

Variant Parametric Types*

A Flexible Subtyping Scheme for Generics

Atsushi Igarashi

Graduate School of Informatics, Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Mirko Viroli

DEIS, Università degli Studi di Bologna
mviroli@deis.unibo.it

August 4, 2003

Abstract

We develop the mechanism of *variant parametric types*, as a means to enhance synergy between parametric and inclusive polymorphism in object-oriented languages. Variant parametric types are used to control both subtyping between different instantiations of one generic class and the accessibility of their fields and methods. On one hand, one parametric class can be used as either covariant, contravariant, or bivariant by attaching a variance annotation to a type argument. On the other hand, the type system prohibits certain method/field accesses according to variance annotations, when those accesses may otherwise make the program unsafe. By exploiting variant parametric types, a programmer can write generic code abstractions working on a wide range of parametric types in a safe way. For instance, a method that only reads the elements of a container of strings can be easily modified so as to accept containers of any subtype of string.

Technical subtleties in typing for the proposed mechanism are addressed in terms of an intuitive correspondence between variant parametric types and bounded existential types. Then, for a rigorous argument of correctness of the proposed typing rules, we extend FGJ—an existing formal core calculus for Java with generics—with variant parametric types and prove type soundness.

1 Introduction

1.1 Background

The recent development of high-level constructs for object-oriented languages is witnessing renewed interest in the design, implementation, and applications of parametric polymorphism—also known as *generics*. Such interest has been growing mostly due to the emergence and development of the Java programming language. Initially, Java’s designers decided to avoid generic features, and to provide programmers only with inclusive (or subtyping) polymorphism, supported by inheritance. However, as Java was used to build large-scale applications, it became clear that the introduction of parametric polymorphism would have significantly enhanced programmers’ productivity, as well as the readability, maintainability, and safety of programs. In response to Sun Microsystems’ call for proposals for adding generics to the Java programming language [41] a lot of extensions have been

*A preliminary version appeared under the title “On Variance-Based Subtyping for Parametric Types” in Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP2002), Springer LNCS vol. 2374, pages 441–469, 2002.

proposed ([5, 15, 49, 48, 32, 1] to cite some); finally, Bracha, Odersky, Stoutamire, and Wadler’s GJ [5] has been chosen as the reference implementation technique for the first upcoming release of Java with generics. Other than Java, more recently an extension of Microsoft’s .NET Common Language Runtime [34] (CLR) with generics has been studied [42, 33].

Given this growing interest in generics, we believe that studying its language constructs will play a key role in increasing the expressiveness of mainstream programming languages such as Java and C#. In this paper, we explore a technique to enhance the synergy between inclusive and parametric polymorphism, with the goal of increasing expressiveness and reuse in object-oriented languages supporting generics.

1.2 Previous Approaches to Subtyping for Generic Types

In most of current mainstream object-oriented languages—such as Java, C++, and C#—inclusive polymorphism is supported only through inheritance: class C is considered a subtype of class D if and only if C is declared to be a subclass of D . Extensions of these languages with generics usually adopt a subtyping scheme called *pointwise* subtyping, which is a straightforward extension of the monomorphic setting above: for instance, provided that class $\text{Stack}\langle X \rangle$ is a subclass of $\text{Vector}\langle X \rangle$ (where X is a type parameter) a parametric type $\text{Stack}\langle \text{String} \rangle$ is a subtype of $\text{Vector}\langle \text{String} \rangle$; similarly for any type argument. Pointwise subtyping, however, never allows two instantiations of one generic class to be in the subtyping relation. For example, $\text{Vector}\langle \text{Integer} \rangle$ and $\text{Vector}\langle \text{Number} \rangle$ are not related with pointwise subtyping.

Historically, most of well-known attempts to introduce another subtyping scheme for generics were based on the notion of *variance* for classes, which is used to define a subtype relation between different instantiations of the same generic class. Basically, a generic class $C\langle X \rangle$ is said to be *covariant* with respect to X if $S \prec T$ implies $C\langle S \rangle \prec C\langle T \rangle$ (where \prec denotes the subtyping relation), and conversely, $C\langle X \rangle$ is said to be *contravariant* with respect to X , if $S \prec T$ implies $C\langle T \rangle \prec C\langle S \rangle$. Also, $C\langle X \rangle$ is said to be *invariant* when $C\langle S \rangle \prec C\langle T \rangle$ holds only if $S = T$. A familiar example of covariant parametric types is array types in Java, in which, $\text{String}[]$ is a subtype of $\text{Object}[]$ since String is a subtype of Object . Naive introduction of covariant subtyping, however, makes the type system unsound. For example, given a variable v of type $\text{Object}[]$, it may be unsafe to update an element of v with a new Object , because v may actually point to an array of strings.¹ As a result, Java arrays require every assignment to an array to be checked at run-time: if an incompatible value is to be assigned, an exception will be thrown.

There have been several proposals [16, 2, 4, 3] for a sound type system for generics with variance and most of them take a similar approach: in short, covariance and contravariance can be permitted under certain constraints on the occurrences of type variable X within $C\langle X \rangle$ ’s signature. For example, for a generic class $C\langle X \rangle$ to be covariant, X must not appear in an argument type or in a type of a writable instance variable and so on. Conversely, in order for $C\langle X \rangle$ to be contravariant, X must not appear in a return type or in a type of a readable instance variable. Those restrictions are often understood in connection with readability and writability of instance variables. For example, consider a generic collection class whose element type is abstracted as a type parameter; typically, such a class can be covariant if it provides methods only to read elements, while it can be contravariant if it provides methods only to write elements.

This approach, which works in principle, apparently turns out to pose some difficulty in practice. Since variance can be obtained by prohibiting the declaration of possibly dangerous operations,

¹It may be worth pointing out that side-effecting features are not the only source of unsoundness; even purely functional GJ would be unsafe if covariant subtyping was allowed.

programmers will face a tradeoff between a rich set of subtypes thanks to variance and a rich set of methods in a class. Even worse, this scheme for variance would decrease reusability: one may be forced to declare three very similar classes for any kind of collection—an invariant one with methods for both reading and writing elements, a covariant one obtained by dropping methods for reading, and a contravariant one obtained by dropping methods for writing.

1.3 Our Approach

Our approach here is to let programmers defer the decision about which variance is desirable until a class is *employed*, rather than when it is *declared*. To put it more concretely, for any type argument T , a parametric class $C<X>$ may be used to induce the type $C<T>$, which is invariant as usual, but also the types $C<+T>$ and $C<-T>$, which are respectively covariant and contravariant; in exchange for variance, certain (potentially unsafe) member accesses through $C<+T>$ and $C<-T>$ are forbidden. In the case of collection classes above, only methods to read elements are accessible via covariant types and only methods to write elements are accessible via contravariant ones. In other words, our approach amounts to restrict *accessibility*—rather than *definability*—of members of a class by implicit *interfaces* automatically derived by the annotations $+$ and $-$. As a result, it is expected that class designers are released from the burden of taking variance into account and, moreover, class reuse is promoted since one class can be used to derive read-only covariant and write-only contravariant interfaces.

The idea of annotating type arguments for covariance has emerged in the study of *structural virtual types* [45], where a possible application of the idea to generic classes is pointed out. However, a rigorous treatment of the problem—including the study of contravariance, the integration with other language features such as inheritance, and the development of a type system—is still lacking, so it remains unclear how this approach to variance will successfully work in a full-blown language design.

1.4 Contributions of the Paper

We develop the idea above into the mechanism of *variant parametric types* for possible application to widely disseminated, modern object-oriented languages such as Java and C#. Our contributions are summarized as follows:

- We propose a more general subtyping scheme for variance than what has been proposed in the literature: in particular, in the attempt of integrating covariance and contravariance we find it useful to add *bivariance*—the property that two different instantiations are in a subtyping relation regardless of their type arguments.
- We argue the usefulness of variant parametric types. Most notably, they are used to extend the set of acceptable arguments to a method, when it uses arguments in certain disciplined manners. Furthermore, we show that variance is effective also for nested parametric types, such as vectors of vectors.
- We point out an intuitive correspondence between variant parametric types and bounded existential types [36, 14] and give a rationale of design decisions on typing, including access control enforced in exchange for variance and typing in terms of the correspondence. In fact, although variant parametric types are fairly easy to understand in most basic uses, they introduce some surprising subtlety, especially when they are nested. Notice that previous work on variance [45, 16, 2, 4, 3] does not address the interaction with nested parametric

types in a satisfactory manner, even though this case often arise in practice, making the whole safety argument unclear.

- Basing on Featherweight GJ—a generic version of Featherweight Java [23]—we define a formal core language of variance with its syntax, type system, and operational semantics, and prove type soundness.

Compared to the result presented in the early version [25], we have extended the core calculus with generic methods and provided a type soundness proof for the extended calculus and more detailed comparisons with other related language mechanisms.

At the time of writing, the mechanism developed in this paper is considered for the inclusion in the JDK 1.5 official release of the Java programming language and has been implemented in a prototype compiler by Sun Microsystems as an experimental feature (<http://developer.java.sun.com/developer/bugParade/bugs/4856545.html>).

1.5 Outline of the Paper

The remainder of the paper is organized as follows. In Section 2, the classical approach to variance for parametric classes is briefly outlined. Section 3 informally presents the language construct of variant parametric types, and addresses its design issues. Section 4 discusses the applicability and usefulness of these types, by showing examples of applications of covariance, contravariance, bivariance, and their combinations. Section 5 elaborates the interpretation of variant parametric types as a form of bounded existential types, gives a rationale of the basic design, and discusses subtleties of typing. Section 6 presents the core calculus for variant parametric types and proves the soundness of its type system. Section 7 discusses related work and Section 8 presents concluding remarks and perspectives on future work.

2 Classical Approach to Variance for Parametric Classes

Historically, one main approach to flexible inclusive polymorphism for generics was through the mechanism of *variance* for classes, whose limitations made it never appear in widely disseminated object-oriented languages, such as Java and C++.

A generic class $C\langle X \rangle$ is said to be *covariant* in the type parameter X when the subtype relation $C\langle S \rangle \prec C\langle T \rangle$ holds if $S \prec T$. Conversely, $C\langle X \rangle$ is said to be *contravariant* in X when $C\langle S \rangle \prec C\langle T \rangle$ holds if $T \prec S$. General notions of *bivariance* and *invariance* can be defined as well. $C\langle X \rangle$ is said to be *bivariant* in X when $C\langle S \rangle \prec C\langle T \rangle$ for any S and T . $C\langle X \rangle$ is said to be *invariant* in X when $C\langle S \rangle \prec C\langle T \rangle$ holds only when $S = T$. Since variance is a property of each type parameter of a generic class, all these definitions can be easily extended to generic classes with more than one type parameter.

However, not every class can be safely given an arbitrary variance property, as array types in Java demonstrate. Java arrays can be considered a generic class from which the element type is abstracted out as a type parameter; moreover, the array types are defined to be covariant in that type—e.g., `String[]` is a subtype of `Object[]`. However, since arrays provide the operation to update their content, even a well-typed program can lead to a run-time exception, as the following Java code shows:

```
Object[] o=new String[]{"1","2","3"};
o[0]=new Integer(1); // Throws a java.lang.ArrayStoreException
```

The former statement is permitted because of covariance. The latter is also statically accepted, because `Integer` is a subtype of `Object`. However, when the code is executed an exception `java.lang.ArrayStoreException` is thrown: the bytecode interpreter tries to insert an `Integer` instance to where a `String` instance is actually expected. Moreover, the need for run-time checks to intercept wrong insertions of elements considerably slows down the performance of Java arrays.

To recover safety, previous work [16, 2, 4, 3] proposed to pose restrictions on how a type variable can appear in a class definition, according to its variance. Those restrictions are derived from Liskov's substitution principle [28]—for a type `S` to be safely considered a subtype of `T`, an instance of `S` can be passed to where an instance of type `T` is expected without incurring in additional run-time errors. When a class is covariant in the type parameter `X`, for instance, `X` should not appear as type of a public (and writable) field or as an argument type of any public method. Conversely, in the contravariant case, `X` should not appear as type of a public (and readable) field or as return type of any public method. For example, the following class (written in a GJ-like language [5])

```
class Pair<X extends Object, Y extends Object> extends Object{
    private X fst;
    private Y snd;
    Pair(X fst,Y snd){ this.fst=fst; this.snd=snd; }
    void setFst(X fst){ this.fst=fst; }
    Y getSnd(){ return snd; }
}
```

can be safely considered covariant in type variable `Y` and contravariant in type variable `X`, since `Y` appears only as the return type in `getSnd` and `X` appears only as the argument type in `setFst` (except private fields and constructors). In some existing proposals—such as Strongtalk [4, 3] and the NextGen compiler for generics in Java [15]—one has to declare desirable variance to guide typechecking by the compiler, by putting `+` (for covariance) or `-` (for contravariance) before the formal type variables. Following this notation the first line of the declaration above would be:

```
class Pair<+X extends Object, -Y extends Object> extends Object{
```

It is easy to see that any type `Pair<S,T>` can be safely considered a subtype of `Pair<String,Number>` when `S` is a supertype of `String` and `T` is a subtype of `Number`, as the following code reveals.

```
Number getAndSet(Pair<String,Number> c, String s){
    c.setFst(s);
    return c.getSnd();
}
...
Number n=getAndSet(new Pair<Object,Integer>(null, new Integer(1)),"1");
```

In fact, the invocation of `getAndSet` causes the string `"1"` to be safely passed to `setFst`, which expected an `Object`, and an `Integer` object to be returned by `getSnd`, whose return type is `Number`.

However, it is now commonly recognized—see e.g., Day et al. [17]—that the applicability of this mechanism seems not so wide as expected, since type variables typically occur in such positions that forbid both covariance and contravariance. Consider the usual application of generics as collection classes, and their typical signature schema with methods for getting and setting elements, as is shown in the following class `Vector<X>`, which can be neither covariant nor contravariant:

```

class Vector<X>{
    private X[] ar;
    Vector(int size){ ar=new X[size];}
    int size(){ return ar.length; }
    X getElementAt(int i){ return ar[i]; }    // Reading elements disallows contravariance
    void setElementAt(X t,int i){ ar[i]=t; } // Writing elements disallows covariance
}

```

Typically, the type variable occurs as a method return type when the method is used to extract some element of the collection, while the type variable occurs as a method argument type when the method is used to insert new elements into the collection. So, a generic collection class can be considered covariant only if it represents read-only collections, and contravariant only if it represents write-only collections. Bivariance is even more useless since it would be safely applied only to collections whose content is neither readable nor writable.

One possible solution to this problem is to split the class for vectors into two classes: a read-only (hence covariant) vector class `ROVector<+X>` and an (invariant) subclass `Vector<X>` with operations to write.

```

class ROVector<+X>{
    protected X[] ar;
    ROVector(int size){ ar=new X[size];}
    int size(){ return ar.length; }
    X getElementAt(int i){ return ar[i];}
}
class Vector<X> extends ROVector<X>{
    Vector(int size){ super(size);}
    void setElementAt(X t,int i){ ar[i]=t;}
}

```

In this way, for instance, the type `ROVector<Number>` is not only a safe supertype of `ROVector<Integer>` and `ROVector<Float>`, but also of `Vector<Integer>` and `Vector<Float>`. One of the consequences of the observation above is that class designers have to be responsible for the tradeoff between variance and available functionality of classes together with their subclassing hierarchy.

Unfortunately, this approach—which casts a heavy burden on class designers—does not scale up very well in practice. On one hand, in order to deal with contravariance as well, multiple inheritance would be required: `Vector<X>` can be defined by extending the covariant class `ROVector<+X>` and the contravariant one `WOVector<-X>` (which provides only the writing functionality), both classes extending a common base class `NORWVector`. On the other hand, as the number of type parameters increases, the number of classes may exponentially increase, e.g. coding a pair class with two type parameters would require a diamond structure of 16 classes.

A solution to the applicability limitations of the classical approach is hinted in the work on structural virtual types [45]. The idea is to let a programmer specify whether their type arguments are invariant or covariant when a class is *employed* rather than when a class is *defined*: for covariance, the symbol `+` is inserted before the *actual* type argument, e.g., writing `Vector<+Object>`, which behaves similarly to the structural virtual type `Vector[X<Object]` and prohibits write access to the vector of that type. With this syntax, where variance annotations are moved from parametric class definitions to parametric types, the choice of a variance property can be substantially deferred. At least, two main advantages appear to arise from this approach: the applicability of the mechanisms is widened, since covariant (and contravariant) types can be derived from any class—independently of it being e.g. read-only or write-only—and the designers of libraries are released from the burden of making decisions about the tradeoff mentioned above. So, in this paper

we generalize this idea by investigating the inclusion of covariance, contravariance, and bivarience and develop the mechanism of *variant parametric types* in a full-fledged language design.

3 Variant Parametric Types

This section introduces the basic design of *variant parametric types*, including subtyping and rules access restriction due to variance. Variant parametric types are a generalization of standard parametric (or generic) types (such as `Vector<String>` and `Pair<String,Integer>`) where each type parameter may be associated with a *variance annotation*, either `+`, `-`, or `*`, respectively referred to as the *covariance*, *contravariance*, or *bivarience annotation symbol*. Syntactically, a variance annotation symbol precedes a type parameter and introduces the corresponding variance to the argument position: for example, `Vector<+String>` is a subtype of `Vector<+Object>`. Variant parametric types can be arbitrarily nested: a type parameter of a variant parametric type can be a variant parametric type as well as a type variable, as in `Vector<+String>`, `Pair<String,-X>`, and `Vector<Vector<*X>>`. A parametric type where no (outermost) type argument has a variance annotation is called an *invariant type*. `Vector<String>` and `Pair<Vector<+String>,Integer>` are examples of invariant types. These types serve as a role of run-time types of objects. Thus, objects are instantiated by an expression of the form `new C<T1, . . . ,Tn>(. . .)`. Note that T_i need not be an invariant type, as e.g. `new Vector<Vector<+Number>>(. . .)` is permitted. Thus, implementations of generics that reify information on type parameters so as to have an explicit representation of generics at run-time—such as the type-passing compilation of LM [49, 48] or the code-expansion one of NextGen [15]—would have to carry explicit information on variance, too.

Unlike the classical mechanism of variance for classes, described in the previous section, programmers derive covariant, contravariant, and bivariant types from one generic class. Safety is achieved by restricting accesses to fields and methods, instead of constraining their declarations. For example, even when `setElementAt` is declared in `Vector`, a covariant type `Vector<+Number>` can be used in a program; the type system just forbids accessing the method `setElementAt` through it.

3.1 Subtyping

A simple interpretation of a variant parametric type is given as a set of objects, instantiated from invariant types. A type `C<+T>` can be interpreted as the set of all objects of the form `new C<S>(. . .)` where `S` is a subtype of `T`; a type `C<-T>` can be interpreted as the set of all objects of the form `new C<S>(. . .)` where `S` is a supertype of `T`; and, a type `C<*T>` can be interpreted as the union of `C<+T>` and `C<-T>`, hence including all instances of the form `new C<S>(. . .)` for any `S`. Therefore, `Vector<+Integer>` `<:` `Vector<+Number>` directly follows as an inclusion of the sets they denote. Moreover, it is easy to derive subtyping between types that differ only in variance annotations: since an invariant type would correspond to a singleton, then `Vector<Integer>` `<:` `Vector<+Integer>` and `Vector<Integer>` `<:` `Vector<-Integer>` hold, and similarly for `Vector<+Integer>` `<:` `Vector<*Integer>` and `Vector<-Integer>` `<:` `Vector<*Integer>`. In summary, Figure 1 shows the subtyping relation for the class `Vector` and type arguments `Object`, `Number`, and `Integer` (under the usual subtyping relation: `Integer <:` `Number <:` `Object`.) In general, a variant parametric type can be used as a common supertype for many different instantiations of the same generic class.

