Variant Parametric Types: A Flexible Subtyping Scheme for Generics

ATSUSHI IGARASHI
Kyoto University
and
MIRKO VIROLI
Alma Mater Studiorum – Università di Bologna

We develop the mechanism of *variant parametric types*, as a means to enhance synergy between parametric and inclusion polymorphism in object-oriented programming 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 to derive covariant types, contravariant types, and bivariant types (generally called variant parametric types), 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 numbers can be easily modified so as to accept containers of integers, floating point numbers, or any subtype of the number type.

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 Featherweight GJ—an existing formal core calculus for Java with generics—with variant parametric types and prove type soundness.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; *Polymorphism*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*Object-oriented constructs*; *Type structure*

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: generic classes, Java, language design, language semantics, subtyping, variance

This is a revised and extended version of the paper titled "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. This work was supported in part by Grant-in-Aid for Scientific Research for Young Scientists (B) No. 13780203 and by Grant-in-Aid for Scientific Research on Priority Areas Research No. 13224013, both from MEXT of Japan (Igarashi), and from the Italian PRIN 2004 Project "Extensible Object Systems" (Viroli).

Authors' addresses: A. Igarashi, Graduate School of Informatics, Kyoto University, Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan; email: igarashi@kuis.kyoto-u.ac.jp; M. Viroli, DEIS, Università di Bologna, via Venezia 52, 47023 Cesena (FC), Italy; email: mirko.viroli@unibo.it.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

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 an 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 inclusion (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. Since then, a number of extensions were proposed ([Odersky and Wadler 1997; Bracha et al. 1998; Cartwright and Steele Jr. 1998; Viroli and Natali 2000; Viroli 2003; Myers et al. 1997; Agesen et al. 1997] to cite some): Sun Microsystems announced a call for proposals for adding generics to the Java programming language [Sun Microsystems 1998], and finally, Bracha, Odersky, Stoutamire, and Wadler's GJ [Bracha et al. 1998] was chosen as the reference implementation technique for the recent release of Java with generics (JDK 5.0, `http://www.java.sun.com`). Other than Java, more recently an extension of Microsoft's .NET Common Language Runtime (CLR) with generics has been studied [Syme and Kennedy 2001].

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 article, we explore a technique to enhance the synergy between parametric and inclusion 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$^{\#}$—inclusion 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 `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>`; similarly for any type argument. Pointwise subtyping, however, never allows two instantiations of one generic class to be in the subtyping relation. For example, `Vector<Integer>` and `Vector<Number>` 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*, 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>` holds only if `S` = `T`. A familiar example of covariant parametric types is array types in Java, in which, `String[]` is a subtype of `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 `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—but side-effecting features are not the only source of unsoundness, even purely functional GJ would be unsafe if covariant subtyping was allowed. 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 a number of proposals [Cook 1989; America and van der Linden 1990; Bracha and Griswold 1993; Bracha 1996] 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<X>`'s signature. For example, for a generic class `C<X>` 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<X>` 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, actually turns out to pose some difficulty in practice. Since variance can be obtained by prohibiting the declaration of possibly dangerous operations, 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 (or interfaces) 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 derive 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* [Thorup and Torgersen 1999], 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—was still lacking, so it remained unclear how this approach to variance would successfully work in a full-blown language design.

1.4 Contributions

We develop the idea above into the mechanism of *variant parametric types*[1] 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, denoted by annotation symbol `*`.

—We demonstrate the usefulness of variant parametric types by means of examples. 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 [Mitchell and Plotkin 1988; Cardelli and Wegner 1985] and give a rationale of design decisions on 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 [Thorup and Torgersen 1999; Cook 1989; America and van der Linden 1990; Bracha and Griswold 1993; Bracha 1996] 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 [Igarashi et al. 2001a]—we define a formal core language of variance with its syntax, type system, and operational semantics, and prove type soundness.

For the sake of concreteness, we mostly use Java-like notations for the discussion but believe that the idea itself can be applied, possibly with a few adaptations,

[1]The phrase "variant parametric types" may be confused with "variant types," which usually refers to disjoint (or tagged) union types, such as datatypes in ML and Haskell, or variant records in Pascal. We use "variant" for a different meaning, that is, generalization of the four words: invariant, covariant, contravariant, and bivariant.

to other class-based object-oriented languages with nominal type systems as in Java, Eiffel, or C$^{\#}$. Compared to the result presented in the early version of this article [Igarashi and Viroli 2002], 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 proposed in this article is developed into *wildcard types* [Torgersen et al. 2004], shipped with JDK 5.0, an official release of the Java programming language by Sun Microsystems—a few syntactic and semantic differences with respect to our proposal are described in Section 6.

1.5 Outline of the Article

The remainder of the article 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, addresses its design issues, and demonstrates the applicability and usefulness through examples. Section 4 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 5 presents the core calculus for variant parametric types and proves the soundness of its type system. Section 6 discusses related work and Section 7 presents concluding remarks and perspectives on future work.

2. CLASSICAL APPROACH TO VARIANCE FOR PARAMETRIC CLASSES

Historically, one main approach to flexible inclusion polymorphism for generics was through the mechanism of *variance* for classes, which, in spite of several attempts [Meyer 1986; America and van der Linden 1990; Cook 1989; Bracha and Griswold 1993; Bracha 1996; Cartwright and Steele Jr. 1998], was not really adopted in widely disseminated object-oriented languages, such as Java and C++. In this section, we review the approach and discuss its limitations.

As mentioned above, a generic class `C<X>` is said to be *covariant* in the type parameter `X` when the subtype relation `C<S>` <: `C<T>` holds if `S` <: `T`. Conversely, `C<X>` is said to be *contravariant* in `X` when `C<S>` <: `C<T>` holds if `T` <: `S`. General notions of *bivariance* and *invariance* can be defined as well. `C<X>` is said to be *bivariant* in `X` when `C<S>` <: `C<T>` for any `S` and `T`. `C<X>` is said to be *invariant* in `X` when `C<S>` <: `C<T>` 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.

In principle, a generic class can be assigned any variance property by the type system but some of them could be unsafe, 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—e.g., types `Object[]` and `String[]` could be seen as types `Array<Object>` and `Array<String>` where `Array` is a special system class. The Java type system associates to the array types the covariance property—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 first statement is permitted because of covariance. The second is also statically accepted, because `Integer` is a subtype of `Object`. When the code is executed, however, 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.

The problem of understanding when covariance, contravariance and bivariance can be used for a generic class in a safe way has received some interest. Previous work [Cook 1989; America and van der Linden 1990; Bracha and Griswold 1993; Bracha 1996] 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 [Liskov 1988]—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 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 [Bracha et al. 1998])

```
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 [Bracha and Griswold 1993; Bracha 1996] and the NextGen compiler for generics in Java [Cartwright and Steele Jr. 1998]—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. [1995]—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 collection classes, the standard application of generics, and their typical signature schema with methods for getting and setting elements, as is exemplified 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; }
   // Reading elements disallows contravariance
   X getElementAt(int i){ return ar[i]; }
   // Writing elements disallows covariance
   void setElementAt(X t,int i){ ar[i]=t; }
}
```

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. In order to deal with contravariance (and bivariance) as well, a diamond structure of classes (or interfaces, if coded in Java) would be required: `Vector<X>` can be defined by extending the covariant class (interface) `ROVector<+X>` and the contravariant one `WOVector<-X>` (which provides only the writing functionality), both extending a common base `NORWVector`. In particular, as the number of type parameters increases, the number of classes/interfaces may exponentially increase, e.g. coding a pair class with two type parameters would require a diamond structure of *16* classes. It is worth noting that all these issues apply not only to collection classes, but also in general to classes dealing with the production/consumption of data, such as stream classes. That is, covariance (contravariance) with respect to a certain type argument is safe if there are no methods consuming (producing, resp.) objects whose static types are the corresponding type parameter.

A solution to the applicability limitations of the classical approach is hinted in the work on structural virtual types [Thorup and Torgersen 1999]. The idea is to let a programmer specify whether 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—for example, `Vector<`$^{+}$`Object>` 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 mechanism is widened, since covariant (and contravariant) types can be derived from *any* class—independently of how type parameters occur in the signature—and the designers of libraries are released from the burden of making decisions about the tradeoff mentioned above. So, in this article we generalize this idea by investigating the inclusion of covariance, contravariance, and bivariance 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 of access restriction due to variance, along with examples based on `Vector<X>` from the last section. Although most of the examples are concerned with only vectors or vectors of vectors, they can be easily generalized to other kinds of collections, found in e.g. the Java Collection Framework, or even stream classes. Those use cases are indeed found in the JDK 5.0 API as will be mentioned below. (See also Section 6 for applications other than collections.)

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 *bivariance annotation symbol*. A variance annotation symbol, which precedes a type parameter as in `Vector<+String>`, 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 in `Vector<+Vector<*X>>`. A parametric type where no (outermost) type argument has a variance annotation is called *instance type*, or sometimes invariant type. `Vector<String>` and

`Pair<Vector<+String>,Integer>` are examples of instance types. Thus, objects are instantiated by an expression of the form `new C<`T_1`,...,`T_n`>(...)` and instance types play the role of run-time types of objects. Note that each type argument T_i needs not be an instance type—for example, `new Vector<Vector<+Number>>(...)` is permitted. Thus, implementations of generics that reify information on type parameters so as to have an explicit representation of parametric types at run-time—such as the type-passing compilation of LM [Viroli and Natali 2000; Viroli 2003], the code-expansion one of NextGen [Cartwright and Steele Jr. 1998], or the hybrid one proposed for generics in C# [Syme and Kennedy 2001]—would have to carry explicit information on variance, too.

Unlike the classical variance described in the previous section, programmers can 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. As a running example for our discussion on design issues, we consider the class `Vector` reported in the previous section, along with the following three simple methods `fillFrom()`, `fillTo()`, and `sameSize()`.

```
class Vector<X extends Object>{
   ...
   // gets elements from v
   void fillFrom(Vector<+X> v, int start){
       for (int i=0; i<v.size() && i+start<this.size(); i++)
           this.setElementAt(v.getElementAt(i),i+start);
   }
   // puts elements into v
   void fillTo(Vector<-X> v, int start){
       for (int i=0; i<this.size() && i+start<v.size(); i++)
           v.setElementAt(this.getElementAt(i),i+start);
   }
   // checks if v has the same size
   boolean sameSize(Vector<*X> v){
       return this.size()==v.size();
   }
}
```

The intuitive idea is that, in each case, the argument `v` can be safely given a variant parametric type (`Vector<+X>`, `Vector<-X>`, and `Vector<*X>`, respectively)—rather than the usual invariant type `Vector<X>`—since safety is guaranteed by the fact that only a subset of `Vector`'s methods is applied to `v`. In the case of method `fillFrom()`, for instance, `v` can be safely given the covariant type `Vector<+X>` for only `getElementAt()` is invoked on it: the type system would instead forbid accessing the method `setElementAt()`. The other cases are handled similarly, as explained below in detail.

3.1 Subtyping

A simple interpretation of a variant parametric type is given as a set of objects, instantiated from instance types. A type `C<T>` can be interpreted as the set of all objects of the form `new C<T>(...)`; 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 set of all objects of the form `new C<S>(...)` (for *any* `S`). Therefore, `Vector<+Integer> <: Vector<+Number>` directly follows from inclusion of the sets they denote. Moreover, it is easy to derive subtyping between types that differ only in variance annotations: `Vector<Integer> <: Vector<+Integer>` and `Vector<Integer> <: Vector<-Integer>` hold, and similarly `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.

Consider the examples of methods `fillFrom()`, `fillTo()`, and `sameSize()`, and the following definitions (where we further suppose `Float <: Number`):

```
Vector<Number>  v  = new Vector<Number>(30);
Vector<Integer> vi = new Vector<Integer>(10);
Vector<Number>  vn = new Vector<Number>(10);
Vector<Float>   vf = new Vector<Float>(10);
Vector<Object>  vo = new Vector<Object>(10);
```

Since `Vector<Number>`, `Vector<Integer>`, and `Vector<Float>` are subtypes of the type `Vector<+Number>`, we are allowed to store numbers taken from either `vn`, `vi`, and `vf` into `v` by method `fillFrom()`:

```
v.fillFrom(vi,0);  // Vector<Integer> <: Vector<+Number>
v.fillFrom(vn,10); // Vector<Number> <: Vector<+Number>
v.fillFrom(vf,20); // Vector<Float> <: Vector<+Number>
```

Here, notice that the applicability of method `fillFrom()` is clearly widened, as a larger set of vectors can be passed to it in a uniform way thanks to the covariant argument type `Vector<+X>`. Dually, because `Vector<Number>` and `Vector<Object>` are subtypes of `Vector<-Number>`, we can use `fillTo()` to store the numbers of `v` in both the vector `vn` of numbers and the vector `vo` of objects:

```
v.fillTo(vn,0); // Vector<Number> <: Vector<-Number>
v.fillTo(vo,0); // Vector<Object> <: Vector<-Number>
```

In this case, the applicability of the method is widened, too, by using the contravariant type `Vector<-X>` as argument type. Finally, by virtue of the bivariant type `Vector<*X>` we are allowed to check whether any two vectors have the same size through method `sameSize()`:

```
v.sameSize(vn)  // Vector<Number>  <: Vector<*Number>
v.sameSize(vo)  // Vector<Object>  <: Vector<*Object>  <: Vector<*Number>
v.sameSize(vi)  // Vector<Integer> <: Vector<*Integer> <: Vector<*Number>
vf.sameSize(vi) // Vector<Integer> <: Vector<*Number>  <: Vector<*Float>
```

In general, subtyping for variant parametric types is defined as follows. We use the metavariable `v` for variance annotations and use `o` to mean empty (i.e., invariant) variance annotation. Suppose `C` is a generic class that takes n type arguments, and `S` and `T` (possibly with subscripts) are types.

