Investigating the Use of Variance in Object-Oriented Languages

Raoul-Gabriel Urma and Alan Mycroft

Computer Laboratory, University of Cambridge
{raoul.urma,alan.mycroft}@cl.cam.ac.uk

Abstract. Variance constructs were introduced to increase the flexibility of object-oriented programming languages supporting generics. There are two approaches to specify variance: declaration-site variance, which is simple but restrictive, and use-site variance, which is more flexible but more complex. However, it remains unclear how programmers use the additional flexibility provided by variance, and whether they use it at all.
This paper studies how programmers use variance in real programs. We explore the use of use-site variance and declaration-site variance through wildcards in Java and variance annotations in C# and Scala. Our investigation provides empirical answers to the following research questions: how do programmers use variant generics? Which are the most frequently used subtype relations with variant generics? Based on these results we evaluate the usability of use-site and declaration-site variance and whether they can coexist in programming languages.

1 Introduction

Subtype polymorphism is a central feature of object-oriented programming languages. Concretely, a variable of a type `A` may be assigned a value of any subtype of `A`. However, other coexisting features in programming languages can complicate this simple principle. For example, in Java and C#, array subtyping is *covariant*, meaning that type `B[]` is considered a subtype of `A[]` whenever `B` is a subtype of `A`. However, this relation is unsafe [18]. Consider the following Java code:

```
Banana[] bananas = new Banana[5];
Fruit[] fruit = bananas;
fruits[0] = new Apple(); // Runtime exception
peelBanana(bananas[0]); // Apple???
```

The above example is accepted by the Java compiler (assuming that `Apple` and `Banana` are both subtypes of `Fruit`) but causes a runtime exception when executed. The problem occurs when we take an array of `Banana` and assign it to an array of `Fruit`. Since a `Banana` is a subtype of `Fruit`, it seems intuitive to expect that `Banana[]` is also a subtype of `Fruit[]`. Unfortunately, on line 3

when we now try to put an `Apple` into `Fruit[]` we get a runtime exception. Although statically, the array has type `Fruit[]`, its runtime type is `Banana[]` and thus we cannot use it to store an `Apple`. For this reason, some languages like Scala restrict array subtyping [6].

The introduction of parametric polymorphism (*generics* or *parametric types*) in object-oriented languages adds further difficulties: what should the relation of `T<A>` and `T<B>` be for some data type `T`? Variance principles address this question. The relation can be *invariant*, meaning that `T<A>` and `T<B>` are never subtypes. It can be *covariant* meaning that `T<B>` is a subtype of `T<A>` if `B` is a subtype of `A`. It can also be *contravariant* meaning that `T<A>` is a subtype of `T<B>` if `B` is a subtype of `A`. Finally, it can also be *bivariant* meaning that `T<A>` is a subtype of `T<B>` for any type A and B. In order to guarantee no runtime exceptions, most language designers made generics invariant.

However, this can be restrictive and many programming languages provide programmers with constructs to specify variance in a statically checkable way [13]. The classic approach is called *declaration-site variance*, which allows programmers to specify variance on the definition of a generic class [14]. Using this specification, the type system can then statically verify whether the type parameters in the generics are used safely. Typically, the type parameter can be covariant only if it is read-only and contravariant only if write-only, otherwise it can lead to possible runtime exceptions as shown in the Java array example. Even though this approach seems quite straightforward, it forces programmers to design carefully their classes into their covariant and contravariant usage, complicating their usability. This approach is supported by many programming languages like C# and Scala [1,6]. However, Scala designers tackle this problem by recommending additional type bounded parameters on methods conflicting with the variance annotations [16].

The alternative approach, is called *use-site variance* [22], which allows programmers to specify variance on the type usages. This approach is elegant and does not enforce programmers to split their abstractions into different parts. This mechanism is supported by Java and is known as *wildcards* [21, 22]. However, in practice wildcards have intricate inference rules and can have complex type signatures [19]. Because of this, there are guidelines on how to use wildcards for maximal flexibility [10,15]. However, it is unclear whether the flexibility provided is actually worth the additional conceptual overhead for programmers.

