Taming Wildcards in Java's Type System [*]

Technical Report (incomplete)

Ross Tate
University of California, San Diego
rtate@cs.ucsd.edu

Alan Leung
University of California, San Diego
aleung@cs.ucsd.edu

Sorin Lerner
University of California, San Diego
lerner@cs.ucsd.edu

Abstract

Wildcards have become an important part of Java's type system since their introduction 7 years ago. Yet there are still many open problems with Java's wildcards. For example, there are no known sound and complete algorithms for subtyping (and consequently type checking) Java wildcards, and in fact subtyping is suspected to be undecidable because wildcards are a form of bounded existential types. Furthermore, some Java types with wildcards have no joins, making inference of type arguments for generic methods particularly difficult. Although there has been progress on these fronts, we have identified significant shortcomings of the current state of the art, along with new problems that have not been addressed.

In this paper, we illustrate how these shortcomings reflect the subtle complexity of the problem domain, and then present major improvements to the current algorithms for wildcards by making slight restrictions on the usage of wildcards. Our survey of existing Java programs suggests that realistic code should already satisfy our restrictions without any modifications. We present a simple algorithm for subtyping which is both sound and complete with our restrictions, an algorithm for lazily joining types with wildcards which addresses some of the shortcomings of prior work, and techniques for improving the Java type system as a whole. Lastly, we describe various extensions to wildcards that would be compatible with our algorithms.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—Java; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Algorithms, Design, Languages, Theory

Keywords Wildcards, Subtyping, Existential types, Parametric types, Joins, Type inference, Single-instantiation inheritance

1. Introduction

Java 5, released in 2004, introduced a variety of features to the Java programming language, most notably a major overhaul of the type system for the purposes of supporting generics. Although Java has undergone several revisions since, Java generics have remained unchanged since they were originally introduced into the language.

Java generics were a long-awaited improvement to Java and have been tremendously useful, leading to a significant reduction in the amount of unsafe type casts found in Java code. However, while Java generics improved the language, they also made type checking extremely complex. In particular, Java generics came with *wildcards*, a sophisticated type feature designed to address the limitations of plain parametric polymorphism [19].

Wildcards are a simple form of existential types. For example, `List<?>` represents a "list of unknowns", namely a list of objects, all of which have the same unknown static type. Similarly, `List<? extends Number>` is a list of objects of some unknown static type, but this unknown static type must be a subtype of `Number`. Wildcards are a very powerful feature that is used pervasively in Java. They can be used to encode use-site variance of parametric types [7, 17–19], and have been used to safely type check large parts of the standard library without using type casts. Unfortunately, the addition of wildcards makes Java's type system extremely complex. In this paper we illustrate and address three issues of wildcards: subtyping, type-argument inference, and inconsistencies in the design of the type system.

Subtyping with wildcards is surprisingly challenging. In fact, there are no known sound and complete subtyping algorithms for Java, soundness meaning the algorithm accepts only subtypings permitted by Java and completeness meaning the algorithm always terminates and accepts all subtypings permitted by Java. Subtyping with wildcards is even suspected to be undecidable, being closely related to the undecidable problem of subtyping with bounded existential types [21]. In Section 3 we will illustrate this challenge, including examples of programs which make `javac`[†] suffer a stack overflow. In Section 4 we will present our simple subtyping algorithm which is sound and complete given certain restrictions.

Java also includes type-argument inference for generic methods, which again is particularly challenging with wildcards. Without type-argument inference, a programmer would have to provide type arguments each time a generic method is used. Thus, to make generic methods convenient, Java infers type arguments at method-invocation sites. Furthermore, Java can infer types not expressible by users, so that explicit annotation is not always an option. Unfortunately, there is no known sound and complete type-argument inference algorithm. Plus, type-argument inference can even affect the semantics of a program. We illustrate these issues in Section 5 and present our improvements on the state of the art in Section 6.

Wildcards also introduce a variety of complications to Java's type system as a whole. While Java attempts to address these complications, there are yet many to be resolved. In some cases Java is overly restrictive, while in others Java is overly relaxed. In fact, the type-checking algorithm used by `javac` is non-deterministic from the user's perspective due to wildcards. In Section 7 we will illustrate these issues, and in Section 8 we will present our solutions.

A few of our solutions involve imposing restrictions on the Java language. Naturally one wonders whether these restrictions are practical. As such, we have analyzed 9.2 million lines of open-source Java code and determined that *none* of our restrictions are violated. We present our findings in Section 9, along with a number of interesting statistics on how wildcards are used in practice.

[*] This work was supported by NSF grant CCF-0644306 and a Microsoft Research PhD Fellowship.

[†] When we refer to `javac` we mean version 1.6.0_22.

Java is an evolving language, and ideally our algorithms can evolve with it. In Section 10 we present a variety of extensions to Java which preliminary investigations indicate would be compatible with our algorithms. These extensions also suggest that our algorithms could apply to other languages such as C# and Scala.

Many of the above difficulties of wildcards are by no means new and have been discussed in a variety of papers [4, 9, 14, 21]. In response to these challenges, researchers have explored several ways of fixing wildcards. The work by Smith and Cartwright [14] in particular made significant progress on improving algorithms for type checking Java. Throughout this paper we will identify the many contributions of these works. However, we will also identify their shortcomings, motivating the need for our improvements. Although this paper does not solve all the problems with type checking Java, it does significantly improve the state of the art, providing concrete solutions to many of the open issues with wildcards.

2. Background

In early proposals for adding parametric polymorphism to Java, namely GJ [1], one could operate on `List<String>` or on `List<Number>`, yet operating on arbitrary lists was inconvenient because there was no form of variance. One had to define a method with a polymorphic variable `P` and a parameter of type `List<P>`, which seems natural except that this had to be done even when the type of the list contents did not matter. That is, there was no way no refer to all lists regardless of their elements. This can be especially limiting for parametric classes such as `Class<P>` for which the type parameter is not central to its usage. Thus, Java wanted a type system beyond standard parametric polymorphism to address these limitations.

2.1 Wildcards

Wildcards were introduced as a solution to the above problem among others [19]. `List<?>` stands for a list whose elements have an arbitrary unknown static type. Types such as `List<String>`, `List<Number>`, and `List<List<String>>` can all be used as a `List<?>`. The `?` is called a wildcard since it can stand for any type and the user has to handle it regardless of what type it stands for. One can operate on a `List<?>` as they would any list so long as they make no assumptions about the type of its elements. One can get its length, clear its contents, and even get `Object`s from it since in Java all instances belong to `Object`. As such, one might mistake a `List<?>` for a `List<Object>`; however, unlike `List<Object>`, one cannot add arbitrary `Object`s to a `List<?>` since it might represent a `List<String>` which only accepts `String`s or a `List<Number>` which only accepts `Number`s.

