Range Parameterized Types: Use-site Variance without the Existential Questions

Bruno De Fraine
System and Software Engineering Lab (SSEL), Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
bdefrain@vub.ac.be

ABSTRACT
Use-site variance approaches such as Java wildcards allows to flexibly derive many co- and contravariant types from one generic class definition. Safety is achieved by restricting the access to members of the parameterized types, but the definition of proper access rules is complicated by the possible nesting of types. Existing approaches for use-site variance relate parameterized classes to bounded existential types and employ a conversion of the type parameters to fresh type variables in order to type an operation. This technique only indirectly explains the operations of a parameterized class and confronts the programmer with type variables which are introduced by the compiler for the purpose of type checking.

In this paper, we propose a more straightforward, uniform and powerful mechanism to systematically address the access restrictions of parameterized types with use-site variance. Parameterized classes receive two type values for each type parameter, and the operations may be determined by simply substituting these values in covariant resp. contravariant positions. The two type values may be interpreted as the bounds of a type range, and we define a safe and flexible variant subtype relation based on range containment. With the exception of wildcard capture, our approach supports all of the common features of other use-site variance models.

Categories and Subject Descriptors
D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.3 [Logics and Meaning of Programs]: Studies of Program Constructs—Object-oriented constructs; Type structure

General Terms
Design, Languages, Theory

Keywords
generic classes, language design, language semantics, subtyping, variance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION
Generics and generic classes are a common feature in modern statically-typed object-oriented programming languages, such as Java 5 [6], C# 2.0 and Scala [11]. One of the most common applications of generic types is the definition of generic containers, where parameterized types such as List<String> allow for client code that is more robust, readable and maintainable. However, without additional measures this parametric polymorphism mechanism does not optimally integrate with the subtype polymorphism that is traditionally employed by object-oriented type systems. Specifically, there is no subtype relation between different instantiations of the same generic class. This restriction originates from the fact that a generic interface such as List<E> offers both operations to retrieve E instances as well as operations that accept E instances. For instance, the interface of List<Object> will accept any Object instance, while List<String> accepts only String instances, and oppositely, List<String> provides strictly String instances while List<Object> may provide other Object instances as well. Because neither of the two types supports the complete interface of the other type, there is no safe substitution between these two parameterizations of the same generic class, and it is therefore unsound to relate them by subtyping.

It may be noted that this subtype restriction may be relaxed when the generic interface offers either only ‘producer’ operations or only ‘consumer’ operations with respect to the type represented by the type parameter. The traditional approach to enable covariant or contravariant subtyping for parameterized types is therefore to associate a variance with the formal type parameters in the generic type declaration. Consequently, only the definition of members where this type variable appears in respectively a covariant or contravariant position is then allowed for this class. Such declaration-site variance is supported by the type systems of e.g., StrongTalk [2] and Scala [11, Sec. 5.1]. Alternatively, use-site variance approaches consider variance as a property of the actual type parameters in a parameterized type and restrict access to the members rather than constraining the member definitions. This allows to derive many covariant and/or contravariant types from one generic class definition and arguably provides a more flexible mechanism. Two concrete proposals in this area are variant parametric types [8, 9] and Java wildcards [14, 13].

An example of a variant parametric type is List<+Number>, which is the instantiation of the generic List with type Number, where only a limited set of operations are accessible such that List<+Number> supports covariant subtyping.
In order to determine these operations, Igarashi and Viroli propose [8, Sec. 5] to interpret List<<Number>> as the bounded existential type $\exists X. \langle X. \text{List}, X \rangle$. Indeed, List<<Number>> should support only those operations that are guaranteed to work for any type List<T>, where T is a subtype of Number. Following this interpretation, the typing rules for existential types are employed; for example, an operation on List<<Number>> is type checked by converting this type to the ordinary parameterized type List<X> where X is a fresh type variable and the assumption X < Number is added to the type environment (technically, this operation is called the opening of the existential type). In Java 5, wildcard type arguments provide a very similar mechanism that also exploits the connection to existential types in order to type check operations on parameterized types. Additionally, the existential type interpretation is represented more explicitly in the concrete syntax of such wildcard parameterized types: List<T extends Number> is the equivalent of List<<Number>> and may be read as “list of some subtype of Number”.