More generally, subtyping for variant parametric types is defined as follows. Suppose `C` is a generic class that takes n type arguments, and `S` and `T` (possibly with subscripts) are types.

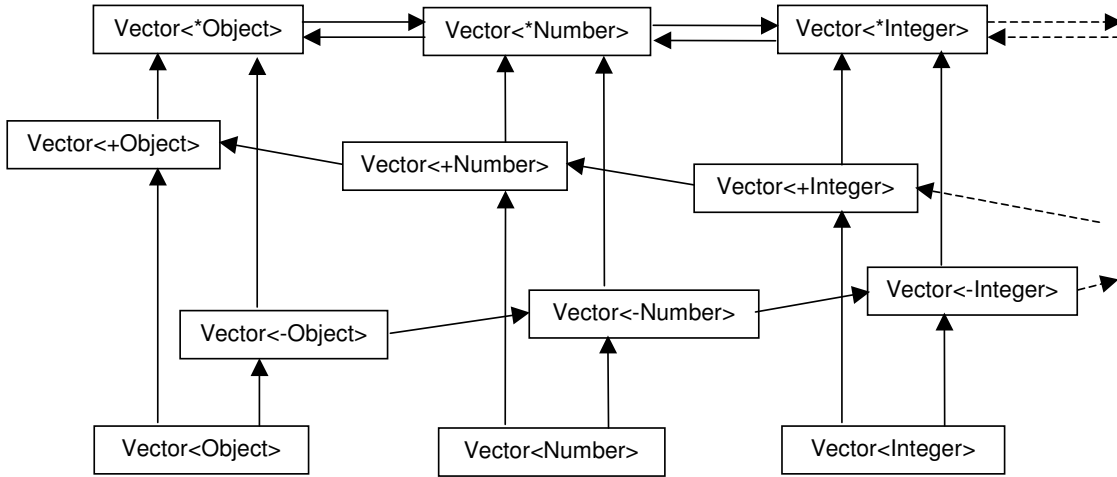


Figure 1: Subtyping graph of variant parametric types

- The following subtype relations hold that involve variant parametric types differing just in the variance annotation symbol on one type parameter:

$$\begin{aligned}
 C\langle \dots, T, \dots \rangle &<: C\langle \dots, +T, \dots \rangle \\
 C\langle \dots, T, \dots \rangle &<: C\langle \dots, -T, \dots \rangle \\
 C\langle \dots, +T, \dots \rangle &<: C\langle \dots, *T, \dots \rangle \\
 C\langle \dots, -T, \dots \rangle &<: C\langle \dots, *T, \dots \rangle
 \end{aligned}$$

More precisely, the first relation means that for any types T_1, T_2, \dots, T_n the relation $C\langle T_1, \dots, T_{i-1}, T_i, T_{i+1}, \dots, T_n \rangle <: C\langle T_1, \dots, T_{i-1}, +T_i, T_{i+1}, \dots, T_n \rangle$ holds, and similarly for the others.

- The following relations hold that involve variant parametric types differing in the instantiation of just one type parameter:

$$\begin{aligned}
 C\langle \dots, S, \dots \rangle &<: C\langle \dots, T, \dots \rangle && \text{if } S <: T \text{ and } T <: S \\
 C\langle \dots, +S, \dots \rangle &<: C\langle \dots, +T, \dots \rangle && \text{if } S <: T \\
 C\langle \dots, -S, \dots \rangle &<: C\langle \dots, -T, \dots \rangle && \text{if } T <: S \\
 C\langle \dots, *S, \dots \rangle &<: C\langle \dots, *T, \dots \rangle && \text{for any } S \text{ and } T
 \end{aligned}$$

(Note that the subtyping relation is *not* anti-symmetric: $\text{Vector}\langle *Object \rangle$ and $\text{Vector}\langle *Integer \rangle$ are subtypes of each other but not (syntactically) equal.)

- Other cases of subtyping between different instantiations of the same generic class can be obtained by the above ones through transitivity. For example, the relation $\text{Pair}\langle String, String \rangle <: \text{Pair}\langle +String, -String \rangle$ can be inferred from

$$\begin{aligned}
 \text{Pair}\langle String, String \rangle &<: \text{Pair}\langle String, -String \rangle \\
 \text{Pair}\langle String, -String \rangle &<: \text{Pair}\langle +String, -String \rangle \\
 \text{Pair}\langle +String, -String \rangle &<: \text{Pair}\langle +Object, -String \rangle.
 \end{aligned}$$

In what follows, the type argument following `*` is often omitted by simply writing e.g. `Vector<*>`: whatever comes after `*` does not have impact on subtyping, due to bivarience, and similarly for member access restriction as shown in the following. In fact, this is also the syntax we propose for an actual language extension featuring variant parametric types.

We conclude this discussion by considering inheritance-based subtyping and its relationship with variance. Subtyping between variant parametric types obtained from different generic classes can be understood by combining the above interpretation of “types as sets of instances” and usual pointwise subtyping. Suppose the following subclass of `Pair`:

```
class Twin<X> extends Pair<X,X> { ... }
```

Since `Twin<T>` is a subtype of `Pair<T,T>` by pointwise subtyping, the invariant type `Pair<T,T>` now denotes not only the set of instances of the form `new Pair<T,T>(...)`, but also those of the form `new Twin<T>(...)`. Similarly, the type `Pair<+S,+T>` includes instances of the form `new Twin<U>(...)` when `U` is a subtype of *both* `S` and `T`, because `Twin<U>` is a subtype of `Pair<U,U>` which is a subtype of `Pair<+S,+T>`: hence `Twin<+U> <: Pair<+S,+T>` when `U <: S` and `U <: T`. From the discussion above, it may seem straightforward to define a subtyping relation involving subclassing, but as we will see in Section 5, however, its actual definition is more subtle than it first appear, especially when nested parameterizations are involved.

3.2 Restrictions on Member Access

As mentioned in the previous sections, certain access restrictions have to be imposed on variant parametric types, according to variance annotations. Here, we will describe the handling of access restriction for simple cases, through the simple interpretation of variant parametric types given above.

We first begin with covariant types. Consider the type `Vector<+Number>`, which denotes the set of instances of the form `new Vector<T>(...)` where `T` is a subclass of `Number`—this set includes, for example, `new Vector<Number>(...)`, `new Vector<Integer>(...)`, or `new Vector<Double>(...)`, but not `new Vector<String>(...)`. Since element types can be *any* subtype of `Number`, we cannot safely insert anything through the type `Vector<+Number>`, prohibiting assignments to fields whose types are exactly the type parameter (which stands for the element type). For much the same reason, a method with an argument type being the type parameter (such as `setElementAt`) cannot be invoked. On the other hand, even if the exact element type of instances in `Vector<+Number>` is unknown, it is safe to allow to read elements through `Vector<+Number>` and to give them the type `Number`, because an upper bound of the element types is known to be `Number`. As a result, `Vector<+Number>` behaves as a type for vectors only with operations to read elements.

Conversely, contravariant types are write-only. Since the type `Vector<-Number>` denotes the set of instances of the form `new Vector<T>(...)` where `T` is a *superclass* of `Number`, it is safe to assign any numbers (subtypes of `Number`) to elements through this type. However, reading elements by field access or `getElementAt` results in an unknown element type and so it is prohibited.

Finally, bivariant types prohibit both reading and writing. Since `Vector<*>` is a supertype of `Vector<+Number>` and `Vector<-Number>`, only operations that are applicable to both subtypes can be allowed for `Vector<*>`: only method `size`—whose signature does not include the type variable—in this case.

Actually, some expressions are rejected due to member access restrictions, even though, in principle, they could be safely executed. If the type structure over which type variables range has the “top” type (for example, `Object` is a supertype of any reference type in Java), it is possible

to allow `getElementAt` to be invoked on `Vector<-T>` or `Vector<*>`, giving the top type to the result. Conversely, if there exists the “bottom” type and a value that belongs to it (for example, the type of `null` is a subtype of any reference type in Java), the method `setElementAt` could be invoked on `Vector<+T>` or `Vector<*>` passing that value as an argument. Similarly, if `C` is a final class, it could be allowed to invoke `setElementAt` on `Vector<+C>` because the set it denotes is the same as `Vector<C>`. Nevertheless, we believe it is more sensible to disallow all those cases so that restrictions on member access caused by variance annotations correspond to the idea of read-only/write-only collection classes—much the same as is expected for the classical approach of variance, discussed in the previous section.

Moreover, at first glance the annotated type parameter `+Object` seems to be able to substitute for the bivariance symbol `*`, since both `Vector<+Object>` and `Vector<*>` denote the same set of instances, namely, all vectors. However, we believe it is better to preserve bivariant types. Firstly, by the help of `*`, subtyping can be understood in a more syntactic, structural manner, without necessarily thinking at the set of instances each type denotes. Secondly, `*` is a concise way to signify “not being accessed,” whereas `+Object` means “being only read as objects.” Bivariance might seem useless, but it turns out to be indeed of some use as we will see in the next section. In particular, we believe that allowing exceptional subtyping rules such as `Vector<-T> <: Vector<+Object>` for any `T` would make the mechanism of variant parametric types harder to understand and sensibly employ.

So far, we have discussed only simple cases, where a type parameter of a class becomes a type of a field or a method argument/return type as it is (without being put inside `<>`). We could explain more complicated cases that involve nested types but it would get harder to think of the set of instances denoted by such types. So, we will defer a more refined view of variant parametric types to Section 5, where we give an informal correspondence to bounded existential types [14]. However, a simple view given in this section is also useful as a first approximation and indeed enough to understand most of practical cases, including the examples in the next section.

4 Basic Applications

In this section, the usefulness of variant parametric types is investigated by focusing on the very case where generic classes represent generic collections, which are one of the basic and significant applications for generics. In particular, we show how variant parametric types can widen the applicability of functionalities over collections by exploiting covariance, contravariance, bivariance, and their combinations. As a reference, we consider classes `Pair<X,Y>` (completed with methods `setSnd` and `getFst`) and `Vector<X>` defined in previous sections.

4.1 Covariance

As discussed in the previous section, the type `Vector<+T>` is a supertype of any type `Vector<S>` if `S` is a subtype of `T`, and writing any elements through this type will be prohibited for safety. Hence, the method `getElementAt` can be invoked on `Vector<+T>`, while `setElementAt` cannot. In other words, if `setElementAt` is not invoked on a variable (or a formal parameter) of type `Vector<T>`, it can be safely replaced with a covariant type `Vector<+T>`, which denotes a larger set of objects than `Vector<T>`. As a first application, consider the following method `fillFrom` for class `Vector`:

```
class Vector<X extends Object>{
    ...
    void fillFrom(Vector<+X> v, int start){
```

```

    // Here no methods with X as argument type are invoked on v
    for (int i=0;i<v.size() && i+start<this.size();i++)
        this.setElementAt(v.getElementAt(i),i+start);
} }

```

The method `fillFrom` in the class `Vector` accepts a vector `v` and inserts all its elements inside the vector receiving the invocation, starting from position `start`. Since this method accepts a vector that is meant to be only read, and whose elements are expected to be given type `X`, the formal argument can be given type `Vector<+X>`, which allows the following use:

```

Vector<Number> vn = new Vector<Number>(20);
Vector<Integer> vi = new Vector<Integer>(10); ...
Vector<Float> vf = new Vector<Float>(10); ...
vn.fillFrom(vi,0); // Permitted for Vector<Integer> <: Vector<+Number>
vn.fillFrom(vf,10); // Permitted for Vector<Float> <: Vector<+Number>

```

In the code above, the vector of numbers `vn` is filled with elements of a vector of integers `vi` and floats `vf`. In summary, the covariant parameterization structure can be exploited to widen the applicability of those methods that take a collection and simply read its elements.

Variant parametric types can also be used in nested parameterizations, as in the following method `fillFromVector`:

```

class Vector<X extends Object>{
    ...
    void fillFromVector(Vector<+Vector<+X>> vv, int pos){
        for (int i=0;i<vv.size();i++){
            Vector<+X> v = vv.getElementAt(i);
            if (pos+v.size()>=this.size()) break;
            this.fillFrom(v,pos);
            pos += v.size();
        } } }

```

The method `fillFromVector` takes a vector of vectors `vv` and puts its inner-level elements into the vector receiving the invocation. For instance, invoking `fillFromVector` on a vector of the kind `["1","2","3","4","5"]` by passing the vector `[["A","B"],["C"]]` and the position 1, would change the receiver to `["1","A","B","C","5"]`. Here, `vv` is given type `Vector<+Vector<+X>>` since the outer vector is accessed only by `getElementAt` and so are inner vectors extracted from `vv`. This type accepts indeed a large set of arguments: when `X` is instantiated to `Number`, for example, subtypes of `Vector<+Vector<+Number>>` include `Vector<Vector<Integer>>`, `Vector<Vector<Float>>`, and `Vector<Vector<+Number>>`. The first two subtypes represent vectors of vectors of a specific kind of numbers. Thus, the following code will be permitted.

```

Vector<Vector<Integer>> vvi=new Vector<Vector<Integer>>(1);
vvi.setElementAt(vi,0);
Vector<Vector<Float>> vvf=new Vector<Vector<Float>>(1);
vvf.setElementAt(vf,0);

vn.fillFromVector(vi,0);
// Permitted for Vector<Vector<Integer>> <: Vector<+Vector<+Number>>
vn.fillFromVector(vf,10);
// Permitted for Vector<Vector<Float>> <: Vector<+Vector<+Number>>

```

The third type `Vector<Vector<+Number>>` represents vectors of read-only vectors containing various kinds of numbers. So, instead of writing the code above, we could first join the vector of integers `vi` and the vector of floats `vf` into another vector of type `Vector<Vector<+Number>>`, as follows.

```
// Creating a receiver and two source vectors
Vector<Number> vn = new Vector<Number>(20);
Vector<Integer> vi = new Vector<Integer>(10); ...
Vector<Float> vf = new Vector<Float>(10); ...
...
// Joining vi and vf and then filling vn
Vector<Vector<+Number>> vvn = new Vector<Vector<+Number>>(2);
vvn.setElementAt(vi,0); // Vector<Integer> <: Vector<+Number>
vvn.setElementAt(vf,1); // Vector<Float> <: Vector<+Number>
vn.fillFromVector(vvn,0);
// Permitted for Vector<Vector<+Number>> <: Vector<+Vector<+Number>>
```

4.2 Contravariance

Contravariance is kind of “dual” of covariance. Type `Vector<-T>` is a supertype of any type `Vector<S>` if `T` is a subtype of `S`, and can be interpreted as the type of all those vectors which is safe to be filled with elements of type `T`, that is, write-only vectors of `T` elements. Hence, the invocation of method `getElementAt` on type `Vector<-T>` is forbidden, while method `setElementAt` can be invoked by passing an object of type `T`. The following method `fillTo` inserts all the elements of the receiver into the vector `v` passed as argument, starting at its index `start`.

```
class Vector<X extends Object>{
    ...
    void fillTo(Vector<-X> v, int start){
        // Here no methods with X as return type are invoked on v
        for (int i=0;i<this.size() && i+start<v.size();i++)
            v.setElementAt(this.getElementAt(i),i+start);
    }
}
```

Since the only method invoked on `v` is `setElementAt`, which expects `X` as the first argument, so `v` can be safely given type `Vector<-X>`, making it possible to write the following code:

```
Vector<Number> vn = new Vector<Number>(20);
Vector<Integer> vi = new Vector<Integer>(10);
Vector<Float> vf = new Vector<Float>(10);
vi.fillTo(vn,0); // Permitted for Vector<Number> <: Vector<-Integer>
vf.fillTo(vn,10); // Permitted for Vector<Number> <: Vector<-Float>
```

Here, `fillTo` is invoked on both `Vector<Integer>` and `Vector<Float>` vectors with an argument of type `Vector<Number>`. Similarly, `fillTo` could be invoked on a `Vector<Integer>` vector, passing elements of type `Vector<Integer>`, `Vector<Number>`, and `Vector<Object>`. In summary, the contravariant parameterization structure can be exploited to widen the applicability of those methods that take a collection and simply write its elements. Notice that accessing some information about such a collection is still permitted as far as it does not involve reading an element, e.g. the size of the vector can be retrieved by invoking `size()`.

Mixing covariance and contravariance can be useful as well. Consider the method `fillToVector` within class `Vector<X>`, which inserts elements of the receiver vector into a data structure of the

kind `Vector<Vector<X>>` provided as argument—which is a dual case with respect to the method `fillFromVector`.

```
class Vector<X extends Object>{
    ...
    void fillToVector(Vector<+Vector<-X>> vv){
        int pos=0;
        for (int i=0;i<vv.size();i++){
            Vector<-X> v=vv.elementAt(i);
            for (int j=0;j<v.size();j++){
                v.setElementAt(this.elementAt(pos++));
                if (pos>=this.size()) return;
            }
        }
    }
}
```

For instance, invoking `fillToVector` on the vector `["1","2","3"]` with the vector `[["A","B"],["C","D"]]` would change the latter to `[["1","2"],["3","D"]]`. The outer vector is just used to access inner vectors through the method `getElementAt`—hence it can be safely declared covariant—while the inner vectors are only updated through the method `setElementAt`—so they can be safely declared contravariant. Thus, the formal argument `vv` can be safely given type `Vector<+Vector<-X>>`. Similarly to `fillFromVector`, when `X` is instantiated to `Number`, we can apply this method to `Vector<Vector<Number>>`, `Vector<Vector<Objects>>` and so on.

4.3 Bivariance

As mentioned in the previous section, a bivariant vector type `Vector<*T>`, abbreviated to `Vector<*>`, is a supertype of both `Vector<+T>` and `Vector<-T>` but no methods that contain the type variable `X` in their signature can be invoked on it. So, `Vector<*>` is meant to represent a sort of “frozen” vector, from which elements can be neither read nor written.

Although the usefulness of such vectors might appear very limited, interesting applications do exist. Consider the following code:

```
Vector<Vector<*>> vv=new Vector<Vector<*>>(2);
Vector<String> vs=new Vector<String>(10);...
Vector<Integer> vi=new Vector<Integer>(15);...
vv.setElementAt(vs,0);
vv.setElementAt(vi,1);
```

Because of the bivariant parameterization, the vectors `vs` and `vi` can be joined together into a vector of type `Vector<Vector<*>>`. Such an object can be exploited e.g. by a functionality that sums the sizes of a vector of vectors, as follows:

```
int countVector(Vector<+Vector<*>> vv){
    int sz=0;
    for (int i=0;i<vv.size();i++){
        sz+=vv.elementAt(i).size();
    }
    return sz;
}
...
int i=countVector(vv); // yields 25
```

Since the inner variance annotation is `*`, inner vectors can be any kind of vector, e.g. `Vector<String>` and `Vector<Integer>`. Moreover, as usual, outer variance annotation can be safely set to `+`, for the outer vector `vv` is just iterated by successively invoking the method `getElementAt`.