Wildcards can also be constrained in order to convey restricted use of the type. For example, the type `List<? extends Number>` is often used to indicate read-only lists of `Number`s. This is because one can get elements of the list and statically know they are `Number`s, but one cannot add `Number`s to the list since the list may actually represent a `List<Integer>` which does not accept arbitrary `Number`s. Similarly, `List<? super Number>` is often used to indicate write-only lists. This time, one cannot get `Number`s from the list since it may actually be a `List<Object>`, but one can add `Number`s to the list. Note, though, that this read-only/write-only usage is only convention and not actually enforced by the type system. One can mutate a `List<? extends Number>` via the `clear` method and one can read a `List<? super Number>` via the `length` method since neither method uses the type parameter for `List`.

Java's subtyping for types with wildcards is very flexible. Not only can a `List<Error>` be used as a `List<? extends Error>`, but a `List<? extends Error>` can even be used as a `List<? extends Throwable>` since `Error` is a subclass of `Throwable`. Similarly, a `List<Throwable>` can be used as a `List<? super Throwable>` which can be used as a `List<? super Error>`. Thus by constraining wildcards above one gets covariant subtyping, and by constraining wildcards below one gets contravariant subtyping. This is known as use-site variance [17] and is one of the basic challenges of subtyping with wildcards. However, it is only the beginning of the difficulties for subtyping, as we will demonstrate in Section 3. Before that, we discuss the connection between wildcards and existential types as it is useful for understanding and formalizing the many subtleties within wildcards.

2.2 Existential Types

Since their inception, wildcards have been recognized as a form of existential types [4, 5, 7, 9, 18, 19, 21]. The wildcard `?` represents an existentially quantified type variable, so that `List<?>` is shorthand for the existentially quantified type $\exists$`X. List<X>`. Existential quantification is dual to universal quantification; a $\forall$`X. List<X>` can be used as a `List<String>` by instantiating `X` to `String`, and dually a `List<String>` can be used as an $\exists$`X. List<X>` by instantiating `X` to `String`. In particular, any list can be used as an $\exists$`X. List<X>`, just like `List<?>`. The conversion from concrete instantiation to existential quantification is often done with an explicit *pack* operation, but in this setting all packs are implicit in the subtype system.

Bounded wildcards are represented using bounded quantification. The wildcard type `List<? extends Number>` is represented by $\exists$`X extends Number. List<X>`. A list belongs to this existential type if its element type is a subtype of `Number`. Examples are `List<Integer>` and `List<Double>`, which also belong to `List<? extends Number>` as expected. Once again this is dual to bounded universal quantification.

Even subtyping of wildcards can be formalized by existential subsumption (dual to universal subsumption [8, 10, 11]). For example, $\exists$`X extends Error. List<X>` subsumes $\exists$`Y extends Throwable. List<Y>` because the type variable `Y` can be instantiated to the type `X` which is a subtype of `Throwable` because `X` is constrained to be a subtype of `Error` which is a subclass of `Throwable`. Often this subsumption relation is accomplished by explicit *open* and *pack* operations. The left side is opened, moving its variables and their constraints into the general context, and then the types in the opened left side are packed into the variables bound in the right side and the constraints are checked to hold for those packed types. In the context of wildcards, the opening process is called wildcard capture [6, 19] and is actually part of the specification for subtyping [6: Chapter 4.10.2].

2.3 Implicit Constraints

This means that implicit constraints offer more than just convenience; they actually increase the expressiveness of wildcards, producing a compromise between the friendly syntax of wildcards and the expressive power of bounded existential types. However, this expressiveness comes at the cost of complexity and is the source of many of the subtleties behind wildcards.

2.4 Notation

For traditional existential types in which all constraints are explicit, we will use the syntax $\exists\Gamma : \Delta.\ \tau$. The set Γ is the set of bound variables, and the set Δ is *all* constraints on those variables. Constraints in Δ have the form $v\ <::\ \tau'$ analogous to `extends` and $v\ ::>\ \tau'$ analogous to `super`. Thus we denote the traditional existential type for the wildcard type `SortedList<? super Integer>` as $\exists$`X:X` $::>$ `Integer, X` $<::$ `Comparable<X>. SortedList<X>`.

For an example with multiple bound variables, consider the following class declaration that we will use throughout this paper:

```
class Super<P, Q extends P> {}
```

We denote the traditional existential type for the wildcard type `Super<?,?>` as $\exists$`X,Y:Y` $<::$ `X. Super<X,Y>`.

3. Non-Termination in Subtyping

The major challenge of subtyping wildcards is non-termination. In particular, with wildcards it is possible for a subtyping algorithm to

```
class C implements List<List<? super C>> {}
```

Is `C` a subtype of `List<? super C>`?

```
Step 0)   C <: List<? super C>
Step 1)   List<List<? super C>> <: List<? super C> (inheritance)
Step 2)   C <: List<? super C> (checking wildcard ? super C)
Step ...  (cycle forever)
```

Figure 1. Example of cyclic dependency in subtyping [9]

```
class C<P> implements List<List<? super C<C<P>>>> {}
```

Is `C<Byte>` a subtype of `List<? super C<Byte>>`?

```
Step 0)   C<Byte> <: List<? super C<Byte>>
Step 1)   List<List<? super C<C<Byte>>>> <: List<? super C<Byte>>
Step 2)   C<Byte> <: List<? super C<C<Byte>>>
Step 3)   List<List<? super C<C<Byte>>>> <: List<? super C<C<Byte>>>
Step 4)   C<C<Byte>> <: List<? super C<C<Byte>>>
Step ...  (expand forever)
```

Figure 2. Example of acyclic proofs in subtyping [9]

always be making progress and yet still run forever. This is because there are many sources of infinity in Java's type system, none of which are apparent at first glance. These sources arise from the fact that wildcards are, in terms of existential types, impredicative. That is, `List<?>` is itself a type and can be used in nearly every location a type without wildcards could be used. For example, the type `List<List<?>>` stands for `List<∃X. List<X>>`, whereas it would represent simply `∃X. List<List<X>>` in a predicative system.

We have identified three sources of infinity due to impredicativity of wildcards, in particular due to wildcards in the inheritance hierarchy and in type-parameter constraints. The first source is infinite proofs of subtyping. The second source is wildcard types that represent infinite traditional existential types. The third source is proofs that are finite when expressed using wildcards but infinite using traditional existential types. Here we illustrate each of these challenges for creating a terminating subtyping algorithm.

3.1 Infinite Proofs

Kennedy and Pierce provide excellent simple examples illustrating some basic difficulties with wildcards [9] caused by the fact that wildcards can be used in the inheritance hierarchy. In particular, their examples demonstrate that with wildcards it is quite possible for the question of whether τ is a subtype of τ' to recursively depend on whether τ is a subtype of τ' in a non-trivial way. Consider the program in Figure 1 and the question "Is `C` is subtype of `List<? super C>`?". The bottom part of the figure contains the steps in a potential proof for answering this question. We start with the goal of showing that `C` is a subtype of `List<? super C>`. For this goal to hold, `C`'s superclass `List<List<? super C>>` must be a subtype of `List<? super C>` (Step 1). For this to hold, `List<List<? super C>>` must be `List` of some supertype of `C`, so `C` must be a subtype of `List<? super C>` (Step 2). This was our original question, though, so whether `C` is a subtype of `List<? super C>` actually depends on itself non-trivially. This means that we can actually prove `C` is a subtype of `List<? super C>` *provided* we use an infinite proof.