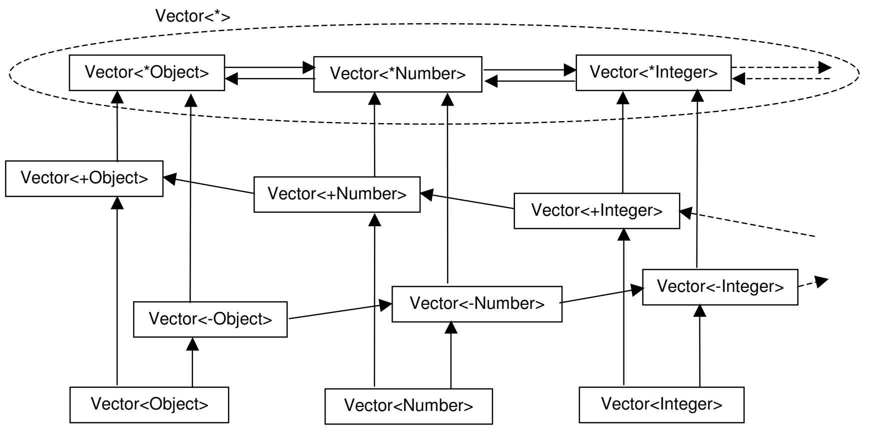

Fig. 1. Subtyping graph of variant parametric types

—The following subtype relation holds that involves variant parametric types differing just in the variance annotation symbol on one type parameter:

$$\texttt{C<...,v}_1\texttt{T,...>} <: \texttt{C<...,v}_2\texttt{T,...>} \text{ if } \texttt{v}_1 \leq \texttt{v}_2$$

where $\texttt{o} \leq \texttt{+} \leq \texttt{*}$ and $\texttt{o} \leq \texttt{-} \leq \texttt{*}$. (Strictly speaking, the ellipses are abused to mean that the ones before/after $\texttt{v}_1\texttt{T}$ and $\texttt{v}_2\texttt{T}$ denote the same sequence of types.)

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

$$\begin{array}{lcll} \texttt{C<..., S,...>} & <: & \texttt{C<..., T,...>} & \text{if } \texttt{S} <: \texttt{T} \text{ and } \texttt{T} <: \texttt{S} \\ \texttt{C<...,+S,...>} & <: & \texttt{C<...,+T,...>} & \text{if } \texttt{S} <: \texttt{T} \\ \texttt{C<...,-S,...>} & <: & \texttt{C<...,-T,...>} & \text{if } \texttt{T} <: \texttt{S} \\ \texttt{C<...,*S,...>} & <: & \texttt{C<...,*T,...>} & \text{for any } \texttt{S} \text{ and } \texttt{T} \end{array}$$

Note that the subtyping relation is *not* anti-symmetric due to the last rule: `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. For example, the relation `Pair<String,String>` <: `Pair<+Object,-String>` can be inferred from

$$\begin{array}{rcl} \texttt{Pair<String,String>} & <: & \texttt{Pair<String,-String>} \\ \texttt{Pair<String,-String>} & <: & \texttt{Pair<+String,-String>} \\ \texttt{Pair<+String,-String>} & <: & \texttt{Pair<+Object,-String>}. \end{array}$$

In what follows, the type argument following `*` is often omitted by simply writing e.g. `Vector<*>` (denoted by the dotted oval in Figure 1): whatever comes after `*` does not have impact on subtyping, due to bivariance, and similarly for member access restriction as discussed below. In fact, this is also the syntax we propose for an actual language extension featuring variant parametric types. (However, we still use the notation `*T` when the type system is introduced in Section 5: its syntactic uniformity makes it easy to formalize typing rules.)

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> { ... }
```

As `Twin<T>` is a subtype of `Pair<T,T>` by pointwise subtyping, the type `Pair<T,T>` now includes 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 4, its actual definition is more involved than it first appears, especially when parametric types are nested.

3.2 Restrictions on Member Access

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

We 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`. Since element types can be *any* subtype of `Number`, we cannot safely insert anything through the type `Vector<+Number>`, and assignments to fields must therefore be prohibited when their 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, elements obtained through `Vector<+Number>` (e.g. by method `getElementAt()`) can be safely given 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 without operations accepting elements. In other words, if a vector is used in a read-only manner, its type can be covariant rather than invariant. In fact, in `fillFrom()`, the argument `v` is used only to invoke method `getElementAt()` to retrieve elements of type `X` and thus it can be safely given type `Vector<+X>`, making the method applicability wider. In the `java.util.Collection` interface of the Java API, the method `addAll()` follows the same pattern as our `fillFrom()` example, so argument type can safely be made covariant.

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 *supertype* of `Number`, it is safe to assign any numbers (subtypes of `Number`) to elements through this type. However, accessing elements by field access or `getElementAt()` results in an unknown element type and so it is prohibited. This is exemplified instead by method `fillTo()`, where argument `v` can be safely given the contravariant type `Vector<-X>` for only `setElementAt()` is invoked on it (with argument `X`).

Similarly to `fillTo()`, contravariance can be used e.g. in the static method `fill()` of class `java.util.Collections`, which inserts an element in all the positions of a (contravariant) collection passed as an argument.

Finally, bivariant types prohibit both producing and consuming elements. Since `Vector<*>` is a supertype of `Vector<+Number>` and `Vector<-Number>`, only operations that are applicable to both subtypes can be allowed for `Vector<*>`. For class `Vector`, only method `size()`—whose signature does not include the type variable—can be applied, as happens in method `sameSize()`.

Actually, some expressions are rejected due to member access restrictions, even though, in theory, 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), 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` can be substituted 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 thinking of the set of instances each type denotes. Secondly, `*` is a concise way to signify "not being accessed," whereas `+Object` means "being read only as objects." Indeed, bivariance turns out to be particularly useful in types with more than one parameter, as shown in the next subsection. 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.

In summary, attaching a variance annotation to an instance type virtually yields an abstract class that provide only safe methods and fields *above* that instance type (in the subtype hierarchy). This cannot be easily achieved in existing class-based languages or by programming idioms like writing wrapper classes to hide unsafe operations.

3.3 Nested Parametric Types

So far, we have discussed only simple cases, where type arguments with variance annotations are non-parametric types. 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 4, where we give an informal correspondence to bounded existential types [Cardelli and Wegner 1985]. However, the informal discussion given in this section is indeed enough to understand many practical cases, including some interesting cases where variant parametric types occur inside another parameterization.

We introduce examples to describe interesting patterns of variant annotations that occur when nested collections are used. Consider the following new methods added to class `Vector`:

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

The method `fillFromVector()` takes a vector of vectors `vv` and puts its inner-level elements into the vector on which it is invoked. For instance, invoking it on a vector `["1","2","3","4","5"]` with `[["A","B"],["C"]]` and `1` as arguments, the first vector would change 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>>`, `Vector<Vector<+Number>>`, and even type `Vector<SubVector<Float>>` where `SubVector<X>` extends `Vector<X>`. 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(vvi,0);
    // Permitted for Vector<Vector<Integer>> <: Vector<+Vector<+Number>>
```

```
vn.fillFromVector(vvf,10);
  // Permitted for Vector<Vector<Float>> <: Vector<+Vector<+Number>>
```

The method `fillToVector()` realizes a dual case: it inserts elements of the receiver vector into a vector of vectors provided as its argument. For instance, invoking it on the vector `["1","2","3"]` with the vector `[["A","B"], ["C","D"]]` as argument 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.

Finally, consider method `fillFromFirst()`, which copies first elements of pairs in a given vector to the receiver: it shows that bivariance appears particularly useful in those cases where a generic class involves more than one type parameter. Similarly to `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, such as `Vector<Pair<Integer,String>>`, `Vector<Pair<Float,Float>>`, and `Vector<Pair<Number,Object>>`. This example suggests an interesting application of bivariance as a mechanism providing "don't care" type arguments.

Even though more complex, these patterns of nesting do occur in the Java Collection Framework. First, class `java.util.Map.Entry`, modelling entries in maps such as hashtables, can come with two type parameters `K` and `V`, one for keys and one for values. Then, an iterator over such entries can be given type `Iterator<+Map.Entry<+K,+V>>`—see e.g. the implementation of method `putAll` in class `java.util.HashMap`.

4. CORRESPONDENCE TO EXISTENTIAL TYPES

In this section, we investigate an informal correspondence of variant parametric types to (bounded) existential types [Mitchell and Plotkin 1988; Cardelli and Wegner 1985; Nordström et al. 1990], which are a type-theoretic basis of (partially) abstract data types (ADTs). Existential types are also used in some work on object encoding [Bruce et al. 1999; Pierce and Turner 1994; Abadi et al. 1996; Bruce 1994; Compagnoni and Pierce 1996], in which types of object *states* are hidden by using existential types. Here, we use existential types to hide (part of) the *interface* of objects. Beginning with a review of 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.

4.1 Existential Types *à la* Mitchell and Plotkin

An (unbounded) existential type is syntactically a type of the form $\exists\mathtt{X}.\mathtt{T}$, 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—the 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

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

where `{...}` is a record type, `A`×`B` stands for the type of pairs of `A` and `B`, and `A`→`B` for the type of functions from `A` to `B`.

A value of an existential type ∃`X.T` is constructed by a pair of a type `U` and a value `v` of [`U`/`X`]`T`: the type `T` where `U` is substituted for the type variable `X`. Such a pair is often written `pack [U,v] as` ∃`X.T` and `U` is sometimes called a *witness type*, since it witnesses the existence of `X`. From the ADT point of view, this pair is considered an ADT package, which is usually given by 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 the same as the one 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, ∃`X.{empty:intlist, ...}` 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, the leakage of a concrete representation of a stack is prohibited.

Bounded existential types introduced by Cardelli and Wegner [1985] allow existential type variables to have upper bounds: an example is ∃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 ∃X<:intbag.{empty:X, ...}. As before, a package is given by a pair of a concrete type and an implementation, but now, the concrete type has to be a subtype of intbag. Here, we assume intlist is a subtype of intbag and so packing r above as

```
stack' = pack [intlist, r] as ∃X<:intbag.{empty:X, ...}
```

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:

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

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

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.

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

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:

```
∃X<:intbag.{empty:X, push:..., pop:...}
      <: ∃X<:Top.{empty:X, push:..., pop:...}
∃X<:intbag.{empty:X, push: ..., pop: ...}
      <: ∃X<:intbag.{empty:X, push:...}
```

Remark. Here, one may notice that this rule is known as one for the full variant of System $F_{\leq}$, in which subtyping is undecidable [Pierce 1994]. Accordingly, even though variant parametric types are rather restricted forms of bounded existential types, the decidability of subtyping of variant parametric types as defined in Section 5 is an open problem.

4.2 Interpreting Variant Parametric Types as Existential Types

Variant parametric types can be explained in terms of existential types above. According to the naive interpretation that `C<+T>` is the set of objects of the from `new C<S>(...)` where `S <: T`, a covariant type `C<+T>` would correspond to the bounded existential type `∃X<:T.C<X>`. As we discuss further below, hiding (partial) information of the type argument will make it impossible to call certain methods, thus realizing access restriction. Contravariant types will require slight generalization: the type `C<-T>` would correspond to the bounded existential type `∃X:>T.C<X>`, where the abstract type `X` has a *lower* bound rather than an upper bound. A bivariant type `C<*T>`—abbreviated to `C<*>`—would simply correspond to the unbounded existential type `∃X.C<X>`. This idea extends naturally to a parametric type with more than one parameter. `D<+S,T>` would correspond to `∃X<:S.D<X,T>` and `D<+S,-T>` to `∃X<:S.∃Y:>T.D<X,Y>`. For nested parametric types, the existential quantifier will appear in bounds: for example, `Vector<+Vector<+Nm>>` would correspond to `∃X<:(∃Y<:Nm.Vector<Y>).Vector<X>` since `Vector<+T>` would be `∃X<:T.Vector<X>` and here `T` is again a type of the form `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.

Remark. In Mitchell-Plotkin's formalization, a value of an existential type is a *pair* of a type and a value whereas the naive interpretation gives only a set of values, namely instances, to a type. However, by ignoring the witness type part, an existential type `∃X<:T.C<X>` denotes a collection of values of the type `C<U>` for *some* (unknown) `U` such that `U` is subtype of `T`—which is closer to the naive interpretation of ours and the "classical" interpretation of the existential quantifier.

4.2.1 *Interpretation of Variance-Based Subtyping.* Covariant and contravariant subtyping are directly attributed to the subtyping rule for existential types above. `C<+S> <: C<+T>` when `S <: T` corresponds to `∃X<:S.C<X> <: ∃X<:T.C<X>` assuming `S <: T`, and similarly for contravariant types. Subtyping that changes variance annotations (`C<+T> <: C<*T>` and `C<-T> <: C<*T>`) can be considered as subtyping between bounded and unbounded existential types: `∃X<:S.C<X> <: ∃X.C<X>` (and `∃X:>S.C<X> <: ∃X.C<X>`) could be allowed since it is just a special kind of relaxing the bounds from `S` to nothing. Finally, subtyping `C<T> <: C<+T>` or `C<T> <: C<-T>` 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<T>` to `C<+T>`, the expression is packed with the witness type `T`, yielding `∃X<:T.C<X>`. For example, when `v` of `Vector<Integer>` is passed to where an argument of `Vector<+Nm>` is expected, it would be represented as:

```
pack [Integer,v] as ∃X<:Nm.Vector<X>
```

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

4.2.2 *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, at least conceptually, has to be `open`ed first. For example, suppose `x` is given type `C<+T>` 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<X>`; since `y` is an instance 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 an `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<:Nm.`$\exists$`Y:>Nm.Pair<X,Y>`. 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<:Nm` and `W:>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)` 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 <: Nm <: 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

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

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 instance 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<:Nm and W:>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 will not be 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 +).

4.2.3 *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>>`. The type itself cannot be the return type as `Z` and `W` would otherwise escape their scope introduced by `open`. Therefore, we need a supertype of `Pair<Pair<Z,W>,Pair<Z,W>>` without `Z` and `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>>`! This 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. In summary, given a type that contains (bounded) type variables, we need to find a supertype that does not contain those type variables, but naive substitution of an actual type argument with its variance does not always work to obtain such a 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 in general. For example, suppose the class `Vector` has method `m()` returning `Vector<Vector<-X>>`.

```
    Vector<Vector<-X>> m() { ... }
```

and consider `x.m()` where `x` is of type `Vector<+Nm>`. In the same manner as before we first obtain `Vector<Vector<-X>>` under `X<:Nm` as its return type, of which we have to find a supertype (without `X`). In this case, both `C<+C<*>>` and `C<-C<-Nm>>` are supertypes of `C<C<-X>>` but neither is a supertype of the other. This phenomenon is unpleasant from the point of view of a bottom-up typecheck algorithm. 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 look 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 `Vector<+Vector<*>>` since an `X`-free supertype of `Vector<-X>` is `Vector<*>` and then `+` is attached before `Vector<*>`. Thus, unfortunately, the context like

```
  Vector<-Vector<-Nm>> v = x.m();
```

that expects `Vector<-Vector<-Nm>>` is rejected even though it would be safe to execute. 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, $\texttt{C}_1\texttt{<C}_2\texttt{<C}_3\texttt{<X,Y>>>}$ with `X<:S` and `Y:>T` has $\texttt{C}_1\texttt{<+C}_2\texttt{<+C}_3\texttt{<+S,-T>>>}$ as a supertype without `X` and `Y`; $\texttt{C}_1\texttt{<-C}_2\texttt{<C}_3\texttt{<X,Y>>>}$ with `X<:S` and `Y:>T` has $\texttt{C}_1\texttt{<*>}$, which is equivalent to $\texttt{C}_1\texttt{<*C}_2\texttt{<+C}_3\texttt{<+S,-T>>>}$, as a supertype without `X` and `Y` (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 [Ghelli and Pierce 1998], where it was introduced to omit a type annotation for the open expression.