While the usage of existential types in this context is a highly interesting insight, the main downside of formulating typing rules for operations on variant parameterized types by ‘delegating’ to a non-variant parameterized type with a variable type argument (and certain assumptions regarding this variable type), is that the conditions for a well-formed operation are specified indirectly, making them hard to reason about by the programmer. We will elaborate on this and some other issues in Section 2. Next, we present a generalization of wildcard parameterized types and variant parameteric types called range parameterized types in Section 3. The members of range parameterized types may be determined by a direct substitution operation (as is the case for ordinary parameterized types), thereby avoiding the issues related to the use of existential types. We will discuss some advanced issues related to our proposal in Section 4 and conclude in Section 5.

2. MOTIVATION

To illustrate some of the difficulties involved with the typing of operations using existential types, we will consider an example in the context of the Java Collections library. The generic Collection interface describes a method to add all of the elements from another Collection, where the type of this collection argument has a wildcard type argument:

```java
interface Collection<E> {
    void addAll(Collection<? extends E> c);
    Iterator<E> iterator();
    ...
}
```

The intuition is that addAll will only retrieve E elements from its argument (by means of the iterator method); it will not add elements to it. Hence, an argument of the type Collection<? extends E> satisfies for addAll.

Now, consider a variable c of wildcard parameterized type Collection<? super Number>. In order to type the operation c.addAll, a Java compiler will apply capture conversion [6, §5.1.10], which means that the type of c is changed to Collection<?>, where Z is a fresh type variable with the lower bound Number. Since the type of c now has an ordinary type argument Z, we may substitute Z for E in the definition of the generic type Collection, to obtain that the operation c.addAll requires an argument of type Collection<? extends Z>. However, this is not a direct answer since Z is not a type that the programmer has firsthand knowledge of. In order to determine whether an argument of a concrete type, say Collection<Integer>, is valid, the programmer must mentally consider whether this concrete type is provably a subtype of Collection<? extends Z> under the assumption that Number < Z.

The diligent reader that follows along with this exercise may derive that Collection<Integer> is indeed such a subtype, but this task is arguably rather hard and it may be noted that there exists a more direct alternative [15]. Given that the programmer has no way of referencing Z, the argument type of c.addAll may (for all practical purposes) be taken to be Collection<? extends Number>, which is the most general subtype of Collection<? extends Z> which does not refer to Z. This type promotion is formalized as a close operation by Igarashi, Viroli and Rimassa in [8], where the close operation is only applied to the result type of an operation, and in [15], where also the argument types are closed. However, in case of nested types, the promotion proposed by these authors may need to discard some type information in order to provide an expressible type [13, Sec. 3.1.3]. Furthermore, there may not exist a unique most general subtype or most specific supertype among the set of expressible types [9, Sec. 4.2.3] (we will revisit these problems in Section 4, after the presentation of our proposal). Finally, even if these expressiveness issues were to be resolved, the consecutive introduction and removal of a type variable (via respectively the open and close operation) is still more difficult to reason about than a direct substitution, as is possible for ordinary parameterized types.

3. RANGE PARAMETERIZED TYPES

In order to give a formal definition of our language, we present syntax and rule definitions in a “featherweight” context (referring to the Featherweight Java and Featherweight GJ calculi [7]). This means that we do not consider such features as nested classes, interfaces, abstract classes and methods, and non-final fields. Nevertheless, Featherweight GJ (or short, FJG) retains essential features such as type variables with bounds and F-bounds, and generic classes and methods. Extensive previous experience with featherweight calculi [9, 13, 3] has indicated this is an appropriate context to evaluate generics features. We also employ some notational conveniences from [7]. We use metavariables c, d for class names, f, g for field names, ℓ for method names, x for term variable names and X, Y for type variable names. We write $\tilde{e}$ as a shorthand for an ordered sequence of elements $e_1, \ldots, e_n$ where the element separator depends on the context. This convention is sometimes extended across binary or ternary constructs where the elements of the sequences should be appropriately ‘zipped’. For example, $\bar{P} \bar{x}$ is a shorthand for $P_1 x_1, \ldots, P_n x_n$. Additionally, we employ the symbol $\triangleleft$ as a shorthand for the keyword extends in parent and bound declarations, and we abbreviate the empty angular brackets

---

1Existential types are developed in [5, 10] as a typing mechanism to restrict direct access to the implementation of abstract data types.