The proof for Figure 1 repeats itself, but there are even subtyping proofs that expand forever without ever repeating themselves. For example, consider the program in Figure 2, which is a simple modification of the program in Figure 1. The major difference is that `C` now has a type parameter `P` and in the superclass the type argument to `C` is `C<P>` (which could just as easily be `List<P>` or `Set<P>` without affecting the structure of the proof). This is known as expansive inheritance [9, 20] since the type parameter `P` expands to the type argument `C<P>` corresponding to that same type parameter `P`. Because of this expansion, the proof repeats itself every four steps except with an extra `C<->` layer so that the proof is acyclic.

Java rejects all infinite proofs [6: Chapter 4.10], and `javac` attempts to enforce this decision, rejecting the program in Figure 1 but suffering a stack overflow on the program in Figure 2. Thus, neither of the subtypings in Figures 1 and 2 hold according to Java. Although this seems natural, as induction is generally preferred over coinduction, it seems that for Java this is actually an inconsistent choice for reasons we will illustrate in Section 3.3. In our type system, infinite proofs are not even possible, avoiding the need to choose whether to accept or reject infinite proofs. Our simple recursive subtyping algorithm terminates because of this.

3.2 Implicitly Infinite Types

Wehr and Thiemann proved that subtyping of bounded impredicative existential types is undecidable [21]. Wildcards are a restricted form of existential types though, so their proof does not imply that subtyping of wildcards is undecidable. However, we have determined that there are wildcard types that actually cannot be expressed by Wehr and Thiemann's type system. In particular, Wehr and Thiemann use *finite* existential types in which all constraints are explicit, but making all implicit constraints on wildcards explicit can actually result in an *infinite* traditional existential type. That is, wildcard types can implicitly represent infinite traditional existential types.

Consider the following class declaration:

```
class Infinite<P extends Infinite<?>> {}
```

The wildcard type `Infinite<?>` translates to an infinite traditional existential type because its implicit constraints must be made explicit. In one step it translates to `∃X : X <:: Infinite<?>. Infinite<X>`, but then the nested `Infinite<?>` needs to be recursively translated which repeats ad infinitum. Thus `Infinite<?>` is implicitly infinite. Interestingly, this type is actually inhabitable by the following class:

```
class Omega extends Infinite<Omega> {}
```

This means that wildcards are even more challenging than had been believed so far. In fact, a modification like the one for Figure 2 can be applied to get a wildcard type which implicitly represents an acyclically infinite type. Because of implicitly infinite types, one cannot expect structural recursion using implicit constraints to terminate, severely limiting techniques for a terminating subtyping algorithm. Our example illustrating this problem is complex, so we leave it until Section 4.4. Nonetheless, we were able to surmount this challenge by extracting implicit constraints lazily and relying only on finiteness of the explicit type.

3.3 Implicitly Infinite Proofs

Possibly the most interesting aspect of Java's wildcards is that finite proofs of subtyping wildcards can actually express infinite proofs of subtyping traditional existential types. This means that subtyping with wildcards is actually more powerful than traditional systems for subtyping with existential types because traditional systems only permit finite proofs. Like before, implicitly infinite proofs can exist because of implicit constraints on wildcards.

To witness how implicitly infinite proofs arise, consider the programs and proofs in Figure 3. On the left, we provide the program and proof in terms of wildcards. On the right, we provide the translation of that program and proof to traditional existential types. The left proof is finite, whereas the right proof is infinite. The key difference stems from the fact that Java does not check implicit con-

Java Wildcards

`class C extends Super<Super<?,?>,C> {}`

Is `C` a subtype of `Super<?,?>`?

Steps of Proof
C $\lessdot$: Super<?,?>
Super<Super<?,?>,C> $\lessdot$: Super<?,?> (inheritance)
(completes)

Traditional Existential Types

class C extends Super<$\exists$X,Y:Y <:: X. Super<X,Y>,C> {}

Is `C` a subtype of $\exists$X$'$,Y$'$:Y$'$ <:: X$'$. Super<X$'$,Y$'$>?

Steps of Proof
C $\lessdot$: $\exists$X$'$,Y$'$:Y$'$ <:: X$'$. Super<X$'$,Y$'$>
Super<$\exists$X,Y:Y <:: X. Super<X,Y>,C> $\lessdot$: $\exists$X$'$,Y$'$:Y$'$ <:: X$'$. Super<X$'$,Y$'$>
C $\lessdot$: $\exists$X,Y:Y <:: X. Super<X,Y>
(repeats forever)

Figure 3. Example of an implicitly infinite subtyping proof

```
        class C<P extends List<? super C<D>>> implements List<P> {}
Decl. 1) class D implements List<C<?>> {}
Decl. 2) class D implements List<C<? extends List<? super C<D>>>> {}
```

Is `C<D>` a valid type?

Using Declaration 1

D $\lessdot$: List<? super C<D>> (check constraint on P)
List<C<?>> $\lessdot$: List<? super C<D>> (inheritance)
C<D> $\lessdot$: C<?> (check wildcard ? super C<D>)
Accepted (implicitly assumes C<D> is valid)

Using Declaration 2

D $\lessdot$: List<? super C<D>>
List<C<? extends List<? super C<D>>>> $\lessdot$: List<? super C<D>>
C<D> $\lessdot$: C<? extends List<? super C<D>>>
D $\lessdot$: List<? super C<D>>
Rejected (implicit assumption above is explicit here)

Figure 4. Example of the inconsistency of rejecting infinite proofs

straints on wildcards when they are instantiated, whereas these constraints are made explicit in the translation to traditional existential types and so need to be checked, leading to an infinite proof.

To understand why this happens, we need to discuss implicit constraints more. Unlike explicit constraints, implicit constraints on wildcards do not need to be checked after instantiation in order to ensure soundness because those implicit constraints must already hold *provided* the subtype is a valid type (meaning its type arguments satisfy the criteria for the type parameters). However, while determining whether a type is valid Java uses subtyping which implicitly assumes all types involved are valid, potentially leading to an implicitly infinite proof. In Figure 3, `Super<Super<?,?>,C>` is a valid type provided `C` is a subtype of `Super<?,?>`. By inheritance, this reduces to whether `Super<Super<?,?>,C>` is a subtype of `Super<?,?>`. The implicit constraints on the wildcards in `Super<?,?>` are not checked because Java implicitly assumes `Super<Super<?,?>,C>` is a valid type. Thus the proof that `Super<Super<?,?>,C>` is valid implicitly assumes that `Super<Super<?,?>,C>` is valid, which is the source of infinity after translation. This example can be modified similarly to Figure 2 to produce a proof that is implicitly acyclically infinite.