4.2.4 *Variance and Inheritance-Based Subtyping.* The operations of open and close are also used for deriving inheritance-based subtyping of 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 instance 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-instance 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`).

5. 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 type system is sound. Our calculus is considered an extension of Featherweight GJ (FGJ for short) by Igarashi et al. [2001a], originally proposed to formally investigate properties of the type system and compilation scheme of GJ [Bracha et al. 1998]. Like FGJ, our extended calculus is functional and supports only minimal features including top-level parametric classes with variant parametric types, fields, parametric methods, F-bounded polymorphism, object instantiation, recursion through `this`, and typecasts.

5.1 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 over type variables; `N`, `P`, and `Q` range over variant parametric 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.

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

Fig. 2. Syntax.

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 $\mathtt{M}_1 \cdots \mathtt{M}_n$ (with no commas). We write $\bullet$ for the empty sequence 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 "$\mathtt{C}_1\ \mathtt{f}_1\mathtt{;}\ \cdots\ \mathtt{C}_n\ \mathtt{f}_n\mathtt{;}$", "$\mathtt{{<}\overline{X}\triangleleft\overline{N}{>}}$" as shorthand for "$\mathtt{{<}X_1\triangleleft N_1,\ \ldots,\ X_n\triangleleft N_n{>}}$", and "$\mathtt{C{<}\overline{vT}{>}}$" for "$\mathtt{C{<}v_1T_1,\ \ldots,v_nT_n{>}}$". 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 write $\mathtt{m} \notin \overline{\mathtt{M}}$ to mean that a method of the name `m` is not included in $\overline{\mathtt{M}}$. As mentioned in Section 3, we introduce the variance annotation `o` for invariance and 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 parametric type $\mathtt{C{<}\overline{vT}{>}}$, we call it an *instance type* and abbreviate it to $\mathtt{C{<}\overline{T}{>}}$.

According to the grammar in Figure 2, a class declaration `L` consists of its name (`C`), type parameters ($\overline{\mathtt{X}}$) with their (upper) bounds ($\overline{\mathtt{N}}$), fields ($\overline{\mathtt{T}}\ \overline{\mathtt{f}}$), and (parametric) methods ($\overline{\mathtt{M}}$)[3]; moreover, every class must explicitly declare its supertype $\mathtt{D{<}\overline{S}{>}}$ with $\triangleleft$ (read `extends`) even if it is `Object`. Note that only an instance type is allowed as a supertype, just as in object instantiation. Since our language supports F-bounded polymorphism [Canning et al. 1989], the bounds $\overline{\mathtt{N}}$ of type variables $\overline{\mathtt{X}}$ can contain $\overline{\mathtt{X}}$ in them. A method definition can be parameterized by type variables $\overline{\mathtt{Y}}$ with bounds $\overline{\mathtt{P}}$. A method body just returns an expression, which is either a variable `x`, field access `e.f`, (polymorphic) method invocation $\mathtt{e.{<}\overline{T}{>}m(\overline{e})}$, object instantiation $\mathtt{new\ C{<}\overline{T}{>}(\overline{e})}$, or typecasts `(T)e`. As we have already mentioned, the type used for an instantiation must be an instance type, hence $\mathtt{C{<}\overline{T}{>}}$. 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

[3]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 roles.

where instantiation of type parameters are kept at run-time, such as the framework of LM [Viroli and Natali 2000; Viroli 2003] or generics for C$^{\#}$ [Syme and Kennedy 2001]. 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 Bracha et al. [1998] 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 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) $CT(\mathtt{C}) =$ `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 transitive closure of the relation between class names obtained from $\triangleleft$ 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.

5.2 Type System

For the typing and reduction rules, we need a few auxiliary definitions, which are shown in Figure 3, to look up the field or method types and the method body of an instance type. As we discussed in the previous section, we never attempt to ask the field or method types of non-instance types. In these rules, we often use an informal notation "..." to avoid introducing extra metavariables not used in any interesting way: for example, with $\mathtt{class\ C{<}\overline{X}\triangleleft\overline{N}{>}\triangleleft D{<}\overline{S}{>}\ \{...\ \ \overline{M}\}}$, we assume field types and names are not significant and all methods are represented by $\overline{\mathtt{M}}$. Similarly for $\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ ...\}$ (methods are uninteresting) and `{...}` (the whole body is uninteresting).

The fields of an instance type $\mathtt{C{<}\overline{T}{>}}$, written $\mathit{fields}(\mathtt{C{<}\overline{T}{>}})$, are a sequence $\overline{\mathtt{S}}\ \overline{\mathtt{f}}$ of corresponding types and field names. 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 instance type $\mathtt{C{<}\overline{T}{>}}$, written $\mathit{mtype}(\mathtt{m}, \mathtt{C{<}\overline{T}{>}})$, returns a signature of the form $\mathtt{{<}\overline{Y}\triangleleft\overline{P}{>}\overline{U}{\rightarrow}U_0}$ where $\overline{\mathtt{Y}}$, $\overline{\mathtt{P}}$, $\overline{\mathtt{U}}$, $\mathtt{U}_0$ are type parameters, their upper bounds, argument types, and a result type, respectively. Here, $\overline{\mathtt{Y}}$ are bound in $\overline{\mathtt{P}}$, $\overline{\mathtt{U}}$ and $\mathtt{U}_0$ and we allow implicit α-conversions on type parameters in a signature.

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

A *type environment* Δ is a finite mapping from type variables to pairs of a

Field lookup:

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \text{(F-Object)}$$

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

Method type lookup:

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>\ U_0\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U_0})} \qquad \text{(MT-Class)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \\ \mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U_0}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U_0}} \qquad \text{(MT-Super)}$$

Method body lookup:

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>U_0\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{x}}.[\overline{\mathtt{V}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e_0}} \qquad \text{(MB-Class)}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \\ \mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}) = \overline{\mathtt{x}}.\mathtt{e_0}\end{array}}{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>},\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{x}}.\mathtt{e_0}} \qquad \text{(MB-Super)}$$

Fig. 3. Field/Method Look-up Functions.

variance annotation except `o` (that is, either `+`, `-`, or `*`) and a type. We write $\mathit{dom}(\Delta)$ for the domain of Δ. When $\mathtt{X} \notin \mathit{dom}(\Delta)$, we write $\Delta, \mathtt{X} : (\mathtt{v}, \mathtt{T})$ for the type environment Δ' such that $\mathit{dom}(\Delta') = \mathit{dom}(\Delta) \cup \{\mathtt{X}\}$ and $\Delta'(\mathtt{X}) = (\mathtt{v}, \mathtt{T})$ and $\Delta'(\mathtt{Y}) = \Delta(\mathtt{Y})$ if $\mathtt{X} \neq \mathtt{Y}$. We often write $\mathtt{X<:T}$ for $\mathtt{X} : (\mathtt{+}, \mathtt{T})$ and $\mathtt{X:>T}$ for $\mathtt{X} : (\mathtt{-}, \mathtt{T})$. When $\Delta(\mathtt{X}) = (\mathtt{v}, \mathtt{N})$ for any $\mathtt{X} \in \mathit{dom}(\Delta)$ (i.e., all the bounds are nonvariable types), we say Δ *has non-variable bounds.*

The type system consists of seven forms of judgments: (1) $\Delta \vdash \mathtt{N} \Uparrow^{\Delta'} \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ for opening a variant parametric type to an instance type; (2) $\mathtt{S} \Downarrow_{\Delta} \mathtt{T}$ for closing a type with some free type variables constrained in Δ to a variant parametric type without them; (3) $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$ for subtyping; (4) $\Delta \vdash \mathtt{T}$ ok for type well-formedness; (5) $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ for typing, where Γ, called an *environment*, is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$; (6) `M` ok in $\mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}$ for typing methods; and (7) `L` ok for typing classes. We abbreviate a sequence of judgments, writing $\Gamma \vdash \overline{\mathtt{S}} \mathrel{\mathtt{<:}} \overline{\mathtt{T}}$ as shorthand for $\Gamma \vdash \mathtt{S}_1 \mathrel{\mathtt{<:}} \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{S}_n \mathrel{\mathtt{<:}} \mathtt{T}_n$ and $\Gamma \vdash \overline{\mathtt{T}}$ 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 disguise. So any operations on values of 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:

$$\Delta \vdash \mathtt{T} \Uparrow^{\emptyset} \mathtt{T} \qquad \text{(O-Refl)}$$

$$\frac{\Delta \vdash \mathtt{S} \Uparrow^{\Delta_1} \mathtt{T} \qquad \Delta, \Delta_1 \vdash \mathtt{T} \Uparrow^{\Delta_2} \mathtt{U}}{\Delta \vdash \mathtt{S} \Uparrow^{\Delta_1, \Delta_2} \mathtt{U}} \qquad \text{(O-Trans)}$$

$$\frac{\mathtt{X} \text{ fresh for } \Delta, \mathtt{C}\texttt{<}\overline{\mathtt{v}}_1\overline{\mathtt{T}}_1\mathtt{,vT,}\overline{\mathtt{v}}_2\overline{\mathtt{T}}_2\texttt{>} \qquad \mathtt{v} \neq \mathtt{o}}{\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{v}}_1\overline{\mathtt{T}}_1\mathtt{,vT,}\overline{\mathtt{v}}_2\overline{\mathtt{T}}_2\texttt{>} \Uparrow^{\mathtt{X:(v,T)}} \mathtt{C}\texttt{<}\overline{\mathtt{v}}_1\overline{\mathtt{T}}_1\mathtt{,oX,}\overline{\mathtt{v}}_2\overline{\mathtt{T}}_2\texttt{>}} \qquad \text{(O-Class)}$$

Close:

$$\frac{\Delta(\mathtt{X}) = (\mathtt{+}, \mathtt{T})}{\mathtt{X} \Downarrow_{\Delta} \mathtt{T}} \qquad \text{(C-Prom)}$$

$$\frac{\mathtt{X} \notin \mathit{dom}(\Delta)}{\mathtt{X} \Downarrow_{\Delta} \mathtt{X}} \qquad \text{(C-TVar)}$$

$$\frac{(\mathtt{w}_i, \mathtt{T}_i{}') = \begin{cases} (\mathtt{v}_i, \mathtt{T}_i) & \text{if } \mathtt{T}_i \Downarrow_{\Delta} \mathtt{T}_i \\ (\mathtt{v}_i \vee \mathtt{+}, \mathtt{U}_i) & \text{if } \mathtt{T}_i \Downarrow_{\Delta} \mathtt{U}_i \text{ and } \mathtt{T}_i \neq \mathtt{U}_i \\ (\mathtt{v}_i \vee \mathtt{v}_i{}', \mathtt{U}_i) & \text{if } \mathtt{T}_i = \mathtt{X} \text{ and } \Delta(\mathtt{X}) = (\mathtt{v}_i{}', \mathtt{U}_i) \end{cases}}{\mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>} \Downarrow_{\Delta} \mathtt{C}\texttt{<}\overline{\mathtt{wT}'}\texttt{>}} \qquad \text{(C-Class)}$$

Fig. 4. Open and Close.

open operation $\Delta \vdash \mathtt{N} \Uparrow^{\Delta'} \mathtt{P}$ is read “under Δ, N is opened to P constrained by Δ'” and that of the close operation $\mathtt{N} \Downarrow_{\Delta} \mathtt{P}$ 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 type arguments with the type variables: for example, List<+Integer> is opened to List<X> with the constraint X<:Integer, written $\vdash$ List<+Integer> $\Uparrow^{\mathtt{X<:Integer}}$ List<X>.

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 $\Delta(\mathtt{X}) = (\mathtt{-}, \mathtt{T})$, 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. Basically, in order to close $\mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>}$, the type arguments $\overline{\mathtt{T}}$ have to be closed first. If $\mathtt{T}_i$ closes to itself—that is, none of the abstract types occurs in $\mathtt{T}_i$—the type argument and its variance annotation remain the same; on the other hand, when $\mathtt{T}_i$ closes to a proper supertype $\mathtt{U}_i$ (i.e. $\mathtt{T}_i \neq \mathtt{U}_i$), the resulting type must be covariant in that argument, thus the least upper bound of + and $\mathtt{v}_i$ is attached. For example, we can derive

$$\mathtt{X} \Downarrow_{\mathtt{X<:Integer}} \mathtt{Integer}$$
$$\mathtt{List\texttt{<}X\texttt{>}} \Downarrow_{\mathtt{X<:Integer}} \mathtt{List\texttt{<}+Integer\texttt{>}}$$
$$\mathtt{List\texttt{<}List\texttt{<}X\texttt{>}\texttt{>}} \Downarrow_{\mathtt{X<:Integer}} \mathtt{List\texttt{<}+List\texttt{<}+Integer\texttt{>}\texttt{>}}$$

Subtyping:

$$\Delta \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{T} \qquad (\text{S-Refl})$$

$$\frac{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T} \qquad \Delta \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{U}}{\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{U}} \qquad (\text{S-Trans})$$

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

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

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

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

Well-formed Types:

$$\Delta \vdash \mathtt{Object}\ \text{ok} \qquad (\text{WF-Object})$$

$$\frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}} \qquad (\text{WF-TVar})$$

$$\frac{\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}\ \{\ldots\} \qquad \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{T}} \mathrel{\mathtt{<:}} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{vT}}\mathtt{>}\ \text{ok}} \qquad (\text{WF-Class})$$

Fig. 5. Subtyping and Well-formed Types.

but cannot derive

$$\mathtt{List<List<X>>} \Downarrow_{\mathtt{X<:Integer}} \mathtt{List<List<+Integer>>}.$$

An exception is the case in which a type argument is a type variable and bounded by `-` or `*`; the type variable itself does not close to anything but the whole type can close 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, we can derive

$$\mathtt{List<X>} \Downarrow_{\mathtt{X:>Integer}} \mathtt{List<-Integer>}$$
$$\mathtt{List<+X>} \Downarrow_{\mathtt{X:>Integer}} \mathtt{List<*Integer>}.$$

Subtyping. A judgment for subtyping $\Delta \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$ is read "`S` is a subtype of `T` under Δ." As usual, the subtyping relation is reflexive and transitive (S-Refl and S-Trans). 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 $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$ is declared to extend another type D<$\overline{\texttt{S}}$>, any (invariant) instantiation C<$\overline{\texttt{T}}$> is a subtype of D<[$\overline{\texttt{T}}/\overline{\texttt{X}}$]$\overline{\texttt{S}}$>. A supertype of a non-instance type is obtained by opening it and closing the supertype of the opened type. For example, under the class declaration

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

we can derive

$$\vdash \texttt{PP<+Nm,-Nm>} <: \texttt{Pair<+Pair<+Nm,-Nm>,+Pair<+Nm,-Nm>>}$$

because

$$\vdash \texttt{PP<+Nm,-Nm>} \Uparrow^{\texttt{Z<:Nm,W:>Nm}} \texttt{PP<Z,W>}$$

and

$$\begin{array}{l}[\texttt{Z/X,W/Y}](\texttt{Pair<Pair<X,Y>,Pair<X,Y>>}) \\ \quad \Downarrow_{\texttt{Z<:Nm,W:>Nm}} \texttt{Pair<+Pair<+Nm,-Nm>,+Pair<+Nm,-Nm>>}\end{array}$$

Finally, the rule S-VAR deals with variance. The first conditional premise means that, if a variance annotation $\texttt{v}_i$ for $\texttt{X}_i$ is either contravariant or invariant, the corresponding type arguments $\texttt{S}_i$ and $\texttt{T}_i$ must satisfy $\Delta \vdash \texttt{T}_i <: \texttt{S}_i$; similarly for the second one.