2In compiler error messages, the javac compiler version 1.6 refers to the type Z with a cryptic name such as capture #137 of ? super Number.
**3.1 Type Structure and Class Declarations**

The central idea of range parameterized types is to provide two type values for each type parameter of a generic class. This leads to the following structure of type expressions:

\[
\begin{align*}
C, D & ::= \ll S-\bar{U} \gg \quad \text{non-variable types} \\
P, S, T, U & ::= X \mid C \mid \text{Null} \quad \text{general types}
\end{align*}
\]

The non-variable type \( \ll S-\bar{U} \gg \) should be interpreted as an instance of the generic class \( \ll X \gg \) where each type variable \( X_i \) has been replaced by \( U_i \) when it appears in covariant positions, and by \( S_i \) in contravariant positions (we define this type substitution formally in Section 3.2). Consequently, an ordinary (non-variant) type argument \( T_i \) corresponds to the situation where \( S_i = U_i = T_i \). In Section 3.3, we additionally explain that the types \( S_i \) and \( U_i \) may be considered as the lower and upper bound of a range of possible type arguments, similar to the interpretation of wildcard type arguments.

Note that we include \( \text{Null} \) to denote the bottom type (or "null type"): as formally defined in Section 3.3. \( \text{Null} \) is a subtype of any type. In Java, the null type is the type of the \( \text{null} \) reference [6, §4.1], where \( \text{null} \) may support any operation by simply raising a \( \text{NullPointerException} \) at runtime. In our context, we do not include the \( \text{null} \) reference and there are no inhabitants of \( \text{Null} \).

The class \( \text{Object} \) is implicitly defined and other classes are declared with the following header structure:

\[
\text{class } \ll X \gg \ll C \gg < D (\cdots)
\]

The type variables each have a bound. Identical to FGJ, the bound of a type variable may not be a type variable, but may be a type expression involving type variables, and may be recursive (to allow the mechanism of \( P \)-bounds [4]). Also as in FGJ, the class members are one fixed constructor and any number of fields and generic methods. Method bodies are type checked using a context where the term variable \( i \) is explicitly defined and other classes are adapted for range parameterized types.

**3.2 Member Access**

The members of a range parameterized type are essentially determined in the same manner as the members of a parameterized type in FGJ, except that a more refined type substitution is employed.

The type substitution function is defined using two auxiliary structures. The first is an environment which maps type variables to a pair of type values, and the other is simply a sign with two possible values:

\[
\begin{align*}
E & ::= X \rightarrow S-\bar{U} \\
s & ::= + \mid -
\end{align*}
\]

We assume an operation \( E(X) \) to access the type values associated with a type variable inside an environment: if the environment contains the mapping \( X_i \rightarrow S_i-U_i \), then \( E(X_i)_s = U_i \) and \( E(X_i)_- = S_i \). Additionally, we define the negated sign \( \neg s \), which will simply turn \( + \) into \( - \) and vice versa.

The type substitution in type \( T \) according to environment \( E \) and sign \( s \) is written \( T[E]_s \), and it is defined by case analysis according to the three forms of types that we defined in the previous section:

\[
T[E]_s = \begin{cases} 
E(X)_s & \text{when } X \in E \\
X & \text{otherwise}
\end{cases}
\]

\[
\ll S-\bar{U} \gg [E]_s = \ll S\rightarrow S\rightarrow\bar{U}\rightarrow\bar{U} \gg [E]_s
\]

\[
\text{Null}[E]_s = \text{Null}
\]

The interpretation of \( T[E]_s \) is that it substitutes covariant (+) or contravariant (−) occurrences of the type variables bound in \( E \) in type \( T \), where \( s \) indicates whether \( T \) itself occurs in a covariant or contravariant position. Note from the second case that the covariant type arguments \( \ll U \gg \) for a parameterized type are considered to be in a position of equal sign as the parameterized type itself, while the contravariant type arguments \( \ll S \gg \) are considered in a position of opposite sign. This ‘flipping’ of the sign in the position of a contravariant type argument is identical to what occurs in some approaches that support declaration-site variance, for example, Scala [11, Sec. 5.1].

We then proceed to define the field and method members of range parameterized types. In Figure 1, we give a definition of the functions \( \text{fields} \) and \( \text{mtype} \) from FGJ, adapted for range parameterized types. The interpretation of the member definitions is the following. The range parameterized type \( \ll S-\bar{U} \gg \) provides the members from the declaration of the generic type \( \ll X \gg \) after a type substitution according to the environment \( X \rightarrow S-\bar{U} \) and a positive or negative sign, respectively co- and contravariant positions. For methods, the method argument types and the bounds of the type arguments constitute contravariant occurrences; the return type constitutes a covariant occurrence. Again, this corresponds to what are considered covariant and contravari-
ant positions in approaches for declaration-site variance\(^3\).

For fields, we have to distinguish between read and write operations. We therefore equip the field function with a sign parameter, which will be \(+\) when the field is being read, and \(−\) when the field is being written. This difference has little importance in a featherweight context, where fields can only be written from within constructors. However, we choose to already make the distinction in anticipation of a more sophisticated type substitution.