Bivariance appears to be more useful in those cases where a generic class involves more than one type parameter. Consider the following method `fillFromFirst` that copies first elements of pairs in a given vector to the receiver.

```
class Vector<X>{
    ...
    void fillFromFirst(Vector<+Pair<+X,*>> vp,int start){
        for (int i=0;i<vp.size() && i+start<this.size();i++)
            this.setElementAt(vp.getElementAt(i).getFst(),i+start);
    }
}
```

As the same reason as `fillFromVector` before, `+` can be attached to `X` and `Pair`. Moreover, since method `fillFromFirst` neither read nor write the second element in each pair, the annotation symbol `*` can be used in place of it. As a result, it is permitted to invoke `fillFromFirst` on pairs of vectors where second elements of pairs are arbitrary.

```
Vector<Number> vn=new Vector<Number>(100);
Vector<Pair<Integer,String>> vpi = ...;
Vector<Pair<Float,Float>> vppi = ...;
Vector<Pair<Number,Object>> vpoi = ...;

vn.fillFromFirst(vpi,0);
vn.fillFromFirst(vppi,10);
vn.fillFromFirst(vpoi,20);
```

This example suggests an interesting application of bivariance as a mechanism providing a partial instantiation to a parametric type.

So, in summary, parameterization with bivariance can be exploited to widen the applicability of those methods that take a collection but that do not access—neither for reading nor for writing—its elements, or at least, a part of them.

5 Correspondence to Existential Types

In this section, we investigate an informal correspondence of variant parametric types to (bounded) existential types [36, 14], which are a type-theoretic basis of (partially) abstract data types (ADTs). Beginning with reviewing the standard formulation of (bounded) existential types, we give, by means of examples, a rationale of the type system for variant parametric types, based on the informal correspondence. The formal type system of the core language is formally defined in the next section.

5.1 Existential Types *à la* Mitchell and Plotkin

An (unbounded) existential type is syntactically a type of the form $\exists X.T$, with the existential quantifier on a type variable `X`. By regarding the identity of `X` as something unknown, as in existential formulas in predicate logic, existential types can be used for some sort of information hiding—encapsulation of implementation by abstract data types. In this scenario, a signature of an ADT can be represented by an existential type where `T` is the type for a set of operations on the ADT and `X` stands for the abstract type. For example, a signature of (purely functional) `Stack` would be represented as

$$\text{StackType} = \exists X.\{\text{empty}:X, \text{push}:(\text{int}*X)\rightarrow X, \text{pop}:X\rightarrow(\text{int}*X)\},$$

where $\{ \dots \}$ is a record type, $A*B$ stands for a type of pairs of A and B , and $A \rightarrow B$ for a type of functions from A to B .

A value of an existential type $\exists X.T$ is constructed by a pair of a type U and a value v of $[U/X]T$ —type T where type variable U is substituted for X . Such a pair is often written `pack [U,v] as $\exists X.T$` and U is sometimes called a witness type, since it witnesses the existence of X . From the ADT point of view, such a value is considered an implementation of an ADT, which is usually given by a pair of a concrete representation of the abstract type and an implementation of operations assuming the concrete type. So, if integer lists are to be used for implementing the stack above, one can use a record of functions

```
r = { empty = nil,
      push(int x, intlist l) = cons(x,l),
      pop(intlist l) = (car(l), cdr(l)) }
```

of type

```
{ empty: intlist,
  push: (int * intlist) → intlist,
  pop: intlist → (int * intlist) }
```

to build the following ADT package:

```
stack = pack [intlist, r] as StackType.
```

Note that the type of `r` is obtained by substituting `intlist` for X in `{empty:X, ...}` of `StackType`. The type annotation following `as` is needed since it depends on programmers' intention which part of the signature is to be abstracted away: for example, $\exists X.\{empty:intlist, \dots\}$ could be given from the same witness type and implementation.

In the original formulation, a value of an existential type can be used by an expression of the form `open p as [X,x] in b`. It unpacks a package `p`, binds the type variable X and the value variable `x` to the witness type and the implementation, respectively, and executes `b`. So, one can create an empty stack, push two integers, and pop one, by

```
open stack as [ST, s] in
  ST x = s.empty;
  x = s.push(1, x);
  x = s.push(2, x);
  return fst(s.pop(x));
```

Since the scope of `ST` is limited within the part after `in`, it does not make sense to export an expression of type `ST` (or type that includes `ST`) outside. Thus, an expression

```
open stack as [ST, s] in
  return s.empty;
```

would not be typed. As a result, it prohibits a concrete representation of a stack from being leaked.

Bounded existential types introduced by Cardelli and Wegner [14] allow existential type variables to have upper bounds: an example is $\exists X<:S.T$, which means T where X is some *subtype* of S . Bounded existential types correspond to partially abstract types: ADTs where partial information of the implementation type is available. For example, when one wants to allow abstract stacks to be regarded as integer bags, their signature can be represented by $\exists X<:intbag.\{empty:X, \dots\}$. As before, the implementation is given by a pair of a concrete type and implementation, but now, the concrete type has to be a subtype of `intbag`. Here, we assume `intbag` is a subtype of `intlist` and so packing `r` above as

```
stack' = pack [intlist, r] as  $\exists X < \text{intbag} . \{ \text{empty} : X, \dots \}$ .
```

will be permitted. Bounded existential types can be used by `open` as before; in `b`, we can use the information $X < S$, allowing a value of the abstract type X to be used as S . Thus, any stack is exported outside as an `intbag`. For example,

```
open stack' as [ST, s] in
  return s.empty;
```

will be accepted as it returns `intbag`. However, it is *not* allowed to invoke `push` with an `intbag` since `intbag < X`—which is the opposite of the assumption—does not hold.

The argument above can be summarized by the following informal typing rules:

$$\frac{\vdash U < S \quad \vdash v \in [U/X]T}{\vdash (\text{pack } [U, v] \text{ as } \exists X < S.T) \in \exists X < S.T} \quad (\text{PACK})$$

$$\frac{\vdash p \in \exists X < S.T \quad X < S, x : T \vdash b \in U \quad X \notin FV(U)}{\text{open } p \text{ as } [X, x] \text{ in } b \in U} \quad (\text{UNPACK})$$

Note that the side condition requires X not to be a free variable of U ; otherwise the hidden type X would escape the abstraction as discussed above.

Furthermore, we can define subtyping between bounded existential types as follows.

$$\frac{\vdash S_1 < S_2 \quad X < S_1 \vdash T_1 < T_2}{\vdash \exists X < S_1.T_1 < \exists X < S_2.T_2}$$

This subtyping rule allows to relax the upper bound of the existential type variable and to promote the type of the implementation. For example, the following subtype relations would hold:

```
 $\exists X < \text{intbag} . \{ \text{empty} : X, \text{push} : \dots, \text{pop} : \dots \} < \exists X < \text{Top} . \{ \text{empty} : X, \text{push} : \dots, \text{pop} : \dots \}$   

 $\exists X < \text{intbag} . \{ \text{empty} : X, \text{push} : \dots, \text{pop} : \dots \} < \exists X < \text{intbag} . \{ \text{empty} : X, \text{push} : \dots \}$ 
```

Remark: Here, experts will notice that this rule is known as one for the full variant of System F_{\leq} , in which subtyping is undecidable [38]. Subtyping of variant parametric types, though, is easily shown to be decidable, as they correspond to rather restricted forms of bounded existential types.

5.2 Interpreting Variant Parametric Types as Existential Types

Variant parametric types can be explained in terms of existential types above. Roughly speaking, a covariant type $C < +T >$ would correspond to the bounded existential type $\exists X < T . C < X >$. Contravariant types will require slight generalization: the type $C < -T >$ would correspond to the bounded existential type $\exists X > T . C < X >$, where the abstract type X has a *lower* bound rather than an upper bound. A bivariate type $C < *T >$ —abbreviated to $C < * >$ —would simply correspond to the unbounded existential type $\exists X . C < X >$. This idea extends naturally to a parametric type with more than one parameter. $D < +S, T >$ would correspond to $\exists X < S . D < X, T >$ and $D < +S, -T >$ to $\exists X < S . \exists Y > T . D < X, Y >$. For a nested parametric type, the existential quantifier will appear in bounds: for example, $\text{Vector} < +\text{Vector} < +\text{Nm} > >$ would correspond to $\exists X < : (\exists Y < : \text{Nm} . \text{Vector} < Y >) . \text{Vector} < X >$ since $\text{Vector} < +T >$ would be $\exists X < : T . \text{Vector} < X >$ and here T is again a type of the form $\text{Vector} < +T' >$. (In what follows, `Number` is often abbreviated to `Nm` for conciseness.)

Bearing the informal correspondence above in mind, we can justify the design decisions made in Section 3 and explain them in more details.

5.2.1 Interpretation of Variance-Based Subtyping

Covariant and contravariant subtyping are directly attributed to the subtyping rule for existential types above. $C\langle+S\rangle \prec C\langle+T\rangle$ when $S \prec T$ corresponds to $\exists X\langle:S.C\langle X\rangle \prec \exists X\langle:T.C\langle X\rangle$ assuming $S \prec T$, and similarly for covariant types. Subtyping that changes variance annotations ($C\langle+T\rangle \prec C\langle* T\rangle$ and $C\langle-T\rangle \prec C\langle* T\rangle$) can be considered as subtyping between bounded and unbounded existential types: $\exists X\langle:S.T \prec \exists X.T$ could be allowed since it is just a special kind of relaxing the bounds from T to nothing. Finally, subtyping $C\langle T\rangle \prec C\langle+T\rangle$ or $C\langle T\rangle \prec C\langle-T\rangle$ that introduces a co- or contra-variant annotation can be explained in terms of (implicit) packing operation: when a type of an expression is changed from $C\langle T\rangle$ to $C\langle+T\rangle$, the expression is packed with the witness type T , yielding $\exists X\langle:T.C\langle X\rangle$. For example, when v of `Vector<Nm>` is passed to where an argument of `Vector<+Nm>` is expected, it would be represented as:

```
pack [Nm,v] as  $\exists X\langle:Nm.Vector\langle X\rangle$ 
```

The discussion above can easily extend to a class with more than one type parameter.

5.2.2 Interpretation of Access Restriction Rules

We exploit the connection also to give correct typing rules for field access and method invocation. Since a variant parametric type corresponds to an existential type, an expression of a variant parametric type, conceptually, has to be opened first. For example, suppose x is given type $C\langle+T\rangle$ and consider an expression $x.m(e)$. Then, this expression would correspond to `open x as [X,y] in y.m(e)`, which first opens x and then invokes the method. In the body of `open`, the type variable X is assumed to have an upper bound T and y to have type $C\langle X\rangle$; since y is an invariant type, the standard typing rules can be applied to $y.m(e)$. Finally, the type of the whole `open` expression will be calculated from the type S of its body $y.m(e)$. Since S may include X , it may be the case that S itself cannot be the type of the whole expression (recall the side condition of the rule `UNPACK`). Thus, we have to find a *X-free supertype* of S . As we will discuss in detail below, there are some subtleties about this calculation.

We will explain typing method invocations in more detail, using a parametric class `Pair`:

```
class Pair<X extends Object, Y extends Object> {
  X fst; Y snd;
  Pair(X fst,Y snd){ this.fst=fst; this.snd=snd; }

  void setFst(X x){ this.fst=x; }
  void setSnd(Y y){ this.snd=y; }
  void copyFst(Pair<X,*> p) { setFst(p.getFst()); }

  ...
}
```

Furthermore, we assume x is given type `Pair<+Nm,-Nm>`, which corresponds to $\exists X\langle:Nm.\exists Y\langle:Nm.Pair\langle X,Y\rangle$. Then, for example, `x.setFst(e)` corresponds to `open x as [Z,W,y] in y.setFst(e)`.² Inside `open`, y stands for the object of type `Pair<Z,W>` under the assumptions $Z\langle:Nm$ and $W\langle:Nm$; method/field types can be easily obtained by simply replacing the type parameters of a class with the actual type arguments. For example, the argument type of `setFst` is Z , that of `setSnd` is W . Now, it turns out that `open x as [Z,W,y] in y.setFst(e)`

²We often abbreviate a sequence of `open` as one that binds multiple type variables at once.

cannot be well typed for any expression e —the argument type of `setFst` is Z , which is assumed to be an unknown subtype of Nm , but the type of e cannot be a subtype of Z . On the other hand, `setSnd` can be invoked with an argument of type Nm (or its subtype) because if T is a subtype of Nm , it is the case that $T \prec: Nm \prec: W$. This is why `+` results in protecting some fields from being written, while `-` allows writing.

A more complex case, where an argument type includes type parameters of the class inside angle brackets, as in `copyFst`, can also be explained in the same way. Suppose x is `Pair<+Nm, -Nm>` as before and, for example, consider the expression `x.copyFst(new Pair<Integer,Float>(...))`. Then, just as before, the argument type of `copyFst` would be `Pair<Z,*>` in which Z is assumed to be a subtype of Nm . Thus, this expression is not allowed because `Pair<Integer,Float>` is not a subtype of `Pair<Z,*>`. In fact, it *should not* be allowed because x may be bound to an instance of `Pair<Float,Object>` and executing the expression above will assign an `Integer` to the field for `Float`. This example shows that a method argument type cannot be obtained by naively substituting for X the type argument `+Nm` together with a variance annotation; in this case, naive substitution would lead to a wrong argument type `Pair<+Nm,*>`, a supertype of `Pair<Integer,Float>`. In the next section, where the type system is formalized, we formalize the operation *open* to obtain an invariant type and type bounds from a variant parametric type.

Now, we turn our attention to how the return type of a method is calculated. Suppose the class `Pair` has some more methods as shown below.

```
class Pair<X extends Object, Y extends Object> {
    ...
    X getFst(){ return this.fst; }
    Y getSnd(){ return this.snd; }

    Pair<Y,X> reverse() {
        return new Pair<Y,X>(this.getSnd(), this.getFst());
    }
    Pair<+Y,X> reverse2() {
        return new Pair<Y,X>(this.getSnd(), this.getFst());
    }
    Pair<Pair<X,Y>, Pair<X,Y>> dup () {
        return new Pair<Pair<X,Y>,Pair<X,Y>>(this, this);
    }
}
```

By the same argument, when a receiver is of type `Pair<+Nm, -Nm>`, the result type of `getFst` is Z , that of `getSnd` is W , that of `reverse` is `Pair<W,Z>`, and that of `dup` is `Pair<Pair<Z,W>,Pair<Z,W>>`, under the assumption $Z \prec: Nm$ and $W \succ: Nm$.

As briefly mentioned above, when the method return type T includes a type variable Z introduced by *open*, a Z -free supertype of T has to be obtained in some way. For example, a Z -free subtype of Z can be obtained from its upper bound Nm , and so the type of `x.getFst()` will be Nm , as expected. On the other hand, it may not be possible to obtain such a supertype, and in that case, the expression is not typed. For example, `x.getSnd` is not typeable because the return type W has only a lower bound and no supertype without W . This is why `-` results in protecting some fields from being read, while `+` allows reading.

Similarly, the return type of `x.reverse()` will be `Pair<-Nm,+Nm>`, since the return type obtained from `Pair<Z,W>` is `Pair<W,Z>`, which can be promoted to a supertype `Pair<-Nm,+Nm>` by exploiting the fact that Z is a subtype of Nm and W is a supertype of Nm . When a variance annotation is attached to the method definition as in `reverse2()`, we have to take care of them by calculating the upper

bound of the annotation in the definition and one from the bound of the type variable. So, the return type of `x.reverse2()` will be `Pair<*,+Nm>` (`*` is the upper bound of `-` and `+`).

5.2.3 Subtleties in Obtaining Return Types

It looks as if the type arguments and variances `+Nm` and `-Nm` were respectively substituted for `X` and `Y` (with a twist on variance annotations), but this naive view is not correct as we will see in the next example `x.dup()`. The return type of this method invocation is `Pair<Pair<Z,W>,Pair<Z,W>>`. However, the type we obtain by the naive substitution—that is, `Pair<Pair<+Nm,-Nm>,Pair<+Nm,-Nm>>`—is *not* a supertype of `Pair<Pair<Z,W>,Pair<Z,W>>`! It is because the two inner occurrences of `Pair` are invariant. So, as long as `Pair<Z,W>` and `Pair<+Nm,-Nm>` are different, those two types are not in the subtype relation. If it were a supertype, another pair could be assigned to the outer pair, making the two first elements of the inner pairs have different types. A correct supertype is a covariant type `Pair<+Pair<+Nm,-Nm>,+Pair<+Nm,-Nm>>`, obtained by attaching `+` to everywhere the type argument is changed by promotion. So, subtlety here is that naive substitution of an actual type argument with its variance does not always work to obtain a return type.

In fact, there is yet another subtlety in calculating a return type: given a type and type variables with bounds, there may be two (or more) *incomparable* supertypes without those type variables. This property is unpleasant from the point of view of a bottom-up typecheck algorithm. For example, suppose the return type of a certain method is `C<C<-X>>` under the assumption `X<:Nm`. Then, both `C<+C<*>>` and `C<-C<-Nm>>` are supertypes of `C<C<-X>>` but neither is a supertype of the other. Since it depends on the context of the method invocation which return type is the expected one, a naive typecheck algorithm would have to try both cases, hampering efficient and modular typechecking.

We avoid this combinatorial explosion problem by mechanically choosing one particular supertype from possible ones, although some programs that looks reasonable in terms of the correspondence developed here may be rejected. The strategy we adopted is actually the same as what we have seen: the bounds with variance annotations are substituted for the type variables but `+` will be attached everywhere substitution causes any changes. In the example above, we obtain `C<+C<*>>` since a `X`-free supertype of `C<-X>` is `C<*>` and then `+` is attached before `C<*>`. Thus, unfortunately, the context that expects `C<-C<-Nm>>` is rejected. We believe our decision is practical because (1) it is easy to understand for those who do not (want to) understand the underlying correspondence to existential types; and (2) the strategy always works and yields a supertype, even if type variables appear in deeper positions. For example, `C1<C2<C3<X,Y>>>` with `X<:S` and `Y:>T` has `C1<+C2<+C3<+S,-T>>>` as a supertype without `X` and `Y`; `C1<-C2<C3<X,Y>>>` with `X<:S` and `Y:>T` has `C1<*>`, which is equivalent to `C1<+C2<+C3<+S,-T>>>`, as a supertype without `X` and `T` (recall that `*` is an upper bound of `+` and `-`); and so on. In the next section, we will formalize this operation for obtaining an abstract-type-free supertype as the operation called *close*. A similar operation is found in a type system for bounded existential types with minimal typing property [20], where it was introduced to omit a type annotation for the open expression.

5.2.4 Variance and Inheritance-Based Subtyping

The operations of open and close are also used for deriving inheritance-based subtyping for variant parametric types. Suppose we declare two subclasses of `Pair`.

```
class Twin<X extends Object> extends Pair<X,X> { ... }
class PP<X extends Object, Y extends Object>
  extends Pair<Pair<X,Y>, Pair<X,Y>> { ... }
```

As in GJ, inheritance-based subtyping for invariant types is simple. A supertype is obtained by substituting the type arguments for type variables in the type after `extends`: for example,

```
Twin<Integer> <: Pair<Integer,Integer>
```

and

```
PP<Integer,String> <: Pair<Pair<Integer,String>,Pair<Integer,String>>.
```

Subtyping for non-invariant types involves open and close, similarly to field and method accesses. For example, `Twin<+Nm>` is a subtype of `Pair<+Nm,+Nm>` because the open operation on `Twin<+Nm>` introduces `Twin<Z>` with `Z<:Nm`, a supertype of `Twin<Z>` is `Pair<Z,Z>`—obtained by substitution of `Z` for `X`—and it closes to `Pair<+Nm,+Nm>`. Similarly, a supertype of `PP<+Nm,-Nm>` is `Pair<+Pair<+Nm,-Nm>,+Pair<+Nm,-Nm>>`, obtained by closing `Pair<Pair<Z,W>,Pair<Z,W>>` (note that `+` before the inner occurrences of `Pair`).

