On Decidability of Nominal Subtyping with Variance

Andrew J. Kennedy
Microsoft Research Cambridge

Benjamin C. Pierce
University of Pennsylvania

Abstract

We investigate the algorithmics of subtyping in the presence of nominal inheritance and variance for generic types, as found in Java 5, Scala 2.0, and the .NET 2.0 Intermediate Language. We prove that the general problem is undecidable and characterize three different decidable fragments. From the latter, we conjecture that undecidability critically depends on the combination of three features that are not found together in any of these languages: contravariant type constructors, class hierarchies in which the set of types reachable from a given type by inheritance and decomposition is not always finite, and class hierarchies in which a type may have multiple supertypes with the same head constructor.

These results settle one case of practical interest: subtyping between ground types in the .NET intermediate language is decidable; we conjecture that our proof can also be extended to show full decidability of subtyping in .NET. For Java and Scala, the decidability questions remain open; however, the proofs of our preliminary results introduce a number of novel techniques that we hope may be useful in further attacks on these questions.

1. Introduction

The core of the subtype relation in most object-oriented programming languages is *nominal*, in the sense that basic inclusions between type constructors are explicitly declared by the programmer. However, most languages also support a modicum of *structural* subtyping; for example, array types in Java and C$^\sharp$ behave covariantly. More recent designs feature richer structural features—in particular, covariant and contravariant subtyping of constructor parameters, such as the *variance* annotations of Scala's generic types and the *wildcard* types supported by Java 5.

Formally, subtyping is typically presented in a declarative style. For many systems, an equivalent syntax-directed presentation is easily derived, and termination of the corresponding subtype checker is easy to demonstrate. For example, in the case of C$^\sharp$ 2.0 and the original "GJ" design for generics in Java [3, 13], it is straightforward to derive an algorithm by building transitivity into the subtyping rules for superclasses and upper bounds and to then prove termination by constructing a measure on subtype judgments that strictly decreases from conclusion to premises in the algorithmic rules.

For more sophisticated systems, such as the original use-site variance proposal for Java [14], the wildcard design of Java 5 [12, 22], and the declaration-site variance of Scala [18], algorithmic subtyping rules have never been presented, and decidability is still an open problem.[1] This is the starting point for our investigation.

Language features Each of these languages supports *generic inheritance*, in which a named class is declared with type parameters and (multiple) supertypes referencing the type parameters. Here are equivalent definitions in .NET 2.0 IL, C$^\sharp$ 2.0, Java 5 and Scala 2.0:

```
.class C<X,Y> extends class D<class E<!X>>    // .NET IL
  implements class I<!Y>
class C<X,Y> : D<E<X>>, I<Y>                  // C#
class C<X,Y> extends D<E<X>> implements I<Y>  // Java
class C[X,Y] extends D[E[X]] with I[Y]        // Scala
```

A second feature shared by these languages is *covariant* and *contravariant* subtyping of generic types. Scala 2.0 and .NET 2.0 IL support *declaration-site variance*, where the variance behaviour of type parameters is declared up-front on the declaration of the class; this feature is also a strong candidate for a future version of C$^\sharp$. For example, here are equivalent headers for a contra/co-variant function type in .NET IL, an imaginary future version of C$^\sharp$, and Scala:

```
.class interface Func<-A,+B>                  // .NET IL
interface Func<-A,+B>                         // C# ?.0
trait Func[-A,+B]                             // Scala
```

In contrast, Java 5 supports *use-site variance*, where annotations on the *use* of a generic type determine its variance behaviour, as in the `cast` function below:

```
interface Func<A,B> { B apply(A a); }
class C { }
class D extends C {
  static Func<? super D, ? extends C>
    cast(Func<? super C, ? extends D> f) { return f; }
}
```

Here, the `? extends` annotation induces covariant subtyping on the type argument, and `? super` annotation induces contravariant subtyping. (In an earlier variance design for Java [14] the annotations were the more concise `+` and `-`.)

For all of these languages, decidability of subtyping is an open problem. The Java 5 design [12, 22] is based on an earlier extension to Java generics whose decidability is not known [14, §4.1]. A core calculus for Scala has been studied recently and type-checking and subtyping proved decidable [8], but variance is not supported. Finally, an extension to C$^\sharp$ 2.0 modeled on variance in .NET IL 2.0 has been studied by the first author and others [10], but without considering generic inheritance in its full generality (in particular, support for multiple instantiation inheritance).

Contributions We present here a collection of results regarding the algorithmics of nominal subtyping in the presence of (declaration-site) variance. First, we show that a general form of

[1] Readers may enjoy attempting to compile the examples in Appendix A using their favorite Java compiler!

the problem, where the subtype hierarchy may involve multiple inheritance from arbitrary supertypes (in particular, from multiple instances of the same type constructor) is undecidable. Second, we show various restricted fragments of the system are decidable: (1) a system with only covariant and invariant constructors (no contravariance); (2) a system (like the .NET CLR) where unbounded expansion of types is disallowed—i.e, where the class hierarchy is restricted so that the set of types reachable from a given type by inheritance and decomposition is always finite; and (3) finally, a system in which (as in Scala and Java) multiple instantiation inheritance is prohibited—class hierarchies are restricted so that a type may not have multiple supertypes with the same head constructor—and where some technical restrictions (detailed below) are imposed on unbounded expansion.

These results settle the decidability issue for one case of practical interest: subtyping between ground types in the .NET intermediate language (which is used for runtime type tests); we conjecture that the argument can be extended without major new insights to the case of subtyping between open types in .NET. For Java and Scala, the decidability questions remain open—we discuss the remaining gaps in Section 6. However, even for these languages, our results do begin to give a feel for the lay of the land, and their proofs introduce a number of novel techniques that we hope may be useful in further attacks on the decision problems.

2. Definitions

Syntax Types, ranged over by T, U, V and W, are of two forms: type variables, ranged over by X, Y, and Z and constructed types $C\texttt{<}\overline{T}\texttt{>}$, where C is a type constructor (such as `List`) and $\overline{T}$ is a list of argument types. We define the *height* of a type by

$$\begin{array}{rcl} \mathit{height}(X) & = & 1 \\ \mathit{height}(C\texttt{<}\overline{T}\texttt{>}) & = & 1 + \max \mathit{height}(\overline{T}) \end{array}$$

As usual, nullary constructor applications are written without brackets (C means $C\texttt{<>}$). It is also convenient to write unary applications without the angle brackets (CT means $C\texttt{<}T\texttt{>}$), and to declare that application associates to the *right* (not to the left, as usual), so that we can write long nested applications without brackets. For example, $CDEX$ is shorthand for $C\texttt{<}D\texttt{<}E\texttt{<}X\texttt{>>>}$.

Class tables We are working in a nominal subtyping world, where the basic inclusions between constructors are explicitly declared in a global *class table*. A class table is a set of class declarations, each of the form

$$\begin{array}{lcl} C\texttt{<}\overline{X}\texttt{>} & <:: & T_1 \\ & & \vdots \\ & & T_n \end{array}$$

where the class name C is unique to the declaration, and the T_is may mention the parameter variables $\overline{X}$. This says that, given a tuple of arguments $\overline{U}$, the application $C\texttt{<}\overline{U}\texttt{>}$ is a subtype of $[\overline{U}/\overline{X}]T_i$ for each i. We write $C\texttt{<}\overline{U}\texttt{>} <:: [\overline{U}/\overline{X}]T_i$ when this holds and define $<::^+$ and $<::^*$ to be the transitive and reflexive transitive closures of this relation. (In Java and C$^\sharp$, exactly one of the supertypes of $C\texttt{<}\overline{U}\texttt{>}$ will be an actual class; the rest will be interfaces. Since we are only interested in subtyping in this paper, we elide the difference.)

Note that the class table is allowed to be both singly and mutually recursive: the supertypes of C can mention C, can mention a class D whose supertypes involve C, etc. Moreoever, in the absence of further restrictions, we have already gone beyond generic inheritance in the style of Java, Scala, and C$^\sharp$, as we have not imposed a single root (such as `java.lang.Object`), and we permit both mixin-style inheritance [1] of the form $C\texttt{<}\overline{X}\texttt{>} <:: X_i$ and multiple-instantiation inheritance (*e.g.*, $C <:: DE_1, DE_2$).

We require that inheritance be *acyclic*, in the following sense: if $C\texttt{<}\overline{T}\texttt{>} <::^+ D\texttt{<}\overline{U}\texttt{>}$ then $C \neq D$. In the presence of mixins it is not immediately apparent how to check acyclicity for a given class table: consider, for example, the definitions $CX <:: X$ and $D <:: CD$. Fortunately, a straightforward procedure does exist [1], and for the remainder of the paper we simply assume that the acyclicity property holds.

Individual parameters of type constructors may be marked as co- or contra-variant. The general form of a class declaration is:

$$\begin{array}{lcl} C\texttt{<}\overline{vX}\texttt{>} & <:: & T_1 \\ & & \vdots \\ & & T_n \end{array}$$

where each v_i is $\circ$ (invariant, also written as an empty string), `+` (covariant), or `-` (contravariant). We write $C\#i$ to stand for the i'th type parameter in the definition of the class C, and $\mathit{var}(C\#i)$ for the variance of that type parameter.

