On Variance-Based Subtyping for Parametric Types

Atsushi Igarashi[1] and Mirko Viroli[2]

[1] Graduate School of Informatics, Kyoto University
Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan
igarashi@kuis.kyoto-u.ac.jp
[2] DEIS, Università degli Studi di Bologna
via Rasi e Spinelli 176, 47023 Cesena (FC), Italy
mviroli@deis.unibo.it

Abstract. We develop the mechanism of *variant parametric types*, inspired by *structural virtual types* by Thorup and Torgersen, 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 visibility 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—which can be either `+`, `-`, or `*`, respectively—to a type argument. On the other hand, the type system prohibits certain method/field accesses through variant parametric types, when those accesses can 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 that it can accept containers of any subtype of string.
The theoretical issues are studied by extending Featherweight GJ—an existing core calculus for Java with generics—with variant parametric types. By exploiting the intuitive connection to bounded existential types, we develop a sound type system for the extended calculus.

1 Introduction

The recent development of high-level constructs for object-oriented languages is witnessing renewed interest in the design, implementation, and applications of parametric polymorphism. For instance, Java's designers initially decided to avoid generic features, and to provide programmers only with the inclusive 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's call for proposals for adding generics to the Java programming language [29] many extensions have been proposed ([5,13,36,25,1] to cite some); finally, Bracha, Stoutamire, Odersky, and Wadler's GJ [5] has been chosen as the reference implementation technique for the first upcoming release of Java with generics. More recently, an extension of Microsoft's .NET Common Language Runtime [26] (CLR) with generics has been studied [30]. Such an extension not only provides C$^{\#}$ with generic classes and methods but also has a potential impact to promote genericity as a key paradigm in the whole CLR framework, because, in principle, *all* the languages supported by CLR can be extended with generic features. In this scenario, it is clear that studying language constructs related to genericity is likely to play a key role in increasing the expressiveness of modern high-level languages such as Java and C$^{\#}$.

Following this research direction, 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.

In most current object-oriented languages—such as Java, C++, and C$^{\#}$—inclusive polymorphism is supported only through inheritance. Their extensions with generics allow a generic class to extend from another generic class [20], and introduce *pointwise* subtyping: for instance, provided that class `Stack<X>` is a subclass of `Vector<X>` (where `X` is a type parameter) a parametric type `Stack<String>` is a subtype of `Vector<String>`.

Historically, most of well-known attempts to introduce another subtyping scheme for generics were based on the notion of *variance*, which is used to define a subtype relation between different instantiations of the same generic class. Basically, a generic class `C<X>` is said to be *covariant* with respect to `X` if `S <: T` implies `C<S> <: C<T>` (where `<:` denotes the subtyping relation), and conversely, `C<X>` is said to be *contravariant* with respect to `X`, if `S <: T` implies `C<T> <: C<S>`. Also, `C<X>` is said to be *invariant* when `C<S> <: C<T>` is derived only if `S` = `T`. For the resulting type system to be sound, covariance and contravariance can be permitted under some constraints on the occurrences of type variable `X` within `C<X>`'s signature. For example, consider a generic collection class whose element type is abstracted as a type parameter. Typically, it can be covariant only if it is read-only, while it can be contravariant only if write-only. Thus, to make use of variance, one has to be more careful about the design of classes.

Recently, Thorup and Torgensen [33] briefly sketched how some flavors of programming with *structural virtual types*, which support safe covariance, can be simulated by an extension of parametric classes. The idea is to specify the variance of each type parameter when the type is *used*, rather than when the class is *declared*. For any type argument `T`, a parametric class `C<X>` induces two types: `C<T>`, which is invariant, and `C<+T>`, which is covariant; in exchange for covariance, certain (potentially unsafe) member accesses through `C<+T>` are forbidden. Thus, it is expected that class designers are released from the burden of taking variance into account, and moreover class reuse is promoted. Unfortunately, it has not been rigorously discussed how this idea works in a full-blown language design.

Our contribution is to generalize this approach, and develop it into the mechanism of *variant parametric types* for possible application to widely disseminated languages such as Java and C$^{\#}$. In particular, we study their basic design and usefulness. Moreover, in order to discuss their fundamental concepts and subtleties rigorously, we introduce a core language based on Featherweight GJ, a generic version of Featherweight Java [20], with a formalization of its syntax, type system, and operational semantics, along with a proof of type soundness.

Variant parametric types are a new form of parametric types, in which any type parameter can come with one of the following variance annotation symbols: `+` for covariance, `-` for contravariance, and `*` for *bivariance*—`C<*T>` is a subtype of `C<*R>` whatever `T` and `R` are. The type argument without an annotation is considered invariant[1]. While types that are invariant in every type argument are used for run-time types of objects, variant parametric types are used for field types, method (argument and return) types, and local variable types.

Roughly speaking, a variant parametric type can be viewed as a set of run-time types: for instance, type `Vector<+Number>` is the type of all those vectors whose elements can be given type `Number`, including `Vector<Number>`, `Vector<Float>`, `Vector<Integer>`, and so on. Thus, both subtype relations `Vector<Number>` <: `Vector<+Number>` and `Vector<Float>` <: `Vector<+Number>` hold (note that `Vector<Float>` is *not* a subtype of `Vector<Number>`). On the other hand, since no hypothesis can be made on what kinds of elements can be stored in a vector of type `Vector<+Number>`, the invocation of methods inserting elements in the vector is statically forbidden. With this language construct, for example, the applicability of those methods accepting an argument of type `Vector<Number>` and only reading its elements can be safely widened by extending the argument type to `Vector<+Number>`, enabling the method to accept generic vectors instantiated with any subtype of `Number`. As a result, while standard variance is (typically) constrained to read-only and write-only generic collection classes, variant parametric types are available for every class; they provide uniform views of different instantiations of a generic class and can be exploited by those code abstractions that access a collection only by reading or by writing its elements.

We also show how some technical subtleties come in, especially when variant parametric types are arbitrarily nested. It seems that in previous work on variance such as [14], such cases are not systematically addressed even though they often arise in practice, making the whole safety argument not very convincing. The key intuition to tackle the subtleties is similarity between variant parametric types and bounded existential types [12]. In the above example of `Vector<+Number>`, the element type is only known to be *some* subtype of `Number`. In this sense, `Vector<+Number>` would correspond to type `∃X<:Number.Vector<X>`. In fact, this intuitive connection is exploited in the whole design of a sound type system for variant parametric types.

[1] In [14], the term "bivariant" was used to mean what here we refer to as "invariant." We decide to adopt the term invariant since we believe it is standard and more appropriate to denote the cases where type arguments cannot be different for an instantiation to be a subtype of another.

The remainder of the paper is organized as follows. In Section 2, the classical approach to variance of parametric types 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, comparing their expressiveness to that of parametric methods in conjunction with bounded polymorphism. Section 5 elaborates the interpretation of variant parametric types as a form of bounded existential types, discussing the subtleties of our language extension. Section 6 presents the core calculus for variant parametric types and provides a sound type system for it. Section 7 discusses related work and section 8 presents concluding remarks and perspectives on future work. For brevity, a detailed proof of type soundness is omitted in this paper; interested readers are referred to a full version of this paper, which will be available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers.html`.