6 Core Calculus for Variant Parametric Types

In this section, we introduce a calculus for class-based object-oriented languages with variant parametric types to prove that the core part of the type system is sound. Our calculus is considered a variant of Featherweight GJ (FGJ for short) by Igarashi, Pierce, and Wadler [23], originally proposed to formally investigate properties of the type system and compilation scheme of GJ [5]. Like FGJ, our extended calculus is functional and supports only minimal features including top-level parametric classes with variant parametric types, object instantiation, field access, method invocation, and typecasts.

6.1 Syntax

The metavariables `A`, `B`, `C`, `D`, and `E` range over class names; `S`, `T`, `U`, and `V` range over types; `X`, `Y`, and `Z` range type variables; `N`, `P`, and `Q` range over variant types; `L` ranges over class declarations; `M` ranges over method declarations; `v` and `w` range over variance annotations; `f` and `g` range over field names; `m` ranges over method names; `x` ranges over variables; and `e` and `d` range over expressions. The abstract syntax of types, class declarations, method declarations, and expressions is given in Figure 2.

We write \bar{f} as shorthand for a possibly empty sequence f_1, \dots, f_n (and similarly for \bar{c} , \bar{x} , \bar{e} , etc.) and write \bar{M} as shorthand for $M_1 \dots M_n$ (with no commas). We write the empty sequence as \bullet and denote concatenation of sequences using a comma. The length of a sequence \bar{x} is written $|\bar{x}|$. We abbreviate operations on pairs of sequences in the obvious way, writing “ $\bar{C} \bar{f}$ ” as shorthand for “ $C_1 f_1, \dots, C_n f_n$ ” and “ $\bar{C} \bar{f};$ ” as shorthand for “ $C_1 f_1; \dots C_n f_n;$ ”, “ $\langle \bar{X} \triangleleft \bar{N} \rangle$ ” as shorthand for “ $\langle X_1 \triangleleft N_1, \dots, X_n \triangleleft N_n \rangle$ ”, and “ $C \langle \bar{v} \bar{T} \rangle$ ” for “ $C \langle v_1 T_1, \dots, v_n T_n \rangle$ ”. Sequences of field declarations, parameter names, type variables, and method declarations are assumed to contain no duplicate names. The empty brackets $\langle \rangle$ are often omitted for conciseness. We denote by $m \notin \bar{M}$ that a method of the name `m` is not included in \bar{M} . We introduce another variance annotation `o` to denote invariance, which would be regarded as default and omitted in the surface language. Then, a partial order \leq on variance annotations is defined: formally, \leq is the least partial order satisfying $o \leq + \leq * \leq - \leq *$. We write $v_1 \vee v_2$ for the least upper bound of v_1 and v_2 . If every v_i is in a variant type $C \langle \bar{v} \bar{T} \rangle$, we call it an *invariant type* and abbreviates to $C \langle \bar{T} \rangle$.

$N ::= C\langle\bar{v}\bar{T}\rangle$	variant parametric types
$T ::= X \mid N$	types
$v ::= o \mid + \mid - \mid *$	variance annotations
$L ::= \text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{S}\rangle\rangle \{ \bar{T} \bar{f}; \bar{M} \}$	class definitions
$M ::= \langle\bar{Y}\langle\bar{P}\rangle T m(\bar{T} \bar{x})\{ \text{return } e; \}$	method definitions
$e ::= x$	variables
$e.f$	field access
$e.m(\bar{e})$	method invocation
$\text{new } C\langle\bar{T}\rangle(\bar{e})$	object instantiation
$(T)e$	typecasts

Figure 2: Syntax

A class declaration consists of its name (`class C`), type parameters (\bar{X}) with their (upper) bounds (\bar{N}), fields ($\bar{T} \bar{f}$), and methods (\bar{M})³; moreover, every class must explicitly declare its supertype $D\langle\bar{S}\rangle$ with \triangleleft (read **extends**) even if it is `Object`. Note that only an invariant type is allowed as a supertype, just as in object instantiation. Since our language supports F-bounded polymorphism [11], the bounds \bar{N} of type variables \bar{X} can contain \bar{X} in them. A method definition can be parameterized by type variables \bar{Y} with bounds \bar{P} . A body of a method just returns an expression, which is either a variable, field access, method invocation, object instantiation, or typecasts. As we have already mentioned, the type used for an instantiation must be an invariant type, hence $C\langle\bar{T}\rangle$. For the sake of generality, we allow the target type T of a typecast expression $(T)e$ to be any type, including a type variable. Thus, we will need an implementation technique where instantiation of type parameters are kept at run-time, such as the framework of LM [49, 48]. Should it be implemented with the type-erasure technique as in GJ, T has to be a non-variable type and a special care will be needed for downcasts (see [5] for more details). We treat `this` in method bodies as a variable, rather than a keyword, and so require no special syntax. As we will see later, the typing rules prohibit `this` from appearing as a method parameter name.

A class table CT is a mapping from class names C to class declarations L ; a *program* is a pair (CT, e) of a class table and an expression. `Object` is treated specially in every program: the definition of `Object` class never appears in the class table and the auxiliary functions to look up field and method declarations in the class table are equipped with special cases for `Object` that return the empty sequence of fields and the empty set of methods. (As we will see later, method lookup functions take a pair of class and method names as arguments; the case for `Object` is just undefined.) To lighten the notation in what follows, we always assume a *fixed* class table CT .

The given class table is assumed to satisfy some sanity conditions: (1) $CT(C) = \text{class } C\dots$ for every $C \in \text{dom}(CT)$; (2) `Object` $\notin \text{dom}(CT)$; (3) for every class name C (except `Object`) appearing anywhere in CT , we have $C \in \text{dom}(CT)$; and (4) there are no cycles in the transitive closure of the relation between class names induced by \triangleleft clauses in CT . By the condition (1), we can identify a class table with a sequence of class declarations in an obvious way; so, in the rules below, we just write `class C ...` to state $CT(C) = \text{class } C \dots$, for conciseness.

³We assume that each class has a trivial constructor that takes the initial (and also final) values of each fields of the class and assigns them to the corresponding fields. In FGJ, such constructors have to be declared explicitly, in order to retain compatibility with GJ. We omit them because they play no other significant role.

Field lookup:	$fields(\mathbf{Object}) = \bullet$	(F-OBJECT)
	$\frac{\text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{U}\rangle \{ \bar{S} \bar{f}; \bar{M} \} \quad fields([\bar{T}/\bar{X}]D\langle\bar{U}\rangle) = \bar{V} \bar{g}}{fields(C\langle\bar{T}\rangle) = \bar{V} \bar{g}, [\bar{T}/\bar{X}]\bar{S} \bar{f}}$	(F-CLASS)
Method type lookup:		
	$\frac{\text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{S}\rangle \{ \dots \bar{M} \} \quad \langle\bar{Y}\langle\bar{P}\rangle U_0 \ m(\bar{U} \ \bar{x}) \{ \text{return } e; \} \in \bar{M}}{mtype(m, C\langle\bar{T}\rangle) = [\bar{T}/\bar{X}](\langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle\rightarrow U_0)}$	(MT-CLASS)
	$\frac{\text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{S}\rangle \{ \dots \bar{M} \} \quad m \notin \bar{M} \quad mtype(m, [\bar{T}/\bar{X}]D\langle\bar{S}\rangle) = \langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle\rightarrow U_0}{mtype(m, C\langle\bar{T}\rangle) = \langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle\rightarrow U_0}$	(MT-SUPER)
Method body lookup:		
	$\frac{\text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{S}\rangle \{ \dots \bar{M} \} \quad \langle\bar{Y}\langle\bar{P}\rangle U_0 \ m(\bar{U} \ \bar{x}) \{ \text{return } e_0; \} \in \bar{M}}{mbody(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle) = \bar{x}. [\bar{V}/\bar{Y}][\bar{T}/\bar{X}]e_0}$	(MB-CLASS)
	$\frac{\text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{S}\rangle \{ \dots \bar{M} \} \quad m \notin \bar{M} \quad mbody(m\langle\bar{V}\rangle, [\bar{T}/\bar{X}]D\langle\bar{S}\rangle) = \bar{x}. e_0}{mbody(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle) = \bar{x}. e_0}$	(MB-SUPER)

Figure 3: Field/Method Look-up Functions

6.2 Type System

For the typing and reduction rules, we need a few auxiliary definitions to look up the field or method types and the method body of an invariant type, which are shown in Figure 3. As we discussed in the previous section, we never attempt to ask the field or method types of non-invariant types.

The fields of an invariant type $C\langle\bar{T}\rangle$, written $fields(C\langle\bar{T}\rangle)$, are a sequence of corresponding types and field names, $\bar{S} \bar{f}$. In what follows, we use the notation $[\bar{T}/\bar{X}]$ for a substitution of T_i for X_i . The type of the method invocation m at an invariant type $C\langle\bar{T}\rangle$, written $mtype(m, C\langle\bar{T}\rangle)$, returns a signature of form $\langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle\rightarrow U_0$ where \bar{Y} , \bar{P} , \bar{U} , U_0 are type parameters, their upper bounds, argument types, and a result type, respectively. Here, \bar{Y} are bound in \bar{P} , \bar{U} and U_0 and we allow implicit α -conversions on type parameters in a signature.

The body of the method invocation m with type arguments \bar{V} at an invariant type $C\langle\bar{T}\rangle$, written $mbody(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle)$, is a pair, written $\bar{x}. e$, of a sequence of parameters \bar{x} and an expression e . (Note that the functions $mtype(m, C\langle\bar{T}\rangle)$ and $mbody(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle)$ are both partial functions: since \mathbf{Object} is assumed to have no methods, both $mtype(m, \mathbf{Object})$ and $mbody(m\langle\bar{V}\rangle, \mathbf{Object})$ are undefined.)

A *type environment* Δ is a finite mapping from type variables to pairs of a variance annotation except \circ (that is, either $+$, $-$, or $*$) and a type. We write $dom(\Delta)$ for the domain of Δ . When $X \notin dom(\Delta)$, we write $\Delta, X : (v, T)$ for the type environment Δ' such that $dom(\Delta') = dom(\Delta) \cup \{X\}$ and $\Delta'(X) = (v, T)$ and $\Delta'(Y) = \Delta(Y)$ if $X \neq Y$. We often write $X\langle:T$ for $X : (+, T)$ and $X\langle:T$ for $X : (-, T)$. When $\Delta(X) = (v, N)$ for any $X \in dom(\Delta)$ (i.e., all the bounds are nonvariable types), we

Open:	$\Delta \vdash T \uparrow^\emptyset T$	(O-REFL)
	$\frac{\Delta \vdash S \uparrow^{\Delta_1} T \quad \Delta, \Delta_1 \vdash T \uparrow^{\Delta_2} U}{\Delta \vdash S \uparrow^{\Delta_1, \Delta_2} U}$	(O-TRANS)
	$\frac{X \text{ fresh for } \Delta, C\langle \bar{v}_1 \bar{T}_1, vT, \bar{v}_2 \bar{T}_2 \rangle \quad v \neq o}{\Delta \vdash C\langle \bar{v}_1 \bar{T}_1, vT, \bar{v}_2 \bar{T}_2 \rangle \uparrow^{X:(v,T)} C\langle \bar{v}_1 \bar{T}_1, oX, \bar{v}_2 \bar{T}_2 \rangle}$	(O-CLASS)
Close:	$\frac{\Delta(X) = (+, T)}{X \downarrow_\Delta T}$	(C-PROM)
	$\frac{X \notin \text{dom}(\Delta)}{X \downarrow_\Delta X}$	(C-TVAR)
	$\frac{(w_i, T_i') = \begin{cases} (v_i, T_i) & \text{if } T_i \downarrow_\Delta T_i \\ (v_i \vee +, U_i) & \text{if } T_i \downarrow_\Delta U_i \text{ and } T_i \neq U_i \\ (v_i \vee v_i', U_i) & \text{if } T_i = X \text{ and } \Delta(X) = (v_i', U_i) \end{cases}}{C\langle \bar{v} \bar{T} \rangle \downarrow_\Delta C\langle \bar{w} \bar{T}' \rangle}$	(C-CLASS)

Figure 4: Open and Close

Subtyping:

$$\Delta \vdash T <: T \quad (\text{S-REFL})$$

$$\frac{\Delta \vdash S <: T \quad \Delta \vdash T <: U}{\Delta \vdash S <: U} \quad (\text{S-TRANS})$$

$$\frac{\Delta(X) = (+, T)}{\Delta \vdash X <: T} \quad (\text{S-UBOUND})$$

$$\frac{\Delta(X) = (-, T)}{\Delta \vdash T <: X} \quad (\text{S-LBOUND})$$

$$\frac{\text{class } C \langle \bar{X} \langle \bar{N} \rangle \langle D \langle \bar{S} \rangle \{ \dots \} \quad \Delta \vdash C \langle \bar{vT} \rangle \uparrow^{\Delta'} C \langle \bar{U} \rangle \quad ([\bar{U}/\bar{X}]D \langle \bar{S} \rangle) \downarrow_{\Delta'} T}{\Delta \vdash C \langle \bar{vT} \rangle <: T} \quad (\text{S-CLASS})$$

$$\frac{\bar{v} \leq \bar{w} \quad \text{if } w_i \leq -, \text{ then } \Delta \vdash T_i <: S_i \quad \text{if } w_i \leq +, \text{ then } \Delta \vdash S_i <: T_i}{\Delta \vdash C \langle \bar{vS} \rangle <: C \langle \bar{wT} \rangle} \quad (\text{S-VAR})$$

Well-formed Types:

$$\Delta \vdash \text{Object ok} \quad (\text{WF-OBJECT})$$

$$\frac{X \in \text{dom}(\Delta)}{\Delta \vdash X \text{ ok}} \quad (\text{WF-TVAR})$$

$$\frac{\text{class } C \langle \bar{X} \langle \bar{N} \rangle \langle D \langle \bar{S} \rangle \{ \dots \} \quad \Delta \vdash \bar{T} \text{ ok} \quad \Delta \vdash \bar{T} <: [\bar{T}/\bar{X}]\bar{N}}{\Delta \vdash C \langle \bar{vT} \rangle \text{ ok}} \quad (\text{WF-CLASS})$$

Figure 5: Subtyping and Well-formed Types

say Δ has non-variable bounds.

The type system consists of five forms of judgments: $\Delta \vdash N \uparrow^{\Delta'} C \langle \bar{T} \rangle$ for opening a variant parametric type to an invariant type; $S \downarrow_{\Delta} T$ for closing a type with some free type variables constrained in Δ to a variant parametric type without them; $\Delta \vdash S <: T$ for subtyping; $\Delta \vdash T \text{ ok}$ for type well-formedness; and $\Delta; \Gamma \vdash e \in T$ for typing, where Γ , called an *environment*, is a finite mapping from variables to types, written $\bar{x}:\bar{T}$. We abbreviate a sequence of judgments, writing $\Gamma \vdash \bar{S} <: \bar{T}$ as shorthand for $\Gamma \vdash S_1 <: T_1, \dots, \Gamma \vdash S_n <: T_n$, $\Gamma \vdash \bar{T} \text{ ok}$ as shorthand for $\Gamma \vdash T_1 \text{ ok}, \dots, \Gamma \vdash T_n \text{ ok}$, and $\Delta; \Gamma \vdash \bar{e} \in \bar{T}$ as shorthand for $\Delta; \Gamma \vdash e_1 \in T_1, \dots, \Delta; \Gamma \vdash e_n \in T_n$.

Open and Close

As already mentioned, variant parametric types are essentially bounded existential types in disguised forms. So any operations on variant parametric types have to “open” the existential type first and the type of the result has to be “closed” in case it involves the abstract type variables. A judgment of the open operation $\Delta \vdash N \uparrow^{\Delta'} P$ is read “under Δ , N is opened to P and constraints Δ ” and that of the close operation $N \downarrow_{\Delta} P$ is read “ N with abstract types in Δ closes to P .” The rules to derive those judgments are shown in Figure 4.

The open operation introduces fresh type variables to represent abstract types and replace non-invariant type arguments with the type variables: for example, $\text{List} \langle +\text{Integer} \rangle$ is opened to $\text{List} \langle X \rangle$ with the constraint $X <: \text{Integer}$, written $\vdash \text{List} \langle +\text{Integer} \rangle \uparrow^{X <: \text{Integer}} \text{List} \langle X \rangle$. The close operation computes a minimal supertype without mentioning the abstract types. The first rule means that, if X ’s upper bound is known, X can be promoted to its bound. (When $\Delta(X) = (-, T)$, on the other hand, it cannot be promoted since an upper bound is unknown.) The second rule means that a type variable not bound in Δ remains the same. The third rule is explained as follows. Basically, in order to close $C \langle \bar{v}\bar{T} \rangle$, the type arguments \bar{T} have to be closed to \bar{T}' first. If T_i is equal to T'_i , which means none of the abstract types occurs in T_i , the type argument and its variance annotation remain the same; on the other hand, when T_i is closed to a proper supertype T'_i (i.e. $T_i \neq T'_i$), the resulting type must be covariant in that argument, thus the least upper bound of $+$ and v_i is attached. For example, under $X <: \text{Integer}$, X closes to Integer , $\text{List} \langle X \rangle$ to $\text{List} \langle +\text{Integer} \rangle$, and $\text{List} \langle \text{List} \langle X \rangle \rangle$ to $\text{List} \langle +\text{List} \langle +\text{Integer} \rangle \rangle$ (not to $\text{List} \langle \text{List} \langle +\text{Integer} \rangle \rangle$). An exception is the case where a type argument is a type variable and bounded by $-$ or $*$; the type variable itself does not close but the whole type can be closed by substituting its bound for the type variable. The least upper bound of variance annotations is attached because the resulting type must have both of their properties. For example, under $X >: \text{Integer}$, $\text{List} \langle X \rangle$ closes to $\text{List} \langle -\text{Integer} \rangle$, and $\text{List} \langle +X \rangle$ to $\text{List} \langle *\text{Integer} \rangle$.

Subtyping

A judgment for subtyping $\Delta \vdash S <: T$ is read “ S is a subtype of T under Δ .” As usual, the subtyping relation is reflexive and transitive. When an upper or lower bound of a type variable is recorded in Δ , the type variable is a subtype or supertype of the bound, respectively (S-UBOUND and S-LBOUND). The rule S-CLASS takes care of inheritance-based pointwise subtyping, described in the previous section. When $C \langle \bar{X} \rangle$ is declared to extend another type $D \langle \bar{S} \rangle$, any (invariant) instantiation $C \langle \bar{T} \rangle$ is a subtype of $D \langle [\bar{T}/\bar{X}]\bar{S} \rangle$. A supertype of a non-invariant type is obtained by opening it and closing the supertype of the opened type. Finally, the rule S-VAR deals with variance. The first conditional premise means that, if a variance annotation v_i for X_i is either contravariant or invariant, the corresponding type arguments S_i and T_i must satisfy $\Delta \vdash T_i <: S_i$; similarly for the second one.

Type Well-formedness

A judgment for type well-formedness is of the form $\Delta \vdash T \text{ ok}$, read “ T is a well-formed type under Δ ”. The rules for type well-formedness are straightforward: (1) `Object` is always well formed; (2) a type variable is well formed if it is in the domain of Δ ; and (3) a variant parametric type $C\langle\bar{v}\bar{T}\rangle$ is well-formed if the type arguments \bar{T} are lower than their bounds, respectively. Note that variance annotations can be any.