Subtyping Figure 1 presents the ground subtype relation $<:$ together with an auxiliary variance-indexed relation $<:_v$ (the last three rules). Rule SUPER is the familiar generic inheritance rule from Generic Java and C$^\sharp$ 2.0. Note the side-condition, which ensures that the subtyping rules are *syntax-directed*; moreover, given the absence of cycles in the inheritance hierarchy, two instantiations of the same class cannot be related by inheritance. Subtyping can be extended to open types through the addition of an axiom $X <: X$ asserting reflexivity for type variables.

Although the rules are syntax-directed, rule SUPER presents a choice for V in its premise if C inherits from multiple superclasses. So a subtype checker may need to employ backtracking.

In rule VAR, the variance annotation of a parameter $C\#i$ determines the behaviour of the i'th argument in an instantiation of C with respect to subtyping. Suppose T_i is a subtype of U_i. If $\mathit{var}(C\#i) = \texttt{+}$ then $C\texttt{<}T_1, \ldots, T_i, \ldots, T_n\texttt{>}$ is a subtype of $C\texttt{<}T_1, \ldots, U_i, \ldots, T_n\texttt{>}$; dually, if $\mathit{var}(C\#i) = \texttt{-}$ then $C\texttt{<}T_1, \ldots, U_i, \ldots, T_n\texttt{>}$ is a subtype of $C\texttt{<}T_1, \ldots, T_i, \ldots, T_n\texttt{>}$.

This *declaration-site* variance is supported by Scala 2.0 [18], and has also been studied as an extension to C$^\sharp$ by the first author and others [10]. Java 5, in contrast, supports *use-site* variance, where the variance behaviour of a type is determined by annotations on its actual instantiation. Suppose that T is a subtype of U. Then $C\texttt{<? extends }T\texttt{>}$ is a subtype of $C\texttt{<? extends }U\texttt{>}$; dually, $C\texttt{<? super }U\texttt{>}$ is a subtype of $C\texttt{<? super }T\texttt{>}$.

In fact, wildcard types in Java are best thought of as *bounded existentials*, so $C\texttt{<? extends }T\texttt{>}$ is essentially $\exists X{<:}T.C\texttt{<}X\texttt{>}$ and $C\texttt{<? super }T\texttt{>}$ is $\exists X{:>}T.C\texttt{<}X\texttt{>}$. This existential interpretation leads to somewhat more complex (and more powerful) subtyping rules, particularly in combination with inheritance and F-bounded type parameters. Nevertheless, it is quite straightforward to define a translation from declaration-site to use-site variance that preserves and reflects subtyping. Hence rather than dive in at the deep end with use-site/wildcard variance, we choose to start our investigation with the declaration-site variety.

Also note that we do not consider *bounds* (also called *constraints*) on type parameters, a feature supported by Java 5, C$^\sharp$ 2.0, .NET IL 2.0, and Scala 2.0. Formally, these require subtyping judgments of the form $\Delta \vdash T <: U$, where Δ is a set of assumptions: upper bounds on type parameters, for Java, C$^\sharp$, and .NET, and both lower and upper bounds, for Scala. We believe that for declaration-site variance, our results can be generalized to open subtyping in the presence of bounds; we return to this point in Section 6.

Properties of subtyping It is easy to prove that ground subtyping is reflexive, by a derivation consisting only of instances of VAR. To prove transitivity, we need to make use of a well-formedness

$$(\textsc{Var})\ \frac{\text{for each } i \quad T_i <:_{var(C\#i)} U_i}{C<\overline{T}> <: C<\overline{U}>}$$

$$(\textsc{Super})\ \frac{C<\overline{X}> <:: V \qquad [\overline{T}/\overline{X}]\,V <: D<\overline{U}>}{C<\overline{T}> <: D<\overline{U}>}\ C \neq D$$

$$\frac{T <: U}{T <:_{+} U} \qquad \frac{}{T <:_{\circ} T} \qquad \frac{U <: T}{T <:_{-} U}$$

Figure 1. The subtyping relation

$$\neg v = \begin{cases} -, \text{ if } v = +, \\ \circ, \text{ if } v = \circ, \\ +, \text{ if } v = - \end{cases}$$

$$\frac{v_i \in \{\circ, +\}}{\overline{vX} \vdash X_i\ ok}$$

$$\frac{\text{for each } i \quad \begin{array}{l} var(C\#i) \in \{\circ, +\} \Rightarrow \overline{vX} \vdash T_i\ ok \\ var(C\#i) \in \{\circ, -\} \Rightarrow \neg\overline{vX} \vdash T_i\ ok \end{array}}{\overline{vX} \vdash C<\overline{T}>\ ok}$$

$$\frac{\overline{vX} \vdash T\ ok}{C<\overline{vX}> <:: T\ ok}$$

Figure 2. Well-formed types and class declarations

condition on supertype declarations that restricts covariant type parameters to appear only in positive positions in the supertype and, dually, restricts contravariant parameters to appear only in negative positions. (This condition is necessary for semantic soundness, but observe that if it were not enforced, one could make the definition $C<+X> <:: NX$ for contravariant N, and have $CT <: CU$ and $CU <: NU$ but not $CT <: NU$.) Formally, we introduce a well-formedness judgment $\overline{vX} \vdash T\ ok$ which can be read "type T respects the variance annotations $\overline{v}$ on type parameters $\overline{X}$." This is shown in Figure 2 together with its application in specifying well-formedness of class declarations. (For the type system proper, well-formedness is also used to restrict field and method signatures appropriately [10].) For example, if class C is invariant in its parameter and N is contravariant, then the declaration $D<-X, +Y> <:: NNY$ is allowed, because $-X, +Y \vdash NNY\ ok$ is derivable, but $D<-X, +Y> <:: NY$ and $D<-X, +Y> <:: CX$ are rejected.

Using these assumptions, we can prove that subtyping is transitive.

LEMMA 1 (Transitivity). *If* $T <: U$ *and* $U <: V$ *then* $T <: V$.

PROOF: See Appendix B. □

It follows that the syntax-directed formulation of subtyping we are working with here is equivalent to a declarative formulation where reflexivity and transitivity are included as separate rules and rule SUPER is replaced by a simpler version that concludes $T <: U$ from premise $T <:: U$. This observation gives some assurance that the decidability issues we are addressing do not arise just from some peculiarity in the way we have formulated our algorithmic rules, but rather are inherent to the notion of subtyping under consideration.

3. Examples

Is the subtype relation we have defined decidable, for any class table? It is not clear, on the face of it, what we should expect. On one hand, this type system has something of the flavor of System $F^{\omega}_{\leq}$ [5, 7, 20], the higher-order extension of System $F_{\leq}$ [6] (the type constructors in the present system are intuitively something like bounded type variables of higher kind in $F^{\omega}_{\leq}$), so we might worry about the possibility of undecidability [19]. Indeed, the possibility of recursion in the class table introduces elements of *F-bounded* quantification [4, 11, 2]. On the other hand, some features of $F_{\leq}$ that play a prominent role in its proof of undecidability (in particular, "free-standing" universal quantifiers, which allow introducing new type variables into the environment) are missing here. And the other complicating feature, the variance annotations, has been shown to be well-behaved (though complex to analyze) in at least some situations [21].

(Appendix A presents Java 5 code for the examples below).

Example 1 The first observation that hints at the full system's undecidability is that the subtype checker defined by the algorithmic rules above does not always halt. (This is not a proof of undecidability, of course—it is always possible that there is another, smarter algorithm that does always terminate.)

Define the classes N and C as follows:

$$\begin{array}{lll} N<-Z> & <:: & \\ C & <:: & NNC \end{array}$$

That is, N is a contravariant constructor with no declared supertypes and C is an invariant nullary constructor whose immediate supertype is NNC.

Now, suppose we want to know whether $C <: NC$. If we try to construct a proof of this fact from the algorithmic subtyping rules (following the steps of the subtyping algorithm), we enter an infinite regress:

$$\begin{array}{lll} & C <: NC & \\ \longrightarrow & NNC <: NC & \text{by }\textsc{Super} \\ \longrightarrow & C <: NC & \text{by }\textsc{Var} \\ \longrightarrow & NNC <: NC & \text{by }\textsc{Super} \\ \longrightarrow & C <: NC & \text{by }\textsc{Var} \\ \longrightarrow & \text{etc.} & \end{array}$$

Of course, this infinite regress is easy to detect. And clearly, if $T <: U$ can only be proved by a derivation containing as a subderivation a proof of $T <: U$, then T is *not* a subtype of U. (Under a *co-inductive* interpretation of the subtyping rules, T would be considered a subtype of U. We believe that this more generous definition is semantically sound; but it does not affect decidability, as an algorithm must in any case detect infinite regress, returning *true* in place of *false*.)