Implicitly infinite proofs are the reason why Java's rejection of infinite proofs is an inconsistent choice. The programs and proofs in Figure 4 are a concrete example illustrating the inconsistency. We provide two declarations for class `D` which differ only in that the constraint on the wildcard is implicit in the first declaration and explicit in the second declaration. Thus, one would expect these two programs to be equivalent in that one would be valid if and only if the other is valid. However, this is not the case in Java because Java rejects infinite proofs. The first program is accepted because the proof that `C<D>` is a valid type is finite. However, the second program is rejected because the proof that `C<D>` is a valid type is infinite. In fact, `javac` accepts the first program but suffers a stack overflow on the second program. Thus Java's choice to reject infinite proofs is inconsistent with its use of implicit constraints. Interestingly, when expressed using traditional existential types, the (infinite) proof for the first program is exactly the same as the (infinite) proof for the second program as one would expect given that they only differ syntactically, affirming that existential types are a suitable formalization of wildcards.

Note that none of the wildcard types in Figures 3 and 4 are implicitly infinite. This means that, even if one were to prevent proofs that are infinite using subtyping rules for wildcards and prevent implicitly infinite wildcard types so that one could translate to traditional existential types, a subtyping algorithm can still always make progress and yet run forever. Our algorithm avoids these problems by not translating to traditional or even finite existential types.

4. Improved Subtyping

Now that we have presented the many non-termination challenges in subtyping wildcards, we present our subtyping rules with a simple sound and complete subtyping algorithm which always terminates even when the wildcard types and proofs involved are implicitly infinite. We impose two simple restrictions, which in Section 9 we demonstrate already hold in existing code bases. With these restrictions, our subtype system has the property that all possible proofs are finite, although they may still translate to infinite proofs using subtyping rules for traditional existential types. Even without restrictions, our algorithm improves on the existing subtyping algorithm since `javac` fails to type check the simple program in Figure 5 that our algorithm determines is valid. Here we provide the core aspects of our subtyping rules and algorithms; the full details can be found in Section A.

4.1 Existential Formalization

We formalize wildcards using existential types. However, we do not use traditional existential types. Our insight is to use a variant that bridges the gap between wildcards, where constraints can be implicit, and traditional existential types, where all constraints must be explicit. We provide the grammar for our existential types, represented by $\overset{\exists}{\tau}$, in Figure 6.

Note that there are two sets of constraints so that we denote our existential types as $\exists\Gamma : \Delta(\Delta).\ \overset{\exists}{\tau}$. The constraints Δ are the constraints corresponding to traditional existential types, combining both the implicit and explicit constraints on wildcards. The constraints Δ are those corresponding to explicit constraints on wildcards, with the parenthetical indicating that only those constraints need to be checked during subtyping.

Our types are a mix of inductive and coinductive structures, meaning finite and infinite. Most components are inductive so that

```
class C<P extends Number> extends ArrayList<P> {}
List<? extends List<? extends Number>> cast(List<C<?>> list)
  {return list;}
```

Figure 5. Example of subtyping incorrectly rejected by `javac`

$$\boxed{\overset{\exists}{\tau}} \qquad \begin{aligned} \overset{\exists}{\tau} &:= v \mid \exists\Gamma : \Delta(\Delta).\ C\langle\overset{\exists}{\tau},\ \ldots,\ \overset{\exists}{\tau}\rangle \\ \Gamma &:= \varnothing \mid \Gamma, v \\ \Delta &:= \varnothing \mid \Delta, v <:: \overset{\exists}{\tau} \mid \Delta, v ::> \overset{\exists}{\tau} \end{aligned}$$

Figure 6. Grammar of our existential types (coinductive)

$$\boxed{\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'}$$

$$\textsc{Sub-Exists}\quad \frac{\begin{array}{c} C\langle P_1,\ \ldots,\ P_m\rangle \text{ is a subclass of } D\langle\bar{\overset{\exists}{\tau}}_1,\ \ldots,\ \bar{\overset{\exists}{\tau}}_n\rangle \\ \mathbb{\Gamma}, \Gamma \xleftarrow{\varnothing} \Gamma' \vdash D\langle\bar{\overset{\exists}{\tau}}_1,\ \ldots,\ \bar{\overset{\exists}{\tau}}_n\rangle[P_1\mapsto\overset{\exists}{\tau}_1, \ldots, P_m\mapsto\overset{\exists}{\tau}_m] \approx_{\theta_i} D\langle\overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle \\ \text{for all } v' \text{ in } \Gamma',\ \text{exists } i \text{ in } 1 \text{ to } n \text{ with } \theta(v') = \theta_i(v') \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \bar{\overset{\exists}{\tau}}_i[P_1\mapsto\overset{\exists}{\tau}_1, \ldots, P_m\mapsto\overset{\exists}{\tau}_m] \cong \overset{\exists}{\tau}'_i[\theta] \\ \text{for all } v <:: \hat{\overset{\exists}{\tau}} \text{ in } \Delta',\ \ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \theta(v) <: \hat{\overset{\exists}{\tau}}[\theta] \\ \text{for all } v ::> \hat{\overset{\exists}{\tau}} \text{ in } \Delta',\ \ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \hat{\overset{\exists}{\tau}}[\theta] <: \theta(v) \end{array}}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash \exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_m\rangle <: \exists\Gamma' : \Delta'(\mathbb{\Delta}').\ D\langle\overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle}$$

$$\textsc{Sub-Var}\quad \frac{}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v <: v} \qquad \frac{v <:: \overset{\exists}{\tau} \text{ in } \mathbb{\Delta} \quad \mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v <: \overset{\exists}{\tau}'} \qquad \frac{v ::> \overset{\exists}{\tau}' \text{ in } \mathbb{\Delta} \quad \mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: v}$$

Figure 7. Subtyping rules for our existential types (inductive and coinductive definitions coincide given restrictions)

we may do structural recursion and still have termination. However, the combined constraints $\mathbb{\Delta}$ are coinductive. This essentially means that they are constructed on demand rather than all ahead of time. This corresponds to only performing wildcard capture when it is absolutely necessary. In this way we can handle wildcards representing implicitly infinite types as in Section 3.2.

4.2 Existential Subtyping

We provide the subtyping rules for our existential types in Figure 7 (for sake of simplicity, throughout this paper we assume all problems with name hiding are taken care of implicitly). The judgement $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'$ means that $\overset{\exists}{\tau}$ is a subtype of $\overset{\exists}{\tau}'$ in the context of type variables $\mathbb{\Gamma}$ with constraints $\mathbb{\Delta}$. The subtyping rules are syntax directed and so are easily adapted into an algorithm. Furthermore, given the restrictions we impose in Section 4.4, the inductive and coinductive definitions coincide, meaning there is no distinction between finite and infinite proofs. From this, we deduce that our algorithm terminates since all proofs are finite.