Type Well-formedness. A judgment for type well-formedness is of the form $\Delta \vdash$ T 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<$\overline{\texttt{vT}}$> 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 (except for casts), and shown in Figure 6. In what follows, we use $\mathit{bound}_\Delta(\texttt{T})$ defined by: $\mathit{bound}_\Delta(\texttt{X}) = \mathit{bound}_\Delta(\texttt{S})$ if $\Delta(\texttt{X}) = (\texttt{+}, \texttt{S})$ and $\mathit{bound}_\Delta(\texttt{N}) = \texttt{N}$. Key rules are T-FIELD and T-METHOD. (In fact, other rules are essentially the same as ones in Featherweight GJ.) When a field or method is accessed, the receiver type is opened and the result type is closed. The typing rule for method invocations checks that the actual type arguments $\overline{\texttt{V}}$ satisfy the bounds $\overline{\texttt{P}}$ and that the type of each actual parameter is a subtype of the corresponding formal. Since the opened receiver type C<$\overline{\texttt{T}}$> 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 result type is obtained by removing abstract types in Δ' with the close operation. The rule T-SCAST is called stupid cast rule; although stupid casts are disallowed in Java proper, it is needed to prove type soundness via subject reduction—see Igarashi et al. [2001a] for more details.

A judgment M ok in C<$\overline{\texttt{X}}$◁$\overline{\texttt{N}}$> for typing methods means that method M is well-typed if it appears in class C<$\overline{\texttt{X}}$◁N>. It requires the auxiliary predicate *override* to check correct method overriding. *override*(m, N, <$\overline{\texttt{Y}}$◁$\overline{\texttt{P}}$>$\overline{\texttt{T}}\rightarrow\texttt{T}_0$) holds if and only if either (1) a method with the same name m is not defined in the superclass N; or (2) it is defined 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

Expression Typing:

$$\Delta;\Gamma \vdash \texttt{x} \in \Gamma(\texttt{x}) \qquad \text{(T-Var)}$$

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

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \Delta \vdash \mathit{bound}_\Delta(\texttt{T}_0) \Uparrow^{\Delta'} \texttt{C<}\overline{\texttt{T}}\texttt{>}\\ \mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\texttt{\to U}_0 \qquad \{\overline{\texttt{Y}}\} \cap \mathit{dom}(\Delta') = \emptyset\\ \Delta \vdash \overline{\texttt{V}} \text{ ok} \qquad \Delta, \Delta' \vdash \overline{\texttt{V}} \texttt{<:} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}}\\ \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta, \Delta' \vdash \overline{\texttt{S}} \texttt{<:} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}} \qquad [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0 \Downarrow_{\Delta'} \texttt{T}\end{array}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \in \texttt{T}} \qquad \text{(T-Invk)}$$

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

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

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \Delta \vdash \texttt{T} \text{ ok}\\ \Delta \vdash \mathit{bound}_\Delta(\texttt{T}_0) \not\texttt{<:} \mathit{bound}_\Delta(\texttt{T}) \qquad \Delta \vdash \mathit{bound}_\Delta(\texttt{T}) \not\texttt{<:} \mathit{bound}_\Delta(\texttt{T}_0)\end{array}}{\Delta;\Gamma \vdash \texttt{(T)e}_0 \in \texttt{T}} \qquad \text{(T-SCast)}$$

Method Typing:

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{N}) = \texttt{<}\overline{\texttt{Z}} \lhd \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}}\texttt{\to U}_0 \text{ implies } [\overline{\texttt{Y}}/\overline{\texttt{Z}}](\overline{\texttt{Q}}, \overline{\texttt{U}}, \texttt{U}_0) = \overline{\texttt{P}}, \overline{\texttt{T}}, \texttt{T}_0}{\mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}}\texttt{\to T}_0)}$$

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

Class Typing:

$$\frac{\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}} \vdash \overline{\texttt{N}}, \texttt{D<}\overline{\texttt{S}}\texttt{>}, \overline{\texttt{T}} \text{ ok} \qquad \overline{\texttt{M}} \text{ ok in } \texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>}}{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{D<}\overline{\texttt{S}}\texttt{> \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}} \text{ ok}} \qquad \text{(T-Class)}$$

Fig. 6. Typing Rules.

and `this` is given type `C<X̄>`. 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 (`L` ok) if all the methods are well typed.

5.3 Operational Semantics

The reduction relation is of the form $\texttt{e} \longrightarrow \texttt{e}'$, read "expression `e` reduces to expression $\texttt{e}'$ in one step." We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

Reduction Rules:

$$\frac{\mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}}{\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i} \qquad \text{(R-Field)}$$

$$\frac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{).<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}/\mathtt{this}]\mathtt{e}_0} \qquad \text{(R-Invk)}$$

$$\frac{\emptyset \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: \mathtt{T}}{\mathtt{(T)new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}} \qquad \text{(R-Cast)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}_0{}'\mathtt{.f}} \qquad \text{(RC-Field)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{e}_0\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \longrightarrow \mathtt{e}_0{}'\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)}} \qquad \text{(RC-Inv-Recv)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{e}_0\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{e}_0\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\ldots\mathtt{,e}_i{}'\mathtt{,}\ldots\mathtt{)}} \qquad \text{(RC-Inv-Arg)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i{}'}{\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\ldots\mathtt{,e}_i\mathtt{,}\ldots\mathtt{)} \longrightarrow \mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\ldots\mathtt{,e}_i{}'\mathtt{,}\ldots\mathtt{)}} \qquad \text{(RC-New-Arg)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'}{\mathtt{(T)e}_0 \longrightarrow \mathtt{(T)e}_0{}'} \qquad \text{(RC-Cast)}$$

Fig. 7. Reduction Rules.

The reduction rules, which are essentially the same as the ones in Featherweight GJ, are given in Figure 7. There are three computation rules, one for field access, one for method invocation, and one for typecasts. Field access $\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{).f}_i$ looks up and obtains field names $\overline{\mathtt{f}}$ of $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ with $\mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})$; then it reduces to the constructor argument $\mathtt{e}_i$ of the corresponding position. Method invocation $\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{).<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{d}}\mathtt{)}$ first looks up $\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})$ and obtains a pair of a sequence of formal arguments $\overline{\mathtt{x}}$ and the method body; then, it reduces to the method body in which $\overline{\mathtt{x}}$ are replaced with the actual arguments $\overline{\mathtt{d}}$ and `this` with the receiver $\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ to stand for replacing $\mathtt{x}_1$ by $\mathtt{d}_1$, …, $\mathtt{x}_n$ by $\mathtt{d}_n$, and `y` by `e` in the expression $\mathtt{e}_0$. The expression $\mathtt{(T)new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$ reduces to the subject $\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$ 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 $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), which also appear in Figure 7.

5.4 Properties

Type soundness (Theorem 5.4.7) is shown through subject reduction and progress properties [Wright and Felleisen 1994]. 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 5.4.1 (SUBJECT REDUCTION). *If* $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$ *where* Δ *has non-variable bounds and* $\texttt{e} \longrightarrow \texttt{e}'$*, then* $\Delta;\Gamma \vdash \texttt{e}' \in \texttt{S}$ *and* $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$ *for some* S.

PROOF. By induction on the derivation of $\texttt{e} \longrightarrow \texttt{e}'$ with a case analysis on the last reduction rule used. The basic structure is similar to the one for Featherweight GJ [Igarashi et al. 2001a].

Here, we show main cases and only state required lemmas; their detailed proofs are shown in Appendix A.

Case R-FIELD. $\quad \texttt{e} = \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).f}_i \qquad \textit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \texttt{e}' = \texttt{e}_i$

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

$$\begin{array}{lll}
\Delta;\Gamma \vdash \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in \texttt{C<}\overline{\texttt{T}}\texttt{>} & \Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \Uparrow^{\emptyset} \texttt{C<}\overline{\texttt{T}}\texttt{>} & \\
\Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} & \Delta \vdash \overline{\texttt{S}} \texttt{ <: } \overline{\texttt{U}} & \texttt{U}_i \Downarrow_{\emptyset} \texttt{U}_i (= \texttt{T}).
\end{array}$$

In particular, $\Delta;\Gamma \vdash \texttt{e}_i \in \texttt{S}_i$ finishes the case.

Case R-INVK. $\quad \texttt{e} = \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{d}}\texttt{)}$
$\qquad\qquad\quad \textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{x}}\texttt{.e}_0$
$\qquad\qquad\quad \texttt{e}' = [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0$

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

$$\begin{array}{lll}
\Delta;\Gamma \vdash \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in \texttt{C<}\overline{\texttt{T}}\texttt{>} & \Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \Uparrow^{\emptyset} \texttt{C<}\overline{\texttt{T}}\texttt{>} & \\
\textit{mtype}(\texttt{m, C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U}_0 & \Delta \vdash \overline{\texttt{V}} \text{ ok} & \Delta \vdash \overline{\texttt{V}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}} \\
\Delta;\Gamma \vdash \overline{\texttt{d}} \in \overline{\texttt{S}} & \Delta \vdash \overline{\texttt{S}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}} & \\
[\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0 \Downarrow_{\emptyset} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0 (= \texttt{T}) & \Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \text{ ok} &
\end{array}$$

Then, we must show that $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0 \in \texttt{T}_0$ for some $\texttt{T}_0$ such that $\Delta \vdash \texttt{T}_0 \texttt{ <: } \texttt{T}$. Main lemmas required are shown below: (1) one to ensure the type of the method body to be executed matches the one obtained by *mtype* (Lemma 5.4.2); and (2) one to ensure substitution preserves typing (Lemma 5.4.3). (In order to show Lemma 5.4.2, we also need preservation of typing and subtyping under *type* substitution, shown in Appendix. So, readers familiar with proofs of subject reduction for typed lambda-calculi like $F_{\leq}$ [Cardelli et al. 1994] will notice many similarities.)

LEMMA 5.4.2. *If* $\textit{mtype}(\texttt{m, C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U}_0$ *and* $\textit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>, C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{x}}\texttt{.e}_0$ *with* $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}$ *ok and* $\Delta \vdash \overline{\texttt{V}}$ *ok and* $\Delta \vdash \overline{\texttt{V}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}}$*, then there exist some* N *and* S *such that* $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{ <: } \texttt{N}$ *and* $\Delta \vdash \texttt{N}$ *ok and* $\Delta \vdash \texttt{S} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0$ *and* $\Delta; \overline{\texttt{x}} : [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}, \texttt{this} : \texttt{N} \vdash \texttt{e}_0 \in \texttt{S}$.

LEMMA 5.4.3 (TERM SUBSTITUTION PRESERVES TYPING). *For any* Δ *that has non-variable bounds, if* $\Delta;\Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e} \in \texttt{T}_0$ *and* $\Delta;\Gamma \vdash \overline{\texttt{d}} \in \overline{\texttt{S}}$ *where* $\Delta \vdash \overline{\texttt{S}} \texttt{ <: } \overline{\texttt{T}}$*, then* $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e} \in \texttt{S}_0$ *for some* $\texttt{S}_0$ *such that* $\Delta \vdash \texttt{S}_0 \texttt{ <: } \texttt{T}_0$.

Then, by Lemma 5.4.2, $\Delta; \overline{\texttt{x}} : [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}, \texttt{this} : \texttt{N} \vdash \texttt{e}_0 \in \texttt{S}_0$ for some N and $\texttt{S}_0$ such that $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{ <: } \texttt{N}$ where $\Delta \vdash \texttt{N}$ ok, and $\Delta \vdash \texttt{S}_0 \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0$ where $\Delta \vdash \texttt{S}_0$ ok. Then, by Weakening (Lemma A.1) and 5.4.3, $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}_0 \in \texttt{T}_0$ for some $\texttt{T}_0$ such that $\Delta \vdash \texttt{T}_0 \texttt{ <: } \texttt{S}_0$. By the rule S-TRANS, we have $\Delta \vdash \texttt{T}_0 \texttt{ <: } \texttt{T}$, finishing the case.

For RC-Field and RC-Invk, we need to show that, if the type of the receiver changes to a subtype, the result of field/method type look-up is "compatible" with the original result in the following sense.

Lemma 5.4.4. *If* Δ *has non-variable bounds and* $\Delta \vdash \mathit{bound}_\Delta(\mathtt{T}) \Uparrow^{\Delta_1} \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ *and* $\mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ *and* $\mathtt{U}_i \Downarrow_{\Delta_1} \mathtt{U}_0$*, then for any* S *such that* $\Delta \vdash \mathtt{S} \texttt{<:} \mathtt{T}$ *and* $\Delta \vdash \mathtt{S}$ *ok, it holds that* $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}) \Uparrow^{\Delta_2} \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ *and* $\mathit{fields}(\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}) = \ldots, \overline{\mathtt{V}}\ \overline{\mathtt{f}}$ *and* $\mathtt{V}_i \Downarrow_{\Delta_2} \mathtt{V}_0$ *and* $\Delta \vdash \mathtt{V}_0 \texttt{<:} \mathtt{U}_0$ *for some* Δ_2, $\overline{\mathtt{f}}$, $\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$, $\overline{\mathtt{V}}$, *and* $\mathtt{V}_0$.