Example 2 Unfortunately, slightly trickier examples lead to more complicated patterns of regress. For example, if we define N as above and consider

$$CX <:: NNCCX$$

then the regress involves longer and longer types as it goes on:

$$\begin{array}{lll} & CT <: NCU & \\ \longrightarrow & NNCCT <: NCU & \text{by }\textsc{Super} \\ \longrightarrow & CU <: NCCT & \text{by }\textsc{Var} \\ \longrightarrow & NNCCU <: NCCT & \text{by }\textsc{Super} \\ \longrightarrow & CCT <: NCCU & \text{by }\textsc{Var} \\ \longrightarrow & NNCCCT <: NCCU & \text{by }\textsc{Super} \\ \longrightarrow & CCU <: NCCCT & \text{by }\textsc{Var} \\ \longrightarrow & NNCCCU <: NCCCT & \text{by }\textsc{Super} \\ \longrightarrow & CCCT <: NCCCU & \text{by }\textsc{Var} \\ \longrightarrow & \text{etc.} & \end{array}$$

In general, there is no way to describe and detect all such patterns of regress.

Example 3 Another observation that illustrates the bad behavior of the subtyping relation is that *successful* derivations can be exponentially larger than the class table. For example, consider the following definitions:

$$\begin{array}{l} C_0X \mathrel{<::} NNX \\ C_1X \mathrel{<::} C_0C_0X \\ \quad\vdots \\ C_nX \mathrel{<::} C_{n-1}C_{n-1}X \end{array}$$

The derivation of the valid subtyping assertion $C_nNT <: NC_nT$ uses 2^{n+1} instances of the VAR rule!

Example 4 Analogously, there are patterns of regress where the "cycle" is exponentially larger than the class table. One such is obtained by making a small change to the declaration of C_n above:

$$\begin{array}{l} C_0X \mathrel{<::} NNX \\ C_1X \mathrel{<::} C_0C_0X \\ \quad\vdots \\ C_nX \mathrel{<::} C_{n-1}C_{n-1}C_nX \end{array}$$

The reduction of $C_nNT <: NC_nT$ enters a cycle (i.e., generates a subgoal that has already been seen) after passing through 2^{n+1} instances of VAR.

4. Undecidability of the general case

To show undecidability of subtyping, we perform a reduction from the *Post Correspondence Problem*, or PCP for short.

The Post Correspondence Problem Let $\{(u_1, v_1), \ldots, (u_n, v_n)\}$ be a set of pairs of non-empty words over a finite alphabet Σ. The Post Correspondence Problem is to determine whether or not there exists a sequence of indices $i_1, \ldots, i_r$ such that $u_{i_1} \cdots u_{i_r} = v_{i_1} \cdots v_{i_r}$.

THEOREM 2. *PCP is undecidable.*

PROOF: See, for example, [15]. □

Reduction The construction uses the following classes:

E	empty sequence
SX	stop
N<-X>	negation
N_i<-X>	negation, for $0 \leqslant i \leqslant n$
LX	letter, for each $L \in \Sigma$
C<X, Y>	crank
B	boot

The first five of these have no declared supertypes; the supertype declarations for the last two are given below. The general idea is that words over Σ are represented as types, while instances of PCP are represented as class tables in which each pair of words from the PCP instance corresponds to a different supertype declaration for the constructor C.

We represent words as sequences of applications of classes, using the nullary class E to represent the empty sequence. So, for example, if $\Sigma = \{P, Q, R\}$ then our class table will contain declarations for classes P, Q, and R, each taking one parameter and with no supertypes, and the word PQP can be represented by the type $PQPE$ (i.e., P<Q<P<E<>>>>).

Let $u{\cdot}T$ denote the representation of a non-empty word u concatenated onto a word represented by type T:

$$\begin{array}{rcl} L{\cdot}T & = & LT \\ (Lu){\cdot}T & = & L\text{<}u{\cdot}T\text{>} \end{array}$$

The representation of a non-empty word u, denoted $\underline{u}$, is defined as

$$\underline{u} = u{\cdot}E.$$

It is easy to see that $u = v$ iff $\underline{u} = \underline{v}$.

The class table entries for C and B are:

$$\begin{array}{lll} C\text{<}X, Y\text{>} \quad \mathrel{<::} & NN_1C\text{<}u_1{\cdot}X, v_1{\cdot}Y\text{>} & (C_1) \\ & N_1NC\text{<}u_1{\cdot}X, v_1{\cdot}Y\text{>} & (C_1') \\ & \quad\vdots & \\ & NN_nC\text{<}u_n{\cdot}X, v_n{\cdot}Y\text{>} & (C_n) \\ & N_nNC\text{<}u_n{\cdot}X, v_n{\cdot}Y\text{>} & (C_n') \\ & NN_0SX & (C_L) \\ & N_0SY & (C_R) \\ & & \\ B \quad \mathrel{<::} & NN_1C\text{<}\underline{u_1}, \underline{v_1}\text{>} & (B_1) \\ & N_1NC\text{<}\underline{u_1}, \underline{v_1}\text{>} & (B_1') \\ & \quad\vdots & \\ & NN_nC\text{<}\underline{u_n}, \underline{v_n}\text{>} & (B_n) \\ & N_nNC\text{<}\underline{u_n}, \underline{v_n}\text{>} & (B_n') \end{array}$$

Now, the given word problem has a solution iff the subtyping judgment $B <: NB$ is derivable. To see why, consider the progress of the proof search procedure starting from $B <: NB$. Its first step must be to use some instance of SUPER on the left-hand side. If it chooses one of the even-numbered supertypes (the ones beginning with N_i), then it will fail on the next step: since none of the N_is have declared supertypes, there is no rule that can derive $N_iS <: NT$. Among the odd-numbered rules, however, it has a free choice—i.e., algorithmically, it must try each of them in turn, backtracking and trying the others if the proof search for the corresponding subgoal fails. So suppose it tries the first one, rule (B_1) in the class table. The active goal then becomes

$$NN_1C\text{<}\underline{u_1}, \underline{v_1}\text{>} <: NB.$$

The next step of proof search necessarily applies rule VAR, yielding

$$B <: N_1C\text{<}\underline{u_1}, \underline{v_1}\text{>}.$$

At this point, there is again just one choice that allows us to make progress: the only way to avoid failing on the very next step is to choose rule (B_1')

$$N_1NC\text{<}\underline{u_1}, \underline{v_1}\text{>} <: N_1C\text{<}\underline{u_1}, \underline{v_1}\text{>},$$

and another use of VAR yields

$$C\text{<}\underline{u_1}, \underline{v_1}\text{>} <: NC\text{<}\underline{u_1}, \underline{v_1}\text{>}.$$

At this point, the proof search algorithm again has a choice to make. It can either apply SUPER with one of the odd-numbered supertypes of C, or apply the special "stop rule" (C_L) (again, the even numbered rules and rule (C_R) all lead to failure on the next step). If it chooses one of the (C_i') rules—say, (C_4)—then an analogous sequence of forced steps leads to the subgoal

$$C\text{<}\underline{u_4u_1}, \underline{v_4v_1}\text{>} <: NC\text{<}\underline{u_4u_1}, \underline{v_4v_1}\text{>}.$$

This process might continue in the same way with yet another of the (C_i) rules, but suppose, instead, that the proof search now chooses the stop rule (C_L):

$$NN_0S\text{<}\underline{u_4u_1}\text{>} <: NC\text{<}\underline{u_4u_1}, \underline{v_4v_1}\text{>},$$

which leads to

$$C\text{<}\underline{u_4u_1}, \underline{v_4v_1}\text{>} <: N_0S\text{<}\underline{u_4u_1}\text{>},$$

from which the only choice that does not fail at the next step is (C_R),

$$N_0S\text{<}\underline{v_4v_1}\text{>} <: N_0S\text{<}\underline{u_4u_1}\text{>},$$

from which VAR yields

$$S\langle\underline{u_4 u_1}\rangle \quad <: \quad S\langle\underline{v_4 v_1}\rangle.$$

Now the proof search immediately succeeds or fails. The constructor S has no declared supertypes and no variance annotation, so the only way $S\langle\underline{u_4 u_1}\rangle <: S\langle\underline{v_4 v_1}\rangle$ can hold is by reflexivity. This rule applies if the two types being compared are exactly the same—i.e., if $\underline{u_4 u_1} = \underline{v_4 v_1}$. Summing up, we can see that there exists some way of constructing a successful subtyping derivation for the judgment $B <: NB$ iff there is some sequence I such that $\underline{u_I} = \underline{v_I}$—that is (since the translation from words to types is injective), such that $u_I = v_I$.

Formalities The rest of the section recapitulates this argument more formally.

Let i and j range over positive integers, used to index the words, and let I and J range over (possibly empty) sequences of positive integers. We write IJ to denote the concatenation of sequences I and J, and we identify i with the single-element sequence containing i. For $I = i_1 \cdots i_r$ we write u_I to denote the concatenated word $u_{i_1} \cdots u_{i_r}$. We begin with a couple of lemmas.