2 Classical Approach to Variance for Parametric Classes

Historically, one main approach to flexible inclusive polymorphism for generics was through the mechanism called *variance*, which is briefly reviewed in this section. The limitation of the approach is also discussed along with a proposed solution, which inspired us to develop our variant parametric types.

A generic class `C<X>` is said to be *covariant* in the type parameter `X` when the subtype relation `C<R>` <: `C<S>` holds if `R` <: `S`. Conversely, `C<X>` is said to be *contravariant* in `X` when `C<R>` <: `C<S>` holds if `S` <: `R`. Less general notions of *bivariance* and *invariance* can be defined as well. `C<X>` is said to be *bivariant* in `X` when `C<R>` <: `C<S>` for any `R` and `S`. `C<X>` is said to be *invariant* in `X` when `C<R>` <: `C<S>` holds only when `R` = `S`. 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 a variance property, as the array types in Java show us. 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 covariant in that type—e.g., `String[]` is a subtype of `Object[]`. However, since arrays provide the operation to update their content, this can lead to a run-time error, as the following Java code shows:

```
Object[] o=new String[]{"1","2","3"};
o[0]=new Integer(1);  // Exception thrown
```

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: at run time, the bytecode interpreter tries to insert an `Integer` instance to where a `String` instance is actually expected.

To recover safety, previous work [14,2,3] proposed to pose restriction on how a type variable can appear in a class definition, according to its variance. In the case of a class 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 `X` appears as the argument type in `setFst` and `Y` appears as the return type in `getSnd`. It is easy to see that any type `Pair<R,S>` can be safely considered a subtype of `Pair<String,Number>` when `R` is a supertype of `String` and `S` 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, as claimed e.g. in [15], the applicability of this mechanism is considered not very wide, since type variables typically occur in such positions that forbid both covariance and contravariance. In fact, 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>`:

```
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. As a result, 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. As a result, class designers have to be responsible for the tradeoff between variance and available functionality of a class.

More recently, Thorup and Torgersen [33] have briefly sketched a possible solution to the applicability limitations of this classical approach. The idea is to let a programmer specify within parametric types whether he/she wants the type argument to be invariant or covariant: for the latter case, 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, the choice of the variance property can be deferred from when a class is defined to when a class is used to derive a type. It seems that two advantages arise from this approach: the applicability of the mechanisms is widened—e.g., from read-only containers to read-write ones—and the designers of libraries are generally released from the burden of making decisions about the tradeoff mentioned above. Unfortunately, aside from the fact that they dealt with only covariance, the informal idea has been not fully explored so far. So, we generalize this idea to include contravariance and bivariance, and investigate how it works in a full-fledged language design.

3 Variant Parametric Types

Now, based on the idea sketched in [33], we introduce the notion of *variant parametric types* and their basic aspects.

Variant parametric types are a generalization of standard parametric (or generic) types where each type parameter may be associated with a *variance annotation*, either `+`, `-`, or `*`, respectively referred to as the *covariance*, *contravariance*, or *bivariance annotation symbol*. Syntactically, a variance annotation symbol precedes the type parameter it refers to and introduces the corresponding variance to the argument position: for example, `Vector<+String>` is a subtype of `Vector<+Object>`. Similarly to the case of generic types, the type parameter of a variant parametric type can be either a variant parametric type or a type variable, as in `Vector<+String>`, `Pair<+String,-Integer>`, and `Vector<Vector<*X>>`. A parametric type where no (outermost) type argument has a variance annotation is called a *fully invariant type* (or just an invariant type). `Vector<String>` and `Pair<Vector<+String>,Integer>` are examples of invariant types. Since the type argument following `*` does not really matter, due to bivariance, it is often omitted by simply writing e.g. `Vector<*>`, which is also the syntax we propose for an actual language extension.

Unlike the standard mechanism of variance, 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`

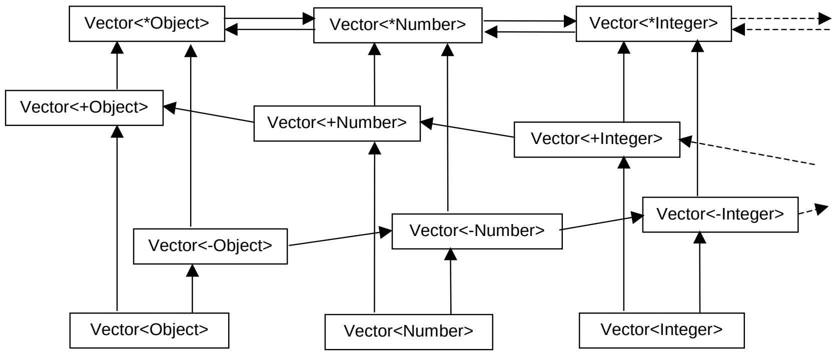

Fig. 1. Subtyping graph of variant parametric types

is declared in `Vector`, `Vector<+Number>` can be used in a program; the type system just forbids accessing the method `setElementAt` through the covariant type `Vector<+Number>`.

A simple interpretation of variant parametric types is given as a set of invariant types. A type `C<+T>` can be interpreted as the set of all invariant types of the form `C<S>` where `S` is a subtype of `T`; a type `C<-T>` can be interpreted as the set of all invariant types of the form `C<S>` where `S` is a supertype of `T`; and, a type `C<*T>` can be interpreted as the set of all invariant types of the form `C<S>`. Hence, `Vector<+Integer>` <: `Vector<+Number>` directly follows as an inclusion of the set 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, `Vector<Integer>` <: `Vector<+Integer>` and `Vector<Integer>` <: `Vector<-Integer>` hold. Similarly, `Vector<+Integer>` <: `Vector<*Integer>` and `Vector<-Integer>` <: `Vector<*Integer>` hold as well. 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`.)

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.

– The following subtype relations hold that involve variant parametric types differing just in the variance annotation symbol on one type parameter[2]:

```
C<..., T,...> <: C<...,+T,...>
C<..., T,...> <: C<...,-T,...>
C<...,+T,...> <: C<...,*T,...>
C<...,-T,...> <: C<...,*T,...>
```

[2] More precisely, e.g. the first relation means that for any types $T_1, T_2, \ldots, T_n$ we have $C<T_1, \ldots, T_{i-1}, T, T_{i+1}, \ldots, T_n>$ <: $C<T_1, \ldots, T_{i-1}, +T, T_{i+1}, \ldots, T_n>$, and similarly for the others.

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