Typing

The typing rules for expressions are syntax directed, with one rule for each form of expression (except for casts), shown in Figure 6. In what follows, we use $bound_{\Delta}(T)$ defined by: $bound_{\Delta}(X) = bound_{\Delta}(S)$ if $\Delta(X) = (+, S)$ and $bound_{\Delta}(N) = N$. Most rules are straightforward. When a field or method is accessed, the receiver type is opened and the result type is closed. The typing rule for method invocations check that the actual type arguments \bar{V} satisfy the bounds \bar{P} and that the type of each actual parameter is a subtype of the corresponding formal. Since the opened receiver type $C\langle\bar{T}\rangle$ and its method type may include abstract types recorded in Δ' , type arguments and argument types are compared under the type environment Δ, Δ' . The rule T-SCAST is called a stupid cast rule; although stupid casts are disallowed in Java proper, it is needed to prove type soundness via subject reduction—see Igarashi, Pierce, and Wadler [23] for more details.

Typing for methods requires the auxiliary predicate *override* to check correct method overriding. $override(m, N, \langle\bar{V}\langle\bar{P}\rangle\bar{T}\rangle \rightarrow T_0)$ holds if and only if a method of the same name m is defined in the superclass N and has the same signature modulo α -conversion. (It would be safe to extend the rule so that the result type can be overridden covariantly, as allowed in GJ). The method body should be given a subtype of the declared result type under the assumption that the formal parameters are given the declared types and `this` is given type $C\langle\bar{X}\rangle$. The environment prohibits `this` from occurring as a parameter name since name duplication in the domain of an environment is not allowed. Finally, a class declaration is well typed if all the methods are well typed.

6.3 Operational Semantics

The reduction relation is of the form $e \longrightarrow e'$, read “expression e reduces to expression e' in one step.” We write \longrightarrow^* for the reflexive and transitive closure of \longrightarrow . The reduction rules are given in Figure 7. There are three computation rules, one for field access, one for method invocation, and one for typecasts. Field access $\text{new } C\langle\bar{T}\rangle(\bar{e}) . f_i$ looks up and obtains field names \bar{f} of $C\langle\bar{T}\rangle$ with $fields(C\langle\bar{T}\rangle)$; then it reduces to the constructor argument e_i of the corresponding position. Method invocation $\text{new } C\langle\bar{T}\rangle(\bar{e}) . m\langle\bar{V}\rangle(\bar{d})$ first looks up $mbody(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle)$ and obtains a pair of a sequence of formal arguments \bar{x} and the method body; then, it reduces to the method body in which \bar{x} are replaced with the actual arguments \bar{d} and `this` with the receiver $\text{new } C\langle\bar{T}\rangle(\bar{e})$. We write $[\bar{d}/\bar{x}, e/y]e_0$ to stand for replacing x_1 by d_1, \dots, x_n by d_n , and y by e in the expression e_0 . The expression $(T)\text{new } C\langle\bar{T}\rangle(\bar{e})$ reduces to the subject $\text{new } C\langle\bar{T}\rangle(\bar{e})$ of typecast if the test succeeds; if not, the evaluation gets stuck, denoting a run-time error (that is, the situation where an exception would be thrown). The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $e \longrightarrow e'$ then $e.f \longrightarrow e'.f$, and the like), which also appear in Figure 7.

Expression Typing:	
$\Delta; \Gamma \vdash x \in \Gamma(x)$	(T-VAR)
$\frac{\Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash bound_{\Delta}(T_0) \uparrow^{\Delta'} C \langle \bar{U} \rangle \quad fields(C \langle \bar{U} \rangle) = \bar{S} \bar{f} \quad S_i \downarrow_{\Delta'} T}{\Delta; \Gamma \vdash e_0.f_i \in T}$	(T-FIELD)
$\frac{\begin{array}{l} \Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash bound_{\Delta}(T_0) \uparrow^{\Delta'} C \langle \bar{T} \rangle \\ mtype(m, C \langle \bar{T} \rangle) = \langle \bar{Y} \langle \bar{P} \rangle \bar{U} \rightarrow U_0 \quad \{\bar{Y}\} \cap dom(\Delta') = \emptyset \\ \Delta \vdash \bar{V} \text{ ok} \quad \Delta, \Delta' \vdash \bar{V} \prec: [\bar{V}/\bar{Y}] \bar{P} \\ \Delta; \Gamma \vdash \bar{e} \in \bar{S} \quad \Delta, \Delta' \vdash \bar{S} \prec: [\bar{V}/\bar{Y}] \bar{U} \quad [\bar{V}/\bar{Y}] U_0 \downarrow_{\Delta'} T \end{array}}{\Delta; \Gamma \vdash e_0.m \langle \bar{V} \rangle (\bar{e}) \in T}$	(T-INVK)
$\frac{\Delta \vdash C \langle \bar{T} \rangle \text{ ok} \quad fields(C \langle \bar{T} \rangle) = \bar{U} \bar{f} \quad \Delta; \Gamma \vdash \bar{e} \in \bar{S} \quad \Delta \vdash \bar{S} \prec: \bar{U}}{\Delta; \Gamma \vdash \text{new } C \langle \bar{T} \rangle (\bar{e}) \in C \langle \bar{T} \rangle}$	(T-NEW)
$\frac{\begin{array}{l} \Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash T \text{ ok} \\ \Delta \vdash bound_{\Delta}(T_0) \prec: bound_{\Delta}(T) \quad \text{or} \quad \Delta \vdash bound_{\Delta}(T) \prec: bound_{\Delta}(T_0) \end{array}}{\Delta; \Gamma \vdash (T) e_0 \in T}$	(T-CAST)
$\frac{\begin{array}{l} \Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash T \text{ ok} \\ \Delta \vdash bound_{\Delta}(T_0) \not\prec: bound_{\Delta}(T) \quad \Delta \vdash bound_{\Delta}(T) \not\prec: bound_{\Delta}(T_0) \end{array}}{\Delta; \Gamma \vdash (T) e_0 \in T}$	(T-SCAST)
Method Typing:	
$\frac{mtype(m, N) = \langle \bar{Z} \langle \bar{Q} \rangle \bar{U} \rightarrow U_0 \text{ implies } [\bar{Y}/\bar{Z}](\bar{Q}, \bar{U}, U_0) = \bar{P}, \bar{T}, T_0}{\text{override}(m, N, \langle \bar{Y} \langle \bar{P} \rangle \bar{T} \rightarrow T_0)}$	
$\frac{\begin{array}{l} \Delta = \bar{X} \prec: \bar{N}, \bar{Y} \prec: \bar{P} \quad \Delta \vdash \bar{P}, \bar{T}, T_0 \text{ ok} \\ \Delta; \bar{x} : \bar{T}, \text{this} : C \langle \bar{X} \rangle \vdash e_0 \in S_0 \quad \Delta \vdash S_0 \prec: T_0 \\ \text{class } C \langle \bar{X} \langle \bar{N} \rangle \langle D \langle \bar{S} \rangle \{ \dots \} \quad \text{override}(m, D \langle \bar{S} \rangle, \langle \bar{Y} \langle \bar{P} \rangle \bar{T} \rightarrow T_0) \end{array}}{\langle \bar{Y} \langle \bar{P} \rangle T_0 \text{ m}(\bar{T} \bar{x}) \{ \text{return } e_0; \} \text{ OK IN } C \langle \bar{X} \langle \bar{N} \rangle}$	(T-METHOD)
Class Typing:	
$\frac{\bar{X} \prec: \bar{N} \vdash \bar{N}, D \langle \bar{S} \rangle, \bar{T} \text{ ok} \quad \bar{M} \text{ OK IN } C \langle \bar{X} \langle \bar{N} \rangle}{\text{class } C \langle \bar{X} \langle \bar{N} \rangle \langle D \langle \bar{S} \rangle \{ \bar{T} \bar{f}; \bar{M} \} \text{ OK}}$	(T-CLASS)

Figure 6: Typing

Reduction Rules:

$$\frac{\text{fields}(\mathbb{C}\langle\bar{T}\rangle) = \bar{U} \bar{f}}{\text{new } \mathbb{C}\langle\bar{T}\rangle(\bar{e}) . f_i \longrightarrow e_i} \quad (\text{R-FIELD})$$

$$\frac{\text{mbody}(\mathbb{m}\langle\bar{V}\rangle, \mathbb{C}\langle\bar{T}\rangle) = \bar{x} . e_0}{\text{new } \mathbb{C}\langle\bar{T}\rangle(\bar{e}) . \mathbb{m}\langle\bar{V}\rangle(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, \text{new } \mathbb{C}\langle\bar{T}\rangle(\bar{e})/\text{this}]e_0} \quad (\text{R-INVK})$$

$$\frac{\emptyset \vdash \mathbb{C}\langle\bar{T}\rangle <: T}{(T)\text{new } \mathbb{C}\langle\bar{T}\rangle(\bar{e}) \longrightarrow \text{new } \mathbb{C}\langle\bar{T}\rangle(\bar{e})} \quad (\text{R-CAST})$$

$$\frac{e_0 \longrightarrow e_0'}{e_0 . f \longrightarrow e_0' . f} \quad (\text{RC-FIELD})$$

$$\frac{e_0 \longrightarrow e_0'}{e_0 . \mathbb{m}\langle\bar{V}\rangle(\bar{e}) \longrightarrow e_0' . \mathbb{m}\langle\bar{V}\rangle(\bar{e})} \quad (\text{RC-INV-RECV})$$

$$\frac{e_i \longrightarrow e_i'}{e_0 . \mathbb{m}\langle\bar{V}\rangle(\dots, e_i, \dots) \longrightarrow e_0 . \mathbb{m}\langle\bar{V}\rangle(\dots, e_i', \dots)} \quad (\text{RC-INV-ARG})$$

$$\frac{e_i \longrightarrow e_i'}{\text{new } \mathbb{C}\langle\bar{T}\rangle(\dots, e_i, \dots) \longrightarrow \text{new } \mathbb{C}\langle\bar{T}\rangle(\dots, e_i', \dots)} \quad (\text{RC-NEW-ARG})$$

$$\frac{e_0 \longrightarrow e_0'}{(T)e_0 \longrightarrow (T)e_0'} \quad (\text{RC-CAST})$$

Figure 7: Reduction Rules

6.4 Properties

Type soundness (Theorem 6.4.3) is shown through subject reduction and progress properties [50]. To state type soundness, we require the notion of *values*, defined by: $v ::= \text{new } C\langle\bar{T}\rangle(v_1, \dots, v_n)$ (n may be 0).

6.4.1 Theorem [Subject Reduction]: If $\Delta; \Gamma \vdash e \in T$ where Δ has non-variable bounds and $e \longrightarrow e'$, then $\Delta; \Gamma \vdash e' \in S$ and $\Delta \vdash S <: T$ for some S .

Proof: See Appendix A. ■

6.4.2 Theorem [Progress]: Suppose e is a well-typed expression.

1. If e includes $\text{new } C\langle\bar{T}\rangle(\bar{e}) . f$ as a subexpression, then $\text{fields}(C\langle\bar{T}\rangle) = \bar{U} \bar{f}$ and $f = f_i$.
2. If e includes $\text{new } C\langle\bar{T}\rangle(\bar{e}) . m\langle\bar{V}\rangle(\bar{d})$ as a subexpression, then $\text{mbody}(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle) = \bar{x} . e_0$ and $|\bar{x}| = |\bar{d}|$.

Proof: Easy. ■

6.4.3 Theorem [Type Soundness]: If $\emptyset; \emptyset \vdash e \in T$ and $e \longrightarrow^* e'$ being a normal form, then e' is either a value v such that $\emptyset; \emptyset \vdash v \in S$ and $\emptyset \vdash S <: T$ for some S , or an expression that includes $(T)\text{new } C\langle\bar{T}\rangle(\bar{e})$ where $\emptyset \vdash C\langle\bar{T}\rangle \not<: T$.

Proof: Follows from Theorems 6.4.1 and 6.4.2. ■

7 Related Language Mechanisms

Parametric Classes and Variance. There have been several languages, such as Eiffel [30, 31], POOL [2] and Strongtalk [4, 3], that support variance for parametric classes; more recently, Cartwright and Steele [15] have discussed possible introduction of variance to NextGen—a proposal for extending Java with generics. Their approaches are different from ours in that, in those languages, variance is a property of *classes*, rather than *types*—variance annotations are attached to the declaration of a type parameter so that a *designer* of a class can express his/her intent about all the parametric types derived from the class. Then, the system can statically check whether the variance declaration is correct: for example, if X is declared to be covariant in a parametric class $C\langle X \rangle$ but used in a method argument type or (writable) field type, the compiler will reject the class. Thus, in order to enhance reusability with variance, library designers must take great care to structure the API, which can be a daunting task. Day et al. [17] even argued that this restriction was too severe and, after all, they have decided to drop variance from their language Theta. On the contrary, in our system, *users* of a parametric class can choose appropriate variance: a class $C\langle X \rangle$, for example, can have arbitrary occurrences of X and induces four different types $C\langle T \rangle$, $C\langle +T \rangle$, $C\langle -T \rangle$, and $C\langle *T \rangle$ with one concrete type argument T . We believe that moving annotations to the use site provides much more flexibility.

In an early design of Eiffel, every parametric type was unsoundly assumed to be covariant. To remedy the problem, Cook [16] proposed to *infer*, rather than *declare*, variance annotations on each type parameters of a given class. For example, if X in the class $C\langle X \rangle$ appears only in a method return type, the type $C\langle T \rangle$ is automatically regarded as covariant, and similarly for contravariant.

This proposal is not adopted in the current design, in which every parametric class is regarded as invariant [19].

Theoretical foundations of this classical approach to variance have been considered in the context of typed λ -calculi [12, 18, 40], where type operators (functions from types to types) are equipped with a variance property, often called polarity. The resulting type system is complicated and its meta-theory is rather hard to investigate. On the other hand, the theoretical basis of variant parametric types is bounded existential types, whose properties are fairly well studied—at least the case without lower bounds.

Parametric Methods. One may wonder if parametric methods with bounded polymorphism can be used for the examples shown in Section 4; indeed, some of them can be easily handled with parametric methods, if a type variable is allowed to be an upper bound of another type variable (which is not the case for GJ). For instance, the method `fillFrom` can be implemented as follows:

```
class Vector<X extends Object>{
    ...
    <Y extends X> void fillFrom(Vector<Y> v, int start){
        for (int i=0;i<v.size() && i+start<size();i++)
            setElementAt(v.getElementAt(i), i+start);
    }
}
...
Vector<Number> vn = new Vector<Number>(20);
Vector<Integer> vi = new Vector<Integer>(10);
Vector<Float> vf = new Vector<Float>(10);
vn.fillFrom<Integer>(vi,0);
vn.fillFrom<Float>(vf,10);
```

Here, the definition of `fillFrom` is parameterized by a type variable `Y`, bounded by an upper bound `X`, and the actual type arguments are explicitly given (inside `<>`) at method invocations.⁴ Similarly, `fillTo` can be expressed by using a *lower bound* of a type parameter:

```
class Vector<X extends Object>{
    ...
    <Y extendedby X> void fillTo(Vector<Y> v, int start){
        for (int i=0;i<size() && i+start<v.size();i++)
            v.setElementAt(getElementAt(i),i+start);
    }
}
```

Here, the keyword `extendedby` means that the type parameter `Y` must be a *supertype* of `X`. In general, it seems that a method taking arguments of variant parametric types can be easily rewritten in terms of parametric methods.

Although the two mechanisms can be used almost interchangeably for some cases, we think variant parametric types and parametric methods are complementary. On one hand, variant parametric types provide a means to specify a set of different instantiations of the same generic classes, making description of methods like `fillFrom` more concise. They also allow to mix different types in one data structure as shown in Section 4. On the other hand, parametric methods can express type dependency among method arguments and results, as in the two methods below:

⁴Some implementation of generics, such as GJ, may feature type parameter inference in method calls, so that an actual instantiation of method type parameters does not have to be provided at the call side. The discussion here, however, is completely independent of this issue, so, for clarity, we stick to the notation where type parameters are explicitly specified.

```
// swapping pos-th element in v1 and v2
<X> void swapElementAt(Vector<X> v1, Vector<X> v2, int pos){...}

// a database-like join operation on tables v1 and v2.
<X,Y,Z> Vector<Pair<X,Z>> join(Vector<Pair<X,Y>> v1,
                             Vector<Pair<Y,Z>> v2){...}
```

The type variable `X` in the method `swapElementAt` expresses dependency between input vectors—it enforces the two vectors `v1` and `v2` to carry the same type of elements. `X`, `Y`, and `Z` in method `join` are used to express dependency among the inputs and outputs. In both cases, such dependencies cannot be expressed by variant parametric types.

As shown above, simulating contravariance with parametric methods requires type parameters with lower bounds. However, their theory has not been well studied and it is not very clear whether or not basic implementation techniques such as type-erasure [5] can be straightforwardly extended to parametric methods with lower bounds. One may argue that the features provided by the methods `fillTo` and `fillFrom` are much the same, so one can easily find the covariant version of any method that uses contravariance, and then implement it using a parametric method only with upper bounds. However, this will force to program in a particular style: that is, for instance, programmers always have to write methods in the collection class consuming elements, instead of in the class producing them. We believe that hampering that freedom would lead to poor programming practice.

Also, we believe that a language with generics should enjoy the combination of both constructs so as to achieve even more power and expressiveness. For instance, the following example shows potential benefits of the combination:

```
<X> Vector<X> unzipleft(Vector<+Pair<+X,*>> vp){
    Vector<X> v=new Vector<X>(vp.size());
    for (int i=0;i<vp.size();i++)
        v.addElementAt(vp.getElementAt(i).getFst(),i);
    return v;
}
```

Method `unzipleft` creates a `Vector<X>` element by unzipping a given list of pairs and taking the first element of each pair. While the type parameter `X` keeps track of dependency between input and output types, variance contributes to widen the range of acceptable arguments, i.e., (1) second elements can be anything (hence `*`); (2) first elements can be any subtype of the expected type `X`; and (3) pairs can be those from a subclass of `Pair`.

Another example is a generic sort method using a parameterized interface `Comparable` and `F`-bounded polymorphism [35]:

```
interface Comparable<X extends Object>{
    int compareTo(X x);
}
...
<X implements Comparable<-X>> void sort(Vector<X> v){
    for (int i=v.size();i>1;i--)
        for (int j=0;j<i-1;j++){
            if (v.getElementAt(j).compareTo(v.getElementAt(j+1))) {
                X temp=v.getElementAt(j);
                v.setElementAt(v.getElementAt(j+1),i);
                v.setElementAt(temp,j+1);
            }
        }
}
```

This signature of `sort` allows a vector of objects comparable to themselves as usual.

```
class Element implements Comparable<Element> {
    int compareTo(Element x) { ... }
}

Vector<Element> v = ...;
sort<Element>(v)    // OK -- Element <: Comparable<Element>
```

Moreover, it also allows a vector of subtype of `Elements` to be sorted, thanks to contravariance: it holds that `MyElement <: Element <: Comparable<Element> <: Comparable<-MyElement>`.

```
class MyElement extends Element{
    ...
    // int compareTo(Element x);
}

Vector<MyElement> v = ...;
sort<MyElement>(v);
```

Note that it would not be allowed without `-` in the argument type of `sort` since `MyElement` is *not* a subtype of `Comparable<MyElement>`.

We should also note that those examples of combining parametric methods and variant parametric types, in principle, could be written only with parametric methods (with lower bounds). For example, `unzipleft` could be written as a parametric method with four type parameters:

```
<X,Y extends X,Z,W extends Pair<Y,Z>> Vector<X> unzipleft(Vector<W> vp){ ... }
```

The signatures of those methods, however, tend to be longer and harder to understand as above. Another, probably more important problem is about actual type arguments at call sites: it is daunting for programmers to explicitly write them and it is hard to design a both effective and efficient type inference algorithm.