LEMMA 3 (Single step). *Suppose*

$$C\langle\underline{u_I}, \underline{v_I}\rangle <: NC\langle\underline{u_I}, \underline{v_I}\rangle$$

is derivable for a non-empty sequence I. Then either $u_I = v_I$, or *there is a proper subderivation whose conclusion is*

$$C\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle <: NC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle$$

for some i.

PROOF: Since $C \neq N$, the derivation must end with an instance of SUPER. There are two cases.

- Declaration (C_L) was used, and so the premise was

  $$NN_0S\underline{u_I} <: NC\langle\underline{u_I}, \underline{v_I}\rangle.$$

  Since both sides begin with N, the final rule in this subderivation must be VAR, with premise

  $$C\langle\underline{u_I}, \underline{v_I}\rangle <: N_0S\underline{u_I}.$$

  Again, the subderivation for this premise must end with an instance of SUPER. None of C_L, C_i, or C'_i lead to a successful derivation (for any i), but C_R does, giving us the premise

  $$N_0S\underline{v_I} <: N_0S\underline{u_I}.$$

  By VAR again, this was derived from

  $$S\underline{u_I} <: S\underline{v_J}.$$

  Finally, by VAR once more, we must have a subderivation of $\underline{u_I} = \underline{v_I}$, and so $u_I = v_I$, as required.
- Declaration (C_i) was used, for some i, so the premise was

  $$NN_iC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle <: NC\langle\underline{u_I}, \underline{v_I}\rangle.$$

  By VAR, this must have been derived from

  $$C\langle\underline{u_I}, \underline{v_I}\rangle <: N_iC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle.$$

  Then by SUPER through declaration C'_i we have premise

  $$N_iNC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle <: N_iC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle$$

  and by VAR we have the premise

  $$C\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle <: NC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle$$

  as required. □

LEMMA 4 (Multiple step). *For any non-empty sequence I, if there is a derivation of*

$$C\langle\underline{u_I}, \underline{v_I}\rangle <: NC\langle\underline{u_I}, \underline{v_I}\rangle$$

then there exists a (not necessarily proper) subderivation whose conclusion is

$$C\langle\underline{u_{JI}}, \underline{v_{JI}}\rangle <: NC\langle\underline{u_{JI}}, \underline{v_{JI}}\rangle$$

such that $u_{JI} = v_{JI}$ for some (possibly empty) J.

PROOF: By induction on the height of the derivation.

From Lemma 3, we have either $u_I = v_I$, in which case we're done (J is the empty sequence), or else we have a proper subderivation whose conclusion is

$$C\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle <: NC\langle\underline{u_{iI}}, \underline{v_{iI}}\rangle$$

for some i. We now apply the induction hypothesis to get a (not necessarily proper) subderivation of

$$C\langle\underline{u_{J'iI}}, \underline{v_{J'iI}}\rangle <: NC\langle\underline{u_{J'iI}}, \underline{v_{J'iI}}\rangle$$

for some J' such that $u_{J'iI} = v_{J'iI}$. This is the result we desire (let $J = J'i$). □

THEOREM 5. *Subtyping is undecidable.*

PROOF: We show that an instance of PCP can be reduced to an instance of subtype validity under some class table.

For the particular instance of PCP, define a class table as described above. We now show that the instance of PCP has a solution if and only if $B <: NB$ is derivable.

($\Rightarrow$). Suppose that I is a solution to the problem, so that $u_I = v_I$. Then $S\langle\underline{u_I}\rangle <: S\langle\underline{v_I}\rangle$ by reflexivity. By two uses of VAR, and SUPER through C_L and C_R we obtain a derivation of $C\langle\underline{u_I}, \underline{v_I}\rangle <: NC\langle\underline{u_I}, \underline{v_I}\rangle$. We now show $B <: NB$ is derivable by induction on the length of I. For the base case, suppose $I = i$. Then we can easily construct a derivation, using VAR twice and SUPER through B_i and B'_i. For the inductive step, suppose that $I = iJ$. By two uses of VAR and SUPER, through C_i and then C'_i, we obtain $C\langle\underline{u_J}, \underline{v_J}\rangle <: NC\langle\underline{u_J}, \underline{v_J}\rangle$. Applying the induction hypothesis gives us the desired result.

($\Leftarrow$). The derivation must end with an instance of SUPER, with premise $NN_iC\langle\underline{u_i}, \underline{v_i}\rangle <: NB$ for some i. This in turn must have been derived using VAR, with premise $B <: NC\langle\underline{u_i}, \underline{v_i}\rangle$. Another use of SUPER and VAR takes us to $C\langle\underline{u_i}, \underline{v_i}\rangle <: N_iC\langle\underline{u_i}, \underline{v_i}\rangle$. Now we can apply Lemma 4, to obtain a derivation of

$$C\langle\underline{u_{Ii}}, \underline{v_{Ii}}\rangle <: NC\langle\underline{u_{Ii}}, \underline{v_{Ii}}\rangle$$

for some I such that $u_{Ii} = v_{Ii}$. In other words, we have some J such that $u_J = v_J$, as required. □

5. Some decidable fragments

The reduction from PCP in the previous section used the following vital ingredients:

- *Contravariance* was used in a "double-negation" fashion to send a term to the opposite side of the subtype assertion and then back again. (Interestingly, it is possible to devise a slightly more complex reduction from PCP that uses only a *single* contravariant constructor.)
- *Unbounded growth* in the size of the subtype assertion was used to accumulate a concatenation of words in the encoding.
- *Multiple instantiation inheritance*—applying a type constructor at different type instantiations in inheritance declarations—was used to encode a choice of words. (Note, though, that the instantiations are *non-overlapping*: instantiation of C does not lead to identification of any of its supertypes.)

In this section, we explore different ways to recover decidability by restricting the class table to eliminate one or more of these ingredients.

5.1 Contravariance

THEOREM 6. *Suppose that no type constructors are contravariant. Then subtyping is decidable.*

PROOF: Define the following order on subtype assertions:

$$\begin{gathered}(T_1 <: U_1) \prec (T_2 <: U_2) \\ \text{iff} \\ \mathit{height}(U_1) < \mathit{height}(U_2) \\ \text{or } (\mathit{height}(U_1) = \mathit{height}(U_2) \text{ and } T_2 <::^+ T_1)\end{gathered}$$

The acyclicity condition on inheritance ensures that the inverse of $<::^+$ is well-founded; so $\prec$ is well-founded also. It is easy to see that this order strictly decreases from conclusion to premises of the subtyping rules: rule VAR reduces the height of both sides of the assertion for covariant parameters, whilst rule SUPER leaves the right-hand-side alone and reduces the left-hand-side with respect to the inheritance ordering. □

5.2 Expansive inheritance

The undecidability reduction made crucial use of the ability to grow the subtype assertion in an unbounded fashion. We now show how to check this growth by restricting the form of types in the class table.

Given a particular class table, a set of types $\mathcal{S}$ is said to be *closed under decomposition and inheritance*—or *inheritance closed*, for short—if (a) whenever $C{<}\overline{T}{>}$ is in $\mathcal{S}$, so are all the $\overline{T}$, and (b) if T is in $\mathcal{S}$ and $T <:: U$, then U is in $\mathcal{S}$. Define the *inheritance closure* of a set $\mathcal{S}$, written $cl(\mathcal{S})$, to be the least inheritance-closed superset of $\mathcal{S}$.

The types appearing in subtype judgments in the subtyping rules are clearly closed with respect to decomposition and inheritance: a derivation of $T <: U$ involves only types in $cl(\{T, U\})$. If the subtype checker remembers a set of "visited goals" and rejects assertions that appear in this set, then it must terminate if the inheritance closure of the types in the original problem is finite. We say that a class table is *finitary* if any finite set of types has a finite inheritance closure with respect to the class table; otherwise, the class table is *infinitary*. The class table in Example 1 is finitary, while that in Example 2 is infinitary: $cl(\{CT\}) = \{N^m C^n T \mid 0 \leqslant m \leqslant 2 \text{ and } n \geqslant 0\}$. Likewise, the "crank" class C in Section 4 induces an infinitary closure.

Even disregarding subtyping, infinite closure presents a problem for language implementers, as they must take care not to create type representations for supertypes in an eager fashion, else non-termination is the result. For example, the .NET Common Language Runtime supports generic instantiation and generic inheritance in its intermediate language targeted by C$^\sharp$. The class loader maintains a hash table of types currently loaded, and when loading a new type it will attempt to load its supertypes, add these to the table, and in turn load the type arguments involved in the supertype.

Fortunately, there is a syntactic characterization of infinitary class tables due to Viroli [23] that can be used to reject such definitions; the specification of the .NET CLR includes such a restriction [9, Partition II, §9.2]. Here we recast Viroli's definitions in our framework and present a (somewhat slicker) proof of correctness.