The bulk of our algorithm lies in SUB-EXISTS, since SUB-VAR just applies assumed constraints on the type variable at hand. The first premise of SUB-EXISTS examines the inheritance hierarchy to determine which, if any, invocations of D that C is a subclass or subinterface of (including reflexivity and transitivity). For Java this invocation is always unique, although this is not necessary for our algorithm. The second and third premises adapt unification to existential types permitting equivalence and including the prevention of escaping type variables. The fourth premise checks that each pair of corresponding type arguments are equivalent for some chosen definition of equivalence such as simple syntactic equality or more powerful definitions as discussed in Sections 7.3 and 8.3. The fifth and sixth premises recursively check that the explicit constraints in the supertype hold after instantiation. Note that only the explicit constraints Δ' in the supertype are checked, whereas the combined implicit and explicit constraints $\mathbb{\Delta}$ in the subtype are assumed. This separation is what enables termination and completeness.

We have no rule indicating that all types are subtypes of `Object`. This is because our existential type system is designed so that such a rule arises as a consequence of other properties. In this case, it arises from the fact `Object` is a superclass of all classes and interfaces in Java and the fact that all variables in Java are implicitly constrained above by `Object`. In general, the separation of implicit and explicit constraints enables our existential type system to adapt to new settings, including settings outside of Java. General reflexivity and transitivity are also consequences of our rules. In

$$\boxed{\mathbb{\Gamma} : \mathbb{\Delta} \vdash \tau <: \tau'}$$

$$\textsc{Sub-Exists}\quad \frac{\begin{array}{c} C\langle P_1,\ \ldots,\ P_m\rangle \text{ is a subclass of } D\langle\bar{\tau}_1,\ \ldots,\ \bar{\tau}_n\rangle \\ \textit{explicit}(\overset{?}{\tau}_1, \ldots, \overset{?}{\tau}_m) = \langle\Gamma; \dot{\Delta}; \tau_1, \ldots, \tau_m\rangle \\ \textit{explicit}(\overset{?}{\tau}'_1, \ldots, \overset{?}{\tau}'_n) = \langle\Gamma'; \dot{\Delta}'; \tau'_1, \ldots, \tau'_n\rangle \\ \textit{implicit}(\Gamma; C; \tau_1, \ldots, \tau_m) = \mathring{\Delta} \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \bar{\tau}_i[P_1\mapsto\tau_1, \ldots, P_m\mapsto\tau_m] = \tau'_i[\theta] \\ \text{for all } v <:: \hat{\tau} \text{ in } \dot{\Delta}',\ \ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathring{\Delta}, \dot{\Delta} \vdash \theta(v) <: \hat{\tau} \\ \text{for all } v ::> \hat{\tau} \text{ in } \dot{\Delta}',\ \ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathring{\Delta}, \dot{\Delta} \vdash \hat{\tau} <: \theta(v) \end{array}}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash C\langle\overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_m\rangle <: D\langle\overset{?}{\tau}'_1,\ \ldots,\ \overset{?}{\tau}'_n\rangle}$$

$$\textsc{Sub-Var}\quad \frac{}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v <: v} \qquad \frac{v <:: \tau \text{ in } \mathbb{\Delta} \quad \mathbb{\Gamma} : \mathbb{\Delta} \vdash \tau <: \tau'}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v <: \tau'} \qquad \frac{v ::> \tau' \text{ in } \mathbb{\Delta} \quad \mathbb{\Gamma} : \mathbb{\Delta} \vdash \tau <: \tau'}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash \tau <: v}$$

Figure 8. Subtyping rules specialized for wildcards

fact, the omission of transitivity is actually a key reason that the inductive and coinductive definitions coincide.

Although we do not need the full generality of our existential types and proofs to handle wildcards, this generality informs which variations of wildcards and existential types would still ensure our algorithm terminates. In Section 10, we will present a few such extensions compatible with our existential types and proofs.

4.3 Wildcard Subtyping

While the existential formalization is useful for understanding and generalizing wildcards, we can specialize the algorithm to wildcards for a more direct solution. We present this specialization of our algorithm in Figure 8, with τ representing a Java type and $\overset{?}{\tau}$ representing a Java type argument which may be a (constrained) wildcard. The function *explicit* takes a list of type arguments that may be (explicitly bound) wildcards, converts wildcards to type variables, and outputs the list of fresh type variables, explicit bounds on those type variables, and the possibly converted type arguments. For example, *explicit*(`Numbers<? super Integer>`) returns $\langle$`X`; `X ::> Integer`; `Numbers<X>`$\rangle$. The function *implicit* takes a list of constrainable type variables, a class or interface name C, and a list of type arguments, and outputs the constraints on those type arguments that are constrainable type variables as prescribed by the

```
class C<P extends List<List<? extends List<? super C<?>>>>>
  implements List<P> {}
```

Is `C<?>` a subtype of `List<? extends List<? super C<?>>>`?

Steps of Proof

$\texttt{C<?>} \mathrel{\triangleleft^{?}:} \texttt{List<? extends List<? super C<?>>>}$

  $\texttt{C<?>} \mapsto \texttt{C<X>}$ with $\texttt{X} <:: \texttt{List<List<? extends List<? super C<?>>>>}$

$\texttt{C<X>} \mathrel{\triangleleft^{?}:} \texttt{List<? extends List<? super C<?>>>}$

$\texttt{X} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<?>>}$

$\texttt{List<List<? extends List<? super C<?>>>>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<?>>}$

$\texttt{C<?>} \mathrel{\triangleleft^{?}:} \texttt{List<? extends List<? super C<?>>>}$

(repeats forever)

Figure 9. Example of infinite proof due to implicitly infinite types

```
class C<P, Q extends P> implements List<List<? super C<List<Q>,?>>> {}
```

Is `C<?,?>` a subtype of `List<? super C<?,?>>`?

Constraints	Subtyping (wildcard capture done automatically)
$\texttt{X}_1 <:: \texttt{X}_0$	$\texttt{C<X}_0\texttt{,X}_1\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<?,?>>}$
$\texttt{Y}_1 <:: \texttt{Y}_0$	$\texttt{C<Y}_0\texttt{,Y}_1\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<List<X}_1\texttt{>,?>>}$
$\texttt{X}_2 <:: \texttt{List<X}_1\texttt{>}$	$\texttt{C<List<X}_1\texttt{>,X}_2\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<List<Y}_1\texttt{>,?>>}$
$\texttt{Y}_2 <:: \texttt{List<Y}_1\texttt{>}$	$\texttt{C<List<Y}_1\texttt{>,Y}_2\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<List<X}_2\texttt{>,?>>}$
$\texttt{X}_3 <:: \texttt{List<X}_2\texttt{>}$	$\texttt{C<List<X}_2\texttt{>,X}_3\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<List<Y}_2\texttt{>,?>>}$
$\texttt{Y}_3 <:: \texttt{List<Y}_2\texttt{>}$	$\texttt{C<List<Y}_2\texttt{>,Y}_3\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<List<X}_3\texttt{>,?>>}$
$\texttt{X}_4 <:: \texttt{List<X}_3\texttt{>}$	$\texttt{C<List<X}_3\texttt{>,X}_4\texttt{>} \mathrel{\triangleleft^{?}:} \texttt{List<? super C<List<Y}_3\texttt{>,?>>}$
	(continue forever)

Figure 10. Example of expansion through implicit constraints

requirements of the corresponding type parameters of C, constraining a type variable by `Object` if there are no other constraints. For example, *implicit*(`X`; `Numbers`; `X`) returns `X` $<::$ `Number`. Thus, applying *explicit* and then *implicit* accomplishes wildcard capture. Note that for the most part $\mathbb{\Delta}$ and Δ combined act as Δ does in Figure 7.