```
C<..., S,...> <: C<..., T,...> if S <: T and T <: S
C<...,+S,...> <: C<...,+T,...> if S <: T
C<...,-S,...> <: C<...,-T,...> if T <: S
C<...,*S,...> <: C<...,*T,...> ∀ S, T
```

(Note that the subtyping relation is *not* anti-symmetric: `Vector<*Object>` and `Vector<*Integer>` 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.

Subtyping of variant parametric types which are instantiations of different generic classes—that is, involving inheritance—is more subtle than it might have been expected, and is discussed in Section 5.

Objects are created from generic classes through an expression of the form `new` $\texttt{C<T}_1\texttt{,...,T}_n\texttt{>(...)}$, without specifying any variant annotation in the (outermost) type arguments (variance annotations can appear inside $\texttt{T}_i$.) Objects created through this expression are given the corresponding invariant parametric type $\texttt{C<T}_1\texttt{,..,T}_n\texttt{>}$. While an invariant parametric type denotes a singleton set and is used for the run-time type of an object, a variant parametric type can be generally used as a common supertype for many different instantiations of the same generic class.

A more refined view of variant parametric types is given as a correspondence to bounded existential types [12], originally used for modeling partially abstract types. Actually, this view, exploited throughout the development of the type system, explains how access restriction is achieved. Intuitively, the covariant type `Vector<+Number>` would correspond to the existential type `∃X<:Number.Vector<X>`, read "`Vector<X>` for *some* `X` which is a subtype of `Number`." Then, an invocation of `setElementAt` on an expression of type `Vector<+Number>` is forbidden because the first argument type is `X`, which is known as some unknown subtype of `Number`, but the actual argument cannot be given a subtype of `X`. Similarly, `Vector<-Number>` would correspond to `∃X:>Number.Vector<X>`, where only the lower bound of the element type is known. Thus, invocation of `getElementAt` is not allowed because its return type would be an abstract type `X`, which cannot be promoted to a concrete type. Finally, `Vector<*>` would correspond to the unbounded existential type `∃X.Vector<X>`, which prevents both `getElementAt` and `setElementAt` from being invoked. Actually, 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 result the top type. However, we decided to disallow it so that member access restrictions and method/field typing becomes much the same as expected for the classical approach, discussed in the previous section.

The intuitive connection is further explored in Section 5 to deal with more complicated use of type variables, in particular when they appear inside parametric types, and to deal with inheritance-based subtyping—type variables naturally appear inside the type specified as the superclass in the `extends` clause.

In summary, variant parametric types provide uniform views over different instantiations of the same generic class. Unlike the standard mechanism reviewed in the previous section, variant parametric types are just a means for static access control, so there is no additional constraints on how classes are declared.

4 On Applicability and Usefulness

In this section, the usefulness of variant parametric types is studied by focusing on those generic classes representing collections—such as classes `Pair<X,Y>` (completed with methods `setSnd` and `getFst`), and `Vector<X>` defined in previous sections—which actually form one of the basic and significant application cases for generic classes.

4.1 Covariance

As discussed in the previous section, type `Vector<+T>` is a supertype of any type `Vector<R>` if `R` as a subtype of `T`, and can be interpreted as the type of all those vectors whose extracted elements can be given type `T`. Hence, the invocation of method `setElementAt` is forbidden on an object with type `Vector<+T>`, while method `getElementAt` can be invoked that returns an object of type `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<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(vi,0); // Permitted for Vector<Integer> <: Vector<+Number>
vn.fillFrom(vf,10);// Permitted for Vector<Float> <: Vector<+Number>
```

Here the method `fillFrom` accepts a vector which is meant to be only read, and whose elements are expected to be given type `X`. As a result, instead of simply declaring formal argument `v` to have type `Vector<X>`, it is more convenient to use type `Vector<+X>`. The corresponding method execution is always safe since the only method invoked on `v` is `getElementAt`, and even more, the applicability of `fillFrom` is extended to a wider set of vectors. In the code above a vector of numbers is filled with elements of vectors of integers and floats. So, in general, the covariant parameterization structure can be exploited to widen the applicability of methods that take a collection and simply read its elements.

Now that we have a mechanism to deal with different instantiations of the same generic class in a uniform way, one may be willing to rely on nested parameterizations so as to further exploit the flavors of collections classes, as in the following method `fillFromVector`:

```
class Vector<X extends Object>{
   ...
   void fillFromVector(Vector<+Vector<+X>> vv){
       int pos=0;
       for (int i=0;i<vv.size();i++){
           Vector<+X> v = vv.getElementAt(i);
           if (pos+v.size()>=size()) break;
           fillFrom(v,pos);
           pos += v.size();
}  }   }
...
Vector<Number> vn = new Vector<Number>(20);
Vector<Integer> vi = new Vector<Integer>(10);
Vector<Float> vf = new Vector<Float>(10);
Vector<Vector<+Number>> vvn = new Vector<Vector<+Number>>(2);
vvn.setElementAt(vi,0);
vvn.setElementAt(vf,1);
vn.fillFromVector(vvn);
```

The method `fillFromVector` takes a vector of vectors and puts its inner-level elements into the vector on which the method is invoked. Since neither the outer vector nor the inner vectors are updated, the argument can be given type `Vector<+Vector<+X>>`. The inner `+`, which means that the inner vectors are read-only, allows different inner vectors to be mixed in one vector: as in the code above, a vector of integers and a vector of floats are put in one vector `vvn`. The outer `+`, which means that the outer vector is also read-only, allows inner vectors passed to `fillFromVector` to extend class `Vector`, as in the following code:

```
class MyVector<X extends Object> extends Vector<X> { ... }
...
Vector<MyVector<+Number>> mvv = ...;
vn.fillFromVector(mvv);
```

4.2 Contravariance

Contravariance has a dual kind of use. Type `Vector<-T>` is a supertype of any type `Vector<R>` if `T` is a subtype of `R`, and can be interpreted as the type of all those vectors which is safe to fill with elements of type `T`. Hence, the invocation of method `getElementAt` on an object of type `Vector<-T>` is forbidden, while method `setElementAt` can be invoked passing an object of type `T`. The following method `fillTo` provides an example similar to the one shown for the covariance case:

```
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<size() && i+start()<v.size();i++)
           v.setElementAt(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);
vi.fillTo(vn,0); // Permitted for Vector<Number> <: Vector<-Integer>
vf.fillTo(vn,10);// Permitted for Vector<Number> <: Vector<-Float>
```

Mixed variance parameterizations can be useful as well. For instance, consider a method `fillToVector` that inserts elements of the receiver vector `this` into a structure of the kind `Vector<Vector<X>>` provided as input—which is a dual case with respect to the method `fillFromVector`. In this case, the method's formal argument `vv` should be safely given type `Vector<+Vector<-X>>`. There, the outer vector is just used to access its elements through method `getElementAt`—hence it can be safely declared covariant—while the inner vectors may only be updated through method `setElementAt`—so they can be safely declared contravariant.

4.3 Bivariance

From a conceptual point of view, the construct of bivariant parametric types comes for free once covariance and contravariance are defined. In fact, for the same reason why one may need to denote by `Vector<+T>` the supertype of each `Vector<R>` with `R <: T`, and conversely for contravariance, then it may also be the case to explicitly denote the supertype of both `Vector<+T>` and `Vector<-T>`, which is represented here by `Vector<*T>`—or more concisely, by `Vector<*>`.