Recently, some programming languages such as Kotlin have suggested a combination of the two techniques by providing declaration-site variance for simple abstractions and *type projections* as a simple use-site variance mechanism [3].

Some other programming languages such as Dart do not give programmers the ability to specify variance and simply allow covariance for generic types in order to avoid cluttering the type system with additional complexity, despite the fact that this requires runtime checks as explained in the previous example [2].

In this paper, we describe the main ideas of our research on investigating how programmers react to the ability to specify variance for themselves. We aim

to provide answers to the following research questions and to inform the design of future programming languages:

- How do programmers use variant generics?
- Which are the most frequently used subtype relations with variant generics?
- When is use-site or declaration-site variance more appropriate and can they coexist in programming languages?

The remainder of this paper is organized as follows: section 2 provides an overview of variance mechanisms in object-oriented languages. Section 3 describes our research goals and methodology. Section 4 describes related work, and section 5, finally, provides our conclusions.

2 Background

In this section, we give a brief overview of declaration-site and use-site variance.

2.1 Declaration-site variance

Declaration-site variance allows programmers to specify variance on the definition of a generic class. For example, Scala programmers can use the `+` annotation for covariant type parameters and `-` for contravariant type parameters. In the example below, `+` indicates that `CoList[Apple]` is to be considered a subtype of `CoList[Fruit]`.

```
class CoList[+T](h: T, t: CoList[T]) {
    def head: T = h
    def tail: CoList[T] = t
    def getHead() : T = h;
```

The typesystem will ensure that the type parameter is used only in safe positions. Typically, covariant type parameters need to be read-only and contravariant parameters write-only. This is why the code below will not type check as `T` is now also used in a contravariant position:

```
class CoList[+T](h: T, t: CoList[T]) {
    def head: T = h;
    def tail: CoList[T] = t;
    def getHead() : T = h;
    // contravariant position
    def prepend(elem: T): CoList[T] = new CoList(elem, this);
}
```

This is somewhat restrictive as the only way to achieve full genericity for library architects is to split abstractions into their covariant and contravariant usages. However, Scala's designers recommend using polymorphic methods with a *type bound* to remedy this problem. In fact, the previous example can be generalised by making the `prepend` method polymorphic and placing a *lower bound* on its type parameter. The code below shows how this is possible:

```
class CoList[+T](h: T, t: CoList[T]) {
    def head: T = h
    def tail: CoList[T] = t
    def getHead() : T = h;
    def prepend[U >: T](elem: U): CoList[U] = new CoList(elem, this);
}
```

The introduction of the type parameter `U` is defined as a supertype of `T` using the notations `>:`. As a result, it is now possible to call the `prepend` method on a `CoList[Apple]` with an argument `Banana`. The result will be a `CoList[Fruit]` and the covariant property of `CoList` is also preserved. This works because `T` now only appears in covariant positions.

2.2 Use-site variance

Use-site variance allows programmers to choose variance when a class is instantiated rather than when it is defined. This mechanism is known as *wildcards* in Java. Safety is achieved by restricting accesses to fields and methods, rather than restricting their declarations.

For example, one can use the wildcard `List<? extends S>` to represent the set of types `List<T>` and `List<? extends T>` where `T` is a subtype of `S`. However, access to methods and fields of `List` which use the type parameter in a contravariant position are now restricted. In essence, this mechanism can be used to create a covariance relation between two generic instances.

Java also supports contravariance through the wildcard `? super S`. in addition, it supports a form of bivariance through the *unbounded wildcard* `<?>`. In practice, this means that `List<?>` is a supertype of `List<T>` for any `T`. Since the actual type parameter is unknown, it also means that full access is provided only to methods and fields in which the type parameter does not appear.

Existential Types Wildcards can be seen as a restricted form of existential types [12, 13, 22]. For example, `List<?>` can be represented as the type `T = ∃ X: List<X>`. As a result, `List<String>` and `List<Integer>` are both subtypes of `T` by instantiating X to `String` and `Integer` respectively.