4.4 Termination

Unfortunately, our algorithm does not terminate without imposing restrictions on the Java language. Fortunately, the restrictions we impose are simple, as well as practical as we will demonstrate in Section 9. Our first restriction is on the inheritance hierarchy.

Inheritance Restriction

For every declaration of a direct superclass or superinterface τ of a class or interface, the syntax `? super` *must not occur within τ.*

Note that the programs leading to infinite proofs in Section 3.1 (and in the upcoming Section 4.5) violate our inheritance restriction. This restriction is most similar to a significant relaxation of the contravariance restriction that Kennedy and Pierce showed enables decidable subtyping for declaration-site variance [9]. Their restriction prohibits contravariance altogether, whereas we only restrict its usage. Furthermore, as Kennedy and Pierce mention [9], wildcards are a more expressive domain than declaration-site variance. We will discuss these connections more in Section 10.1.

Constraints on type parameters also pose problems for termination. The constraint context can simulate inheritance, so by constraining a type parameter `P` to extend `List<List<? super P>>` we encounter the same problem as in Figure 1 but this time expressed in terms of type-parameter constraints. Constraints can also produce implicitly infinite types that enable infinite proofs even when our inheritance restriction is satisfied, such as in Figure 9 (which again causes `javac` to suffer a stack overflow). To prevent problematic forms of constraint contexts and implicitly infinite types, we restrict the constraints that can be placed on type parameters.

Parameter Restriction

For every parameterization `<`P_1 `extends` τ_1`, ...,` P_n `extends` τ_n`>`, *every syntactic occurrence in τ_i of a type* C`<..., ? super` τ`, ...>` *must be at a covariant location in τ_i.*

Note that our parameter restriction still allows type parameters to be constrained to extend types such as `Comparable<? super P>`, a well known design pattern. Also note that the inheritance restriction is actually the conjunction of the parameter restriction and Java's restriction that no direct superclass or superinterface may have a wildcard as a type argument [6: Chapters 8.1.4 and 8.1.5].

With these restrictions we can finally state our key theorem.

Subtyping Theorem. *Given the inheritance and parameter restrictions, the algorithm prescribed by the rules in Figure 8 always terminates. Furthermore it is a sound and complete implementation of the subtyping rules in the Java language specification [6: Chapter 4.10.2] provided all types are valid according to the Java language specification [6: Chapter 4.5].*[‡]

Proof. Here we only discuss the reasons for our restrictions; the full proofs can be found in Section B. The first thing to notice is that, for the most part, the supertype shrinks through the recursive calls. There are only two ways in which it can grow: applying a lower-bound constraint on a type variable via SUB-VAR, and checking an explicit lower bound on a wildcard via SUB-EXISTS. The former does not cause problems because of the limited ways a type variable can get a lower bound. The latter is the key challenge because it essentially swaps the subtype and supertype which, if unrestricted, can cause non-termination. However, we determined that there are only two ways to increase the number of swaps that can happen: inheritance, and constraints on type variables. Our inheritance and parameter restrictions prevent this, capping the number of swaps that can happen from the beginning and guaranteeing termination. □

4.5 Expansive Inheritance

Smith and Cartwright conjectured that prohibiting expansive inheritance as defined by Kennedy and Pierce [9] would provide a sound and complete subtyping algorithm [14]. This is because Kennedy and Pierce built off the work by Viroli [20] to prove that, by prohibiting expansive inheritance, any infinite proof of subtyping in their setting would have to repeat itself; thus a sound and complete algorithm could be defined by detecting repetitions.

Unfortunately, we have determined that prohibiting expansive inheritance as defined by Kennedy and Pierce does not imply that all infinite proofs repeat. Thus, their algorithm adapted to wildcards does not terminate. The problem is that implicit constraints can cause an indirect form of expansion that is unaccounted for.

Consider the class declaration in Figure 10. According to the definition by Kennedy and Pierce [9], this is not expansive inheritance since `List<Q>` is the type argument corresponding to `P` rather than to `Q`. However, the proof in Figure 10 never repeats itself. The key observation to make is that the context, which would be fixed in Kennedy and Pierce's setting, is continually expanding in this setting. In the last step we display, the second type argument of `C` is a subtype of `List<? extends List<? extends List<?>>>`, which will keep growing as the proof continues. Thus Smith and Cartwright's conjecture for a terminating subtyping algorithm does not hold. In our technical report we identify syntactic restrictions that would be necessary (although possibly still not sufficient) to adapt Kennedy

[‡] See Section 7.4 for a clarification on type validity.

```
class Var {
  boolean mValue;
  void addTo(List<? super Var> trues, List<? super Var> falses)
    {(mValue ? trues : falses).add(this);}
}
```

Figure 11. Example of valid code erroneously rejected by `javac`

and Pierce's algorithm to wildcards [16]. However, these restrictions are significantly more complex than ours, and the adapted algorithm would be strictly more complex than ours, so we do not expect this to be a practical alternative.

5. Challenges of Type-Argument Inference

So far we have discussed only one major challenge of wildcards, subtyping, and our solution to this challenge. Now we present another major challenge of wildcards, inference of type arguments for generic methods, with our techniques to follow in Section 6.

5.1 Joins

Java has the expression `cond ? t : f` which evaluates to `t` if `cond` evaluates to `true`, and to `f` otherwise. In order to determine the type of this expression, it is useful to be able to combine the types determined for `t` and `f` using a $join(\tau, \tau')$ function which returns the most precise common supertype of τ and τ'. Unfortunately, not all pairs of types with wildcards have a join (even if we allow intersection types). For example, consider the types `List<String>` and `List<Integer>`, where `String` implements `Comparable<String>` and `Integer` implements `Comparable<Integer>`. Both `List<String>` and `List<Integer>` are a `List` of something, call it `X`, and in both cases that `X` is a subtype of `Comparable<X>`. So while both `List<String>` and `List<Integer>` are subtypes of simply `List<?>`, they are also subtypes of `List<? extends Comparable<?>>` and of `List<? extends Comparable<? extends Comparable<?>>>` and so on. Thus their join using only wildcards is the undesirable infinite type `List<? extends Comparable<? extends Comparable<? extends ...>>>`.