We do not specify new typing rules for term expressions or member declarations. It is one of the central contributions of our proposal that the typing of operations on range parameterized types may simply be explained using the typing rules from FGJ. We have only updated the definitions of the functions fields and mtype in order to incorporate a more sophisticated type substitution.

### 3.3 Variant Subtyping

The subtyping rules for range parameterized types are defined in Figure 2. In this figure, the metavariable \(\Delta\) represents a type environment which contains upper bound assumptions for type variables. The first four rules have similar counterparts in FGJ. In addition, rule \textsc{Sub-Null} simply states that \texttt{Null} is a subtype of any type. Finally, rule \textsc{Sub-Range} enables variant subtyping: the parameterized type \(\texttt{<$S$-$U$} >\) behaves covariantly with respect to changes to the type values \(\bar{U}\) and contravariantly for changes to the type values \(S\). In other words, the type \(\texttt{<$S$-$U$} >\) is a supertype of \(\texttt{<$S'$-$U'$} >\) when the ranges \(S' < U\) contain the respective ranges \(S < \bar{U}\), as indicated by lower lower bounds \((S' < S)\) and higher upper bounds \((\bar{U} < \bar{U}'\)). Furthermore, because of the restrictions on run-time values, the inhabitants of a type \(\texttt{<$S$-$U$} >\) will be the values new \(\texttt{<$T$} >\)\((\ldots)\) where \(S < T < U\) (plus similar values for any subclass of \(c\)). Due to the transitivity of the subtype relation, only range parameterized types where \(S < U\) can therefore be inhabited (with the exception of the inhabitance by the \texttt{null} reference, if it is considered). These facts explain why one may consider the type values \(S < U\) as ranges of possible type arguments. In Java, subtyping between wildcard parameterized types is similarly defined based on the concept of type argument containment [6, §4.5.1.1]. Due to their equivalent subtype relations, we may draw the following correspondence between

\(^3\) It also corresponds to well-known subtype rules for function types and type abstractions from systems such as \(F_c\) [12, Ch. 26].

(A similar correspondence may be defined for variant parametric types.) Note that, using Java wildcards, the programmer cannot express an equivalent for a range parameterized type \(\texttt{<$S$-$U$} >\) where \(S \neq \texttt{Null}\) and \(U \neq \texttt{Object}\) and \(S \neq U\), although such a type is considered internally by some type checkers; such a type is also included in the Wild FJ calculus [13] as \(\texttt{<> <U} >\).

One may wonder where the rule \textsc{Sub-Range} derives its validity, i.e., why is safe substitution guaranteed for values of these types? Informally, we aim to provide some insight in this matter through a connection to a system with declaration-site variance. For a generic class declaration \(cX\) in our system, suppose that one defines a corresponding class \(c'<Y', +Z'\) with declaration-site variance. The class \(c'\) is defined with methods with signatures identical to those of \(c\), except that any use of \(X\) in the signatures is replaced by \(Z\) in covariant positions and by \(Y\) in contravariant positions (this corresponds to the variance that is declared for type variables \(Y\) and \(Z\)). Now, we note that the interface of the range parameterized type \(\texttt{<$S$-$U$} >\) is precisely the same as that of \(c'<Y', +Z'\) with declaration-site variance. The class \(c'\) offers the same interface on the outside, the declaration of the generic classes \(c\) and \(c'\) follows different rules on the inside: as explained in Section 3.1, the bodies of methods declared in \(c\) have access to a \texttt{this} variable of type \(cX\)\(\ldots\), whereas the methods of \(c'\) have a \texttt{this} variable of type \(c'X\)\(\ldots\), with no relation between \(X\) and \(Y\). The method declarations for \(c\) are therefore not generally admitted for class \(c'\).

### 3.4 Revisited Example

We now revisit the example from Section 2, in the context of range parameterized types. Using our formalism, the generic type \texttt{Collection} may be defined as follows:

```java
interface Collection<E> {
    void addAll(Collection<Null-E> c);
    Iterator<E-E> iterator();
    ...
}
```

The argument of addAll is of type \texttt{Collection<Null-E>}, which is sufficiently specific for the implementation of this method to allow the retrieval of elements of type \(E\) from this argument. Concretely, the \texttt{iterator} method for this type has the return type \texttt{Iterator<Null-E>}, which type, in turn, offers a \texttt{next} method with return type \(E\) (since this return type is a covariant position). To recreate the scenario of Section 2, we consider a variable \(c\) of type
Collection<? extends Number>, which corresponds to the wildcard parameterized type Collection<? super Number>. The member access rules for range parameterized types directly indicate that the argument type of c.addAll is the type Collection<null-Number>. This corresponds to precisely the desired argument type Collection<? extends Number> that we named in Section 2.