Polymorphic Methods In the context of methods, wildcards can often be replaced by additional type parameters. For example, the following method:

```
public static void aMethod(List<? extends Number> list) {...}
```

can be written using a bounded type parameter:

```
public static <T extends Number> void aMethod(List<T> list) {...}
```

However, Java does not support *lower bound* on type parameters (e.g `<T super Number>`), whereas wildcards support it. This is one of the reason why wildcards are preferred in terms of expressiveness [7].

3 Research Method

In this section we give an overview of the research questions we aim to answer as well as our methodology. In addition, we briefly describe early results.

3.1 Research Questions

Our investigation starts by studying how programmers make use of variance constructs. We aim to provide empirical answers to the following questions: are variance constructs mostly used with built-in libraries or also with user-defined data types? Is there a general pattern on how variant constructs are used? Furthermore, we investigate claims about the additional flexibility provided by use-site variance over declaration-site variance in practice. In addition, we examine whether programmers follow the recommended guidelines of use-site variance constructs.

Next, we look at which are the most frequently used subtype relations with variant generics. Under what situations do programmers make use of covariance, contravariance and bivariance?

Finally building on these results, we evaluate when use-site or declaration-site variance is beneficial for programmers and whether they can be combined.

3.2 Methodology

Our study proceeds as follows: first, we build a corpus of relevant software in Java, C# and Scala. Second, we perform an automated static analysis on each corpus to report uses of variance constructs. Next, we build a sample data by storing the results from the analysis phase in a database. Finally, we report the results by querying the database and manually inspecting relevant use cases.

Corpora We use three corpora for our study:

- A Java corpus based on a curated version of the Qualitas Corpus [20]. We selected only a subset for our study, excluding programs not making use of wildcards. In addition, we excluded some of the programs that needed to be fixed in order to compile. A number of programs contained missing dependencies or referenced old repositories. We also extended the corpus with a few additional established open source software: *pcgen*, *voldemort*, *clojure* and *ceylon-language*.
- A .NET corpus based on a list of open-source software maintained by the community [4].
- A Scala corpus created for testing static analysis tools and maintained by the community [5].

Analysis We perform different analysis suited to the specific programming languages of our corpora.

- We write a Java annotation processor to report uses of wildcards in type expressions, assignments and method invocations.
- We perform a static analysis on .NET bytecode to report uses of variance annotations.
- We write a Scala compiler plugin [8] to analyse the abstract syntax tree of programs and report uses of variance annotations.

3.3 Early Results

We have already sampled 3211 uses of wildcards in type expressions from a corpus of 33 open source Java software, which comprises over 4.1 million lines of code. We found that the vast majority of wildcards in type expressions specified bivariance `<?>` (62 percent). Covariance `? extends` accounted for 35 percent and contravariance `? super` for only 3 percent.

4 Related Work

4.1 Adding Wildcards to the Java Programming Language

In this work, Torgersen et al. introduce *wildcards* as a new language construct to increase the flexibility of object-oriented languages with parameterized classes [22]. They present wildcards as a means to abstract over different kinds of parametric instances to exploit their common properties. In addition, they demonstrate how it is a cleaner and more expressive enhancement to *polymorphic methods* described previously [11]. For example, they note that a method taking a `List` as argument with its element type conforming to a bound is forced to provide a dummy type variable, even though it is irrelevant in the body of the method:

```
<T extends Number> void aMethod(List<T> list){ ...}
```

However, using a wildcard that method argument can be expressed as a a type of `List` whose element type is irrelevant:

```
void aMethod(List<? extends Number> list){ ...}
```

Finally, they present *wildcard capture*, a mechanism to improve type inference in polymorphic methods and how it relates to existential types.

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

In this work, Altidor et al. present a unified framework for reasoning about variance, that benefits both from definition-site and use-site variance [9]. They describe their framework by extending the Java type system with a mechanism that infers the definition-site variance of type parameters, while also allowing use-site variance annotations on any generics instances.

In relation to our work, they applied this technique to six Java libraries and found that 37 percent of the wildcard uses could be eliminated without sacrificing either type safety or the generality of types. In addition, they found that 32 percent of the generics defined can be allowed to enjoy variant subtyping without users having to annotate them with wildcards.

Based on these results, the authors conclude that the need for the flexibility of use-site variance is not as strong as previously thought.