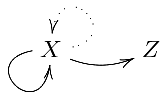
Define a *type parameter dependency graph* as follows. The vertices are all the type parameters to classes in the class table; we will denote these interchangeably by either $C\#i$—the i'th formal type parameter to class C—or by their alphabetic names, which we assume are (α-converted to be) distinct. Boolean-labeled edges represent uses of formal type parameters in class instantiations in the class table. For each declaration $C{<}\overline{X}{>} <:: T$ and each subterm $D{<}\overline{T}{>}$ of T,

- if $T_j = X_i$ add a *non-expansive* edge $C\#i \xrightarrow{0} D\#j$;
- if X_i is a proper subterm of T_j add an *expansive* edge $C\#i \xrightarrow{1} D\#j$.

We write $X \rightarrow Y$ if $X \xrightarrow{e} Y$ for some $e \in \{0, 1\}$ and write $\rightarrow^+$ for the transitive closure of $\rightarrow$.

Infinitary class tables are characterized precisely by those graphs that contain a cycle with at least one expansive edge. Consider Example 2 and its graph, using dotted arrows for non-expansive edges and solid arrows for expansive ones:

$$\begin{array}{lll} N{<}{-}Z{>} & <:: & \\ C{<}X{>} & <:: & NNCCX \end{array}$$

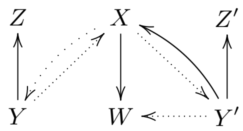


$$\begin{tikzcd} X \arrow[loop above, dotted] \arrow[loop left] \arrow[r] & Z \end{tikzcd}$$

The type parameter X appears in three edges: a non-expansive edge represents its use as argument to the second occurrence of C, an expansive cyclic edge represents its use inside the argument to the first occurrence of C, and an expansive edge to Z represents its use inside the argument to the two occurrences of N.

Now consider a more complex class table and its graph:

$$\begin{array}{lll} C{<}X{>} & <:: & DP{<}X, X{>} \\ P{<}Y, Y'{>} & <:: & Q{<}CY, CDY'{>} \\ Q{<}Z, Z'{>} & <:: & \\ D{<}W{>} & <:: & \end{array}$$



$$\begin{tikzcd} Z & X \arrow[dl, dotted, bend left=10] \arrow[d] \arrow[dr, dotted] & Z' \\ Y \arrow[u] \arrow[ur, dotted, bend left=10] & W & Y' \arrow[l, dotted] \arrow[u] \arrow[ul, bend right] \end{tikzcd}$$

The type parameter X is used in three ways: its appearance in the instantiation of D is represented by an edge to the "sink" node W, its use as first argument to P is represented by an edge in a non-expansive cycle through Y, and its use as second argument to P is represented by an edge in an expansive cycle through Y'. The existence of this latter expansive cycle implies that the class table is infinitary; indeed $cl(\{CT\}) \supseteq \{CD^m T \mid m \geqslant 0\}$.

In order to show that expansiveness is a *sufficient* condition for infinitary closure, we make use of the following relationship between inheritance closure and type parameter dependency graphs.

LEMMA 7. *Suppose $\mathcal{S}$ is inheritance closed and $C{<}\overline{T}{>} \in \mathcal{S}$.*

1. *If $C\#i \xrightarrow{0} D\#j$ then $D{<}\overline{U}{>} \in \mathcal{S}$ for some $\overline{U}$ with $U_j = T_i$.*
2. *If $C\#i \xrightarrow{1} D\#j$ then $D{<}\overline{U}{>} \in \mathcal{S}$ for some $\overline{U}$ such that T_i is a proper subterm of U_j.*

PROOF:

1. From the definition of a type parameter dependency graph, we must have $C{<}\overline{X}{>} <:: V$ for some V with some type $D{<}\overline{V}{>}$ a subterm of V and $V_j = X_i$. By inheritance closure of $\mathcal{S}$ we have $[\overline{T}/\overline{X}]\, V \in \mathcal{S}$ and then by decomposition closure we have $D{<}[\overline{T}/\overline{X}]\overline{V}{>} \in \mathcal{S}$, from which the result follows.
2. Similar. □

THEOREM 8. *If a class table is expansive, then it is infinitary.*

PROOF: Suppose that the class table is expansive. Then its type parameter dependency graph contains a cycle, at least one of whose edges (say the first) is expansive—i.e., either $C\#i \xrightarrow{1} C\#i$ (a one-cycle), or $C\#i \xrightarrow{1} D\#j \rightarrow^+ C\#i$. By repeated use of Lemma 7 we can see that, if $C{<}\overline{T}{>} \in \mathcal{S}$ and $\mathcal{S}$ is inheritance closed, then $C{<}\overline{U}{>} \in \mathcal{S}$ for some $\overline{U}$ with T_i a proper subterm of U_i. In other words, for any instantiation of C in $\mathcal{S}$ there is a larger one; it follows that $\mathcal{S}$ is infinite. □

Showing that expansiveness is a *necessary* condition is a bit more tricky. We need two further notions.

First, we rank the vertices in the graph, assigning each type parameter X a natural number $\mathit{level}(X)$ with the following property.

Suppose $X \rightarrow Y$. If $Y \rightarrow^+ X$ then $level(X) = level(Y)$, else $level(X) > level(Y)$. (One means of assigning levels is first to identify nodes in strongly-connected components and then to topologically sort the resulting DAG.) For the example above, we can make the assignment $level(Z) = level(Z') = level(W) = 0$ and $level(Y) = level(Y') = level(X) = 1$.

Second, we introduce the notion of a *path*. A particular subterm of a type can be identified by a path, which we represent as a sequence of formal type parameters, writing ϵ for the empty path and $X.p$ (or $C\#i.p$) for the path consisting of the formal type parameter X (or $C\#i$) concatenated onto the path p. We interpret a path p as a partial function from terms to subterms, as follows:

$$\frac{}{\epsilon(T) = T} \qquad \frac{p(T_i) = U}{(C\#i.p)(C\texttt{<}\overline{T}\texttt{>}) = U}$$

For example, let $T = C\texttt{<}DU, V\texttt{>}$. Then $(C\#1.\epsilon)(T) = DU$, and $(C\#1.D\#1.\epsilon)(T) = U$. We say that *$p$ is a path in T* if $p(T)$ is defined.

THEOREM 9. *If a class table is infinitary, then it is expansive.*

PROOF: We argue the contrapositive—that non-expansive class tables are finitary. Let $\mathcal{S}$ be a finite set of types. Let δ be a bound on the height of types in $\mathcal{S}$ and in the class table (if $T \in \mathcal{S}$ then $height(T) \leqslant \delta$; if $C\texttt{<}\overline{X}\texttt{>} <:: T$ then $height(T) \leqslant \delta$) and let L be the number of levels (so $0 \leqslant level(X) < L$ for any formal type parameter X). We prove that the height of types in $cl(\mathcal{S})$ is bounded by δL; then, because the set of types of a certain height is finite, it follows that $cl(\mathcal{S})$ must be finite.

Let $\phi(p)$ hold for a path p if it can be divided into a sequence of (possibly empty) sequences of type parameters whose levels are bounded by $0, \ldots, L-1$ and whose lengths are bounded by δ. That is, $\phi(p)$ means that p has the form $\overline{X}_0\overline{X}_1 \cdots \overline{X}_{L-1}$, with $level(\overline{X}_l) \leqslant l$ and $|\overline{X}_l| \leqslant \delta$ for $0 \leqslant l < L$. Let $\phi(T)$ hold for a type T if $\phi(p)$ holds for every path p in T. It is easy to see that $\phi(T)$ implies $height(T) \leqslant \delta L$.

Clearly ϕ holds for the types in $\mathcal{S}$. We show that ϕ is preserved under the closure operations used to construct $cl(\mathcal{S})$. For decomposition, it's clear that, if $\phi(C\texttt{<}\overline{T}\texttt{>})$ holds, then so does $\phi(T_i)$ for each i. For inheritance, suppose that $C\texttt{<}\overline{X}\texttt{>} <:: U$; we must show that $\phi(C\texttt{<}\overline{T}\texttt{>})$ implies $\phi([\overline{T}/\overline{X}]U)$. If U is simply a type parameter (mixin inheritance), then the result follows directly. Otherwise, consider a path p in $[\overline{T}/\overline{X}]U$. There are two possibilities. First, p could be simply a path in U that maps to a non-variable subterm ($p(U) = D\texttt{<}\overline{U}\texttt{>}$ for some D and $\overline{U}$). In this case we know that $|p| \leqslant \delta$ and so we have $\phi(p)$ immediately. Otherwise, $p = p'.q$ for some non-empty p' and q such that $p'(U) = X_i$ and q is a path in T_i. Hence $C\#i.q$ is a path in $C\texttt{<}\overline{T}\texttt{>}$, and so from $\phi(C\texttt{<}\overline{T}\texttt{>})$ we can deduce $\phi(C\#i.q)$, or written another way, $\phi(X_i.q)$. Now if $level(X_i) = k$ then $q = \overline{Y_k}.\overline{Y_{k+1}}.\ldots.\overline{Y_{L-1}}$, with $level(Y_l) \leqslant l$ for $k \leqslant l < L$ and with $|\overline{Y_k}| < \delta$ and $|\overline{Y_l}| \leqslant \delta$ for $k < l < L$. Suppose $p' = \overline{Z}.Z$. By the definition of the type parameter dependency graph, we know that $X_i \xrightarrow{1} Z_j$ for each j and that $X_i \xrightarrow{0} Z$. Because the graph contains no expansive cycles (that is, we do not have $Z_j \rightarrow^+ X_i$), we can deduce that $level(Z_j) < level(X_i) = k$ for each j. Finally, because $|\overline{Z}| < \delta$, we can see that $p = \overline{Z}.Z.\overline{Y_k}.\ldots.\overline{Y_{L-1}}$ satisfies ϕ, as required. $\square$