No methods that contain the type variable `X` in their signature can be invoked on an expression of `Vector<*>`, so `Vector<*>` is meant to represent a sort of "frozen" vector, which can be neither read nor written. For instance, bivariance can be exploited to build a method 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.getElementAt(i).size();
   return sz;
}
```

More useful are those cases where a generic class involves more than one type parameter. Consider the following method:

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

Since method `fillFromFirst` does not actually read or write the second element in each pair, its type can be any type, so annotation symbol `*` can be used in place of it. This example suggests an interesting application for bivariance as a mechanism providing a parametric type's partial instantiation.

4.4 Comparison with Parametric Methods

One may wonder if parametric methods with bounded polymorphism can be used for the examples shown above; indeed, some of them can be easily handled with parametric methods. 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. 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 to a parametric methods.

Although they can be used almost interchangeably for some cases, we think variant parametric types and parametric methods are complementary mechanisms. On one hand, variant parametric types provides a uniform viewpoint over different instantiations of the same generic classes, making it possible to mix different types in one data structure, as we have seen in `Vector<+Vector<+X>>` and `Vector<+Vector<-X>>`. On the other hand, parametric methods can better express type dependency among method arguments and results, as in the two methods below:

```
// 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 method `swapElementAt` requires the arguments to be vectors carrying the same type of elements, while in method `join` variables `X`, `Y`, and `Z` 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 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, in general, this will force programmers to always write that kind of methods in the container class receiving 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 the argument and projecting on the left side. The use of variance widens its applicability, *(i)* abstracting from the type of the right side, *(ii)* allowing the left side to be a subtype of the expected type `X`, and *(iii)* also allowing subclasses of `Pair` to be used for zipping. We leave further studies on the combination of parametric methods and variance for future work.

5 Variant Parametric Types and Existential Types

In this section, we further investigate the informal connection between variant parametric types and (bounded) existential types [12], used to describe partially abstract types. The basic intuition was that, for example, `Vector<+Nm>` (in what follows, `Number` is often abbreviated to `Nm` for conciseness) would correspond to the bounded existential type `∃X<:Nm.Vector<X>`. Since only the upper bound `Nm` of `X` is known and its identity is unknown, a method that takes `X` as an argument cannot be invoked, while the field of type `X` can be accessed and given type `Nm`. Similarly, the type `Vector<*T>`—abbreviated to `Vector<*>`—would correspond to the unbounded existential type `∃X.Vector<X>`, and the type `Vector<-Nm>` to the bounded existential type `∃X:>Nm.Vector<X>`, where the abstract type `X` has a *lower* bound. For a nested parametric type, the existential quantifier comes inside, too: for example, `Vector<+Vector<+Nm>>` would correspond to `∃X<:(∃Y<:Nm.Vector<Y>).Vector<X>`.

5.1 Existential Types *à la* Mitchell and Plotkin

We first begin with reviewing the standard formulation of (bounded) existential types [27]. A value of an existential type `∃X<:S.T` is considered an ADT package, consisting of a pair of a hidden witness (or implementation) type `U`, which is a subtype of `S`, and an ADT implementation `v` of type [`U`/`X`]`T`. The expression `pack [U,v] as ∃X<:S.T` creates such a package; the expression `open o as [X,x] in b` unpacks a package `o` and binds the type variable `X` and the value variable `x` to the witness type and the implementation, where their scope is `b`. The typing rules for those expressions are informally written as follows:

$$\frac{\vdash \mathtt{U} <: \mathtt{S} \qquad \vdash \mathtt{v} \in [\mathtt{U}/\mathtt{X}]\mathtt{T}}{\vdash (\mathtt{pack\ [U,v]\ as\ }\exists\mathtt{X<:S.T}) \in \exists\mathtt{X<:S.T}}$$

$$\frac{\vdash \mathtt{o} \in \exists\mathtt{X<:S.T} \qquad \mathtt{X<:S}, \mathtt{x}:\mathtt{T} \vdash \mathtt{b} \in \mathtt{U} \qquad \mathtt{X} \notin FV(\mathtt{U})}{\mathtt{open\ o\ as\ [X,x]\ in\ b} \in \mathtt{U}}$$

Note that the side condition requires `X` not to be a free variable of `U`; otherwise the hidden type `X` would escape the abstraction. The following rule describes subtyping between bounded existential types[3]:

$$\frac{\vdash \mathtt{S}_1 <: \mathtt{S}_2 \qquad \mathtt{X<:S}_1 \vdash \mathtt{T}_1 <: \mathtt{T}_2}{\vdash \exists\mathtt{X<:S}_1.\mathtt{T}_1 <: \exists\mathtt{X<:S}_2.\mathtt{T}_2}$$

[3] This rule is known as one for the full variant of System $F_{\leq}$, in which subtyping is undecidable. Subtyping of variant parametric types, though, is easily shown to be decidable: they correspond to rather restricted forms of bounded existential types.

5.2 Variant Parametric Types as Existential Types

Variant parametric types can be explained in terms of the formulation above. For example, covariant and contravariant subtyping directly corresponds to subtyping mentioned above. On the other hand, when a variance annotation is introduced, as in `Vector<Nm>` <: `Vector<+Nm>`, it is considered a packing `pack [Nm,v] as ∃X<:Number.Vector<X>`.

When an operation (a field access or a method invocation) is performed on a variant parametric type, it is considered as if `open` were implicitly inserted. Consider the following class

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

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

and assume `x` is given type `Pair<+Nm,-Nm>`, which corresponds to `∃X<:Nm,Y:>Nm.Pair<X,Y>`. Then, for example, `x.setFst(e)` corresponds to `open x as [Z,W,y] in y.setFst(e)`. Inside `open`, method/field types can be easily obtained by simply replacing the type parameters of a class with the actual type arguments of the opened type. For example, the argument type of `setFst` is `Z`, that of `setSnd` is `W`, and that of `copyFst` is `Pair<Z,+Object>`. Similarly, 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>>`. Now, it turns out that `open x as [Z,W,y] in y.setFst(e)` is not well typed for any expression `e`: the argument type of `setFst` is `Z`, which is constrained by `Z<:Nm`, but the type of `e` cannot be a subtype of `Z` [4]. 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` <: `W`. Thus, `+` and `-` can be used to protect particular fields from being written or read. For much the same reason as `setFst`, the method call `p.copyFst(new Pair<Integer,Float>(...))` is not allowed because the argument type of `copyFst` would be `Pair<Z,*>`, which is not a super type of `Pair<Integer,Float>`. Actually, it *should not* be allowed because `p` may be bound to `new Pair<Float,Object>` and executing the expression above

[4] One exception could be `null`, which could be given the bottom type, which is a subtype of *any* type (even of unknown abstract type).

will assign an `Integer` to the field that holds `Float`. This example shows that a method type cannot be obtained by naively substituting for `X` the type argument `+Nm` together with a variance annotation; it would lead to a wrong argument type `Pair<+Nm,+Object>`, a supertype of `Pair<Integer,Float>`. In the next section, we formalize transformation from variant parametric types to invariant types with constrained abstract types as the operation *open*: for example, `Pair<+Nm,-Nm>` is opened to `Pair<Z,W>` with `Z<:Nm` and `W:>Nm`.

