheriting the upper bound of `Number`. This way the programmer could freely take elements from `top` and put them back into `top`. Furthermore, the programmer could pass `top` to generic methods like `reverseView` even in a type system without capture. Nonetheless, the Kotlin team felt this would be too unfamiliar to their target audience of Java programmers, and so has opted for the more convenient capture system instead.

4.3 Constraints

Lastly, we touch upon how constraints on type parameters of a generic class/interface affect validity and variance of types. The answer is simple: they have *no* effect on either the validity of type arguments or the variance of the types. Suppose `Ordered<E extends Comparable<in E>>` is simply an iterable with the additional informal guarantee that the iteration will be in increasing order. Contrary to what C# and Scala reject, `Ordered` can perfectly safely be declared covariant, and, regardless of variance, `Ordered<Object>` is a valid type despite the fact that `Object` does not extend

	Java	C#	Scala	Defaulting	Layering [2]	Joining [1]	Mixed	Mixed and Consistent
Can express use-site variance	✓	✗	✓	✓	✓	✓	✓	✓
Has declaration-site variance	✗	✓	✓	✓	✓	✓	✓	✓
C`<`τ`>` $<:$ C`<out` τ`>`	✓	-	✓	✗	✓	✓	✓	✓
C`<out` τ`>` $<:$ C`<out` τ'`>` implies $\tau <: \tau'$	✗	-	✗	✓	✗	✗	✓	✓
`Invariant<`τ`>` has no strict subtypes disregarding inheritance	✓	✓	✗	✗	✗	✗	✗	✓
Equivalent types are essentially syntactically identical	✗	✓	✗	✓	✗	✓	✓	✓
Any C`<out` τ`>` is a C`<`v`>` for some v subtype of τ	✓	-	✓	✓	✓	✗	✓	✓
Any C`<out` τ`>` was allocated as C`<`v`>` for some v subtype of τ	✓	-	✗	✓	✗	✗	✓	✓
C`<in` τ `out` τ'`>` is expressible	✗	✗	✓	✓	✓	✗	✓	✓
C`<in` τ `out` τ`>` is valid implies C`<`τ`>` is valid	-	-	✗	✓	✓	-	✓	✓
`Sorted<?>` $<:$ `Sorted<out Comparable<?>>`	✓	-	✗	✗	✗	✗	✗	✗

Figure 4. Comparison of features and properties of various nominal type systems with variance

`Comparable<in Object>`. The only time constraints need to be checked for type arguments is when allocating an instance (and consequently when one class/interface inherits another). These relaxations are consistent with the existential type model and therefore sound, and there are numerous reasons to make these relaxations.

First, by not incorporating type-parameter constraints into variance checking we improve programmers' ability to take advantage of declaration-site variance, such as with `Ordered`.

Second, by not always checking type arguments we make type checking more efficient.

Third, by not always checking type arguments we need not worry about how type manipulations affect validity. As a nonexample, in Scala `Ordered[Any]` is considered invalid but the type `Ordered[_>:Any<:Any]` is considered valid even though Scala considers the two types to be the same and even seems to internally transform the latter valid type into the former supposedly invalid type.

Fourth, and possibly most important from personal experience, not always checking type arguments means that adding constraints to a generic class/interface does not force one to propagate those constraints through every existing generic method that makes polymorphic use of that class/interface, even those that do not allocate any instances of the class, which is most often the case. This is especially frustrating when the generic methods that needlessly need to be modified are in a code base the programmer has no control over.

So, to summarize, relaxing the incorporation of constraints into a type system results in a more convenient, efficient, consistent, and changeable type system.

5. Conclusion

We have presented mixed-site variance, a type system that combines the best of both use-site variance and declaration-site variance while avoiding the subtle complexities of existing versions of both and existing attempts to combine the two. This system was driven by industrial experience and controlled by academic experience, resulting in a design amenable to both perspectives. Figure 4 illustrates our contribution by showing how existing systems hold up to a few expectations from users, implementers, and formalizers, with mixed-site variance consistently satisfying all these expectations except the one we purposely sacrificed due to its irrelevance in practice. Our core concept is that use-site variances should only weaken declaration-site variances, since otherwise ambiguities and complications arise. Our primary lesson is that avoiding implicit constraints allows types to stay simple throughout type checking. Our major complexity is that maintaining simple types requires careful treatment of use-site capture. While nothing in this type system is ground breaking, the appeal lies in its simplicity despite the surrounding complexities. Thus it has already been integrated into the developing industrial programming language, Kotlin, whose designers and users have provided the feedback resulting in the choices made in this paper.

There is still much to be done for generics. For example, type inference is in increasing demand. Scala has taken a very aggressive approach towards inference, but this has also made type inference and checking rather unpredictable, which has been problematic for industry adoption. Thus, there is incentive to investigate type systems with decidable intraprocedural inference (interprocedural inference is not desired much in practice due to software-engineering concerns). This is quite challenging though, since even just type-argument inference for generic-method invocation has proven difficult [12, 14]. One step in that direction is to add intersection (`&`) and union (`|`) types. In single-instantiation-inheritance [10] models, these additions have some interesting interactions with mixed-site variance. For example, `List<String>|List<Integer>` can be made equivalent to `List<in String&Integer out String|Integer>`. Note that this is a new way for inconsistent types to be introduced through manipulations of consistent types. Nonetheless, preliminary investigations suggest that mixed-site variance in a slightly more expressive variation of single-instantiation inheritance can greatly simplify type checking in the presence of intersection and union types. Although Kotlin does not plan to incorporate such features now, as they are inexpressible in Java, these features may eventually form the basis for a rich object-oriented language with strong decidability and inference properties.

Acknowledgements We thank Andrey Breslav and the rest of the Kotlin team at JetBrains for their collaborative efforts. Their openness to suggestion and inquiry made for an excellent opportunity to explore, and their industry insights and experience provided invaluable guidance.

References

[1] John Altidor, Shan Shan Huang, and Yannis Smaragdakis. Taming the wildcards: Combining definition- and use-site variance. In *PLDI*, 2011.

[2] John Altidor, Christoph Reichenbach, and Yannis Smaragdakis. Java wildcards meet definition-site variance. In *ECOOP*, 2012.

[3] Joshua Bloch. *Effective Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2008.

[4] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *FTfJP*, 2009.

[5] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *ECOOP*, 2008.

[6] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, 1989.

[7] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *ECOOP*, 2006.

[8] James Gosling, Bill Joy, Guy Steel, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley Professional, third edition, June 2005.

[9] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ECOOP*, 2002.

[10] Andrew Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

[11] Martin Odersky. The Scala language specification version 2.9, May 2010.

[12] Daniel Smith and Robert Cartwright. Java type inference is broken: Can we fix it? In *OOPSLA*, 2008.

[13] Christopher Strachey. Fundamental concepts in programming languages. Lecture notes for the International Summer School in Computer Programming, August 1967.

[14] Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in Java's type system. In *PLDI '11: Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*, New York, NY, USA, 2011. ACM.

[15] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP*, 1999.

[16] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *FOOL*, 2005.

[17] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC*, 2004.

[18] Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *APLAS*, 2009.