4. Advanced Issues

**Nested types.** As explained in Section 2, it is reported in [9, Sec. 4.2.3] and [13, Sec. 3.1.3] that certain expressiveness issues arise for the close operation proposed by Igarashi, Viroli and Rimassa in the case of nested parameterized types. In order to illustrate how range parameterized types deal with this situation, we revisit two of the corner cases that are considered by the authors of these papers. Consider the following hypothetical generic type where the return type of two methods is a nested range parameterized type:

```java
interface Box<X> {
    Box<Box<X-Number>> nest();
    Box<Box<X-Object>> nest2();
}
```

We now consider the parameterized type Box<null-Number>. The signatures of methods nest and nest2 for this type are immediately obtained through type substitution:

```java
Box<Box<Number-null>> - Box<null-Number>> nest();
Box<Box<Number-Object>> - Box<null-Object>> nest2();
```

These return types are not expressible as variant parametric types or wildcard parameterized types and they can therefore not be the result of the close operation in these systems. Instead, an expressible supertype is employed, which we may obtain from the above range parameterized types by either changing the upper type argument to the trivial Object or the lower type argument to the trivial Null. In case of nest, the choice is obvious, since the lower type argument Box<Number-null> is also not expressible as a variant parametric type (it is also only inhabited by null since Number !: Null). However, in case of nest2, both options are viable and the close operation has two possible results which correspond to the range parameterized types Box<null - Box<null-Object>> and Box<Box<Number-Object>> - Object.

In contrast to the above problems, range parameterized types provide a system that is closed with respect to its own expressiveness: there is always a single best argument or result type for an operation. Now, it is imaginable that the expressiveness of variant parametric types or wildcard parameterized types were to be extended such that the above return types may be expressed. However, in that case the argument remains that it is more direct to obtain these results by means of a substitution operation than through the consecutive introduction and removal of a type variable (via respectively the open and close operation).

**Wildcard capture.** Java wildcards allow the programmer to directly exploit the fact that wildcard parameterized types are type checked by means of a capture conversion operation. This may be illustrated by considering the variable 1 of type List<?>. Since the inhabitants of this type will always be of type List<T> for some type T, we know that a copy operation between the elements of the same List<? instance will be sound. This operation is not directly supported, but the wildcard capture mechanism [14, 13] allows to open the type T as a type variable within the scope of generic method that implements the operation:

```java
<X> void copy(List<X> l) { 1.set(1, l.get(0)); }
```

For the invocation of copy(1), the type of 1 is capture converted to List<X>. Due to wildcard capture, the type argument of the generic method invocation is inferred to be Z (since the programmer cannot himself refer to Z, this entails that the type inference is no longer only a convenience mechanism).

Admittedly, this functionality is not supported in our proposal, and it would seem that it can only be added by interpreting the range parameterized type List<null-Object> as the existential type \( \exists X \in \text{null-Object}. \text{List}<X>X \) and introducing an explicit open operation for this type. It appears that the additional power of existential types is warranted in these situations. However, the situation remains that the complexity of an existential typing is uncalled for in the case of standard operations. In our opinion, an application of range parameterized type, together with an existential type mechanism for these limited specific situations, would best fit the famous credo by Alan Kay: “simple things should be simple, complex things should be possible”.

5. Conclusions and Future Work

In this paper, we propose range parameterized types, a new approach for the use-site variance of generic classes and interfaces. The members of a range parameterized type are directly determined through a type substitution, similar to approaches without variance, yet these types provide flexible variant subtyping. This improves over existing approaches for use-site variance which specify the conditions for well-formed operations indirectly and confront the programmer with type variables which are introduced by the compiler for the purpose of type checking.

We speculate that our approach is type sound and we provide an informal safety argument. It is clear however that a complete formal safety proof is required to support this claim. We have begun work to develop a machine-verified version of such a proof using the Coq proof assistant [1]. At the time of writing, a development of preservation and progress properties of the Featherweight GJ calculus has been completed, and it is our hope that range parameterized types may be integrated in a straightforward manner because of their proximity in language features to FGJ. To our knowledge, this would be the first use-site variance mechanism that is evaluated at this level of rigor: while there are full type soundness proofs for variant parametric types [9] and wildcards [3], the correctness of these proofs has not yet been mechanically verified.

6. References