If a type of a body of unpacking includes the abstract type variables, the type of the whole expression has to be promoted to the least supertype without them, in order for abstract types not to escape from their scope. When such promotion is not possible, the expression is not typed. For example, consider `p.getFst()`, whose return type is `Z` with a constraint `Z<:Nm`. The type of the whole expression is the least supertype of `Z` without mentioning `Z`, i.e., its upper bound `Nm`. On the other hand, `p.getSnd` is not typeable because the return type `W` cannot be promoted to a supertype without `W`. Similarly, the return type of `p.reverse()` 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`.

It looks as if the type arguments and variances `+Nm` and `-Nm` were substituted for `X` and `Y`, respectively, but this naive view is wrong as we will see in the next example `p.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>>`! That's 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. The correct supertype is a covariant type `Pair<+Pair<+Nm,-Nm>,+Pair<+Nm,-Nm>>`; in general, `+` is attached to everywhere the type argument is changed by promotion. So, naive substitution of an actual type argument with its variance for a type variable does not work, again. 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 [17]; it was introduced to omit a type annotation for the open expression.

The notions of opening and closing are also used for deriving inheritance-based subtyping. Suppose we declare two subclasses of `Pair`.

```
class PairX<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,

```
PairX<Integer> <: Pair<Integer,Integer>
```

and

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

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

5.3 Comparison with Explicit Use of Existential Types

As a last remark, it would be interesting to see the full power of existential types. By using unpacking appropriately, more expressions are typeable. For example, suppose `x` is given type `∃X.Vector<X>` and a programmer wants to get an element from `x` and put it to `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 `open`s, 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 can easily turn programming into a cumbersome task.

6 Core Calculus for Variant Parametric Types

We introduce a core calculus for class-based object-oriented languages with variant parametric types. Our calculus is considered a variant of Featherweight GJ (FGJ for short) by Igarashi, Pierce, and Wadler [20], 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. For simplicity, polymorphic methods, found in FGJ, are omitted so as to focus on the basic mechanism of variant parametric types.

$\mathtt{N} ::= \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}$	variant parametric types
$\mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$	types
$\mathtt{v} ::= \mathtt{o} \mid \mathtt{+} \mid \mathtt{-} \mid \mathtt{*}$	variance annotations
$\mathtt{L} ::= \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ }\overline{\mathtt{M}}\mathtt{\ \}}$	class definitions
$\mathtt{M} ::= \mathtt{T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}}$	method definitions
$\mathtt{e} ::= \mathtt{x}$	variables
$\mid\ \mathtt{e.f}$	field access
$\mid\ \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)}$	method invocation
$\mid\ \mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$	object instantiation
$\mid\ \mathtt{(T)e}$	typecasts

Fig. 2. Syntax

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 $\overline{\mathtt{f}}$ as shorthand for a possibly empty sequence $\mathtt{f}_1,\ldots,\mathtt{f}_n$ (and similarly for $\overline{\mathtt{C}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{e}}$, etc.) and write $\overline{\mathtt{M}}$ as shorthand for $\mathtt{M}_1\ldots\mathtt{M}_n$ (with no commas). We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{\mathtt{x}}$ is written $|\overline{\mathtt{x}}|$. We abbreviate operations on pairs of sequences in the obvious way, writing "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" as shorthand for "$\mathtt{C}_1\ \mathtt{f}_1,\ldots,\mathtt{C}_n\ \mathtt{f}_n$" and "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}$" as shorthand for "$\mathtt{C}_1\ \mathtt{f}_1\mathtt{;}\ldots\mathtt{C}_n\ \mathtt{f}_n\mathtt{;}$", "$\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}$" as shorthand for "$\mathtt{<X}_1 \triangleleft \mathtt{N}_1,\ldots,\mathtt{X}_n \triangleleft \mathtt{N}_n\mathtt{>}$", and "$\mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}$" for "$\mathtt{C<v}_1\mathtt{T}_1\mathtt{,}\ldots\mathtt{,v}_n\mathtt{T}_n\mathtt{>}$". Sequences of field declarations, parameter names, type variables, and method declarations are assumed to contain no duplicate names. The empty brackets <> are often omitted for conciseness. We denote by $\mathtt{m} \not\in \overline{\mathtt{M}}$ that a method of the name m is not included in $\overline{\mathtt{M}}$. We introduce another variance annotation o to denote invariant, 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 $\mathtt{o} \leq \mathtt{+} \leq \mathtt{*}$ and $\mathtt{o} \leq \mathtt{-} \leq \mathtt{*}$. We write $\mathtt{v}_1 \vee \mathtt{v}_2$ for the least upper bound of $\mathtt{v}_1$ and $\mathtt{v}_2$. If every $\mathtt{v}_i$ is o in a variant type $\mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}$, we call it an *invariant type* and abbreviates to $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$.

A class declaration consists of its name (class C), type parameters ($\overline{\mathtt{X}}$) with their (upper) bounds ($\overline{\mathtt{N}}$), fields ($\overline{\mathtt{T}}\ \overline{\mathtt{f}}$), and methods ($\overline{\mathtt{M}}$)[5]; moreover, every class

[5] We 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.

must explicitly declare its supertype $\mathtt{D}$<$\overline{\mathtt{S}}$> 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 $\overline{\mathtt{N}}$ of type variables $\overline{\mathtt{X}}$ can contain $\overline{\mathtt{X}}$ in them. 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 $\mathtt{C}$<$\overline{\mathtt{T}}$>. 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 [36]. 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 $(CT, \mathtt{e})$ 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 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) $CT(\mathtt{C}) =$ `class C...` for every $\mathtt{C} \in dom(CT)$; (2) $\mathtt{Object} \notin dom(CT)$; (3) for every class name `C` (except `Object`) appearing anywhere in CT, we have $\mathtt{C} \in dom(CT)$; and (4) there are no cycles in the reflexive and 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 $CT(\mathtt{C}) =$ `class C ...`, for conciseness.

6.2 Type System

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

The fields of an invariant type $\mathtt{C}$<$\overline{\mathtt{T}}$>, written *fields*($\mathtt{C}$<$\overline{\mathtt{T}}$>), are a sequence of corresponding types and field names, $\overline{\mathtt{S}}\ \overline{\mathtt{f}}$. In what follows, we use the notation $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$ for a substitution of $\mathtt{T}_i$ for $\mathtt{X}_i$. The type of the method invocation `m` at an invariant type $\mathtt{C}$<$\overline{\mathtt{T}}$>, written *mtype*($\mathtt{m}, \mathtt{C}$<$\overline{\mathtt{T}}$>), returns a pair $\overline{\mathtt{U}}$->$\mathtt{U}_0$ of argument types $\overline{\mathtt{U}}$, and a result type $\mathtt{U}_0$.