Lemma 5.4.5. *If* Δ *has non-variable bounds and* $\Delta \vdash \mathtt{T}$ *ok and* $\Delta \vdash \mathit{bound}_\Delta(\mathtt{T}) \Uparrow^{\Delta_1} \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{U}_0$ *and* $\Delta \vdash \overline{\mathtt{V}}, \overline{\mathtt{W}}$ *ok and* $\Delta, \Delta_1 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$ *and* $\Delta, \Delta_1 \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ *and* $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_1} \mathtt{V}_0$*, then for any* S *such that* $\Delta \vdash \mathtt{S} \texttt{<:} \mathtt{T}$ *and* $\Delta \vdash \mathtt{S}$ *ok, we have* $\Delta \vdash \mathit{bound}_\Delta(\mathtt{S}) \Uparrow^{\Delta_2} \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}$ *and* $\mathit{mtype}(\mathtt{m}, \mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}'\texttt{>}\overline{\mathtt{U}}' \to \mathtt{U\_'0}$ *and* $\Delta, \Delta_2 \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}'$ *and* $\Delta, \Delta_2 \vdash \overline{\mathtt{W}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}'$ *and* $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_2} \mathtt{V}_0{}'$ *and* $\Delta \vdash \mathtt{V}_0{}' \texttt{<:} \mathtt{V}_0$.

Given these lemmas, the following two cases are easy.

Case RC-Field. $\quad \mathtt{e} = \mathtt{e}_0\texttt{.}\mathtt{f}_i \quad \mathtt{e}' = \mathtt{e}_0{}'\texttt{.}\mathtt{f}_i \quad \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$

By the rule T-Field, we have

$$\begin{array}{ll} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 & \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \\ \mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} & \mathtt{S}_i \Downarrow_{\Delta'} \mathtt{T} \end{array}$$

By the induction hypothesis, $\Delta; \Gamma \vdash \mathtt{e}_0{}' \in \mathtt{T}_0{}'$ for some $\mathtt{T}_0{}'$ such that $\Delta \vdash \mathtt{T}_0{}' \texttt{<:} \mathtt{T}_0$. By Lemma 5.4.4, $\Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0{}') \Uparrow^{\Delta''} \mathtt{D}\texttt{<}\overline{\mathtt{U}}\texttt{>}$, $\mathit{fields}(\mathtt{D}\texttt{<}\overline{\mathtt{U}}\texttt{>}) = \ldots, \overline{\mathtt{V}}\ \overline{\mathtt{f}}$, and $\mathtt{V}_i \Downarrow_{\Delta''} \mathtt{V}_i{}'$ and $\Delta \vdash \mathtt{V}_i{}' \texttt{<:} \mathtt{T}$. Therefore, by the rule T-Field, $\Delta; \Gamma \vdash \mathtt{e}_0{}'\texttt{.f} \in \mathtt{V}_i{}'$, finishing the case.

Case RC-Inv-Recv. $\quad \mathtt{e} = \mathtt{e}_0\texttt{.<}\overline{\mathtt{V}}\texttt{>m(}\overline{\mathtt{e}}\texttt{)} \quad \mathtt{e}' = \mathtt{e}_0{}'\texttt{.<}\overline{\mathtt{V}}\texttt{>m(}\overline{\mathtt{e}}\texttt{)} \quad \mathtt{e}_0 \longrightarrow \mathtt{e}_0{}'$

By the rule T-Invk, we have

$$\begin{array}{lll} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 & \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} & \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{U}_0 & & \\ \Delta \vdash \overline{\mathtt{V}} \text{ ok} & \Delta \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} & \Delta \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \Delta, \Delta' \vdash \overline{\mathtt{S}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} & [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta'} \mathtt{T}. & \end{array}$$

By the induction hypothesis, $\Delta; \Gamma \vdash \mathtt{e}_0{}' \in \mathtt{T}_0{}'$ for some $\mathtt{T}_0{}'$ such that $\Delta \vdash \mathtt{T}_0{}' \texttt{<:} \mathtt{T}_0$. By Lemma 5.4.5,

$$\begin{array}{ll} \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}'_0) \Uparrow^{\Delta''} \mathtt{D}\texttt{<}\overline{\mathtt{W}}\texttt{>} & \mathit{mtype}(\mathtt{m}, \mathtt{D}\texttt{<}\overline{\mathtt{W}}\texttt{>}) = \texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}'\texttt{>}\overline{\mathtt{U}}' \to \mathtt{U}_0{}' \\ \Delta \vdash \overline{\mathtt{V}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}' & \Delta, \Delta'' \vdash \overline{\mathtt{S}} \texttt{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}' \\ [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0{}' \Downarrow_{\Delta''} \mathtt{T}' & \Delta \vdash \mathtt{T}' \texttt{<:} \mathtt{T}. \end{array}$$

Then, by the rule T-Invk, $\Delta; \Gamma \vdash \mathtt{e}_0{}'\texttt{.<}\overline{\mathtt{V}}\texttt{>m(}\overline{\mathtt{e}}\texttt{)} \in \mathtt{T}'$, finishing the case.

Case R-Cast, RC-Cast, RC-Inv-Arg, RC-New-Arg. Easy. $\square$

Theorem 5.4.6 (Progress). *Suppose* e *is a well-typed expression.*

(1) *If* e *has* $\texttt{new C<}\overline{\mathtt{T}}\texttt{>(}\overline{\mathtt{e}}\texttt{).f}$ *as a subexpression, then* $\mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} = \mathtt{f}_i$.

(2) *If* e *has* $\texttt{new C<}\overline{\mathtt{T}}\texttt{>(}\overline{\mathtt{e}}\texttt{).<}\overline{\mathtt{V}}\texttt{>m(}\overline{\mathtt{d}}\texttt{)}$ *as a subexpression, then* $\mathit{mbody}(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \overline{\mathtt{x}}\texttt{.}\mathtt{e}_0$ *and* $|\overline{\mathtt{x}}| = |\overline{\mathtt{d}}|$.

PROOF. Immediate from the typing rules for object instantiation, field access, and method invocation. □

THEOREM 5.4.7 (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)new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`)` *where* $\emptyset \vdash$ `C<`$\overline{\texttt{T}}$`>` $\not<:$ `T`.

PROOF. Follows from Theorems 5.4.1 and 5.4.6. □

6. RELATED LANGUAGE MECHANISMS

Parametric Classes and Variance. There have been several languages, such as Eiffel [Meyer 1986; 1992], POOL [America and van der Linden 1990], Strongtalk [Bracha and Griswold 1993; Bracha 1996] and NextGen [Cartwright and Steele Jr. 1998], that support variance for parametric classes. 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 that 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, which can be a daunting task. Day et al. [1995] 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 appropriate 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 [1989] proposed to *infer*, rather than *declare*, variance annotations on each type parameters of a given class. 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 of Eiffel, in which every parametric class is regarded as invariant [Interactive Software Engineering 2001].

Theoretical foundations of this classical approach to variance have been considered in the context of typed λ-calculi [Cardelli 1990; Duggan and Compagnoni 1999; Steffen 1998], where type operators (functions from types to types) are equipped with a variance property, often called polarity. (Other pieces of work [Barthe and van Raamsdonk 2000; Barthe and Frade 1999] consider ML-like datatypes with subtyping and covariant type operators; they also fall into this category.) The resulting type system is rather complicated and its meta-theory is 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 for types with upper bounds.

Parametric Methods. One may wonder if parametric methods with bounded polymorphism can be used for the examples shown in Section 3; 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.<Integer>fillFrom(vi,0);
vn.<Float>fillFrom(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 super 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 `super` 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. Moreover, even those examples of combining parametric methods and variant parametric types, in principle, could be written only with parametric methods (with lower bounds). For example, a method to copy elements of a list to another list can be written as a parametric method with three type parameters:

```
<E, E1 super E, E2 extends E> void copy(List<E1> dest, List<E2> src){...}
```

rather than one with one type parameter and variant parametric types:

```
<E> void copy(List<-E> dest, List<+E> src){...}
```

Although it looks as if variant parametric types can be dispensed with by using parametric methods, we believe they are complementary machinery.

On one hand, variant parametric types provide a means to specify a set of different instantiations of the same generic classes in a succinct way. One problem of heavy use of parametric methods is that the signatures of those methods tend to be longer and harder to understand. 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. Another merit of variant parametric types is that they allow to mix different kinds of elements in one data structure: for example, using `Vector<+Vector<+Number>>` for fields allows to store vectors of integers, vectors of floating numbers, etc., in one (outer) vector.

On the other hand, parametric methods can 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 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.

We should note also that simulating contravariance with parametric methods requires type parameters with lower bounds. However, they are found in very few languages (a recent exception is Scala [Odersky et al. 2004]) and their theory has not been well investigated so far. 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 such a way that they always consume elements. 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. Let us show an example combining nested parametric types, bivariance, and parametric methods:

```
<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`.

Virtual Types. As we have already mentioned, the idea of variant parametric types has emerged from structural virtual types proposed by Thorup and Torgersen [1999]. In a language with virtual types [Madsen and Møller-Pedersen 1989; Thorup 1997], 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 [1998] 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 many 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 Thorup and Torgersen [1999], 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 Igarashi and Pierce [2002], programming with virtual types is shown to be simulated (to some degree) by exploiting bounded and manifest existential types [Harper and Lillibridge 1994; Leroy 1994], 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 bivariance. Other differences are as follows. On one hand, virtual types seem more suitable for programming with extensible mutually recursive classes/interfaces [Bruce et al. 1998; Thorup and Torgersen 1999; Bruce and Vanderwaart 1999]. 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 Thorup and Torgersen [1999]) how to encode such programming in structural virtual types.

Existential Types in Object-Oriented Languages. Aside from object encoding, the idea of existential types can actually be seen in several mechanisms for object-oriented languages, often in disguised (and limited) form. We will discuss such examples here. Note that, however, just like our interpretation of variant parametric types, those existential types we will mention do not come with witness types and are close to the classical interpretation of the quantifier, mentioned before.

In the language $\mathcal{LOOM}$ [Bruce et al. 1997], subtyping is dropped in favor of matching [Bruce 1994; Bruce et al. 1995], 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 Bruce et al. [1997], `#T` is considered a "match-bounded" existential type ∃`X<#T.X`, where `X<#T` stands for "`X` matches `T`."

Raw types [Bracha et al. 1998] of GJ are also close to bounded existential types [Igarashi et al. 2001b]. 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 import old Java code into GJ. `Vector` is considered a supertype of every parametric type `Vector<T>` and behaves like ∃`X<:Object.Vector<X>` in the sense that reading elements gives `Object` and writing elements would not always be safe. One significant difference is that certain unsafe operations including writing elements into a raw vector are permitted only with a compiler warning.

Explicit Use of Existential Types. It would also 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 ∃`X.Vector<X>` 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.

Wildcards in Java 5.0. Based on our proposal here, the latest release of the Java Programming Language (shipped with JDK 5.0) is extended with the typing mechanism called *wildcards* [Torgersen et al. 2004] as well as generics. Wildcards are types of the form `C<? extends T>`, `C<? super T>`, or `C<?>`. Each of them corresponds to `C<+T>`, `C<-T>`, or `C<*>`, respectively, and obey (almost) the same subtyping rules and access restriction we have described. One notable difference is that one can read elements from `Vector<? super T>` and even `Vector<?>` to give type `Object`. Similarly, it is permitted to put `null` as an element of `Vector<? extends T>`. So, access restriction and variance do not exactly match. As a byproduct, `Vector<? extends Object>` and `Vector<?>` are compatible types (one type is a subtype of the other and implicit casts are allowed in both directions), so are `Vector<? super Object>` and `Vector<Object>`.

In the course of the development, it turns out that naive combination of variance and parametric methods (as is formalized in this article) has practical limitations. Suppose there is a (static) method

```
<X> Set<X> unmodifiableSet(Set<X> set) {...}
```

that constructs a read-only view of a given set. This method, however, cannot be applied to `Set<?>` in our language because the actual type argument for `X` would have to be the unknown element type, which cannot be expressed in the language. Writing this method as

```
Set<?> unmodifiableSet(Set<?> set) {...}
```

does not solve the problem. It can take any sets, including `Set<?>`, but the result type is always `Set<?>` and the element type infomation of the input is lost.

Torgersen et al. observed that it was actually safe to invoke `unmodifiableSet()` on `Set<?>` and so simply allowed such invocations. As for specifying actual type arguments, the unknown element type of `Set<?>` does not have to be written down in a program, because Java allows actual type arguments to be omitted! Thus, invocation of `unmodifiableSet()` is written just as:

```
Set<?> set = ...;
set = unmodifiableSet(set);
```

Then, a natural question is "what is really the omitted type argument?" One possible answer, which again uses the correspondence to existential types, is that the type argument is the hidden abstract type `X` of `∃X.Set<X>`. So, it can be interpreted as

```
∃X.Set<X> set = ...;
set =
  open set as [Y,x] in <Y>unmodifiableSet(x);
```

Since types hidden under `?` is "captured" by a formal type argument of a polymorphic method, this mechanism is called *wildcard capture*. The calculus we have presented in the previous section cannot directly express wildcard capture. Extending the calculus to deal with wildcard capture will be interesting future work.

It is also reported that wildcard capture has contributed to give more appropriate interfaces to library methods. `Collections.shuffle()`, which takes a list and shuffles its elements, is given the following signature in the GJ library:

```
<X> void shuffle(List<X> list)
```

which allows arbitrary lists to be shuffled. From the users' point of view, however, a more concise one

```
void shuffle(List<?> list)
```

is arguably desirable because it does not have a type argument, which in fact does not contribute to the result type. Wildcard capture makes it possible to switch from the former to the latter—the class will have the former as a private method and the latter as a "bridge" method, which only calls the former by using wildcard capture.

We close this section by showing some interesting use of wildcards in Java library classes, developed by Sun Microsystems. All the method signature can be found in JDK5.0.

The class `Collections` consists of static methods that operate on or return collections. It includes a static method `sort()` of the following signature

```
<T implements Comparable<? super T>> void sort(List<T> list)
```

for list sorting. Here, the type variable `T` representing the element type is given the bound `Comparable<? super T>` where `Comparable` is defined as:

```
interface Comparable<T>{
   public int compareTo(T o);
}
```

Thus, roughly speaking, this signature of `sort()` allows sorting of a list of objects comparable to themselves. For example, it is used as follows:

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

List<Elm> v = ...;
Collection.<Elm>sort(v)
   // OK -- Elm <: Comparable<Elm> <: Comparable<? super Elm>
```

It is quite beneficial to use the contravariant type `Comparable<? super T>`, instead of `Comparable<T>`, especially when element types have subtypes: it also allows a vector of subtype of `Elm` to be sorted, thanks to contravariance.

```
class MyElm extends Elm {
  ...
  // int compareTo(Elm x);   is inherited
}

List<MyElm> v = ...;
Collection.<MyElm>sort(v);
```

Note that it holds that

```
MyElm <: Elm <: Comparable<Elm> <: Comparable<? super MyElm>.
```

The invocation of `sort()` above would not be allowed without using "`? super`," since `MyElm` is *not* a subtype of `Comparable<MyElm>`.