4.3 Java Generics Adoption: How New Features are Introduced, Championed, or Ignored

In this work, Parnin et al. conduct a large study to investigate the use of generics in Java [17]. They provide empirical answers to several claims including:

- The introduction of user-defined generics classes reduce code-duplication.
- When generics are introduced in a codebase, the number of type casts will be reduced.

In relation to our work, they provide a features breakdown of generics. For example, they found that the use of Collections types accounts for over 90 percent of parameterized types across all of the codebases examined. They also found that 33 percent of user-defined generic type declarations had a single parameterization. In addition, they note that wildcards make up 10 percent of uses of parametric types.

4.4 Using the OpenJDK to Investigate Covariance in Java

In this work, we presented an empirical study of the use of covariant arrays in Java [23]. We showed that this property is rarely used by programmers. In addition, we found that 75 percent of covariant arrays uses were caused by calls to Java library methods and only 25 percent were user-defined. Furthermore, all of the uses we found could be transformed and made completely type-safe using generics. We therefore argue that covariant arrays should not be included in future programming languages.

5 Summary

Variance constructs were introduced to increase the flexibility of object-oriented programming languages supporting generics. However, it remains unclear how

programmers use the additional flexibility provided by variance, and whether they use it at all.

The research outlined in this paper intends to explore how programmers make use of variance constructs in real programs.

We first gave an overview of variance mechanisms in object-oriented languages. Then we explained the research questions we aim to answer and the associated research methodology, placing both of these in the context of previous work.

Acknowledgments We would like to thank Alex Buckley, Dominic Orchard and Janina Voigt for their positive comments and interesting discussions related to this research. This work was supported by Qualcomm through a PhD studentship.

References

1. C# specification. http://www.microsoft.com/download/en/details.aspx?id=7029.
2. Dart specification. http://www.dartlang.org/docs/spec/latest/dart-language-specification.pdf.
3. Kotlin. http://confluence.jetbrains.net/display/Kotlin/Welcome.
4. Open Source Software in C#. http://csharp-source.net/.
5. Scala corpus. https://github.com/alacscala/scala-corpus.
6. Scala specification. http://www.scala-lang.org/docu/files/ScalaReference.pdf.
7. Type Variables, JLS SE 7. http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.4.
8. Writing scala compiler plugins. http://www.scala-lang.org/node/140.
9. Altidor, J., Huang, S. S., and Smaragdakis, Y. Taming the wildcards: combining definition- and use-site variance. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 602–613.
10. Bloch, J. *Effective Java*. Prentice Hall, 2008.
11. Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), OOPSLA '98, ACM, pp. 183–200.
12. Cameron, N., and Drossopoulou, S. On subtyping, wildcards, and existential types. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs* (New York, NY, USA, 2009), FTfJP '09, ACM, pp. 4:1–4:7.
13. Igarashi, A., and Viroli, M. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems 28*, 5 (2006), 795–847.
14. Kennedy, A., Russo, C., Emir, B., and Yu, D. Variance and Generalized Constraints for C# Generics. In *Proc. of ECOOP 2006* (2006).
15. Naftalin, M., and Wadler, P. *Java Generics and Collections*. O'Reilly Media, 2006.
16. Odersky, M., Spoon, L., and Venners, B. *Programming in Scala*. Artima Press, 2011.

17. Parnin, C., Bird, C., and Murphy-Hill, E. Java Generics Adoption: How New Features are Introduced, Championed, or Ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (New York, NY, USA, 2011), MSR '11, ACM, pp. 3–12.
18. Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.
19. Smith, D., and Cartwright, R. Java type inference is broken: can we fix it? In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications* (New York, NY, USA, 2008), OOPSLA '08, ACM, pp. 505–524.
20. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)* (Dec. 2010).
21. Torgersen, M., Ernst, E., and Hansen, C. P. Wild FJ. In *Proc. of FOOL 2005* (2005).
22. Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G., and Gafter, N. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM symposium on Applied computing* (New York, NY, USA, 2004), SAC '04, ACM, pp. 1289–1296.
23. Urma, R.-G., and Voigt, J. Using the OpenJDK to investigate covariance in Java. *Oracle Java Magazine* (2012).