Virtual Types. As we have already mentioned, the idea of variant parametric types has emerged from structural virtual types proposed by Thorup and Torgersen [45]. In a language with virtual types [29, 44], a type can be declared as a member of a class, just as well as fields and methods, and the virtual type member can be overridden in subclasses. For example, a generic bag class `Bag` has a virtual type member `ElmTy`, which is bound to `Object`; specific bags can be obtained by declaring a subclass of `Bag`, overriding `ElmTy` by their concrete element types.

Since the original proposals of virtual types were unsafe and required run-time checks, Torgersen [46] developed a safe type system for virtual types by exploiting two kinds of type binding: open and final. An open type member is overridable but the identity of the type member is made abstract, prohibiting unsafe accesses such as putting elements into a bag whose element type is unknown; a final type member cannot be overridden in subclasses but the identity of the type member is manifest, making concrete bags. In a pseudo Java-like language with virtual types, a generic bag class and concrete bag classes can be written as follows.

```

class Bag {
    type ElmTy <: Object; // open binding
    ElmTy get() { ... }
    void put(ElmTy e) { ... }
    // No element can be put into a Bag.
}
class StringBag extends Bag {
    type ElmTy == String; // final binding
    // Strings can be put into a String Bag.
}
class IntegerBag extends Bag { type ElmTy == Integer; }

```

One criticism on this approach was that concrete bags were often obtained only by overriding the type member, making lots of small subclasses of a bag. Structural virtual types are proposed to remedy this problem: type bindings can be described in a type expression and a number of concrete types are derived from one class. For example, a programmer can instantiate `Bag[ElmTy==Integer]` to make an integer bag, where the `[]` clause describes a type binding. In addition, `Bag[ElmTy==Integer] <: Bag[ElmTy<:Object]` and `Bag[ElmTy==String] <: Bag[ElmTy<:Object]` hold as is expected.

In their paper [45], it is briefly (and informally) discussed how structural virtual types can be imported to parametric classes, the idea on which our development is based. The above programming is achieved by making `ElmTy` a type parameter to the class `Bag`, rather than a member of a class.

```

class Bag<ElmTy extends Object> extends Object {
    ElmTy get() { ... }
    void put(ElmTy e) { ... }
}

```

Then, an integer bag is obtained by `new Bag<Integer>()` and `Bag<Integer> <: Bag<+Object>` holds. In other words, the type `Bag<Integer>` corresponds to `Bag[ElmTy==Integer]` and `Bag<+Object>` to `Bag[ElmTy<:Object]`. This similarity is not just superficial: as in [22], programming with virtual types is shown to be simulated (to some degree) by exploiting bounded and manifest existential types [21, 26], on which our formal type system is also partly based.

Thus, variant parametric types can be considered a generalization of the idea above with contravariance and bivarience. Other differences are as follows. On one hand, virtual types seem more suitable for programming with extensible mutually recursive classes/interfaces [7, 45, 10]. On the other hand, our system allows a (partially) instantiated parametric class to be extended: as we have already discussed, a programmer can declare a subclass `PP<X,Y>` that inherits from `Pair<Pair<X,Y>,Pair<X,Y>>`. It is not very clear (at least, from their paper [45]) how to encode such programming in structural virtual types.

Existential Types in Object-Oriented Languages. The idea of existential types can actually be seen in several mechanisms for object-oriented languages, often in disguised (and limited) forms.

In the language *LOOM* [8], subtyping is dropped in favor of matching [6, 9], thus losing subsumption. To recover the flexibility of subsumption to some degree, the notion of “hash” types `#T` is introduced, which stand for the set of types that match `T`. As is pointed out in [8], `#T` is considered a “match-bounded” existential type $\exists X<\#T.X$, where `X<\#T` stands for “`X` matches `T`.”

The notion of exact types [7] is introduced in a proposal of a Java-like language with both parametric classes and virtual types. In that language, a class of the name `C` induces two types

C and $@C$; the type C denotes a set of objects created only from the class C , while $@C$ denotes a set of objects created from C or its subclasses. In this sense, $@C$ can be considered the existential type $\exists X \prec C.X$, where \prec denotes subclassing (not subtyping). A similar idea can be found in an old version of the type system of Sather [27], where the symbol $\$$ replaces $@$ and is used to enable efficient method dispatching. In the current design of Sather [43], $\$$ is used for names of abstract classes, from which subclasses can be derived, while names without the leading $\$$ are used for concrete classes, from which objects can be instantiated but subclasses cannot be derived.

Raw types [5] of GJ are also close to bounded existential types [24]. In GJ, the class `Vector<X>`, for example, induces the raw type `Vector` as well as parametric types including `Vector<Integer>` and `Vector<String>`. The raw type `Vector` is typically used by legacy classes written in monomorphic Java, making it smooth to importing old Java code into GJ. `Vector` is considered a supertype of every parametric type `Vector<T>` and behaves somehow like `$\exists X \prec \text{Object}.\text{Vector}\langle X \rangle$` . One significant difference is that certain unsafe operations putting elements into a raw vector are permitted with a compiler warning.

Explicit Use of Existential Types. As a last remark, it would be interesting to evaluate the full power of existential types in object-oriented programming. In fact, by using unpacking appropriately, more expressions are typeable. For example, suppose x is given type `$\exists X.\text{Vector}\langle X \rangle$` and a programmer wants to get an element from x and put it back to (another position) of x . Actually, the expression

```
open x as [Y,y] in y.setElementAt(y.getElementAt(0), 1)
```

would be well-typed (if y is read-only) since inside `open`, the elements are all given an identical type Y . On the other hand, in our language,

```
x.setElementAt(x.getElementAt(0), 1)
```

is not typeable since this expression would correspond to

```
open x as [Y,y] in
  y.setElementAt(open x as [Z,z] in z.getElementAt(0), 1)
```

which introduces two `opens`, and Y and Z are distinguished. Although there is an advantage when using existential types explicitly, we think that allowing programmers to directly insert `open` operations may make programming more cumbersome.

8 Conclusion and Future Work

In this paper, we have presented the language construct of *variant parametric types* as an extension of class-based object-oriented languages supporting generics. Variant parametric types promote inclusive polymorphism for generic types, by providing a uniform view over different instantiations of a generic class. With variant parametric types, unlike most of previous work on variance for generics in object-oriented languages, decision on which variance is desirable is deferred until a type is induced from a generic class. Thereby, reusability and scalability have been significantly enhanced.

Variant parametric types generally make it possible to widen the applicability of methods when arguments are used in certain limited ways in the method body. Furthermore, bivariance—which has not been taken into account in previous work—seems to be fairly useful for parametric types

with more than one type parameter since the bivariance annotation can represent “wild card” when used in a method signature.

We have also pointed out how variant parametric types can be subtle, especially when parametric types are nested. A key idea in the development of the type system is to exploit similarity between variant parametric types and bounded existential types. For a rigorous argument of safety of the type system, we have developed a core calculus of variant parametric types, based on Featherweight GJ [23], and proved type soundness.

Implementation issues are not addressed in this paper and left for future work. It is worth noting, however, that existing implementation techniques seem to allow extensions to variant parametric types in a straightforward manner.

For instance, the type-erasure technique—which is one of the basic approaches to implementing generics, used in both Java (GJ [5]) and the .NET CLR ([42])—can be directly exploited for dealing with variant parametric types, for variance annotations can be simply erased in the translated code as well as type arguments⁵. In fact, as already mentioned, an implementation is already available: Sun Microsystems has recently released a prototype compiler for Java 1.5, including generics with a variance-enabled type system as an experimental feature⁶.

Other advanced implementation techniques where type arguments are maintained at run-time, such as LM translator [49, 48, 47], can be extended to variance as well. In this case, variant parametric types can be supported by simply implementing subtyping in which variance is taken into account. Since the current design allows run-time type arguments to be variant parametric types, it will be important to estimate extra overhead to manage information on variance annotations. One of the basic implementation issues would be the development of an efficient algorithm for subtyping variant parametric types. For example, the approach presented in Raynaud and Thierry [39] and Placz and Vitek [37] may be worth investigating.

Unlike the core calculus presented here, GJ allows (in fact, forces) programmers to omit actual type arguments of parametric methods and the compiler infers appropriate type arguments. As in Featherweight GJ, we have side-stepped this issue by making type arguments explicit and regarding the calculus as an intermediate language after the type inference stage. Still, type inference in the presence of variant parametric types should be investigated. Unfortunately, however, it is reported that there are more cases in which the GJ-style local type inference fails, due to the richer subtyping hierarchy introduced by variance⁷—in short, variance-based subtyping yields more type arguments that make a given method invocation typeable, but it is less likely that there is a best one among them.

Other future work includes further evaluation of the expressiveness of variant parametric type through large-scale applications.

Acknowledgements

We thank ECOOP2002 anonymous referees for useful comments, Gilad Bracha and Benjamin Pierce for helping us clarify about related work, and David Griswold and people involved in the development of the prototype compiler at Sun Microsystems for having discussion with us. This work was

⁵The language has to be somewhat constrained due to the lack of run-time type arguments, though: for example, the target of typecasts cannot be a type variable and so on.

⁶The concrete syntax in the prototype is different from what we propose here: for example, `C<+T>` is written `C<? extends T>`, which makes its existential nature more explicit.

⁷Personal communication with Gilad Bracha, Erik Ernst, Martin Odersky, Mads Torgersen, and other people involved in the implementation of Sun Microsystems’ prototype.

supported in part by Grant-in-Aid for Scientific Research on Priority Areas Research No. 13224013 (Igarashi).

References

- [1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*, pages 49–65, Atlanta, GA, October 1997.
- [2] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the OOPSLA/ECOOP*, pages 161–168, Ottawa, Canada, October 1990.
- [3] Gilad Bracha. The Strongtalk type system for Smalltalk. In *Proceedings of the OOPSLA '96 Workshop on Extending the Smalltalk Language*, 1996. Also available electronically through <http://java.sun.com/people/gbracha/nwst.html>.
- [4] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, pages 215–230, Washington, DC, October 1993.
- [5] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pages 183–200, Vancouver, BC, October 1998.
- [6] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. Preliminary version in POPL 1993, under the title “Safe type checking in a statically typed object-oriented programming language”.
- [7] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, LNCS 1445, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.
- [8] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good “match” for object-oriented languages. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, pages 104–127, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [9] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes on Computer Science*, pages 27–51, Aarhus, Denmark, August 1995. Springer-Verlag.
- [10] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proc. of the 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, volume 20 of *Electronic Notes in Theoretical Computer Science*, New Orleans, LA, April 1999. Elsevier. Available through <http://www.elsevier.nl/locate/entcs/volume20.html>.
- [11] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FPCA '89)*, pages 273–280, September 1989.
- [12] Luca Cardelli. Notes about F_{\leq}^{ω} . Unpublished manuscript, October 1990.
- [13] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Preliminary version in TACS '91 (Sendai, Japan, pp. 750–770).
- [14] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

- [15] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pages 201–215, Vancouver, BC, October 1998.
- [16] William Cook. A proposal for making Eiffel type-safe. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, pages 57–70, Nottingham, England, July 1989. Cambridge University Press.
- [17] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '95)*, pages 156–168, Austin, TX, October 1995.
- [18] Dominic Duggan and Adriana Compagnoni. Subtyping for object type constructors. In *Informal Proceedings of the the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL6)*, January 1999.
- [19] Interactive Software Engineering. An Eiffel tutorial. Available through <http://www.eiffel.com/doc/online/eiffel50/intro/language/tutorial-00.html>, 2001.
- [20] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.
- [21] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 123–137, Portland, OR, January 1994.
- [22] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 2002. An earlier version in *Proc. of the 13th ECOOP*, Springer LNCS 1628, pages 161–185, 1999.
- [23] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001. A preliminary summary appeared in *Proc. of OOPSLA '99*, pages 132–146, Denver, CO, October 1999.
- [24] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Informal Proceedings of the 8th International Workshop on Foundations of Object-Oriented Languages (FOOL8)*, London, England, January 2001. Available through <http://www.cs.williams.edu/~kim/FOOL/FOOL8.html>.
- [25] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In Boris Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP2002)*, volume 2374 of *Lecture Notes on Computer Science*, pages 441–469, Málaga, Spain, June 2002. Springer-Verlag.
- [26] Xavier Leroy. Manifest types, modules and separate compilation. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 109–122, Portland, OR, January 1994.
- [27] Chu-Cheow Lim and A. Stolcke. Sather language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, University of California, Berkeley, May 1991.
- [28] Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), May 1988.
- [29] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '89)*, pages 397–406, New Orleans, LA, 1989.
- [30] Bertrand Meyer. Genericity versus inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '86)*, pages 391–405, September 1986.
- [31] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

- [32] Andrew C. Meyers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145. ACM, 1997.
- [33] Microsoft. Generic for C# and .NET CLR. Available through <http://research.microsoft.com/projects/clrgen/>, 2002.
- [34] Microsoft Corporation. The .NET Common Language Runtime. Information available through <http://msdn.microsoft.com/net/>, 2001.
- [35] Sun Microsystems. Prototype for JSR014: Adding generics to the Java programming language v. 2.0. Available through http://developer.java.sun.com/developer/earlyAccess/adding_generics/index.html, 2003.
- [36] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. of the 12th ACM POPL*, 1985.
- [37] Krzysztof Palacz and Jan Vitek. Subtype tests in real time. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP2003)*, volume 2743 of *Lecture Notes on Computer Science*, pages 378–404, Darmstadt, Germany, July 2003. Springer-Verlag.
- [38] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Preliminary version in POPL '92.
- [39] Olivier Raynaud and Eric Thierry. A quasi optimal bit-vector encoding of tree hierarchies: Application to efficient type inclusion tests. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP2001)*, volume 2072 of *Lecture Notes on Computer Science*, pages 165–180. Springer-Verlag, June 2001.
- [40] Martin Steffen. Polarized higher-order subtyping. In *Proceedings of the the Types working group Workshop on Subtyping, inheritance and modular development of proofs*, August 1997.
- [41] Sun Microsystems. Adding generic types to the Java programming language. Java Specification Request JSR-000014, <http://jcp.org/jsr/detail/014.jsp>, 1998.
- [42] Don Syme and Andrew Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, June 2001.
- [43] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. In Jurg Gutknecht, editor, *Programming Languages and System Architectures*, LNCS 782, pages 208–227. Springer-Verlag, November 1993.
- [44] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [45] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, LNCS 1628, pages 186–204, Lisbon, Portugal, June 1999. Springer-Verlag.
- [46] Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998. Available through <http://www.cs.williams.edu/~kim/FOOL/FOOL5.html>.
- [47] Mirko Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Proceedings of the ACM Symposium on Applied Computing*, pages 610–619, March 2001.

- [48] Mirko Viroli. A type-passing approach for the implementation of parametric methods in Java. *The Computer Journal*, 46(3), 2003.
- [49] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA2000)*, pages 146–165, October 2000.
- [50] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A Proof of Theorem 6.4.1

The basic structure of the proof is similar to the one for Featherweight GJ [23]. The main lemmas required are ones that state preservation of subtyping and typing under *type* substitution (Lemmas A.4 and A.12) and one that states preservation of typing under term substitution (Lemma A.13). (Readers familiar with proofs of subject reduction for typed lambda-calculi like F_{\leq} [13] will notice many similarities). All of them are proved by straightforward induction on a derivation of $\Delta \vdash S <: T$ or $\Delta; \Gamma \vdash e \in T$.

In the following proof, the underlying class table is assumed to be ok.

A.1 Lemma [Weakening]: Suppose $\Delta, \bar{X} <: \bar{N} \vdash \bar{N}$ ok and $\Delta \vdash U$ ok.

1. If $\Delta \vdash S <: T$, then $\Delta, \bar{X} <: \bar{N} \vdash S <: T$.
2. If $\Delta \vdash S$ ok, then $\Delta, \bar{X} <: \bar{N} \vdash S$ ok.
3. If $\Delta; \Gamma \vdash e \in T$, then $\Delta; \Gamma, x : U \vdash e \in T$ and $\Delta, \bar{X} <: \bar{N}; \Gamma \vdash e \in T$.

Proof: By straightforward induction on the derivation of $\Delta \vdash S <: T$ and $\Delta \vdash S$ ok and $\Delta; \Gamma \vdash e \in T$, respectively. ■

A.2 Lemma [Narrowing]:

1. If $\Delta, X <: S \vdash T_1 <: T_2$ and $\Delta \vdash U <: S$, then $\Delta, X <: U \vdash T_1 <: T_2$.
2. If $\Delta, X :> S \vdash T_1 <: T_2$ and $\Delta \vdash S <: U$, then $\Delta, X :> U \vdash T_1 <: T_2$.
3. If $\Delta, X : (*, S) \vdash T_1 <: T_2$, then $\Delta, X : (v, U) \vdash T_1 <: T_2$.

Proof: By induction on derivation of $\Delta, X <: S \vdash T_1 <: T_2$ and $\Delta, X :> S \vdash T_1 <: T_2$ and $\Delta, X : (*, S) \vdash T_1 <: T_2$, respectively. ■

The following lemmas (Lemmas A.3–A.5, and A.12) state that type substitution preserves derivability of judgments about typing.

A.3 Lemma:

1. If $\Delta_1 \vdash \bar{S} <: [\bar{S}/\bar{X}]\bar{N}$ and $\Delta_1 \vdash \bar{S}$ ok and $\Delta_1, \bar{X} <: \bar{N}, \Delta_2 \vdash C <: \bar{v}T > \uparrow^{\Delta'} C <: \bar{w}U >$ with none of \bar{X} appearing in Δ_1 and none of type variables in $dom(\Delta')$ appearing in \bar{S} , then $\Delta_1, [\bar{S}/\bar{X}]\Delta_2 \vdash [\bar{S}/\bar{X}]C <: \bar{v}T > \uparrow^{[\bar{S}/\bar{X}]\Delta'} [\bar{S}/\bar{X}]C <: \bar{w}U >$.
2. If $S \downarrow_{\Delta} T$ where $dom(\Delta)$ and \bar{X} are distinct, then $[\bar{S}/\bar{X}]S \downarrow_{[\bar{S}/\bar{X}]\Delta} [\bar{S}/\bar{X}]T$,

3. $[\overline{S}/\overline{X}]fields(\mathbb{C}\langle\overline{T}\rangle) = fields([\overline{S}/\overline{X}]\mathbb{C}\langle\overline{T}\rangle)$, and
4. $[\overline{S}/\overline{X}]mtype(\mathfrak{m}, \mathbb{C}\langle\overline{T}\rangle) = mtype(\mathfrak{m}, [\overline{S}/\overline{X}]\mathbb{C}\langle\overline{T}\rangle)$.

Proof: By straightforward induction on the derivation of $\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash \mathbb{C}\langle\overline{vT}\rangle \uparrow^{\Delta'} \mathbb{C}\langle\overline{wU}\rangle$ and $S \downarrow_{\Delta} \mathbb{C}\langle\overline{T}\rangle$ and $fields(\mathbb{C}\langle\overline{T}\rangle)$ and $mtype(\mathbb{T})$, respectively. ■

A.4 Lemma [Type Substitution Preserves Subtyping]: If $\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash S \langle: T$ and $\Delta_1 \vdash \overline{U} \langle: [\overline{U}/\overline{X}]\overline{N}$ with $\Delta_1 \vdash \overline{U}$ ok and none of \overline{X} appearing in Δ_1 , then $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S \langle: [\overline{U}/\overline{X}]T$.

Proof: By induction on the derivation of $\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash S \langle: T$.

Case S-REFL, S-TRANS, S-VAR:

Easy.