A *type environment* Δ is a finite mapping from type variables to pairs of a variance except `o` (that is, either `+`, `-`, or `*`) and a type. We write $dom(\Delta)$ for the domain of Δ. When $\mathtt{X} \notin dom(\Delta)$, we write $\Delta, \mathtt{X} : (\mathtt{v}, \mathtt{T})$ for the type environment

Field lookup:

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{U}}\mathtt{>\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathit{fields}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{U}}\mathtt{>}) = \overline{\mathtt{V}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{V}}\ \overline{\mathtt{g}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{U}_0\ \mathtt{m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\mathtt{->}\mathtt{U}_0)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>})}$$

Open:

$$\frac{\begin{array}{l}\text{if } (\mathtt{v}_i, \mathtt{T}_i) \neq (\mathtt{w}_i, \mathtt{U}_i), \text{then}\\ \quad \mathtt{w}_i = \mathtt{o} \qquad \mathtt{U}_i = \mathtt{X}_i \notin \mathit{dom}(\Delta) \cup \{\overline{\mathtt{U}}\} \setminus \{\mathtt{U}_i\} \qquad \Delta'(\mathtt{X}_i) = (\mathtt{v}_i, \mathtt{T}_i)\end{array}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>} \Uparrow^{\Delta'} \mathtt{C<}\overline{\mathtt{wU}}\mathtt{>}}$$

Close:

$$\frac{\Delta(\mathtt{X}) = (\mathtt{+}, \mathtt{T})}{\mathtt{X} \Downarrow_{\Delta} \mathtt{T}} \qquad \frac{\mathtt{X} \notin \mathit{dom}(\Delta)}{\mathtt{X} \Downarrow_{\Delta} \mathtt{X}} \qquad \frac{\overline{\mathtt{T}} \Downarrow_{\Delta} \overline{\mathtt{T}}' \qquad \mathtt{w}_i = \begin{cases} \mathtt{v}_i & \text{if } \mathtt{T}_i = \mathtt{T'}_i \\ \mathtt{+}\vee\mathtt{v}_i & \text{otherwise}\end{cases}}{\mathtt{C<}\overline{\mathtt{vT}}\mathtt{>} \Downarrow_{\Delta} \mathtt{C<}\overline{\mathtt{wT}'}\mathtt{>}}$$

Subtyping:

$$\Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{T} \qquad \frac{\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \qquad \Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{U}}{\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{U}} \qquad \frac{\Delta(\mathtt{X}) = (\mathtt{+}, \mathtt{T})}{\Delta \vdash \mathtt{X} \mathrel{<:} \mathtt{T}} \qquad \frac{\Delta(\mathtt{X}) = (\mathtt{-}, \mathtt{T})}{\Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{X}}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\}} \qquad \Delta \vdash \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>} \Uparrow^{\Delta'} \mathtt{C<}\overline{\mathtt{U}}\mathtt{>} \qquad ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}) \Downarrow_{\Delta'} \mathtt{T}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>} \mathrel{<:} \mathtt{T}}$$

$$\frac{\overline{\mathtt{v}} \leq \overline{\mathtt{w}} \qquad \text{if } \mathtt{w}_i \leq -, \text{then } \Delta \vdash \mathtt{T}_i \mathrel{<:} \mathtt{S}_i \qquad \text{if } \mathtt{w}_i \leq +, \text{then } \Delta \vdash \mathtt{S}_i \mathrel{<:} \mathtt{T}_i}{\Delta \vdash \mathtt{C<}\overline{\mathtt{vS}}\mathtt{>} \mathrel{<:} \mathtt{C<}\overline{\mathtt{wT}}\mathtt{>}}$$

Type Well-formedness:

$$\Delta \vdash \mathtt{Object}\ \text{ok} \qquad \frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}} \qquad \frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\}}\\ \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{T}} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\end{array}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}\ \text{ok}}$$

Fig. 3. Auxiliary Definitions for Typing

Δ' such that $dom(\Delta') = dom(\Delta) \cup \{\mathtt{X}\}$ and $\Delta'(\mathtt{X}) = (\mathtt{v}, \mathtt{T})$ and $\Delta'(\mathtt{Y}) = \Delta(\mathtt{Y})$ if $\mathtt{X} \neq \mathtt{Y}$. We often write `X<:T` for $\mathtt{X} : (\mathtt{+}, \mathtt{T})$ and `X:>T` for $\mathtt{X} : (\mathtt{-}, \mathtt{T})$.

The type system consists of five forms of judgments: $\Delta \vdash \mathtt{N} \Uparrow^{\Delta'} \mathtt{P}$ for opening a variant parametric type to an invariant type; $\mathtt{N} \Downarrow_{\Delta} \mathtt{P}$ for closing an invariant type with some free constrained type variables to a variant parametric type without them; $\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ for subtyping; $\Delta \vdash \mathtt{T}$ ok for type well-formedness; and $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ for typing, where Γ, called an *environment*, is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$. We abbreviate a sequence of judgments, writing $\Gamma \vdash \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}}$ as shorthand for $\Gamma \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{S}_n \mathrel{<:} \mathtt{T}_n$, $\Gamma \vdash \overline{\mathtt{T}}$ ok as shorthand for $\Gamma \vdash \mathtt{T}_1$ ok$, \ldots, \Gamma \vdash \mathtt{T}_n$ ok, and $\Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{T}}$ as shorthand for $\Delta; \Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1, \ldots, \Delta; \Gamma \vdash \mathtt{e}_n \in \mathtt{T}_n$.

Open and Close. As already mentioned, variant parametric types are essentially bounded existential types in disguised forms and so any operations on variant parametric types have to "open" the existential type first. A judgement of the open operation $\Delta \vdash \mathtt{N} \Uparrow^{\Delta'} \mathtt{P}$ is read "under Δ, `N` is opened to `P` and constraints Δ'." The open operation introduces fresh type variables to represent abstract types and replace non-invariant type arguments with the type variables: for example, `List<+Integer>` is opened to `List<X>` with the constraint `X<:Integer`, written $\vdash \mathtt{List{<}{+}Integer{>}} \Uparrow^{\mathtt{X<:Integer}} \mathtt{List{<}X{>}}$.

When the result of an operation involves the abstract types (type variables under constraints) introduced by open, it needs to be "closed" so that the abstract types do not escape; a judgment of the close operation $\mathtt{N} \Downarrow_{\Delta} \mathtt{P}$ is read "`N` with abstract types in Δ closes to `P`." The close operation computes the least 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 $\Delta(\mathtt{X}) = (\mathtt{-}, \mathtt{T})$, 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. In order to close $\mathtt{C{<}\overline{vT}{>}}$, the type arguments $\overline{\mathtt{T}}$ have to be closed to $\overline{\mathtt{T}}'$ first; when $\mathtt{T}_i$ is closed to a proper supertype $\overline{\mathtt{T}}_i{}'$ (i.e. $\mathtt{T}_i \neq \mathtt{T}_i{}'$), the resulting type must be covariant in that argument, thus the least upper bound of `+` and $\mathtt{v}_i$ is attached. For example, under `X<:Integer`, `X` closes to `Integer`, `List<X>` to `List<+Integer>`, and `List<List<X>>` to `List<+List<+Integer>>` (not to `List<List<+Integer>>`).

Subtyping. A judgment for subtyping $\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ is read "`S` is a subtype of `T` under Δ." As usual, the subtyping relation includes the reflexive and transitive closure of the relation induced by $\triangleleft$ clauses. 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 (the third and fourth rules). The next rule is for subtyping based on subclassing. When $\mathtt{C{<}\overline{X}{>}}$ is declared to extend another type $\mathtt{D{<}\overline{S}{>}}$, any (invariant) instantiation $\mathtt{C{<}\overline{T}{>}}$ is a subtype of $\mathtt{D{<}[\overline{T}/\overline{X}]\overline{S}{>}}$. A supertype of a non-invariant type is obtained by opening it and closing the supertype of the opened type. Finally, the last rule deals with variance. The first conditional premise means that, if a variance annotation $\mathtt{v}_i$ for $\mathtt{X}_i$ is either contravariant or invariant, the corresponding type arguments $\mathtt{S}_i$ and $\mathtt{T}_i$ must satisfy $\Delta \vdash \mathtt{T}_i \mathrel{<:} \mathtt{S}_i$; similarly for the second one.