COROLLARY 10. *Non-expansive subtyping is decidable.*

5.3 Multiple instantiation inheritance

We now consider the third ingredient of the reduction from PCP: multiple instantiation inheritance. Java 5 prohibits it:

$$(\textsc{SuperVar})\ \frac{T <::^* D\texttt{<}\overline{T}\texttt{>} \qquad \text{for each } i,\ T_i <:_{var(D\#i)} U_i}{T <: D\texttt{<}\overline{U}\texttt{>}}$$

$$\frac{T <: U}{T <:_+ U} \qquad \frac{}{T <:_\circ T} \qquad \frac{U <: T}{T <:_- U}$$

Figure 3. Deterministic subtyping

"A class may not at the same time be a subtype of two interface types which are different invocations of the same generic interface, or an invocation of a generic interface and a raw type naming that same generic interface." [12, §8.1.5]

To put it another way, generic instantiations are uniquely determined by the inheritance relation: if $T <::^* C\texttt{<}\overline{U}\texttt{>}$ and $T <::^* C\texttt{<}\overline{V}\texttt{>}$ then $\overline{U} = \overline{V}$. The Java 5 specification goes on to say:

"This requirement was introduced in order to support translation by type erasure."

C$^\sharp$ 2.0, which does not erase types, has no such restriction. Instead, it merely requires supertypes to be non-overlapping: if $C\texttt{<}\overline{X}\texttt{>} <:: T$ and $C\texttt{<}\overline{X}\texttt{>} <:: U$, then, for all instantiations $\overline{V}$, if $[\overline{V}/\overline{X}]T = [\overline{V}/\overline{X}]U$, then $T = U$.

If multiple instantiation inheritance is prohibited, we can reformulate subtyping so that derivations are unique. Figure 3 presents the revised rules, in which SUPERVAR combines variance and inheritance in a single rule. It is easy to show that the rules define the same relation. (For an example of non-unique derivations under *single* instantiation inheritance in the original system, consider the declarations $CX <:: IX$ and $D <:: CE, IE$ and the assertion $D <: IE$. In the revised rules, such non-determinism is 'hidden' inside the $<::^*$ side-condition on rule SUPERVAR.)

Of course, uniqueness of derivations in a syntax-directed system does not imply decidability: derivations in $F_{\leq}$ are unique, but subtyping is undecidable [19]. However, it does simplify the problem somewhat, as the search for a derivation does not involve a choice of paths, but must determine merely whether a particular linear path is finite (and, if so, whether it ends in a successful state).

Fortunately, for all examples in Section 3—and for many other more complex ones—it is possible to detect the infinite regress using a notion of *accessibility*, which we describe next. (Unfortunately, we do not yet have a general result for all class tables without multiple instantiation inheritance; our proof requires an additional, somewhat artificial, technical condition on class tables, described below. It is possible that, without this restriction, the problem is still undecidable.)

Consider Example 2 once more. Observe that the types T and U actually play no role in the reduction: they can be replaced by arbitrary types without affecting the validity of the subtype assertion. Indeed, at any point in the reduction, all but the first occurrence of C in the types can be replaced without affecting validity, because that part of the type is never "accessed." To make things a little clearer, we adapt the example to use a different type inside C. The definitions and type parameter dependency graph are as follows:

$$\begin{array}{lll} N\texttt{<-}Z\texttt{>} & <:: & \\ D\texttt{<}Y\texttt{>} & <:: & \\ C\texttt{<}X\texttt{>} & <:: & NNCDX \end{array}$$

$$\begin{tikzcd} Y & X \arrow[l, dotted] \arrow[loop above] \arrow[r, bend right] & Z \end{tikzcd}$$

Using the new rules from Figure 3, the pattern of regress is as follows:

$$\begin{array}{lll} & CT <: NCU & \\ \longrightarrow & CU <: NCDT & \text{by SUPERVAR} \\ \longrightarrow & CDT <: NCDU & \text{by SUPERVAR} \\ \longrightarrow & CDU <: NCDDT & \text{by SUPERVAR} \\ \longrightarrow & CDDT <: NCDDU & \text{by SUPERVAR} \\ \longrightarrow & \text{etc.} & \end{array}$$

Notice that C is invariant and recursive through X: the former property means that it cannot be "passed through" using variance to reach a goal involving its arguments—the only thing we can do with it is to use the inheritance relation to replace it (on the left of $<:$) by some supertype—and the latter means that, when an instantiation of C is replaced by something else in this way, the type replacing it is another instantiation of C itself or of another class in mutual recursion with C.

Of course, a type parameter may be used both recursively and non-recursively in different subterms of the same type, or in different supertype declarations. To obtain our (preliminary) decidability result, we impose a somewhat artificial restriction on the class table: if a type parameter X appears in an expansive cycle in the type parameter dependency graph, then (a) it is an invariant type parameter, and (b) it appears exactly once in a single supertype declaration. We call such a type parameter *expansive-recursive*.

We can now formalize the idea of *accessibility* by defining a relation $\sim$ between types, read "same accessible parts" and defined as follows: $C\texttt{<}\overline{T}\texttt{>} \sim D\texttt{<}\overline{U}\texttt{>}$ if $C = D$ and for each i either $C\#i$ is expansive-recursive or else $T_i \sim U_i$. We extend the definition to subtype judgments: $(T <: T') \sim (U <: U')$ iff $T \sim U$ and $T' \sim U'$. Alternatively, we say that a path p is *accessible* if it does not contain an expansive-recursive type parameter. It is easy to show that $T \sim U$ iff T and U have the same set of accessible paths.

The following technical lemma relates this notion of accessibility to the inheritance relation.

LEMMA 11. *Suppose $T <::^* U$ and $T \sim T'$. Then $T' <::^* U'$ for some U' such that $U \sim U'$.*

PROOF: By induction on the number of steps of inheritance used to derive $T <::^* U$.

- Suppose $U = T$. Then set $U' = T'$ and we're done.
- Suppose $T = C\texttt{<}\overline{T}\texttt{>}$ and $T' = C\texttt{<}\overline{T'}\texttt{>}$, with $C\texttt{<}\overline{X}\texttt{>} <:: V$ and $[\overline{T}/\overline{X}]\,V <::^* U$. We will show that, if p is an accessible path in V, then $[\overline{T}/\overline{X}](p(V)) \sim [\overline{T}'/\overline{X}](p(V))$; setting $p = \epsilon$ then gives us $[\overline{T}/\overline{X}]\,V \sim [\overline{T'}/\overline{X}]\,V$, from which the desired result follows by the induction hypothesis.

  We proceed by an inner induction on the size of term $p(V)$.
  - First, suppose $p(V) = X_i$ for some i. We deduce that X_i is not expansive-recursive by the following argument. Suppose it were. Then p would represent the only occurrence of X_i (by our linearity restriction), and the only edges from X_i in the type parameter dependency graph would be to nodes in p, as they represent all arguments to constructors in V for which X_i is a subterm. By the definition of expansive-recursive edges, one of these edges would belong to an expansive cycle in the graph, and hence one of the type parameters in p would itself be expansive-recursive. This contradicts our assumption that p is accessible; hence X_i is not expansive-recursive. This being the case, we immediately have $T_i \sim T_i'$ because $C\texttt{<}\overline{T}\texttt{>} \sim C\texttt{<}\overline{T'}\texttt{>}$.
  - Next, suppose $p(V)$ is some compound type $D\texttt{<}\overline{V}\texttt{>}$. We must show that, for each V_i with $D\#i$ not expansive-recursive, $[\overline{T}/\overline{X}]\,V_i \sim [\overline{T}'/\overline{X}]\,V_i$. This follows by an application of the inner induction hypothesis on the (accessible) path $p.D\#i$. ☐

For the rest of the argument, it is convenient to introduce a notation for "reduction" between subtyping goals. We write $J \longrightarrow J'$ if judgments J and J' are respectively the conclusion and premise of an instance of SUPERVAR. (We have been relying on the same informal intuition all along, of course, in arguments of the form "subtyping judgment J_1 can only be provable if J_2 is," but the notion is particularly easy to formalize in the present setting, where we have imposed enough conditions to make proof search deterministic.) This relation is specified by the following inference rules:

$$\frac{T <::^* D\texttt{<}\overline{V}\texttt{>} \qquad var(D\#i) = +}{T <: D\texttt{<}\overline{U}\texttt{>} \longrightarrow V_i <: U_i}$$

$$\frac{T <::^* D\texttt{<}\overline{V}\texttt{>} \qquad var(D\#i) = -}{T <: D\texttt{<}\overline{U}\texttt{>} \longrightarrow U_i <: V_i}$$

The following key lemma captures our intuition that only the accessible part of a type affects reducibility.