`javac` addresses this by using an algorithm for finding *some* common supertype of τ and τ' which is not necessarily the most precise. This strategy is incomplete, as we even saw in the classroom when it failed to type check the code in Figure 11. This simple program fails to type check because `javac` determines that the type of `(mValue ? trues : falses)` is `List<?>` rather than the obvious `List<? super Var>`. In particular, `javac`'s algorithm may even fail to return τ when both arguments are the same type τ.

Smith and Cartwright take a different approach to joining types. They extend the type system with union types [14]. That is, the join of `List<String>` and `List<Integer>` is just `List<String> | List<Integer>` in their system. $\tau \mid \tau'$ is defined to be a supertype of both τ and τ' and a subtype of all common supertypes of τ and τ'. Thus, it is by definition the join of τ and τ' in their extended type system. This works for the code in Figure 11, but in Section 5.2 we will demonstrate the limitations of this solution.

Another direction would be to find a form of existential types beyond wildcards for which joins always exist. For example, using traditional existential types the join of `List<String>` and `List<Integer>` is just $\exists$`X : X` $<::$ `Comparable<X>. List<X>`. However, our investigations suggest that it may be impossible for an existential type system to have both joins and decidable subtyping while being expressive enough to handle common Java code. Therefore, our solution will differ from all of the above.

```
<P> P getFirst(List<P> list) {return list.get(0);}
Number getFirstNumber(List<? extends Number> nums)
  {return getFirst(nums);}
Object getFirstNonEmpty(List<String> strs, List<Object> obs)
  {return getFirst(!strs.isEmpty() ? strs : obs);}
Object getFirstNonEmpty2(List<String> strs, List<Integer> ints)
  {return getFirst(!strs.isEmpty() ? strs : ints);}
```

Figure 12. Examples of capture conversion

5.2 Capture Conversion

Java has generic methods as well as generic classes [6: Chapter 8.4.4]. For example, the method `getFirst` in Figure 12 is generic with respect to `P`. Java attempts to infer type arguments for invocations of generic methods [6: Chapter 15.12.2.7], hence the uses of `getFirst` inside the various methods in Figure 12 do not need to be annotated with the appropriate instantiation of `P`. Interestingly, this enables Java to infer type arguments that cannot be expressed by the user. Consider `getFirstNumber`. This method is accepted by `javac`; `P` is instantiated to the type variable for the wildcard `? extends Number`, an instantiation of `P` that the programmer cannot explicitly annotate because the programmer cannot explicitly name the wildcard. Thus, Java is implicitly opening the existential type `List<? extends Number>` to `List<X>` with `X` $<::$ `Number` and then instantiating `P` as `X` so that the return type is `X` which is a subtype of `Number`. This ability to implicitly capture wildcards, known as capture conversion [6: Chapter 5.1.10], is important to working with wildcards but means type inference has to determine when to open a wildcard type.

Smith and Cartwright developed an algorithm for type-argument inference intended to improve upon `javac` [14]. Before going into their algorithm and showing some of its limitations, let us first go back to Figure 11. Notice that the example there, although originally presented as a join example, can be thought of as an inference example by considering the `? :` operator to be like a generic method. In fact, Smith and Cartwright have already shown that type-argument inference inherently requires finding common supertypes of two types [14], a process that is often performed using joins. Thus the ability to join types is closely intertwined with the ability to do type-argument inference. Smith and Cartwright's approach for type-argument inference is based on their union types, which we explained in Section 5.1. Their approach to type inference would succeed on the example from Figure 11, because they use a union type, whereas `javac` incorrectly rejects that program.

Although Smith and Cartwright's approach to type-argument inference improves on Java's approach, their approach is not strictly better than Java's. Consider the method `getFirstNonEmpty` in Figure 12. `javac` accepts `getFirstNonEmpty`, combining `List<String>` and `List<Object>` into `List<?>` and then instantiating `P` to the captured wildcard. Smith and Cartwright's technique, on the other hand, fails to type check `getFirstNonEmpty`. They combine `List<String>` and `List<Object>` into `List<String> | List<Object>`. However, there is no instantiation of `P` so that `List<P>` is a supertype of the union type `List<String> | List<Object>`, so they reject the code. What their technique fails to incorporate in this situation is the capture conversion permitted by Java. For the same reason, they also fail to accept `getFirstNonEmpty2`, although `javac` also fails on this program for reasons that are unclear given the error message. The approach we will present is able to type check all of these examples.

5.3 Ambiguous Types and Semantics

In Java, the type of an expression can affect the semantics of the program, primarily due to various forms of overloading. This is particularly problematic when combining wildcards and type-argument inference. Consider the program in Figure 13. Notice

```
<P> List<P> singleton(P elem) {return null;}
<Q extends Comparable<?>> Q foo(List<? super Q> list) {return null;}
String typeName(Comparable<?> c) {return "Comparable";}
String typeName(String s) {return "String";}
String typeName(Integer i) {return "Integer";}
String typeName(Calendar c) {return "Calendar";}
boolean ignore() {...};
String ambiguous() {
  return typeName(foo(singleton(ignore() ? "Blah" : 1)));
}
```

Figure 13. Example of ambiguous typing affecting semantics

that the value returned by `ambiguous` depends solely on the type of the argument to `typeName`, which is the return type of `foo` which depends on the inferred type arguments for the generic methods `foo` and `singleton`. Using `javac`'s typing algorithms, `ambiguous` returns `"Comparable"`. Using Smith and Cartwright's typing algorithms [14], `ambiguous` returns either `"String"` or `"Integer"` depending on how the types are (arbitrarily) ordered internally. In fact, the answers provided by `javac` and by Smith and Cartwright are not the only possible answers. One could just as well instantiate `P` to `Object` and `Q` to `Calendar` to get `ambiguous` to return `"Calendar"`, even though a `Calendar` instance is not even present in the method.

The above discussion shows that, in fact, *all four* values are plausible, and which is returned depends on the results of type-argument inference. Unfortunately, the Java specification does not provide clear guidance on what should be done if there are multiple valid type arguments. It does however state the following [6: Chapter 15.12.2.7]: "The type-inference algorithm should be viewed as a heuristic, designed to perform well in practice." This would lead one to believe that, given multiple valid type arguments, an implementation can heuristically pick amongst them, which would actually make any of the four returned values a correct implementation of `ambiguous`. This is not only surprising, but also leads to the unfortunate situation that by providing `javac` with smarter static typing algorithms one may actually change the semantics of existing programs. This in turn makes improving the typing algorithms in existing implementations a risky proposition.

6. Improving Type-Argument Inference

Here we present an algorithm for joining wildcards as existential types which addresses the limitations of union types and which is complete provided the construction is used in restricted settings. We also describe preliminary techniques for preventing ambiguity due to type-argument inference as discussed in Section 5.3.