Case S-CLASS:

Easily follows from Lemma A.3 (1) and (2).

Case S-UBOUND: $S = X \quad T = (\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2)(X)$

The case where $X \in dom(\Delta_1) \cup dom(\Delta_2)$ is immediate. On the other hand, if $X = X_i$, then, by assumption, we have $\Delta_1 \vdash U_i \langle: [\overline{U}/\overline{X}]N_i$. Finally, Lemma A.1 finishes the case.

Case S-LBOUND:

Immediately follows from the fact that $X \in dom(\Delta_1) \cup dom(\Delta_2)$. ■

A.5 Lemma [Type Substitution Preserves Type Well-Formedness]: If $\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash T$ ok and $\Delta_1 \vdash \overline{U} \langle: [\overline{U}/\overline{X}]\overline{N}$ with $\Delta_1 \vdash \overline{U}$ ok and none of \overline{X} appearing in Δ_1 , then $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]T$ ok.

Proof: By induction on the derivation of $\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash T$ ok, with a case analysis on the last rule used.

Case WF-OBJECT:

Trivial.

Case WF-VAR: $T = X \quad X \in dom(\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2)$

The case $X \in X_i$ follows from $\Delta_1 \vdash \overline{U}$ ok and Lemma A.1; otherwise immediate.

Case WF-CLASS: $T = \mathbb{C}\langle\overline{T}\rangle \quad \Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash \overline{T}$ ok $\quad \Delta_1, \overline{X}\langle:\overline{N}, \Delta_2 \vdash \overline{T} \langle: [\overline{T}/\overline{Y}]\overline{P}$
 $\text{class } \mathbb{C}\langle\overline{Y}\langle\overline{P}\rangle\langle\overline{D}\langle\overline{S}\rangle \{ \dots \}$

By the induction hypothesis, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{T}$ ok. On the other hand, by Lemma A.4, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{T} \langle: [\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{P}$. Since $\overline{Y}\langle:\overline{P} \vdash \overline{P}$ by the rule T-CLASS, \overline{P} does not include any of \overline{X} as a free variable. Thus, $[\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{P} = [[\overline{U}/\overline{X}]\overline{T}/\overline{Y}]\overline{P}$, and finally, we have $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \mathbb{C}\langle[\overline{U}/\overline{X}]\overline{T}\rangle$ ok by WF-CLASS. ■

A.6 Lemma: Suppose Δ has non-variable bounds and is of the form $\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2$. If $\Delta \vdash T$ ok and $\Delta_1 \vdash \overline{U} \langle: [\overline{U}/\overline{X}]\overline{N}$ with $\Delta_1 \vdash \overline{U}$ ok and none of \overline{X} appearing in Δ_1 . Then, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T) \langle: [\overline{U}/\overline{X}](bound_{\Delta}(T))$.

Proof: The case where T is a nonvariable type is trivial. The case where T is a type variable X and $X \in dom(\Delta_1) \cup dom(\Delta_2)$ is also easy. Finally, if T is a type variable X_i , then $bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T) = U_i$ and $[\overline{U}/\overline{X}](bound_{\Delta_1, \overline{X}\langle:\overline{N}, \Delta_2}(T)) = [\overline{U}/\overline{X}]N_i$; the assumption $\Delta_1 \vdash \overline{U} \langle: [\overline{U}/\overline{X}]\overline{N}$ and Lemma A.1 finish the proof. ■

To prove Lemma A.12, we require several more auxiliary lemmas. Particularly interesting ones are Lemmas A.10 and A.11 which essentially state that field access and method invocation remain well typed if the type of the target is changed to a subtype. The following three lemmas are needed for them.

A.7 Lemma [Close Yields a Supertype without Local Type Variables]: If $\Delta, \Delta' \vdash S$ ok and $S \Downarrow_{\Delta'} T$, then $\Delta, \Delta' \vdash S <: T$ and $\Delta \vdash T$ ok.

Proof: By structural induction on S . ■

A.8 Lemma: Suppose $\Delta, \Delta_1 \vdash C<\bar{S}>$ ok and $C<\bar{S}> \Downarrow_{\Delta_1} T$ and $\Delta \vdash T \Uparrow^{\Delta_2} C<\bar{U}>$ and $\Delta_3 \vdash S_0$ ok where $dom(\Delta_i)$ ($i = 1, 2, 3$) are distinct from each other.

1. If $[\bar{U}/\bar{X}]S_0 \Downarrow_{\Delta_2} S_0'$ and \bar{X} do not appear in Δ_1 or Δ_2 , then $[\bar{S}/\bar{X}]S_0 \Downarrow_{\Delta_1} S_0'$.
2. If $\Delta, \Delta_2 \vdash U_0 <: [\bar{U}/\bar{X}]S_0$ and $\Delta \vdash U_0$ ok with \bar{X} not appearing in Δ or Δ_1 or Δ_2 , then $\Delta, \Delta_1 \vdash U_0 <: [\bar{S}/\bar{X}]S_0$.

Proof: By inspection of the derivations $C<\bar{S}> \Downarrow_{\Delta_1} T$ and $\Delta \vdash T \Uparrow^{\Delta_2} C<\bar{U}>$, for each S_i , one of the following properties holds:

1. $S_i \Downarrow_{\Delta_1} S_i$ and U_i is equal to S_i ,
2. $S_i \Downarrow_{\Delta_1} S'_i$ (so $\Delta, \Delta_1 \vdash S_i <: S'_i$ by Lemma A.7) and $S_i \neq S'_i$ and U_i is a type variable Y and $\Delta_2(Y) = (+, S'_i)$, or
3. S_i is a type variable Z and $\Delta_1(Z) = (+, V)$ and U_i is a type variable Y and $\Delta_2(Y) = (+, V)$.

Now, the first part is easily shown by induction on S_0 . For the second part, from the inspection above, there exist $\Delta', \Delta'_1, \bar{Y}, \bar{T}, \bar{S}''$ such that $\Delta_1 = \Delta', \Delta'_1$ and $\Delta_2 = \Delta', \bar{Y} <: \bar{T}$ and $\bar{S} = [\bar{S}''/\bar{Y}]\bar{U}$ and $\Delta, \Delta_1 \vdash \bar{S}'' <: \bar{T}$. (Take all S_i and S'_i where the second case applies as \bar{S}'' and \bar{T} , respectively.) Then, it follows from Lemma A.4 that $\Delta, \Delta', \Delta'_1 \vdash [\bar{S}''/\bar{Y}]U_0 <: [\bar{S}''/\bar{Y}][\bar{U}/\bar{X}]S_0$. Since \bar{Y} do not appear in U_0 or S_0 , we have $[\bar{S}''/\bar{Y}]U_0 = U_0$ and $[\bar{S}''/\bar{Y}][\bar{U}/\bar{X}]S_0 = [\bar{S}/\bar{X}]S_0$, finishing the proof. ■

A.9 Lemma:

1. If $S \Downarrow_{\Delta', X:(v,T)} S''$ and $\Delta \vdash T$ ok where $dom(\Delta', X:(v,T))$ are fresh w.r.t. Δ , then $[T/X]S \Downarrow_{\Delta'} S'$ and $\Delta \vdash S' <: S''$.
2. If $\Delta \vdash T <: U$ and $S \Downarrow_{\Delta', X<:U} S''$ where $dom(\Delta', X<:U)$ are fresh for Δ , then $S \Downarrow_{\Delta', X<:T} S'$ and $\Delta \vdash S' <: S''$.
3. If $\Delta \vdash U <: T$ and $S \Downarrow_{\Delta', X>:U} S''$ where $dom(\Delta', X>:U)$ are fresh for Δ , then $S \Downarrow_{\Delta', X>:T} S'$ and $\Delta \vdash S' <: S''$.
4. If $S \Downarrow_{\Delta', X:(*,U)} S''$ where $dom(\Delta') \cup \{X\}$ are fresh for Δ , then $S \Downarrow_{\Delta', X:(v,T)} S'$ and $\Delta \vdash S' <: S''$.

Proof: By structural induction on S . ■

A.10 Lemma: If Δ has non-variable bounds and $\Delta \vdash bound_{\Delta}(T) \Uparrow^{\Delta_1} C<\bar{T}>$ and $fields(C<\bar{T}>) = \bar{U} \bar{f}$ and $U_i \Downarrow_{\Delta_1} U_0$, then for any S such that $\Delta \vdash S <: T$ and $\Delta \vdash S$ ok, it holds that $\Delta \vdash bound_{\Delta}(S) \Uparrow^{\Delta_2} D<\bar{S}>$ and $fields(D<\bar{S}>) = \bar{V} \bar{f}, \dots$ and $V_i \Downarrow_{\Delta_2} V_0$ and $\Delta \vdash V_0 <: U_0$ for some $\Delta_2, \bar{f}, D<\bar{S}>, \bar{V}$, and V'_0 .

Proof: By induction on the derivation $\Delta \vdash S <: T$ with the case analysis on the last rule used.

Case S-REFL:

Trivial.

Case S-UBOUND:

Trivial because $bound_{\Delta}(S) = bound_{\Delta}(T)$.

Case S-LBOUND:

Cannot happen.

Case S-TRANS:

Easy.

Case S-CLASS: $\text{class } D\langle\bar{X}\langle\bar{N}\rangle\langle C\langle\bar{S}'\rangle \{ \dots \} \\ S = D\langle\bar{v}\bar{W}\rangle \quad \Delta \vdash S \uparrow^{\Delta_2} D\langle\bar{W}'\rangle \quad [\bar{W}'/\bar{X}]C\langle\bar{S}'\rangle \downarrow_{\Delta_2} T$

It follows that we can take U_0 as V_0 from Lemma A.8(1) applied to the fact that, $fields(D\langle\bar{W}'\rangle) = fields(C\langle[\bar{W}'/\bar{X}]\bar{S}'\rangle)$, $\bar{T} \bar{g}$ for some \bar{T} and \bar{g} and $C\langle[\bar{W}'/\bar{X}]\bar{S}'\rangle \downarrow_{\Delta_2} T$ and $\Delta \vdash T \uparrow^{\Delta_1} C\langle\bar{T}\rangle$.

Case S-VAR: $S = C\langle\bar{v}\bar{S}'\rangle \quad T = C\langle\bar{w}\bar{T}'\rangle \quad \bar{v} \leq \bar{w} \\ \text{if } w_i \leq -, \text{ then } \Delta \vdash T'_i \prec: S'_i \quad \text{if } w_i \leq +, \text{ then } \Delta \vdash S'_i \prec: T'_i$

We show only the case where $v_i S'_i$ and $w_i T'_i$ are identical for all but one i . The proof easily extends to general cases.

Subcase: $w_i = o$

Follows from the fact that $v_i = o$ and $\Delta \vdash T'_i \prec: S'_i$ and $\Delta \vdash S'_i \prec: T'_i$.

Subcase: $w_i = +$

We have, $\Delta \vdash S'_i \prec: T'_i$ and v_i must be either $+$ or o . If v_i is $+$, then we have $\Delta \vdash S \uparrow^{\Delta_1, X \prec: S'_i} C\langle\bar{T}\rangle$ where $\Delta'_1, X \prec: T'_i = \Delta_1$. By Lemma A.9(2), $U_i \downarrow_{\Delta'_1, X \prec: S'_i} V'_i$ and $\Delta \vdash V'_i \prec: U'_i$. The other case for $v_i = o$ is similar (use Lemma A.9).

Subcase: $w_i = -$ or $w_i = *$

Similar. ■

A.11 Lemma: If Δ has non-variable bounds and $\Delta \vdash T$ ok and $\Delta \vdash bound_{\Delta}(T) \uparrow^{\Delta_1} C\langle\bar{T}\rangle$ and $mtype(m, C\langle\bar{T}\rangle) = \langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle \rightarrow U_0$ and $\Delta \vdash \bar{V}, \bar{W}$ ok and $\Delta, \Delta_1 \vdash \bar{V} \prec: [\bar{V}/\bar{Y}]\bar{P}$ and $\Delta, \Delta_1 \vdash \bar{W} \prec: [\bar{V}/\bar{Y}]\bar{U}$ and $[\bar{V}/\bar{Y}]U_0 \downarrow_{\Delta_1} V_0$, then for any S such that $\Delta \vdash S \prec: T$ and $\Delta \vdash S$ ok, we have $\Delta \vdash bound_{\Delta}(S) \uparrow^{\Delta_2} D\langle\bar{S}\rangle$ and $mtype(m, D\langle\bar{S}\rangle) = \langle\bar{Y}\langle\bar{P}'\rangle\bar{U}'\rangle \rightarrow U'_0$ and $\Delta, \Delta_2 \vdash \bar{V} \prec: [\bar{V}/\bar{Y}]\bar{P}'$ and $\Delta, \Delta_2 \vdash \bar{W} \prec: [\bar{V}/\bar{Y}]\bar{U}'$ and $[\bar{V}/\bar{Y}]U_0 \downarrow_{\Delta_2} V'_0$ and $\Delta \vdash V'_0 \prec: V_0$.

Proof: By induction on the derivation of $\Delta \vdash S \prec: T$ with the case analysis on the last rule used.

Case S-REFL:

Trivial.

Case S-UBOUND:

Trivial because $bound_{\Delta}(S) = bound_{\Delta}(T)$.

Case S-LBOUND:

Cannot happen.

Case S-TRANS:

Easy.

Case S-CLASS: $\text{class } D \langle \bar{X} \triangleleft \bar{N} \rangle \triangleleft C \langle \bar{S}' \rangle \{ \dots \}$
 $S = D \langle \bar{v}\bar{W} \rangle \quad \Delta \vdash S \uparrow^{\Delta_2} D \langle \bar{W}' \rangle \quad [\bar{W}'/\bar{X}]C \langle \bar{S}' \rangle \downarrow_{\Delta_2} T$

By T-CLASS and T-METHOD, $mtype(D \langle \bar{W}' \rangle) = mtype(C \langle [\bar{W}'/\bar{X}]\bar{S}' \rangle)$. Thus, the conclusion follows from Lemma A.8 and the fact that $C \langle [\bar{W}'/\bar{X}]\bar{S}' \rangle \downarrow_{\Delta_2} T$ and $\Delta \vdash T \uparrow^{\Delta_1} C \langle \bar{T} \rangle$.

Case S-VAR: $S = C \langle \bar{v}\bar{S}' \rangle \quad T = C \langle \bar{w}\bar{T}' \rangle \quad \bar{v} \leq \bar{w}$
 if $w_i \leq -$, then $\Delta \vdash T'_i \triangleleft S'_i$
 if $w_i \leq +$, then $\Delta \vdash S'_i \triangleleft T'_i$

We show only the case where $v_i S'_i$ and $w_i T'_i$ are identical for all but one i . The proof easily extends to general cases.

Subcase: $w_i = 0$

Follows from the fact that $v_i = 0$ and $\Delta \vdash T'_i \triangleleft S'_i$ and $\Delta \vdash S'_i \triangleleft T'_i$.

Subcase: $w_i = +$

We have $\Delta \vdash S'_i \triangleleft T'_i$ and v_i must be either $+$ or 0 . If v_i is $+$, then it follows from Lemma A.2 that $\Delta, \Delta_2 \vdash \bar{v} \triangleleft: [\bar{V}/\bar{Y}]\bar{P}'$ and $\Delta, \Delta_2 \vdash \bar{w} \triangleleft: [\bar{V}/\bar{Y}]\bar{U}'$ and we have $\Delta \vdash S \uparrow^{\Delta'_1, X \triangleleft: S'_i} C \langle \bar{T} \rangle$ where $\Delta'_1, X \triangleleft: T'_i = \Delta_1$. By Lemma A.9(2), $[\bar{V}/\bar{Y}]\bar{U}_0 \downarrow_{\Delta'_1, X \triangleleft: S'_i} V_0'$ and $\Delta \vdash V_0' \triangleleft: V_0$. The other case for $v_i = 0$ is similar (use Lemmas A.4 and A.9(1) instead of Lemmas A.2 and A.9(2), respectively).

Subcase: $w_i = -$ or $w_i = *$

Similar. ■

A.12 Lemma [Type Substitution Preserves Typing]: If both Δ_1 and Δ_2 have non-variable bounds and $\Delta_1, \bar{X} \triangleleft: \bar{N}, \Delta_2; \Gamma \vdash e \in T$ and $\Delta_1 \vdash \bar{U} \triangleleft: [\bar{U}/\bar{X}]\bar{N}$ where $\Delta_1 \vdash \bar{U}$ ok and none of \bar{X} appears in Δ_1 , then $\Delta_1, [\bar{U}/\bar{X}]\Delta_2; [\bar{U}/\bar{X}]\Gamma \vdash [\bar{U}/\bar{X}]e \in S$ for some S such that $\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash S \triangleleft: [\bar{U}/\bar{X}]T$.

Proof: By induction on the derivation of $\Delta_1, \bar{X} \triangleleft: \bar{N}, \Delta_2; \Gamma \vdash e \in T$ with a case analysis on the last rule used. In what follows, let $\Delta = \Delta_1, \bar{X} \triangleleft: \bar{N}, \Delta_2$.

Case T-VAR:

Trivial.

Case T-FIELD: $e = e_0 . f_i \quad \Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash bound_{\Delta}(T_0) \uparrow^{\Delta'} C \langle \bar{T} \rangle$
 $fields(C \langle \bar{T} \rangle) = \bar{U} \bar{f} \quad U_i \downarrow_{\Delta'} T$

By the induction hypothesis, $\Delta_1, [\bar{U}/\bar{X}]\Delta_2; [\bar{U}/\bar{X}]\Gamma \vdash [\bar{U}/\bar{X}]e_0 \in S_0$ and $\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash S_0 \triangleleft: [\bar{U}/\bar{X}]T_0$ for some S_0 . By Lemma A.6,

$$\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash bound_{\Delta_1, [\bar{U}/\bar{X}]\Delta_2}([\bar{U}/\bar{X}]T_0) \triangleleft: [\bar{U}/\bar{X}](bound_{\Delta_1, \bar{X} \triangleleft: \bar{N}, \Delta_2}(T_0)).$$

Then, by S-TRANS,

$$\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash bound_{\Delta_1, [\bar{U}/\bar{X}]\Delta_2}(S_0) \triangleleft: [\bar{U}/\bar{X}](bound_{\Delta_1, \bar{X} \triangleleft: \bar{N}, \Delta_2}(T_0)).$$

By Lemma A.10, $\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash bound_{\Delta}(S_0) \uparrow^{\Delta''} D \langle \bar{S} \rangle$ and $fields(D \langle \bar{S} \rangle) = \bar{S} \bar{f}, \dots$, and $S_i \downarrow_{\Delta''} T'$ and $\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash T' \triangleleft: [\bar{U}/\bar{X}]T$. Thus, by the rule T-FIELD, $\Delta_1, [\bar{U}/\bar{X}]\Delta_2; [\bar{U}/\bar{X}]\Gamma \vdash [\bar{U}/\bar{X}]e_0 . f_i \in T'$.