Type Well-Formedness. A judgment for type well-formedness is of the form $\Delta \vdash \texttt{T}$ ok, read "$\texttt{T}$ 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 $\texttt{C<}\overline{\texttt{vT}}\texttt{>}$ is well-formed if the type arguments $\overline{\texttt{T}}$ 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, shown in Figure 4. In what follows, we use $\mathit{bound}_\Delta(\texttt{T})$ defined by: $\mathit{bound}_\Delta(\texttt{X}) = \texttt{S}$ if $\Delta(\texttt{X}) = (\texttt{+},\texttt{S})$ and $\mathit{bound}_\Delta(\texttt{N}) = \texttt{N}$. Most rules are straightforward. When a field or method is accessed, the type on which the operation is performed is opened and the result type is closed. The typing rules for constructor/method invocations check that the type of each actual parameter is a subtype of the corresponding formal.

Typing for methods requires the auxiliary predicate *override* to check correct method overriding. $\mathit{override}(\texttt{m}, \texttt{N}, \overline{\texttt{T}}\texttt{->}\texttt{T}_0)$ holds if and only if a method of the same name `m` is defined in the supertype `N` and has the same argument and return types (it would be safe to extend the rule to handle overriding of the result type 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 $\texttt{C<}\overline{\texttt{X}}\texttt{>}$. 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 Properties

Type soundness (Theorem 3) is shown through subject reduction and progress properties [37], after defining the reduction relation $\texttt{e} \longrightarrow \texttt{e}'$, read "expression `e` reduces to expression $\texttt{e}'$ in one step." The definiton of the reduction relation and proofs of type soundness are found in the full version of the paper[6]. To state type soundness, we require the notion of values, defined by: $v ::= \texttt{new C<}\overline{\texttt{T}}\texttt{>}(v_1, \ldots, v_n)$ (n may be 0).

Theorem 1 (Subject Reduction). *If* $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$ *and* $\texttt{e} \longrightarrow \texttt{e}'$, *then* $\Delta; \Gamma \vdash \texttt{e}' \in \texttt{S}$ *and* $\Delta \vdash \texttt{S} \mathrel{\texttt{<:}} \texttt{T}$ *for some* `S`.

Theorem 2 (Progress). *Suppose* `e` *is a well-typed expression.*

1. *If* `e` *includes* $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}$ *as a subexpression, then* $\mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{f}}$ *and* $\texttt{f} = \texttt{f}_i$.
2. *If* `e` *includes* $\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$ *as a subexpression, then* $\mathit{mbody}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{x}}\texttt{.e}_0$ *and* $|\overline{\texttt{x}}| = |\overline{\texttt{d}}|$.

[6] Strictly speaking, for run-time expressions, we would need another typing rule for stupid casts [20]—that is, casts that turn out to fail in the course of execution; we have omitted it just for brevity.

Expression Typing:

$$\Delta;\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \qquad \mathtt{S}_i \Downarrow_{\Delta'} \mathtt{T}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\mathtt{->U}_0 \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}}\ \mathtt{<:}\ \overline{\mathtt{U}} \qquad \mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}}$$

$$\frac{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok} \qquad \mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}}\ \mathtt{<:}\ \overline{\mathtt{U}}}{\Delta;\Gamma \vdash \mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0)\ \mathtt{<:}\ \mathit{bound}_\Delta(\mathtt{T}) \quad \text{or} \quad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T})\ \mathtt{<:}\ \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma \vdash \mathtt{(T)e}_0 \in \mathtt{T}}$$

Method Typing:

$$\frac{\mathit{mtype}(\mathtt{m}, \mathtt{N}) = \overline{\mathtt{U}}\mathtt{->U}_0 \text{ implies } \overline{\mathtt{U}}\mathtt{,U}_0 = \overline{\mathtt{T}}\mathtt{,T}_0}{\mathit{override}(\mathtt{m}, \mathtt{N}, \overline{\mathtt{T}}\mathtt{->T}_0)}$$

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}_0\ \text{ok} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S}_0 \qquad \Delta \vdash \mathtt{S}_0\ \mathtt{<:}\ \mathtt{T}_0 \\ \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\}} \qquad \mathit{override}(\mathtt{m}, \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}, \overline{\mathtt{T}}\mathtt{->T}_0)\end{array}}{\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{return\ e}_0\mathtt{;\}\ OK\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}}$$

Class Typing:

$$\frac{\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}, \overline{\mathtt{T}}\ \text{ok} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}}{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\mathtt{\}\ OK}$$

Fig. 4. Typing

Theorem 3 (Type Soundness). *If* $\emptyset;\emptyset \vdash \mathtt{e} \in \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$ *being a normal form, then* $\mathtt{e}'$ *is either a value* v *such that* $\Delta;\Gamma \vdash v \in \mathtt{S}$ *and* $\emptyset \vdash \mathtt{S}\ \mathtt{<:}\ \mathtt{T}$ *for some* $\mathtt{S}$ *or an expression that includes* $\mathtt{(T)new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$ *where* $\emptyset \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \not\mathtt{<:}\ \mathtt{T}$.

7 Related Work

Parametric Classes and Variance. There have been several languages, such as POOL [2] and Strongtalk [4,3], that support variance for parametric classes; more recently, Cartwright and Steele [13] have discussed the possibility of the introduction of variance to NextGen—a proposal for extending Java with generics. The approach there is different from ours in that variance is a property of *classes*, rather than *types*. In these languages, the variance annotation is 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<X>` 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, casting a heavy burden on them. Day et al. [15] even argued that this restriction was too severe and, after all, they have decided to drop variance from the their language Theta. On the contrary, in our system, *users* of a parametric class can choose a variance: a class `C<X>`, for example, can have arbitrary occurrences of `X` and induces four different types `C<T>`, `C<+T>`, `C<-T>`, and `C<*T>` 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 [14] proposed to *infer*, rather than *declare*, variance of type parameters of a given class without annotations. For example, if `X` in the class `C<X>` appears only in a method return type, the type `C<T>` 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 [16].

Virtual Types. As we have already mentioned, the idea of variant parametric types has emerged from structural virtual types proposed by Thorup and Torgersen [33]. In a language with virtual types [24,32], 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 runtime checks, Torgersen [34] 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, 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 it was often the case that concrete bags were 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 [33], 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 [19], programming with virtual types is shown to be simulated (to some degree) by exploiting bounded and manifest existential types [18,22], on which our formal type system is also partly based.