LEMMA 12. *Suppose J_1 and J_2 are subtyping judgments with $J_1 \sim J_2$. If $J_1 \longrightarrow J_1'$, then $J_2 \longrightarrow J_2'$ for some J_2' such that $J_1' \sim J_2'$.*

PROOF: Suppose $J_1 = T <: D\texttt{<}\overline{U}\texttt{>}$ and $J_2 = T' <: D\texttt{<}\overline{U}'\texttt{>}$. Then J_1 is reducible only if $var(D\#i) \neq \circ$ for some i. Suppose $var(D\#i) = +$ (the case for $var(D\#i) = -$ is similar). We must have $T <::^* D\texttt{<}\overline{V}\texttt{>}$ for some $\overline{V}$ and $J_1' = V_i <: U_i$. By Lemma 11 we have $T' <::^* U'$ for some V' such that $D\texttt{<}\overline{V}\texttt{>} \sim V'$, so $V' = D\texttt{<}\overline{V'}\texttt{>}$ for some $\overline{V'}$, and hence $J_2 \longrightarrow J_2'$ where $J_2' = V_i' <: U_i'$. Finally, since $D\texttt{<}\overline{U}\texttt{>} \sim D\texttt{<}\overline{U'}\texttt{>}$ and $D\texttt{<}\overline{V}\texttt{>} \sim D\texttt{<}\overline{V'}\texttt{>}$ it follows immediately from the definition of $\sim$ that $U_i \sim U_i'$ and $V_i \sim V_i'$, as required. ☐

COROLLARY 13. *If $J \longrightarrow^+ J'$ and $J \sim J'$ then $J \longrightarrow^\infty$.*

We now have a sufficient condition for non-termination: if we encounter a goal that matches a goal already seen up to accessibility, then we can return *false* immediately.

Example 4 demonstrates that many reductions may occur before such a matching goal is reached. Fortunately, we can show that the number of reductions is bounded: although the inaccessible part of a type can grow unboundedly (as in Example 2), the accessible part cannot. Hence, there are only a finite number of possible types up to accessibility. This is the key to decidability.

Let δ be a bound on the height of a supertype (if $C\texttt{<}\overline{X}\texttt{>} <:: T$ then $height(T) \leqslant \delta$), and let L be the number of levels (so $0 \leqslant level(X) < L$). Given a particular subtyping problem $T <: U$, we can always arrange for δ to be larger than both $height(T)$ and $height(U)$.

Define a notion of *accessible height* as follows.

$$acc(T) = \max\{|p| \mid p \text{ is an accessible path in } T\}$$

We now show that the accessible height of subtype judgments is bounded by δL.

LEMMA 14. *If $acc(J) \leqslant \delta L$ and $J \longrightarrow J'$ then $acc(J') \leqslant \delta L$.*

PROOF: Let $\phi(p)$ hold for a path p if it can be divided into (possibly empty) sequences of type parameters whose levels are bounded by $0, \ldots, L{-}1$ and whose lengths are bounded by δ. That is, p has the form $\overline{X}_0\overline{X}_1 \cdots \overline{X}_{L-1}$ such that $level(\overline{X}_i) \leqslant i$, and $|\overline{X}_i| \leqslant \delta$ for $0 \leqslant i < L$. Let $\phi(T)$ hold for a type T if $\phi(p)$ holds for every accessible path p in T. It is easy to see that $\phi(T)$ implies $acc(T) \leqslant \delta L$.

We now show that, if $\phi(T)$ and $\phi(U)$, and $T <: U \longrightarrow T' <: U'$, then $\phi(T')$ and $\phi(U')$. The argument is very similar to that pursued in Theorem 9, but this time relies on the fact that accessible paths contain no expansive-recursive type parameters. $\square$

COROLLARY 15. *Suppose that the class table makes no use of multiple instantiation inheritance and that any expansive-recursive type parameters are invariant and used exactly once. Then subtyping is decidable.*

6. Discussion

Starting with only basic restrictions on class tables (acyclicity and well-formedness with respect to variance), we have proved the following results about subtyping under inheritance and declaration-site variance:

- The general problem is undecidable (§4).
- If the class table makes no use of contravariance, then subtyping is decidable (§5.1).
- If the class table is not expansive, then subtyping is decidable (§5.2).
- If there is no use of multiple instantiation inheritance, and all expansive-recursive type parameters are invariant and linear, then subtyping is decidable (§5.3).

Of these results, decidability of non-expansive subtyping can be applied directly to ground subtyping in .NET (that is, subtyping between closed types). It is straightforward to generalize the result to incorporate other features of the .NET type system such as covariant arrays, invariant managed pointer types, and boxed types. (Decidability even of ground subtyping is important, as subtype tests at runtime involve un-erased generic types.) To generalize to open subtyping, as employed by the .NET CLR verifier (type-checker), we must support upper bounds on type parameters, and the associated subtyping rule ($\Delta \vdash X <: U$ can be derived from premise $\Delta \vdash T <: U$ if $X <: T \in \Delta$). For instance, an analog of Example 1 can be given using bounds:[2] $X <: NNX \vdash X <: NX$. Observe, though, that bounds alone cannot induce arbitrary expansion of types, because type parameters cannot range over type constructors: they are not higher-kinded. Put another way, there is no analog of Example 2 using bounds. We believe that our decidability result generalizes to the complete type system of .NET 2.0.

Supertype declarations in Java, C$^\sharp$ and Scala have the form $C\texttt{<}\overline{X}\texttt{>} <:: T$. The *constraint-bounded polymorphism* studied by Litvinov [17, 16] also supports "lower bounds" of the form $T <:: C\texttt{<}\overline{X}\texttt{>}$. It is easy to adapt the reduction from PCP to show that subtyping is undecidable in the presence of covariance and both upper and lower bounds in the class table, without the need for contravariance.

One obvious direction for future work is to generalize the result of Section 5.3, removing the special requirement on expansive-recursive type parameters. If expansive subtyping in the absence of multiple instantiation inheritance turned out to be decidable, then perhaps this could be combined with the existing result on a subset of Scala [8] to prove that Scala has decidable subtyping.

An undecidability result for expansive subtyping, on the other hand, would imply that both Scala 2.0 and Java 5 have undecidable subtyping. A fix for Scala might be to apply the non-expansiveness restriction on class tables discussed in Section 5.2. We do not believe that this restriction adversely affects expressivity—at least, we have been unable to devise any practical application of expansive inheritance.

[2] Thanks to Martin Odersky for observing this.

For Java 5, other features must be explored. First, variance behaviour for wildcard types is actually a *consequence* of the interpretation—intuitively, and in the subtyping rules—of wildcards as bounded existential types [22]. It is not immediately apparent how to adapt our decidability results to these very different rules. Second, the combination of Java's wildcards and F-bounded type parameters is particularly intricate, and can lead to unbounded growth in the *typing context* during subtype checking.

Finally, our undecidability result highlights a hazard in extending the type systems of any of these languages: if some future version of Java or Scala were to support multiple instantiation inheritance, or the .NET CLR were adapted to support expansive inheritance through lazy loading of supertypes, then subtyping would (most definitely!) be undecidable.

Acknowledgements

The work was begun while the second author was a visiting researcher at Microsoft Research, Cambridge. We would like to thank Nick Benton, Gavin Bierman, Karl Mazurak, Martin Odersky, Claudio Russo, and the members of the Penn PL Club for helpful discussions, and the FOOL reviewers for their extremely careful and detailed comments.

References

[1] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Anaheim, California, October 2003. ACM.

[2] P. Baldan, G. Ghelli, and A. Raffaetà. Basic theory of F-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, Oct. 1998.

[4] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded quantification for object-oriented programming. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 273–280, Sept. 1989.

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

[6] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Summary in TACS '91 (Sendai, Japan, pp. 750–770).

[7] A. B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, Sept. 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

[8] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In *Proc. MFCS*, Springer LNCS, Sept. 2006.

[9] ECMA International. ECMA Standard 335: Common Language Infrastructure, 3rd edition, June 2005. Available at `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[10] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C♯ generics. In *European Conference on Object-Oriented Programming (ECOOP), Nantes, France*, July 2006.

[11] G. Ghelli. Termination of system F-bounded: A complete proof. *Information and Computation*, 139(1):39–56, 1997.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, June 2005.

[13] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN*

Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Oct. 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[14] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(5), September 2006.

[15] N. D. Jones. *Computability and Complexity From a Programming Perspective*. The MIT Press, 1997.

[16] V. Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Vancouver, October 1998. ACM.

[17] V. Litvinov. *Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages*. PhD thesis, University of Washington, 2003.

[18] M. Odersky and M. Zenger. Scalable component abstractions. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*. ACM, 2005.

[19] B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994. Summary in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*.

[20] B. C. Pierce and M. Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[21] M. Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1998.