Another interesting example is found outside the collection framework. The class `Class<X>` represents the class of classes and interfaces. It is parameterized so that it is statically expressible in the type system which class is represented: an instance of the type `Class<C>` keeps information on the class `C`. It includes the following two methods

```
public X newInstance()
```

and

```
public Class<? super X> getSuperClass()
```

Method `newInstance()` creates a new object of class `X` by invoking the constructor without arguments (raising an exception if this is not defined). Method `getSuperClass()` returns the `Class` representation of the receiver's superclass: since the resulting type is a supertype of `T`, the method return value can be given the contravariant type `Class<? super X>`.

7. CONCLUSION AND FUTURE WORK

In this article, 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 inclusion 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 derived 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 "don't care" when used in a method signature.

The present idea is adapted to the latest release of the Java programming language, and have been implemented (but in a different guise, namely wildcards). Those types are indeed intensively used in library classes, witnessing the usefulness of variant parametric types.

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 [Igarashi et al. 2001a], and have proved type soundness.

Implementation issues are not addressed in this article 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 GJ [Bracha et al. 1998], and the .NET CLR [Syme and Kennedy 2001]—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 arguments[4]. In fact, the latest Java compiler, based on GJ, is implemented by using this technique. Other advanced implementation techniques where type arguments are maintained at run-time, such as LM translator [Viroli and Natali 2000; Viroli 2003], 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 approaches presented in Raynaund and Thierry [2001] and Palacz and Vitek [2003] may be worth investigating.

Unlike the core calculus presented here, Java 5.0 allows 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. It seems that it is harder than type inference without variant parametric types[5]—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.

APPENDIX

A. PROOFS OF LEMMAS REQUIRED FOR THEOREM 5.4.1

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

Lemma A.1 (Weakening). *Suppose* $\Delta, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}} \vdash \overline{\mathtt{N}}$ *ok and* $\Delta \vdash \mathtt{U}$ *ok.*

(1) *If* $\Delta \vdash \mathtt{S} \texttt{ <: } \mathtt{T}$*, then* $\Delta, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}} \vdash \mathtt{S} \texttt{ <: } \mathtt{T}$.

(2) *If* $\Delta \vdash \mathtt{S}$ *ok, then* $\Delta, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}} \vdash \mathtt{S}$ *ok.*

(3) *If* $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$*, then* $\Delta; \Gamma, \mathtt{x} : \mathtt{U} \vdash \mathtt{e} \in \mathtt{T}$ *and* $\Delta, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}; \Gamma \vdash \mathtt{e} \in \mathtt{T}$.

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

Lemma A.2 (Narrowing).

[4]The 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.

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

(*1*) *If* $\Delta, \mathtt{X}\texttt{<:}\mathtt{S} \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$ *and* $\Delta \vdash \mathtt{U} \texttt{ <: } \mathtt{S}$*, then* $\Delta, \mathtt{X}\texttt{<:}\mathtt{U} \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$.

(*2*) *If* $\Delta, \mathtt{X}\texttt{:>}\mathtt{S} \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$ *and* $\Delta \vdash \mathtt{S} \texttt{ <: } \mathtt{U}$*, then* $\Delta, \mathtt{X}\texttt{:>}\mathtt{U} \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$.

(*3*) *If* $\Delta, \mathtt{X} : (*, \mathtt{S}) \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$*, then* $\Delta, \mathtt{X} : (\mathtt{v}, \mathtt{U}) \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$.

PROOF. By induction on derivation of $\Delta, \mathtt{X}\texttt{<:}\mathtt{S} \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$ and $\Delta, \mathtt{X}\texttt{:>}\mathtt{S} \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$ and $\Delta, \mathtt{X} : (*, \mathtt{S}) \vdash \mathtt{T}_1 \texttt{ <: } \mathtt{T}_2$, respectively. □

The following lemmas (Lemmas A.3–A.5, and A.10) state that type substitution preserves derivability of judgments about typing.

LEMMA A.3.

(*1*) *If* $\Delta_1 \vdash \overline{\mathtt{S}} \texttt{ <: } [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *and* $\Delta_1 \vdash \overline{\mathtt{S}}$ *ok and* $\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>} \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{wU}}\texttt{>}$ *with none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1 *and none of type variables in dom*(Δ') *appearing in* $\overline{\mathtt{S}}$*, then* $\Delta_1, [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>} \Uparrow^{[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\Delta'} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{wU}}\texttt{>}$.

(*2*) *If* $\mathtt{S} \Downarrow_\Delta \mathtt{T}$ *where dom*(Δ) *and* $\overline{\mathtt{X}}$ *are distinct, then* $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S} \Downarrow_{[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\Delta} [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{T}$,

(*3*) $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \mathit{fields}([\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$*, and*

(*4*) $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$.

PROOF. By straightforward induction on the derivation of $\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{C}\texttt{<}\overline{\mathtt{vT}}\texttt{>} \Uparrow^{\Delta'} \mathtt{C}\texttt{<}\overline{\mathtt{wU}}\texttt{>}$ and $\mathtt{S} \Downarrow_\Delta \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$ and $\mathit{fields}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$ and $\mathit{mtype}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})$, respectively. □

LEMMA A.4 (TYPE SUBSTITUTION PRESERVES SUBTYPING). *If* $\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{S} \texttt{ <: } \mathtt{T}$ *and* $\Delta_1 \vdash \overline{\mathtt{U}} \texttt{ <: } [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1 \vdash \overline{\mathtt{U}}$ *ok and none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S} \texttt{ <: } [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

PROOF. By induction on the derivation of $\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{S} \texttt{ <: } \mathtt{T}$.

Case S-REFL, S-TRANS, S-VAR. Easy.

Case S-CLASS. Easily follows from Lemma A.3 (1) and (2).

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

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

Case S-LBOUND. Immediately follows from the fact that $\mathtt{X} \in \mathit{dom}(\Delta_1) \cup \mathit{dom}(\Delta_2)$. □

LEMMA A.5 (TYPE SUBSTITUTION PRESERVES TYPE WELL-FORMEDNESS). *If* $\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \mathtt{T}$ *ok and* $\Delta_1 \vdash \overline{\mathtt{U}} \texttt{ <: } [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1 \vdash \overline{\mathtt{U}}$ *ok and none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$ *ok.*

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

Case WF-OBJECT. Trivial.

Case WF-VAR. $\mathtt{T} = \mathtt{X} \quad \mathtt{X} \in \mathit{dom}(\Delta_1, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \Delta_2)$

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

Case WF-CLASS. $\quad \mathtt{T} = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{T}}\ \mathrm{ok}$
$\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2 \vdash \overline{\mathtt{T}}\ \mathtt{<:}\ [\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$
$\mathtt{class\ C<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{S}}\mathtt{>\ \{...\}}$

By the induction hypothesis, Δ_1, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$ ok. On the other hand, by Lemma A.4, Δ_1, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\ \mathtt{<:}\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$. Since $\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \vdash \overline{\mathtt{P}}$ by the rule T-CLASS, $\overline{\mathtt{P}}$ does not include any of $\overline{\mathtt{X}}$ as a free variable. Thus, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} = [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}}$, and finally, we have Δ_1, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{C<}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\mathtt{>}$ ok by WF-CLASS. □

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

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

To prove that type substitution preserves typing (Lemma A.10), we require several more auxiliary lemmas including Lemmas 5.4.4 and 5.4.5, already discussed in Section 5.

LEMMA A.7 (CLOSE YIELDS A SUPERTYPE W/O LOCAL TYPE VARIABLES). *If $\Delta, \Delta' \vdash \mathtt{S}$ ok and $\mathtt{S} \Downarrow_{\Delta'} \mathtt{T}$, then $\Delta, \Delta' \vdash \mathtt{S}\ \mathtt{<:}\ \mathtt{T}$ and $\Delta \vdash \mathtt{T}$ ok.*

PROOF. By structural induction on $\mathtt{S}$. □

LEMMA A.8. *Suppose $\Delta, \Delta_1 \vdash \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}$ ok and $\mathtt{C<}\overline{\mathtt{S}}\mathtt{>} \Downarrow_{\Delta_1} \mathtt{T}$ and $\Delta \vdash \mathtt{T} \Uparrow^{\Delta_2} \mathtt{C<}\overline{\mathtt{U}}\mathtt{>}$ and $\Delta_3 \vdash \mathtt{S}_0$ ok where $dom(\Delta_i)$ $(i = 1, 2, 3)$ are distinct from each other.*

(*1*) *If $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 \Downarrow_{\Delta_2} \mathtt{S}_0{}'$ and $\overline{\mathtt{X}}$ do not appear in Δ_1 or Δ_2, then $[\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0 \Downarrow_{\Delta_1} \mathtt{S}_0{}'$.*

(*2*) *If $\Delta, \Delta_2 \vdash \mathtt{U}_0\ \mathtt{<:}\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0$ and $\Delta \vdash \mathtt{U}_0$ ok with $\overline{\mathtt{X}}$ not appearing in Δ or Δ_1 or Δ_2, then $\Delta, \Delta_1 \vdash \mathtt{U}_0\ \mathtt{<:}\ [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0$.*

PROOF. By inspection of the derivations $\mathtt{C<}\overline{\mathtt{S}}\mathtt{>} \Downarrow_{\Delta_1} \mathtt{T}$ and $\Delta \vdash \mathtt{T} \Uparrow^{\Delta_2} \mathtt{C<}\overline{\mathtt{U}}\mathtt{>}$, for each $\mathtt{S}_i$, one of the following properties holds:

(1) $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}_i$ and $\mathtt{U}_i$ is equal to $\mathtt{S}_i$,

(2) $\mathtt{S}_i \Downarrow_{\Delta_1} \mathtt{S}_i{}'$ (so $\Delta, \Delta_1 \vdash \mathtt{S}_i\ \mathtt{<:}\ \mathtt{S}_i{}'$ by Lemma A.7) and $\mathtt{S}_i \neq \mathtt{S}_i{}'$ and $\mathtt{U}_i$ is a type variable $\mathtt{Y}$ and $\Delta_2(\mathtt{Y}) = (\mathtt{+}, \mathtt{S}_i{}')$, or

(3) $\mathtt{S}_i$ is a type variable $\mathtt{Z}$ and $\Delta_1(\mathtt{Z}) = (\mathtt{+}, \mathtt{V})$ and $\mathtt{U}_i$ is a type variable $\mathtt{Y}$ and $\Delta_2(\mathtt{Y}) = (\mathtt{+}, \mathtt{V})$.

Now, the first part is easily shown by induction on $\mathtt{S}_0$. For the second part, from the inspection above, there exist $\Delta', \Delta_1', \overline{\mathtt{Y}}, \overline{\mathtt{T}}, \overline{\mathtt{S}}''$ such that $\Delta_1 = \Delta', \Delta_1'$ and $\Delta_2 = \Delta', \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{T}}$ and $\overline{\mathtt{S}} = [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ and $\Delta, \Delta_1 \vdash \overline{\mathtt{S}}''\ \mathtt{<:}\ \overline{\mathtt{T}}$. (Take all $\mathtt{S}_i$ and $\mathtt{S}_i{}'$ where the second case applies as $\overline{\mathtt{S}}''$ and $\overline{\mathtt{T}}$, respectively.) Then, it follows from Lemma A.4 that $\Delta, \Delta', \Delta_1' \vdash [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}]\mathtt{U}_0\ \mathtt{<:}\ [\overline{\mathtt{S}}''/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0$. Since $\overline{\mathtt{Y}}$ do not apper in $\mathtt{U}_0$ or $\mathtt{S}_0$, we have $[\overline{\mathtt{S}}'/\overline{\mathtt{Y}}]\mathtt{U}_0 = \mathtt{U}_0$ and $[\overline{\mathtt{S}}''/\overline{\mathtt{Y}}][\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}_0 = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\mathtt{S}_0$, finishing the proof. □

LEMMA A.9.

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

(2) *If* $\Delta \vdash \texttt{T} <: \texttt{U}$ *and* $\texttt{S} \Downarrow_{\Delta',\texttt{X<:U}} \texttt{S}''$ *where* $dom(\Delta', \texttt{X<:U})$ *are fresh for* Δ*, then* $\texttt{S} \Downarrow_{\Delta',\texttt{X<:T}} \texttt{S}'$ *and* $\Delta \vdash \texttt{S}' <: \texttt{S}''$*.*

(3) *If* $\Delta \vdash \texttt{U} <: \texttt{T}$ *and* $\texttt{S} \Downarrow_{\Delta',\texttt{X:>U}} \texttt{S}''$ *where* $dom(\Delta', \texttt{X:>U})$ *are fresh for* Δ*, then* $\texttt{S} \Downarrow_{\Delta',\texttt{X:>T}} \texttt{S}'$ *and* $\Delta \vdash \texttt{S}' <: \texttt{S}''$*.*

(4) *If* $\texttt{S} \Downarrow_{\Delta',\texttt{X}:(*,\texttt{U})} \texttt{S}''$ *where* $dom(\Delta') \cup \{\texttt{X}\}$ *are fresh for* Δ*, then* $\texttt{S} \Downarrow_{\Delta',\texttt{X}:(\texttt{v},\texttt{T})} \texttt{S}'$ *and* $\Delta \vdash \texttt{S}' <: \texttt{S}''$*.*

Proof. By structural induction on S. □

Now, we prove Lemmas 5.4.4 and 5.4.5.