Thus, our variant parametric types can be considered a generalization of the idea above with contravariance and bivariance. Other differences are as follows. On one hand, virtual types seem more suitable for programming with extensible mutually recursive classes/interfaces [7,33,10]. 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 [33]) how to encode such programming in structural virtual types. More rigorous comparisons are interesting but left for future work.

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 $\mathcal{LOOM}$ [8], subtyping is dropped in favor of matching [6,9], thus losing subsumption. To recover the flexibility of subsumption to some degree, they introduced the notion of the "hash" type `#T`, which stands for the set of types that match `T`. As is pointed out in [8], `#T` is considered a "match-bounded" existential type ∃`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` does a set of objects created from `C` or its subclasses. In this sense, `@C` can be considered the existential type $\exists$`X<:C,X`, where `<:` denotes subclassing[7].

Raw types [5] of GJ are also close to bounded existential types [21]. 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<:Object.Vector<X>`. One significant difference is that certain unsafe operations putting elements into a raw vector are permitted with a compiler warning.

8 Conclusions and Future Work

In this paper, we have presented a language construct based on *variant parametric types* as an extension to common object-oriented languages with support for generic classes. Variant parametric types are used to exploit inclusive polymorphism for generic types, namely, providing a uniform view over different instantiations of a generic class.

Variant parametric types generally make it possible to widen the applicability of methods accessing a subset of a generic instance's members—e.g., when a method only reads the elements of a generic collection passed as argument. Furthermore, variant parametric types seem to increase the expressiveness in declaring more complex parameterizations, featuring nesting variant parametric types and partial instantiations through the bivariance symbol `*`. For a rigorous argument of type soundness, we have developed a core calculus of variant parametric types, based on Featherweight GJ [20]. A key idea in the development of the type system is to exploit similarity between variant parametric types and bounded existential types.

Implementation issues are not addressed in this paper which are likely to be a subject of future work. It is worth noting, however, that existing implementation techniques seem to allow for straightforward extensions providing variant parametric types. For instance, the type-erasure technique—which is one of the basic implementation approaches for generics, in both Java (GJ [5]) and the .NET CLR ([30])—can be directly exploited for dealing with variant parametric types, for they can be simply erased in the translated code just as standard parametric types. (The language would be constrained due to the lack of run-time

[7] A similar idea can be found in an old version of the type system of Sather [23], where the symbol `$` replaces `@` and is used to enable efficient method dispatching. In the current design of Sather [31], `$` 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.

type arguments, though: for example, the target of typecasts cannot be a type variable.) Other advanced translation techniques where exact generic types are maintained at run time, such as LM translator [36,35], can be extended as well. In this case, variant parametric types can be supported by simply implementing a subtyping strategy in which variance-based subtyping 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. In general, one of the basic implementation issues would be to find an optimized algorithm for subtyping variant parametric types. For example, the approach presented in [28] may be worth investigating.

Other future work includes the further evaluation of the variant parametric type's expressiveness through large-scale applications and more rigorous comparisons with related constructs such as virtual types [33].

References

1. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proc. of ACM OOPSLA*, 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 *Proc. of OOPSLA/ECOOP*, pages 161–168, Ottawa, Canada, October 1990.
3. Gilad Bracha. The Strongtalk type system for Smalltalk. In *Proc. of the OOPSLA96 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 *Proc. of ACM OOPSLA*, 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 *Proc. of ACM OOPSLA*, 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 *Proc. of the 12th ECOOP*, 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 *Proc. 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, *Proc. of ECOOP*, LNCS 952, 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 *Proc. of ACM FPCA*, pages 273–280, September 1989.
12. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
13. Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proc. of ACM OOPSLA*, pages 201–215, Vancouver, BC, October 1998.
14. William Cook. A proposal for making Eiffel type-safe. In *Proc. of the 3rd ECOOP*, pages 57–70, Nottingham, England, July 1989. Cambridge University Press.
15. Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. of ACM OOPSLA*, pages 156–168, Austin, TX, October 1995.
16. Interactive Software Engineering. An Eiffel tutorial. Available through `http://www.eiffel.com/doc/online/eiffel50/intro/language/tutorial-00.html`, 2001.
17. Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.
18. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. of ACM POPL*, pages 123–137, Portland, OR, January 1994.
19. 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.
20. 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.
21. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Informal Proc. 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`.
22. Xavier Leroy. Manifest types, modules and separate compilation. In *Proc. of ACM POPL*, pages 109–122, Portland, OR, January 1994.
23. 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.
24. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proc. of ACM OOPSLA*, pages 397–406, New Orleans, LA, 1989.
25. Andrew C. Meyers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. of ACM POPL*, pages 132–145. ACM, 1997.
26. Microsoft Corporation. The .NET Common Language Runtime. Information available through `http://msdn.microsoft.com/net/`, 2001.
27. 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.

28. Olivier Raynaund and Eric Thierry. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proc. of the 15th ECOOP*, LNCS 2072, pages 165–180. Springer-Verlag, June 2001.
29. Sun Microsystems. Adding generic types to the Java programming language. Java Specification Request JSR-000014, `http://jcp.org/jsr/detail/014.jsp`, 1998.
30. Don Syme and Andrew Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *Proc. of ACM PLDI*. ACM, June 2001.
31. 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.
32. Kresten Krab Thorup. Genericity in Java with virtual types. In *Proc. of the 11th ECOOP*, LNCS 1241, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.
33. Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proc. of the 13th ECOOP*, LNCS 1628, pages 186–204, Lisbon, Portugal, June 1999. Springer-Verlag.
34. Mads Torgersen. Virtual types are statically safe. In *Proc. 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`.
35. Mirko Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Proc. of the ACM Symposium on Applied Computing*, pages 610–619, March 2001.
36. Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Proc. of ACM OOPSLA*, pages 146–165, Oct 2000.
37. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.