[22] M. Torgensen, E. Ernst, and C. P. Hansen. Wild FJ. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2005.

[23] M. Viroli. On the recursive generation of parametric types. Technical Report DEIS-LIA-00-002, Università di Bologna, 2000.

A. Java examples

We present the Java 5 equivalent of the examples from Section 3.

Example 1 This one causes `javac 1.5` to run out of stack. On `javac 1.6.0-beta2` the program is rejected (correctly).

```
class N<Z> { }
class C extends N<N<? super C>> {
  N<? super C> cast(C c) { return c; }
}
```

Example 2 The following example causes both `javac 1.5` and `javac 1.6.0-beta2` to run out of stack.

```
class T { }
class N<Z> { }
class C<X> extends N<N<? super C<C<X>>>> {
  N<? super C<T>> cast(C<T> c) { return c; }
}
```

Example 3 The following example causes `javac 1.5` to run out of stack. If the test uses `C7` instead, the program is accepted (correctly). On `javac 1.6.0-beta2` the failure happens at `C13`.

```
class T { }
class N<Z> { }
class C0<X> extends N<N<? super X>> { }
class C1<X> extends C0<C0<X>> { }
class C2<X> extends C1<C1<X>> { }
class C3<X> extends C2<C2<X>> { }
class C4<X> extends C3<C3<X>> { }
class C5<X> extends C4<C4<X>> { }
class C6<X> extends C5<C5<X>> { }
class C7<X> extends C6<C6<X>> { }
class C8<X> extends C7<C7<X>> { }
class Test {
  N<? super C8<T>> cast(C8<N<? super T>> c)
  { return c; }
}
```

Example 4 Finally, this example causes `javac 1.5` to run out of stack. On `javac 1.6.0-beta2` the program is rejected (correctly); but again, fails at `C13`.

```
class T { }
class N<Z> { }
class C0<X> extends N<N<? super X>> { }
class C1<X> extends C0<C0<X>> { }
class C2<X> extends C1<C1<X>> { }
class C3<X> extends C2<C2<X>> { }
class C4<X> extends C3<C3<X>> { }
class C5<X> extends C4<C4<X>> { }
class C6<X> extends C5<C5<X>> { }
class C7<X> extends C6<C6<X>> { }
class C8<X> extends C7<C7<C8<X>>> { }
class Test {
  N<? super C8<T>> cast(C8<N<? super T>> c)
  { return c; }
}
```

B. Proof of Transitivity

The proof that the subtype relation is transitive relies on one technical lemma:

LEMMA 16. *Suppose* $\overline{vX} \vdash C\texttt{<}\overline{T}\texttt{>}$ *ok. If* $C\texttt{<}\overline{wY}\texttt{>} <:: U$ *then* $\overline{vX} \vdash [\overline{T}/\overline{Y}]\, U$ *ok.*

PROOF: We prove the following, from which the result follows because by well-formedness of class declarations we have $\overline{wY} \vdash U$ *ok*.

1. If $\overline{wY} \vdash V$ *ok* then $\overline{vX} \vdash [\overline{T}/\overline{Y}]\, V$ *ok*.
2. If $\neg\overline{wY} \vdash V$ *ok* then $\neg\overline{vX} \vdash [\overline{T}/\overline{Y}]\, V$ *ok*.

We proceed by simultaneous induction on both derivations. □

With this in hand, we are ready for the main proof of transitivity.

PROOF OF LEMMA 1: Suppose the derivation of $T <: U$ has size m and the derivation of $U <: V$ has size n. We proceed by induction on $m+n$. When both derivations end in rule VAR or when the first ends in rule SUPER, the result follows by straightforward applications of the induction hypothesis.

The interesting case is when the first derivation ends in rule VAR and the second derivation ends in rule SUPER. Suppose $T = C\texttt{<}\overline{T}\texttt{>}$ and $U = C\texttt{<}\overline{U}\texttt{>}$. Then we have derivations concluding as follows:

$$\frac{\text{for each } i \quad T_i <:_{v_i} U_i}{C\texttt{<}\overline{T}\texttt{>} <: C\texttt{<}\overline{U}\texttt{>}} \qquad \frac{C\texttt{<}\overline{vX}\texttt{>} <:: V_0 \quad [\overline{U}/\overline{X}]\, V_0 <: V}{C\texttt{<}\overline{U}\texttt{>} <: V}$$

By well-formedness of class declarations, we know $\overline{vX} \vdash V_0$ *ok*. We will now show that $[\overline{T}/\overline{X}]\, V_0 <: V$, from which the result follows using an instance of rule SUPER. To do this, we essentially transform the derivation of $[\overline{U}/\overline{X}]\, V_0 <: V$ into a derivation of $[\overline{T}/\overline{X}]\, V_0 <: V$ by replacing each subderivation of the form $[\overline{U}/\overline{X}]X_i <: W$ by a derivation of $T_i <: W$ and replace each subderivation of the form $W <: [\overline{U}/\overline{X}]X_i$ by $W <: T_i$.

Under the assumptions from rule VAR above (namely, that $T_i <:_{v_i} U_i$ for each i), we prove the following. For any types W and W_0, (1) if $\overline{vX} \vdash W_0$ *ok*, and $[\overline{U}/\overline{X}]\, W_0 <: W$ has a derivation of size smaller than n, then $[\overline{T}/\overline{X}]\, W_0 <: W$ is derivable; and (2) if $\neg\overline{vX} \vdash W_0$ *ok*, and $W <: [\overline{U}/\overline{X}]\, W_0$ has a derivation of size smaller than n, then $W <: [\overline{T}/\overline{X}]\, W_0$ is derivable. We proceed by induction on both subtype derivations simultaneously.

- Suppose $W_0 = X_i$. For (1) well-formedness of W_0 tells us that $v_i \in \{\circ, +\}$. Consider the case when $v_i = +$. Then we have a derivation of $U_i <: W$ of size smaller than n; we can apply the outer induction hypothesis to get a derivation of $T_i <: W$, as required. Now suppose $v_i = \circ$. We must have $U_i = W$ so $T_i <: W$ follows trivially. For (2) well-formedness of W_0 tells us that $\neg v_i \in \{\circ, +\}$, that is $v_i \in \{\circ, -\}$. Consider the case when $v_i = -$. Then we have a derivation of $W <: U_i$ of size smaller than n; we can apply the outer induction hypothesis to get a derivation of $W <: T_i$, as required. Again, $v_i = \circ$ is trivial.
- Suppose $W_0 = D\texttt{<}\overline{W}\texttt{>}$. There are two sub-cases to consider.
  - The derivation ends with an instance of VAR, so $W = D\texttt{<}\overline{W'}\texttt{>}$ for some $\overline{W'}$. We show (1), and (2) is similar. Then we must have for each j that $[\overline{U}/\overline{X}]\, W_j <:_{var(D\#j)} W'_j$. Suppose $var(D\#j) = +$. Then by the well-formedness derivation for W_0 we know that $\overline{vX} \vdash W_j$ *ok*, so we can apply the inner induction hypothesis part (1) to obtain $[\overline{T}/\overline{X}]\, W_j <: W'_j$. Suppose $var(D\#j) = -$. Then by the well-formedness derivation for W_0 we know that $\neg\overline{vX} \vdash W_j$ *ok*, so we can apply the inner induction hypothesis part (2) to obtain $W'_j <: [\overline{T}/\overline{X}]\, W_j$. Finally suppose $var(D\#j) = \circ$. We have $\overline{vX} \vdash W_j$ *ok and* $\neg\overline{vX} \vdash W_j$ *ok* and $[\overline{U}/\overline{X}]\, W_j = W'_j$. By a simple induction on the well-formedness derivations we can deduce that $[\overline{T}/\overline{X}]\, W_j = [\overline{U}/\overline{X}]\, W_j$. Hence we have shown for all j that $[\overline{T}/\overline{X}]\, W_j <:_{var(D\#j)} W'_j$, and the result follows by an application of rule VAR.

- For (1), the derivation must conclude with the following instance of SUPER:

$$\frac{D\texttt{<}\overline{Y}\texttt{>} <:: T_0 \quad [[\overline{U}/\overline{X}]\overline{W}/\overline{Y}]\,T_0 <: W}{[\overline{U}/\overline{X}]D\texttt{<}\overline{W}\texttt{>} <: W}$$

  By Lemma 16, we have $\overline{vX} \vdash [\overline{W}/\overline{Y}]\,T_0$ *ok*. Hence $[\overline{T}/\overline{X}][\overline{W}/\overline{Y}]\,T_0 <: W$ follows by an application of the inner induction hypothesis part (1) to the premise above.

  For (2), we have a derivation ending in:

$$\frac{E\texttt{<}\overline{Y}\texttt{>} <:: T_0 \quad [\overline{V}/\overline{Y}]\,T_0 <: [\overline{U}/\overline{X}]\,W_0}{E\texttt{<}\overline{V}\texttt{>} <: [\overline{U}/\overline{X}]\,W_0}$$

  The result follows by applying the inner induction hypothesis part (2) to the premise. □