Proof of Lemma 5.4.4. By induction on the derivation $\Delta \vdash \texttt{S} <: \texttt{T}$ with the case analysis on the last rule used.

Case S-Refl. Trivial.

Case S-UBound. Trivial because $bound_\Delta(\texttt{S}) = bound_\Delta(\texttt{T})$.

Case S-LBound. Cannot happen.

Case S-Trans. Easy.

Case S-Class. $\texttt{class D<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{C<}\overline{\texttt{S}'}\texttt{> \{...\}}$
$\texttt{S} = \texttt{D<}\overline{\texttt{v}}\overline{\texttt{W}}\texttt{>} \qquad \Delta \vdash \texttt{S} \Uparrow^{\Delta_2} \texttt{D<}\overline{\texttt{W}'}\texttt{>} \qquad [\overline{\texttt{W}'}/\overline{\texttt{X}}]\texttt{C<}\overline{\texttt{S}'}\texttt{>} \Downarrow_{\Delta_2} \texttt{T}$

It follows that we can take $\texttt{U}_0$ as $\texttt{V}_0$ from Lemma A.8(1) applied to the fact that, $\mathit{fields}(\texttt{D<}\overline{\texttt{W}'}\texttt{>}) = \mathit{fields}(\texttt{C<}[\overline{\texttt{W}'}/\overline{\texttt{X}}]\overline{\texttt{S}'}\texttt{>}), \overline{\texttt{T}}\ \overline{\texttt{g}}$ for some $\overline{\texttt{T}}$ and $\overline{\texttt{g}}$ and $\texttt{C<}[\overline{\texttt{W}'}/\overline{\texttt{X}}]\overline{\texttt{S}'}\texttt{>} \Downarrow_{\Delta_2} \texttt{T}$ and $\Delta \vdash \texttt{T} \Uparrow^{\Delta_1} \texttt{C<}\overline{\texttt{T}}\texttt{>}$.

Case S-Var. $\texttt{S} = \texttt{C<}\overline{\texttt{v}\texttt{S}'}\texttt{>} \qquad \texttt{T} = \texttt{C<}\overline{\texttt{w}\texttt{T}'}\texttt{>} \qquad \overline{\texttt{v}} \le \overline{\texttt{w}}$
if $\texttt{w}_i \le \texttt{-}$, then $\Delta \vdash \texttt{T}_i{}' <: \texttt{S}_i{}'$ $\qquad$ if $\texttt{w}_i \le \texttt{+}$, then $\Delta \vdash \texttt{S}_i{}' <: \texttt{T}_i{}'$

We show only the case where $\texttt{v}_i\texttt{S}_i{}'$ and $\texttt{w}_i\texttt{T}_i{}'$ are identical for all but one i. The proof easily extends to general cases.

Subcase. $\texttt{w}_i = \texttt{o}$

Follows from the fact that $\texttt{v}_i = \texttt{o}$ and $\Delta \vdash \texttt{T}_i{}' <: \texttt{S}_i{}'$ and $\Delta \vdash \texttt{S}_i{}' <: \texttt{T}_i{}'$.

Subcase. $\texttt{w}_i = \texttt{+}$

We have, $\Delta \vdash \texttt{S}_i{}' <: \texttt{T}_i{}'$ and $\texttt{v}_i$ must be either + or o. If $\texttt{v}_i$ is +, then we have $\Delta \vdash \texttt{S} \Uparrow^{\Delta_1',\texttt{X<:S}_i{}'} \texttt{C<}\overline{\texttt{T}}\texttt{>}$ where $\Delta_1', \texttt{X<:T}_i{}' = \Delta_1$. By Lemma A.9(2), $\texttt{U}_i \Downarrow_{\Delta_1',\texttt{X<:S}_i{}'} \texttt{V}_i{}'$ and $\Delta \vdash \texttt{V}_i{}' <: \texttt{U}_i{}'$. The other case for $\texttt{v}_i = \texttt{o}$ is similar (use Lemma A.9).

Subcase. $\texttt{w}_i = \texttt{-}$ or $\texttt{w}_i = \texttt{*}$

Similar. □

Proof of Lemma 5.4.5. By induction on the derivation of $\Delta \vdash \texttt{S} <: \texttt{T}$ with the case analysis on the last rule used.

Case S-Refl. Trivial.

Case S-UBound. Trivial because $bound_\Delta(\texttt{S}) = bound_\Delta(\texttt{T})$.

Case S-LBound. Cannot happen.

Case S-Trans. Easy.

Case S-CLASS. $\quad$ `class D<`$\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}$`>`$\triangleleft$`C<`$\overline{\mathtt{S}'}$`> {...}`

$\mathtt{S} = \mathtt{D<}\overline{\mathtt{v}\mathtt{W}}\mathtt{>} \qquad \Delta \vdash \mathtt{S} \Uparrow^{\Delta_2} \mathtt{D<}\overline{\mathtt{W}'}\mathtt{>} \qquad [\overline{\mathtt{W}'}/\overline{\mathtt{X}}]\mathtt{C<}\overline{\mathtt{S}'}\mathtt{>} \Downarrow_{\Delta_2} \mathtt{T}$

By T-CLASS and T-METHOD, $\mathit{mtype}(\mathtt{D<}\overline{\mathtt{W}'}\mathtt{>}) = \mathit{mtype}(\mathtt{C<}[\overline{\mathtt{W}'}/\overline{\mathtt{X}}]\overline{\mathtt{S}'}\mathtt{>})$. Thus, the conclusion follows from Lemma A.8 and the fact that $\mathtt{C<}[\overline{\mathtt{W}'}/\overline{\mathtt{X}}]\overline{\mathtt{S}'}\mathtt{>} \Downarrow_{\Delta_2} \mathtt{T}$ and $\Delta \vdash \mathtt{T} \Uparrow^{\Delta_1} \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$.

Case S-VAR. $\quad \mathtt{S} = \mathtt{C<}\overline{\mathtt{v}\mathtt{S}'}\mathtt{>} \qquad \mathtt{T} = \mathtt{C<}\overline{\mathtt{w}\mathtt{T}'}\mathtt{>} \qquad \overline{\mathtt{v}} \leq \overline{\mathtt{w}}$
if $\mathtt{w}_i \leq \mathtt{-}$, then $\Delta \vdash \mathtt{T}_i{}' \mathrel{<:} \mathtt{S}_i{}'$
if $\mathtt{w}_i \leq \mathtt{+}$, then $\Delta \vdash \mathtt{S}_i{}' \mathrel{<:} \mathtt{T}_i{}'$

We show only the case where $\mathtt{v}_i\mathtt{S}_i{}'$ and $\mathtt{w}_i\mathtt{T}_i{}'$ are identical for all but one i. The proof easily extends to general cases.

Subcase. $\quad \mathtt{w}_i = \mathtt{o}$

Follows from the fact that $\mathtt{v}_i = \mathtt{o}$ and $\Delta \vdash \mathtt{T}_i{}' \mathrel{<:} \mathtt{S}_i{}'$ and $\Delta \vdash \mathtt{S}_i{}' \mathrel{<:} \mathtt{T}_i{}'$.

Subcase. $\quad \mathtt{w}_i = \mathtt{+}$

We have $\Delta \vdash \mathtt{S}_i{}' \mathrel{<:} \mathtt{T}_i{}'$ and $\mathtt{v}_i$ must be either `+` or `o`. If $\mathtt{v}_i$ is `+`, then it follows from Lemma A.2that $\Delta, \Delta_2 \vdash \overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}'}$ and $\Delta, \Delta_2 \vdash \overline{\mathtt{W}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}'}$ and we have $\Delta \vdash \mathtt{S} \Uparrow^{\Delta_1', \mathtt{X<:S}_i{}'} \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ where $\Delta_1', \mathtt{X<:T}_i{}' = \Delta_1$. By Lemma A.9(2), $[\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}_0 \Downarrow_{\Delta_1', \mathtt{X<:S}_i{}'} \mathtt{V}_0{}'$ and $\Delta \vdash \mathtt{V}_0{}' \mathrel{<:} \mathtt{V}_0$. The other case for $\mathtt{v}_i = \mathtt{o}$ is similar (use Lemmas A.4 and A.9(1) instead of LemmasA.2 and A.9(2), respectively).

Subcase. $\quad \mathtt{w}_i = \mathtt{-}$ or $\mathtt{w}_i = \mathtt{*}$

Similar. $\quad \square$

LEMMA A.10 (TYPE SUBSTITUTION PRESERVES TYPING). *If both Δ_1 and Δ_2 have non-variable bounds and $\Delta_1, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \Delta_2; \Gamma \vdash \mathtt{e} \in \mathtt{T}$ and $\Delta_1 \vdash \overline{\mathtt{U}} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ where $\Delta_1 \vdash \overline{\mathtt{U}}$ ok and none of $\overline{\mathtt{X}}$ appears in Δ_1, then $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e} \in \mathtt{S}$ for some $\mathtt{S}$ such that $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.*

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

Case T-VAR. $\quad$ Trivial.

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

By the induction hypothesis, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0$ and $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$ for some $\mathtt{S}_0$. By Lemma A.6,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\, \Delta_2}(\mathtt{T}_0)).$$

Then, by S-TRANS,

$$\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}},\, \Delta_2}(\mathtt{T}_0)).$$

By Lemma 5.4.4, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_\Delta(\mathtt{S}_0) \Uparrow^{\Delta''} \mathtt{D<}\overline{\mathtt{S}}\mathtt{>}$ and $\mathit{fields}(\mathtt{D<}\overline{\mathtt{S}}\mathtt{>}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}, \ldots$, and $\mathtt{S}_i \Downarrow_{\Delta''} \mathtt{T}'$ and $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{T}' \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$. Thus, by the rule T-FIELD, $\Delta_1, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}'$.

Case T-Invk.

$$\begin{array}{ll} \mathtt{e} = \mathtt{e}_0\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} & \Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) \Uparrow^{\Delta'} \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} & \mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{W}}\to\mathtt{W}_0 \\ \{\overline{\mathtt{Y}}\} \cap \mathit{dom}(\Delta') = \emptyset & \Delta \vdash \overline{\mathtt{V}} \text{ ok} \\ \Delta, \Delta' \vdash \overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} & \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \Delta, \Delta' \vdash \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}} & [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0 \Downarrow_{\Delta'} \mathtt{T} \end{array}$$

By the induction hypothesis,

$$\begin{array}{l} \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0 \in \mathtt{S}_0 \\ \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{S}_0 \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0 \end{array}$$

and

$$\begin{array}{l} \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2; [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} \in \overline{\mathtt{S}}' \\ \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \overline{\mathtt{S}}' \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} \end{array}$$

for some $\mathtt{S}_0$ and $\overline{\mathtt{S}}'$. By Lemmas A.6, A.5, A.4 and A.3, it is easy to show

$$\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_\Delta(\mathtt{T}_0))$$

and

$$\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \text{ ok}$$

and

$$\begin{array}{l} \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\Delta_2, \Delta') \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}} \end{array}$$

and

$$\begin{array}{l} \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}\mathtt{>}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}\to[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_0 \\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{W}_0 \Downarrow_{[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta'} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T} \end{array}$$

respectively.

Then, by Lemma 5.4.5 and the fact that $[\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{T} = [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T})$ for any $\mathtt{T}$, we have

$$\begin{array}{l} \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathit{bound}_{\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}(\mathtt{S}_0) \Uparrow^{\Delta''} \mathtt{D<}\overline{\mathtt{T}}'\mathtt{>} \\ \mathit{mtype}(\mathtt{m}, \mathtt{D<}\overline{\mathtt{T}}'\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}'\mathtt{>}\overline{\mathtt{W}}'\to{\mathtt{W}_0}' \\ \Delta_1,\, ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2), \Delta'' \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \mathrel{<:} [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta_1,\, ([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2), \Delta'' \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} \mathrel{<:} [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{W}}' \\ [[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}/\overline{\mathtt{Y}}]{\mathtt{W}_0}' \Downarrow_{\Delta''} \mathtt{T}' \\ \Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash \mathtt{T}' \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}. \end{array}$$

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

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

and, by the rule T-Invk,

$$\Delta_1,\, [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2 \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\mathtt{.<}\overline{\mathtt{V}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T}'$$

as required.

Case T-New, T-Cast, T-SCast. Easy. □

Now that Lemma A.10 is proved, we are ready to show Lemmas 5.4.2 and 5.4.3.

Proof of Lemma 5.4.2. By induction on the derivation of $\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{x}}.\mathtt{e}_0$.

Case MB-Class. $\quad$ `class C<`$\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}$`>`$\triangleleft$`D<`$\overline{\texttt{S}}$`> {...` $\overline{\texttt{M}}$`}`
`<`$\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}'$`> U`$_0{}'$ `m(`$\overline{\texttt{U}}'$ $\overline{\texttt{x}}$`){ return e`$_0{}'$`; }` $\in \overline{\texttt{M}}$

Then, $\mathit{mtype}(\texttt{m},\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}}\triangleleft([\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{P}}')\texttt{>}([\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{U}}')\rightarrow([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{U}_0{}')$, that is, $\overline{\texttt{P}} = [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{P}}'$ and $\overline{\texttt{U}} = [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{U}}'$ and $\texttt{U}_0 = [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{U}_0{}'$ and $\texttt{e}_0 = [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{e}_0{}'$. Now, let $\Gamma = \overline{\texttt{x}} : \overline{\texttt{U}}', \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>}$ and $\Delta' = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}, \overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}}$. By the rules T-Class and T-Method, we have $\Delta';\Gamma \vdash \texttt{e}_0{}' \in \texttt{S}_0$ and $\Delta' \vdash \texttt{S}_0 \texttt{ <: } \texttt{U}_0{}'$ for some $\texttt{S}_0$. Since $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}$ ok, we have $\Delta \vdash \overline{\texttt{T}} \texttt{ <: } [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}$ by the rule WF-Class. By Lemmas A.1, A.4, and A.10,

$$\Delta \vdash [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{S}_0 \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{U}_0{}'$$

and

$$\Delta; \overline{\texttt{x}} : [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{U}}', \texttt{this} : \texttt{C<}\overline{\texttt{T}}\texttt{>} \vdash [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{e}_0 \in \texttt{S}_0{}'$$

where

$$\Delta \vdash \texttt{S}_0{}' \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{S}_0.$$

Letting $\texttt{N} = \texttt{C<}\overline{\texttt{T}}\texttt{>}$ and $\texttt{S} = \texttt{S}_0{}'$ finishes the case. (Note that, without loss of generality, we can assume $[\overline{\texttt{V}}/\overline{\texttt{Y}}][\overline{\texttt{T}}/\overline{\texttt{X}}] = [\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{V}}/\overline{\texttt{Y}}]$.)