Case T-INVK: $e = e_0 . m \langle \bar{V} \rangle (\bar{e}) \quad \Delta; \Gamma \vdash e_0 \in T_0$
 $\Delta \vdash bound_{\Delta}(T_0) \uparrow^{\Delta'} C \langle \bar{T} \rangle \quad mtype(m, C \langle \bar{T} \rangle) = \langle \bar{V} \triangleleft \bar{P} \rangle \bar{W} \rightarrow W_0$
 $\{\bar{Y}\} \cap dom(\Delta') = \emptyset \quad \Delta \vdash \bar{V}$ ok $\quad \Delta, \Delta' \vdash \bar{V} \triangleleft: [\bar{V}/\bar{Y}]\bar{P}$
 $\Delta; \Gamma \vdash \bar{e} \in \bar{S} \quad \Delta, \Delta' \vdash \bar{S} \triangleleft: [\bar{V}/\bar{Y}]\bar{W} \quad [\bar{V}/\bar{Y}]W_0 \downarrow_{\Delta'} T$

By the induction hypothesis,

$$\begin{aligned} \Delta_1, [\bar{U}/\bar{X}]\Delta_2; [\bar{U}/\bar{X}]\Gamma \vdash [\bar{U}/\bar{X}]e_0 \in S_0 \\ \Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash S_0 <: [\bar{U}/\bar{X}]T_0 \end{aligned}$$

and

$$\begin{aligned} \Delta_1, [\bar{U}/\bar{X}]\Delta_2; [\bar{U}/\bar{X}]\Gamma \vdash [\bar{U}/\bar{X}]\bar{e} \in \bar{S}' \\ \Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash \bar{S}' <: [\bar{U}/\bar{X}]\bar{S} \end{aligned}$$

for some S_0 and \bar{S}' . By Lemmas A.6, A.5, A.4 and A.3, it is easy to show

$$\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash \text{bound}_{\Delta_1, [\bar{U}/\bar{X}]\Delta_2}(S_0) <: [\bar{U}/\bar{X}](\text{bound}_{\Delta}(T_0))$$

and

$$\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash [\bar{U}/\bar{X}]\bar{v} \text{ ok}$$

and

$$\begin{aligned} \Delta_1, [\bar{U}/\bar{X}](\Delta_2, \Delta') \vdash [\bar{U}/\bar{X}]\bar{v} <: [\bar{U}/\bar{X}][\bar{V}/\bar{Y}]\bar{P} \\ \Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash [\bar{U}/\bar{X}]\bar{S} <: [\bar{U}/\bar{X}][\bar{V}/\bar{Y}]\bar{W} \end{aligned}$$

and

$$\begin{aligned} \text{mtype}(m, [\bar{U}/\bar{X}]\mathbf{C} < \bar{T} >) = < \bar{Y} \triangleleft [\bar{U}/\bar{X}]\bar{P} > [\bar{U}/\bar{X}]\bar{W} \rightarrow [\bar{U}/\bar{X}]\mathbf{W}_0 \\ [\bar{U}/\bar{X}][\bar{V}/\bar{Y}]\mathbf{W}_0 \downarrow_{[\bar{U}/\bar{X}]\Delta'} [\bar{U}/\bar{X}]\mathbf{T} \end{aligned}$$

respectively.

Then, by Lemma A.11 and the fact that $[\bar{U}/\bar{X}][\bar{V}/\bar{Y}]\mathbf{T} = [[\bar{U}/\bar{X}]\bar{V}/\bar{Y}](\bar{U}/\bar{X})\mathbf{T}$ for any \mathbf{T} , we have

$$\begin{aligned} \Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash \text{bound}_{\Delta_1, [\bar{U}/\bar{X}]\Delta_2}(S_0) \uparrow^{\Delta''} \mathbf{D} < \bar{T}' > \\ \text{mtype}(m, \mathbf{D} < \bar{T}' >) = < \bar{Y} \triangleleft \bar{P}' > \bar{W}' \rightarrow \mathbf{W}_0' \\ \Delta_1, ([\bar{U}/\bar{X}]\Delta_2), \Delta'' \vdash [\bar{U}/\bar{X}]\bar{v} <: [[\bar{U}/\bar{X}]\bar{V}/\bar{Y}]\bar{W}' \\ \Delta_1, ([\bar{U}/\bar{X}]\Delta_2), \Delta'' \vdash [\bar{U}/\bar{X}]\bar{S} <: [[\bar{U}/\bar{X}]\bar{V}/\bar{Y}]\bar{W}' \\ [[\bar{U}/\bar{X}]\bar{V}/\bar{Y}]\mathbf{W}_0' \downarrow_{\Delta''} \mathbf{T}' \\ \Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash \mathbf{T}' <: [\bar{U}/\bar{X}]\mathbf{T}. \end{aligned}$$

By Lemma A.1 and the rule S-TRANS,

$$\Delta_1, [\bar{U}/\bar{X}]\Delta_2, \Delta'' \vdash \bar{S}' <: [[\bar{U}/\bar{X}]\bar{V}/\bar{Y}]\bar{W}'$$

and, by the rule T-METHOD,

$$\Delta_1, [\bar{U}/\bar{X}]\Delta_2 \vdash [\bar{U}/\bar{X}]e_0.\text{m} < \bar{V} > (\bar{e}) \in \mathbf{T}'$$

as required.

Case T-NEW, T-CAST, T-SCAST:

Easy. ■

A.13 Lemma [Term Substitution Preserves Typing]: If Δ has non-variable bounds and $\Delta; \Gamma, \bar{x} : \bar{T} \vdash e \in \mathbf{T}$ and $\Delta; \Gamma \vdash \bar{d} \in \bar{S}$ where $\Delta \vdash \bar{S} <: \bar{T}$, then $\Delta; \Gamma \vdash [\bar{d}/\bar{x}]e \in \mathbf{S}$ for some \mathbf{S} such that $\Delta \vdash \mathbf{S} <: \mathbf{T}$.

Proof: By induction on the derivation of $\Delta; \Gamma, \bar{x} : \bar{T} \vdash e \in \mathbf{T}$ with a case analysis on the last rule used.

Case T-VAR: $e = x$

The case where $x \in \text{dom}(\Gamma)$ is immediate, since $[\bar{d}/\bar{x}]x = x$. On the other hand, if $x = x_i$ and $T = T_i$, then $\Delta; \Gamma \vdash d_i \in S_i$ finishing the case.

Case T-FIELD: $e = e_0.f_i$ $\Delta; \Gamma, \bar{x} : \bar{T} \vdash e_0 \in T_0$ $\Delta \vdash \text{bound}_\Delta(T_0) \uparrow^{\Delta'} C \langle \bar{U} \rangle$
 $\text{fields}(C \langle \bar{U} \rangle) = \bar{S} \bar{f}$ $S_i \downarrow_{\Delta'} T$

By the induction hypothesis, $\Delta; \Gamma \vdash [\bar{d}/\bar{x}]e_0 \in S_0$ for some S_0 such that $\Delta \vdash S_0 \prec T_0$. By Lemma A.10, $\Delta \vdash \text{bound}_\Delta(S_0) \uparrow^{\Delta'} D \langle \bar{V} \rangle$, $\text{fields}(D \langle \bar{V} \rangle) = \bar{W} \bar{f}, \dots$, and $W_i \downarrow_{\Delta'} W_i'$ and $\Delta \vdash W_i' \prec T$. Therefore, by the rule T-FIELD, $\Delta; \Gamma \vdash [\bar{d}/\bar{x}]e_0.f_i \in W_i'$.

Case T-INVK: $e = e_0.m \langle \bar{V} \rangle (\bar{e})$ $\Delta; \Gamma, \bar{x} : \bar{T} \vdash e_0 \in T_0$
 $\Delta \vdash \text{bound}_\Delta(T_0) \uparrow^{\Delta'} C \langle \bar{T} \rangle$ $\text{mtype}(m, C \langle \bar{T} \rangle) = \langle \bar{Y} \langle \bar{P} \rangle \bar{U} \rightarrow U_0$
 $\{\bar{Y}\} \cap \text{dom}(\Delta') = \emptyset$ $\Delta \vdash \bar{V} \text{ ok}$ $\Delta, \Delta' \vdash \bar{V} \prec: [\bar{V}/\bar{Y}] \bar{P}$
 $\Delta \Gamma, \bar{x} : \bar{T} \vdash \bar{e} \in \bar{S}$ $\Delta, \Delta' \vdash \bar{S} \prec: [\bar{V}/\bar{Y}] \bar{U}$ $[\bar{V}/\bar{Y}] U_0 \downarrow_{\Delta'} T$

By the induction hypothesis, $\Delta; \Gamma \vdash [\bar{d}/\bar{x}]e_0 \in T_0'$ for some T_0' such that $\Delta \vdash T_0' \prec T_0$ and $\Delta; \Gamma \vdash [\bar{d}/\bar{x}]\bar{e} \in \bar{S}'$ for some \bar{S}' such that $\Delta \vdash \bar{S}' \prec \bar{S}$. By Lemma A.11, we have

$$\begin{array}{lll} \Delta \vdash \text{bound}_\Delta(T_0') \uparrow^{\Delta'} D \langle \bar{T}' \rangle & \text{mtype}(m, D \langle \bar{T}' \rangle) = \langle \bar{Y} \langle \bar{P}' \rangle \bar{U}' \rightarrow U_0' & \Delta, \Delta'' \vdash \bar{V} \prec: [\bar{V}/\bar{Y}] \bar{P}' \\ \Delta, \Delta'' \vdash \bar{S}' \prec: [\bar{V}/\bar{Y}] \bar{U}' & [\bar{V}/\bar{Y}] S_0' \downarrow_{\Delta'} T' & \Delta \vdash T' \prec T \end{array}$$

Therefore, by the rule T-METHOD, $\Delta; \Gamma \vdash [\bar{d}/\bar{x}]e \in T'$, finishing the case.

Case T-NEW, T-CAST, T-SCAST:

Easy. ■

A.14 Lemma: If $\text{mtype}(m, C \langle \bar{T} \rangle) = \langle \bar{Y} \langle \bar{P} \rangle \bar{U} \rightarrow U_0$ and $\text{mbody}(m \langle \bar{V} \rangle, C \langle \bar{T} \rangle) = \bar{x}.e_0$ with $\Delta \vdash C \langle \bar{T} \rangle \text{ ok}$ and $\Delta \vdash \bar{V} \text{ ok}$ and $\Delta \vdash \bar{V} \prec: [\bar{V}/\bar{Y}] \bar{P}$, then there exist some N and S such that $\Delta \vdash C \langle \bar{T} \rangle \prec: N$ and $\Delta \vdash N \text{ ok}$ and $\Delta \vdash S \prec: [\bar{V}/\bar{Y}] U_0$ and $\Delta; \bar{x} : [\bar{V}/\bar{Y}] \bar{U}, \text{this} : N \vdash e_0 \in S$.

Proof: By induction on the derivation of $\text{mbody}(m \langle \bar{V} \rangle, C \langle \bar{T} \rangle) = \bar{x}.e_0$.

Case MB-CLASS: $\text{class } C \langle \bar{X} \rangle \langle \bar{N} \rangle \langle D \langle \bar{S} \rangle \rangle \{ \dots \bar{M} \}$
 $\langle \bar{Y} \langle \bar{P}' \rangle \rangle U_0' m(\bar{U}' \bar{x}) \{ \text{return } e_0'; \} \in \bar{M}$

Then, $\text{mtype}(m, C \langle \bar{T} \rangle) = \langle \bar{Y} \langle ([\bar{T}/\bar{X}] \bar{P}') \rangle ([\bar{T}/\bar{X}] \bar{U}') \rangle ([\bar{T}/\bar{X}] U_0')$, that is, $\bar{P} = [\bar{T}/\bar{X}] \bar{P}'$ and $\bar{U} = [\bar{T}/\bar{X}] \bar{U}'$ and $U_0 = [\bar{T}/\bar{X}] U_0'$ and $e_0 = [\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] e_0'$. Now, let $\Gamma = \bar{x} : \bar{U}', \text{this} : C \langle \bar{X} \rangle$ and $\Delta' = \bar{X} \prec: \bar{N}, \bar{Y} \prec: \bar{P}$. By the rules T-CLASS and T-METHOD, we have $\Delta'; \Gamma \vdash e_0' \in S_0$ and $\Delta' \vdash S_0 \prec: U_0'$ for some S_0 . Since $\Delta \vdash C \langle \bar{T} \rangle \text{ ok}$, we have $\Delta \vdash \bar{T} \prec: [\bar{T}/\bar{X}] \bar{N}$ by the rule WF-CLASS. By Lemmas A.1, A.4, and A.12,

$$\Delta \vdash [\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] S_0 \prec: [\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] U_0'$$

and

$$\Delta; \bar{x} : [\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] \bar{U}', \text{this} : C \langle \bar{T} \rangle \vdash [\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] e_0 \in S_0'$$

where

$$\Delta \vdash S_0' \prec: [\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] S_0.$$

Letting $N = C \langle \bar{T} \rangle$ and $S = S_0'$ finishes the case. (Note that, without loss of generality, we can assume $[\bar{V}/\bar{Y}] [\bar{T}/\bar{X}] = [\bar{T}/\bar{X}] [\bar{V}/\bar{Y}]$.)

Case MB-SUPER: $\text{class } C\langle\bar{X}\langle\bar{N}\rangle\langle D\langle\bar{S}\rangle \{ \dots \bar{M} \} \quad m \notin \bar{M}$

Immediate from the induction hypothesis and the fact that $\Delta \vdash C\langle\bar{T}\rangle <: [\bar{T}/\bar{X}]D\langle\bar{S}\rangle$. \blacksquare

Proof of Theorem 6.4.1: By induction on the derivation of $e \longrightarrow e'$ with a case analysis on the reduction rule used.

Case R-FIELD: $e = \text{new } C\langle\bar{T}\rangle(\bar{e}) . f_i \quad \text{fields}(C\langle\bar{T}\rangle) = \bar{T} \bar{f} \quad e' = e_i$

By the rules T-FIELD, T-NEW, and O-REFL, we have

$$\begin{array}{l} \Delta; \Gamma \vdash \text{new } C\langle\bar{T}\rangle(\bar{e}) \in C\langle\bar{T}\rangle \quad \Delta \vdash C\langle\bar{T}\rangle \uparrow^\emptyset C\langle\bar{T}\rangle \\ \Delta; \Gamma \vdash \bar{e} \in \bar{S} \quad \Delta \vdash \bar{S} <: \bar{U} \quad U_i \Downarrow_\emptyset U_i (= T). \end{array}$$

In particular, $\Delta; \Gamma \vdash e_i \in S_i$ finishes the case.

Case R-INVK: $e = \text{new } C\langle\bar{T}\rangle(\bar{e}) . m\langle\bar{V}\rangle(\bar{d}) \quad \text{mbody}(m\langle\bar{V}\rangle, C\langle\bar{T}\rangle) = \bar{x} . e_0$
 $e' = [\bar{d}/\bar{x}, \text{new } C\langle\bar{T}\rangle(\bar{e})/\text{this}]e_0$

By the rules T-INVK, T-NEW, and O-REFL, we have

$$\begin{array}{l} \Delta; \Gamma \vdash \text{new } C\langle\bar{T}\rangle(\bar{e}) \in C\langle\bar{T}\rangle \quad \Delta \vdash C\langle\bar{T}\rangle \uparrow^\emptyset C\langle\bar{T}\rangle \quad \text{mtype}(m, C\langle\bar{T}\rangle) = \langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle \rightarrow U_0 \\ \Delta \vdash \bar{V} \text{ ok} \quad \Delta \vdash \bar{V} <: [\bar{V}/\bar{Y}]\bar{P} \quad \Delta; \Gamma \vdash \bar{d} \in \bar{S} \\ \Delta \vdash \bar{S} <: [\bar{V}/\bar{Y}]\bar{U} \quad [\bar{V}/\bar{Y}]U_0 \Downarrow_\emptyset [\bar{V}/\bar{Y}]U_0 (= T) \quad \Delta \vdash C\langle\bar{T}\rangle \text{ ok} \end{array}$$

By Lemma A.14, $\Delta; \bar{x} : [\bar{V}/\bar{Y}]\bar{U}$, $\text{this} : N \vdash e_0 \in S_0$ for some N and S_0 such that $\Delta \vdash C\langle\bar{T}\rangle <: N$ where $\Delta \vdash N \text{ ok}$, and $\Delta \vdash S_0 <: [\bar{V}/\bar{Y}]U_0$ where $\Delta \vdash S_0 \text{ ok}$. Then, by Lemmas A.1 and A.13, $\Delta; \Gamma \vdash [\bar{d}/\bar{x}, \text{new } C\langle\bar{T}\rangle(\bar{e})/\text{this}]e_0 \in T_0$ for some T_0 such that $\Delta \vdash T_0 <: S_0$. By the rule S-TRANS, we have $\Delta \vdash T_0 <: T$, finishing the case.

Case R-CAST, RC-CAST, RC-INV-ARG, RC-NEW-ARG:

Easy.

Case RC-FIELD: $e = e_0 . f_i \quad e' = e_0' . f_i \quad e_0 \longrightarrow e_0'$

By the rule T-FIELD, we have

$$\Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash \text{bound}_\Delta(T_0) \uparrow^{\Delta'} C\langle\bar{T}\rangle \quad \text{fields}(C\langle\bar{T}\rangle) = \bar{S} \bar{f} \quad S_i \Downarrow_{\Delta'} T$$

By the induction hypothesis, $\Delta; \Gamma \vdash e_0' \in T_0'$ for some T_0' such that $\Delta \vdash T_0' <: T_0$. By Lemma A.10, $\Delta \vdash \text{bound}_\Delta(T_0') \uparrow^{\Delta''} D\langle\bar{U}\rangle$, $\text{fields}(D\langle\bar{U}\rangle) = \bar{V} \bar{f}, \dots$, and $V_i \Downarrow_{\Delta''} V_i'$ and $\Delta \vdash V_i' <: T$. Therefore, by the rule T-FIELD, $\Delta; \Gamma \vdash e_0' . f \in V_i'$, finishing the case.

Case RC-INV-RECV: $e = e_0 . m\langle\bar{V}\rangle(\bar{e}) \quad e' = e_0' . m\langle\bar{V}\rangle(\bar{e}) \quad e_0 \longrightarrow e_0'$

By the rule T-INVK, we have

$$\begin{array}{l} \Delta; \Gamma \vdash e_0 \in T_0 \quad \Delta \vdash \text{bound}_\Delta(T_0) \uparrow^{\Delta'} C\langle\bar{T}\rangle \quad \text{mtype}(m, C\langle\bar{T}\rangle) = \langle\bar{Y}\langle\bar{P}\rangle\bar{U}\rangle \rightarrow U_0 \\ \Delta \vdash \bar{V} \text{ ok} \quad \Delta \vdash \bar{V} <: [\bar{V}/\bar{Y}]\bar{P} \quad \Delta \vdash \bar{e} \in \bar{S} \\ \Delta, \Delta' \vdash \bar{S} <: [\bar{V}/\bar{Y}]\bar{U} \quad [\bar{V}/\bar{Y}]U_0 \Downarrow_{\Delta'} T. \end{array}$$

By the induction hypothesis, $\Delta; \Gamma \vdash e_0' \in T_0'$ for some T_0' such that $\Delta \vdash T_0' <: T_0$. By Lemma A.11,

$$\begin{array}{l} \Delta \vdash \text{bound}_\Delta(T_0') \uparrow^{\Delta''} D\langle\bar{W}\rangle \quad \text{mtype}(m, D\langle\bar{W}\rangle) = \langle\bar{Y}\langle\bar{P}'\rangle\bar{U}'\rangle \rightarrow U_0' \quad \Delta \vdash \bar{V} <: [\bar{V}/\bar{Y}]\bar{P}' \\ \Delta, \Delta'' \vdash \bar{S} <: [\bar{V}/\bar{Y}]\bar{U}' \quad [\bar{V}/\bar{Y}]U_0' \Downarrow_{\Delta''} T' \quad \Delta \vdash T' <: T. \end{array}$$

Then, by the rule T-INVK, $\Delta; \Gamma \vdash e_0' . m\langle\bar{V}\rangle(\bar{e}) \in T'$, finishing the case. \blacksquare