Case MB-Super. $\quad$ `class C<`$\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}$`>`$\triangleleft$`D<`$\overline{\texttt{S}}$`> {...` $\overline{\texttt{M}}$`}` $\quad \texttt{m} \notin \overline{\texttt{M}}$

Immediate from the induction hypothesis and the fact that $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{ <: } [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{S}}\texttt{>}$.
□

Proof of Lemma 5.4.3. By induction on the derivation of $\Delta;\Gamma,\overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e} \in \texttt{T}$ with a case analysis on the last rule used.

Case T-Var. $\quad \texttt{e} = \texttt{x}$

The case where $\texttt{x} \in \mathit{dom}(\Gamma)$ is immediate, since $[\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{x} = \texttt{x}$. On the other hand, if $\texttt{x} = \texttt{x}_i$ and $\texttt{T} = \texttt{T}_i$, then $\Delta;\Gamma \vdash \texttt{d}_i \in \texttt{S}_i$ finishing the case.

Case T-Field. $\quad \texttt{e} = \texttt{e}_0\texttt{.f}_i \qquad \Delta;\Gamma,\overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}_0 \in \texttt{T}_0$
$\Delta \vdash \mathit{bound}_\Delta(\texttt{T}_0) \Uparrow^{\Delta'} \texttt{C<}\overline{\texttt{U}}\texttt{>} \qquad \mathit{fields}(\texttt{C<}\overline{\texttt{U}}\texttt{>}) = \overline{\texttt{S}}\ \overline{\texttt{f}}$
$\texttt{S}_i \Downarrow_{\Delta'} \texttt{T}$

By the induction hypothesis, $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}_0 \in \texttt{S}_0$ for some $\texttt{S}_0$ such that $\Delta \vdash \texttt{S}_0 \texttt{ <: } \texttt{T}_0$. By Lemma 5.4.4, $\Delta \vdash \mathit{bound}_\Delta(\texttt{S}_0) \Uparrow^{\Delta'} \texttt{D<}\overline{\texttt{V}}\texttt{>}$, $\mathit{fields}(\texttt{D<}\overline{\texttt{V}}\texttt{>}) = \overline{\texttt{W}}\ \overline{\texttt{f}}, \ldots$, and $\texttt{W}_i \Downarrow_{\Delta''} \texttt{W}_i{}'$ and $\Delta \vdash \texttt{W}_i{}' \texttt{ <: } \texttt{T}$. Therefore, by the rule T-Field, $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}_0\texttt{.f}_i \in \texttt{W}_i{}'$.

Case T-Invk. $\quad \texttt{e} = \texttt{e}_0\texttt{.<}\overline{\texttt{V}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \qquad \Delta;\Gamma,\overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e}_0 \in \texttt{T}_0$
$\Delta \vdash \mathit{bound}_\Delta(\texttt{T}_0) \Uparrow^{\Delta'} \texttt{C<}\overline{\texttt{T}}\texttt{>} \qquad \mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}_0$
$\{\overline{\texttt{Y}}\} \cap \mathit{dom}(\Delta') = \emptyset \qquad \Delta \vdash \overline{\texttt{V}}$ ok
$\Delta,\Delta' \vdash \overline{\texttt{V}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}} \qquad \Delta\Gamma,\overline{\texttt{x}} : \overline{\texttt{T}} \vdash \overline{\texttt{e}} \in \overline{\texttt{S}}$
$\Delta,\Delta' \vdash \overline{\texttt{S}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}} \qquad [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}_0 \Downarrow_{\Delta'} \texttt{T}$

By the induction hypothesis, $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}_0 \in \texttt{T}_0{}'$ for some $\texttt{T}_0{}'$ such that $\Delta \vdash \texttt{T}_0{}' \texttt{ <: } \texttt{T}_0$ and $\Delta;\Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\overline{\texttt{e}} \in \overline{\texttt{S}}'$ for some $\overline{\texttt{S}}'$ such that $\Delta \vdash \overline{\texttt{S}}' \texttt{ <: } \overline{\texttt{S}}$. By Lemma 5.4.5, we have

$$\begin{array}{ll} \Delta \vdash \mathit{bound}_\Delta(\texttt{T}_0{}') \Uparrow^{\Delta''} \texttt{D<}\overline{\texttt{T}}'\texttt{>} & \mathit{mtype}(\texttt{m}, \texttt{D<}\overline{\texttt{T}}'\texttt{>}) = \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}'\texttt{>}\overline{\texttt{U}}'\rightarrow\texttt{U}_0{}' \\ \Delta,\Delta'' \vdash \overline{\texttt{V}} \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}}' & \Delta,\Delta'' \vdash \overline{\texttt{S}}' \texttt{ <: } [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}' \\ [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{S}_0{}' \Downarrow_{\Delta''} \texttt{T}' & \Delta \vdash \texttt{T}' \texttt{ <: } \texttt{T} \end{array}$$

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

Case T-New, T-Cast, T-SCast. Easy. □

ACKNOWLEDGMENTS

We thank anonymous reviewers 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.

REFERENCES

Abadi, M., Cardelli, L., and Viswanathan, R. 1996. An interpretation of objects and object types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM Press, St. Petersburg Beach, FL, 396–409.

Agesen, O., Freund, S. N., and Mitchell, J. C. 1997. Adding type parameterization to the Java language. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*. ACM Press, Atlanta, GA, 49–65.

America, P. and van der Linden, F. 1990. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications/European Conference on Object-Oriented Programming(OOPSLA/ECOOP'90)*. ACM Press, Ottawa, Canada, 161–168.

Barthe, G. and Frade, M. J. 1999. Constructor subtyping. In *Proceedings of the 8th European Symposium on Programming (ESOP'99)*. Lecture Notes on Computer Science, vol. 1576. Springer-Verlag, Amsterdam, Netherlands, 109–127.

Barthe, G. and van Raamsdonk, F. 2000. Constructor subtyping in the calculus of inductive constructions. In *Proceedings of the 3rd International Conference on Foundations of Software Science and Computation Structures (FoSSaCS2000)*. Lecture Notes on Computer Science, vol. 1784. Springer-Verlag, Berlin, Germany, 17–34.

Bracha, G. 1996. The Strongtalk type system for Smalltalk. In *Proceedings of the OOPSLA'96 Workshop on Extending the Smalltalk Language*. San Jose, CA. Also available electronically through `http://bracha.org/nwst.html`.

Bracha, G. and Griswold, D. 1993. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*. ACM Press, Washington, DC, 215–230.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*. ACM Press, Vancouver, BC, 183–200.

Bruce, K. B. 1994. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Funct. Program. 4,* 2 (Apr.), 127–206. Preliminary version in POPL 1993, under the title "Safe type checking in a statically typed object-oriented programming language".

Bruce, K. B., Cardelli, L., and Pierce, B. C. 1999. Comparing object encodings. *Inf. Comput. 155*, 108–133. A special issue with papers from *Theoretical Aspects of Computer Software (TACS)*, September, 1997.

Bruce, K. B., Odersky, M., and Wadler, P. 1998. A statically safe alternative to virtual types. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*. Lecture Notes on Computer Science, vol. 1445. Springer-Verlag, Brussels, Belgium, 523–549.

Bruce, K. B., Petersen, L., and Fiech, A. 1997. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes on Computer Science, vol. 1241. Springer-Verlag, Jyväskylä, Finland, 104–127.

Bruce, K. B., Schuett, A., and van Gent, R. 1995. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the 9th European Conference on Object-Oriented*

Programming (ECOOP'95), W. Olthoff, Ed. Lecture Notes on Computer Science, vol. 952. Springer-Verlag, Aarhus, Denmark, 27–51.

Bruce, K. B. and Vanderwaart, J. C. 1999. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Proceedings of the 15th Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*. Electronic Notes in Theoretical Computer Science, vol. 20. Elsevier, New Orleans, LA. Available through `http://www.elsevier.nl/locate/entcs/volume20.html`.

Canning, P., Cook, W., Hill, W., Olthoff, W., and Mitchell, J. 1989. F-bounded quantification for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FPCA'89)*. ACM Press, London, England, 273–280.

Cardelli, L. 1990. Notes about $F^{\omega}_{<:}$. Unpublished manuscript.

Cardelli, L., Martini, S., Mitchell, J. C., and Scedrov, A. 1994. An extension of system F with subtyping. *Inf. Comput. 109,* 1–2, 4–56. Preliminary version in *Proceedings of Theoretical Aspects of Computer Software (TACS'91)*, LNCS 526, Sendai, Japan, 750–770.

Cardelli, L. and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17,* 4 (Dec.), 471–522.

Cartwright, R. and Steele Jr., G. L. 1998. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*. ACM Press, Vancouver, BC, 201–215.

Compagnoni, A. B. and Pierce, B. C. 1996. Higher-order intersection types and multiple inheritance. *Math. Struct. Comput. Sci. 6*, 469–501.

Cook, W. 1989. A proposal for making Eiffel type-safe. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*. Cambridge University Press, Nottingham, England, 57–70.

Day, M., Gruber, R., Liskov, B., and Myers, A. C. 1995. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*. ACM Press, Austin, TX, 156–168.

Duggan, D. and Compagnoni, A. 1999. Subtyping for object type constructors. In *Informal Proceedings of the 6th International Workshop on Foundations of Object-Oriented Languages (FOOL6)*. San Antonio, TX. Available through `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL6.html`.

Ghelli, G. and Pierce, B. 1998. Bounded existentials and minimal typing. *Theor. Comput. Sci. 193*, 75–96.

Harper, R. and Lillibridge, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*. ACM Press, Portland, OR, 123–137.

Igarashi, A. and Pierce, B. C. 2002. Foundations for virtual types. *Inf. Comput. 175,* 1 (May), 34–49. An earlier version appeared in *Proc. of the 13th ECOOP*, Springer LNCS 1628, pages 161–185, 1999.

Igarashi, A., Pierce, B. C., and Wadler, P. 2001a. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23,* 3 (May), 396–450. A preliminary summary appeared in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146, Denver, CO, October 1999.

Igarashi, A., Pierce, B. C., and Wadler, P. 2001b. A recipe for raw types. In *Informal Proceedings of the 8th International Workshop on Foundations of Object-Oriented Languages (FOOL8)*. London, England. Available through `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL8.html`.

Igarashi, A. and Viroli, M. 2002. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP2002)*, B. Magnusson, Ed. Lecture Notes on Computer Science, vol. 2374. Springer-Verlag, Málaga, Spain, 441–469.

Interactive Software Engineering. 2001. An Eiffel tutorial. Available through http://www.eiffel.com/doc/online/eiffel50/intro/language/tutorial-00.ht%ml.

Leroy, X. 1994. Manifest types, modules and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*. ACM Press, Portland, OR, 109–122.

Liskov, B. 1988. Data abstraction and hierarchy. *ACM SIGPLAN Notices 23,* 5 (May), 17–34.

Madsen, O. L. and Møller-Pedersen, B. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*. ACM Press, New Orleans, LA, 397–406.

Meyer, B. 1986. Genericity versus inheritance. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'86)*. ACM Press, Portland, OR, 391–405.

Meyer, B. 1992. *Eiffel: The Language*. Prentice Hall, Upper Saddle River, NJ.

Mitchell, J. C. and Plotkin, G. D. 1988. Abstract types have existential types. *ACM Trans. Program. Lang. Syst. 10,* 3, 470–502. Preliminary version appeared in *Proc. of the 12th ACM POPL,* 1985.

Myers, A. C., Bank, J. A., and Liskov, B. 1997. Parameterized types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, Paris, France, 132–145.

Nordström, B., Petersson, K., and Smith, J. M. 1990. *Programming in Martin-Löf's Type Theory*. Oxford University Press, Oxford, UK. Out of print. An electronic version is available at http://www.cs.chalmers.se/Cs/Research/Logic/book.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. 2004. An overview of the Scala programming language. Tech. Rep. IC/2004/64, École Polytechnique Fédérale de Lausanne, Switzerland.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, Paris, France, 146–159.

Palacz, K. and Vitek, J. 2003. Subtype tests in real time. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP2003)*, L. Cardelli, Ed. Lecture Notes on Computer Science, vol. 2743. Springer-Verlag, Darmstadt, Germany, 378–404.

Pierce, B. C. 1994. Bounded quantification is undecidable. *Inf. Comput. 112,* 1 (July), 131–165. 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.

Pierce, B. C. and Turner, D. N. 1994. Simple type-theoretic foundations for object-oriented programming. *J. Funct. Program. 4,* 2 (Apr.), 207–247.

Raynaund, O. and Thierry, E. 2001. 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)*. Lecture Notes on Computer Science, vol. 2072. Springer-Verlag, Budapest, Hungary, 165–180.

Steffen, M. 1998. Polarized higher-order subtyping. Ph.D. thesis, Universität Erlangen-Nürnberg.

Sun Microsystems. 1998. Adding generic types to the Java programming language. Java Specification Request JSR-000014, http://jcp.org/jsr/detail/014.jsp.

Syme, D. and Kennedy, A. 2001. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM Press, Snowbird, UT. Related information is available through http://research.microsoft.com/projects/clrgen/.

Thorup, K. K. 1997. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes on Computer Science, vol. 1241. Springer-Verlag, Jyväskylä, Finland, 444–471.

THORUP, K. K. AND TORGERSEN, M. 1999. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*. Lecture Notes on Computer Science, vol. 1628. Springer-Verlag, Lisbon, Portugal, 186–204.

TORGERSEN, M. 1998. Virtual types are statically safe. In *Informal Proceedings of the 5th International Workshop on Foundations of Object-Oriented Languages(FOOL5)*. San Diego, CA. Available through `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL5.html`.

TORGERSEN, M., HANSEN, C. P., ERNST, E., VON DER AHÉ, P., BRACHA, G., AND GAFTER, N. 2004. Adding wildcards to the Java programming language. In *Proceedings of the ACM Symposium on Applied Computing (SAC'04)*. ACM Press, Nicosia, Cyprus, 1289–1296.

VIROLI, M. 2003. A type-passing approach for the implementation of parametric methods in Java. *Comput. J. 46,* 3, 263–294.

VIROLI, M. AND NATALI, A. 2000. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*. ACM Press, Minneapolis, MN, 146–165.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115,* 1 (Nov.), 38–94.

Received August 2003; revised May 2004; revised November 2004; accepted November 2004