6.1 Lazily Joining Wildcards

As we mentioned in Section 5.1, it seems unlikely that there is an existential type system for wildcards with both joins and decidable subtyping. Fortunately, we have determined a way to extend our type system with a *lazy* existential type that solves many of our problems. Given a potential constraint on the variables bound in a lazy existential type we can determine whether that constraint holds. However, we cannot enumerate the constraints on the variables bound in a lazy existential type, so lazy existential types must be used in a restricted manner. In particular, for any use of $\tau <: \tau'$, lazy existential types may only be used in covariant locations in τ and contravariant locations in τ'. Maintaining this invariant means that τ' will never be a lazy existential type. This is important because applying SUB-EXISTS requires checking all of the constraints of τ', but we have no means of enumerating these constraints for a lazy existential type. Fortunately, `cond ? t : f` as well as unambiguous type-argument inference only need a join for covariant locations of the return type, satisfying our requirement.

So suppose we want to construct the join ($\sqcup$) of captured wildcard types $\exists\Gamma:\Delta.\ C$<τ_1, …, τ_m> and $\exists\Gamma':\Delta'.\ D$<$\tau'_1$, …, τ'_n>. Let $\{E_i\}_{i \text{ in } 1 \text{ to } k}$ be the set of minimal raw superclasses and superinterfaces common to C and D. Let each E_i<$\bar{\tau}^i_1$, …, $\bar{\tau}^i_{\ell_i}$> be the superclass of C<P_1, …, P_m>, and each E_i<$\hat{\tau}^i_1$, …, $\hat{\tau}^i_{\ell_i}$> the superclass of D<P'_1, …, P'_n>. Compute the anti-unification [12, 13] of all $\bar{\tau}^i_j[P_1 \mapsto \tau_1, \ldots, P_m \mapsto \tau_m]$ with all $\hat{\tau}^i_j[P'_1 \mapsto \tau'_1, \ldots, P'_n \mapsto \tau'_n]$, resulting in $\overset{\sqcup}{\tau}{}^i_j$ with fresh variables $\Gamma_\sqcup$ and assignments θ and θ' such that each $\overset{\sqcup}{\tau}{}^i_j[\theta]$ equals $\bar{\tau}^i_j[P_1 \mapsto \tau_1, \ldots, P_m \mapsto \tau_m]$ and each $\overset{\sqcup}{\tau}{}^i_j[\theta']$ equals $\hat{\tau}^i_j[P'_1 \mapsto \tau'_1, \ldots, P'_n \mapsto \tau'_n]$. For example, the anti-unification of the types `Map<String,String>` and `Map<Integer,Integer>` is `Map<`v`,`v`>` with assignments $v \mapsto$ `String` and $v \mapsto$ `Integer`. The join, then, is the lazy existential type

$$\begin{array}{l}\exists\Gamma_\sqcup : \langle \theta \mapsto \Gamma : \Delta; \theta' \mapsto \Gamma' : \Delta' \rangle. \\ \quad E_1\texttt{<}\overset{\sqcup}{\tau}{}^1_1\texttt{,}\ \ldots\texttt{,}\ \overset{\sqcup}{\tau}{}^1_{\ell_1}\texttt{>}\ \texttt{\&}\ \ldots\ \texttt{\&}\ E_k\texttt{<}\overset{\sqcup}{\tau}{}^k_1\texttt{,}\ \ldots\texttt{,}\ \overset{\sqcup}{\tau}{}^k_{\ell_k}\texttt{>}\end{array}$$

The lazy constraint $\langle \theta \mapsto \Gamma : \Delta; \theta' \mapsto \Gamma' : \Delta' \rangle$ indicates that the constraints on $\Gamma_\sqcup$ are the constraints that hold in context $\Gamma : \Delta$ after substituting with θ and in context $\Gamma' : \Delta'$ after substituting with θ'. Thus the total set of constraints is not computed, but there is a way to determine whether a constraint is in this set. Note that this is the join because Java ensures the $\bar{\tau}$ and $\hat{\tau}$ types will be unique.

Capture conversion can be applied to a lazy existential type, addressing the key limitation of union types that we identified in Section 5.2. The lazy constraint $\langle \theta \mapsto \Gamma : \Delta; \theta' \mapsto \Gamma' : \Delta' \rangle$ is simply added to the context. The same is done when SUB-EXISTS applies with a lazy existential type as the subtype. When SUB-VAR applies for $v <: \tau'$ with v constrained by a lazy constraint rather than standard constraints, one checks that both $\theta(v) <: \tau'[\theta]$ holds and $\theta'(v) <: \tau'[\theta']$ holds, applying the substitutions to relevant constraints in the context as well. A similar adaptation is also made for $\tau <: v$. This extended algorithm is still guaranteed to terminate.

With this technique, we can type check the code in Figure 11 that `javac` incorrectly rejects as well as the code in Figure 12 including the methods that Smith and Cartwright's algorithm incorrectly rejects. For example, for `getFirstNonEmpty2` we would first join `List<String>` and `List<Integer>` as the lazy existential type

$$\exists\texttt{X} : \langle\{\texttt{X} \mapsto \texttt{String}\} \mapsto \varnothing : \varnothing; \{\texttt{X} \mapsto \texttt{Integer}\} \mapsto \varnothing : \varnothing\rangle.\ \texttt{List<X>}$$

This type would then be capture converted so that the type parameter `P` of `getFirst` would be instantiated with the lazily constrained type variable `X`. Although not necessary here, we would also be able to determine that the constraint `X` <:: `Comparable<X>` holds for the lazily constrained type variable.

Occasionally one has to join a type with a type variable. For this purpose, we introduce a specialization of union types. This specialization looks like $\tau_\sqcup(v_1 \mid \ldots \mid v_n)$ or $\tau_\sqcup(\tau \mid v_1 \mid \ldots \mid v_n)$ where each v_i is not lazily constrained and $\tau_\sqcup$ is a supertype of some wildcard capture of each upper bound of each type variable (and of τ if present) with the property that any other non-variable τ' which is a supertype of each v_i (and τ) is also a supertype of $\tau_\sqcup$. A type τ' is a supertype of this specialized union type if it is a supertype of $\tau_\sqcup$ or of each v_i (and τ). Note that $\tau_\sqcup$ might not be a supertype of any v_i or of τ and may instead be the join of the upper bounds of each v_i (plus τ) *after* opening the lazy existential type. This subtlety enables capture conversion to be applied unambiguously when called for. Unfortunately, we cannot join a type with a type variable that is lazily constrained because we cannot enumerate its upper bounds.

6.2 Inferring Unambiguous Types

We believe that the Java language specification should be changed to prevent type-argument inference from introducing ambiguity into the semantics of programs. Since the inferred return type is what determines the semantics, one way to prevent ambiguity would be to permit type-argument inference only when a most precise return type can be inferred, meaning the inferred return type is a subtype of all other return types that could arise from valid type arguments for the invocation at hand. Here we discuss how such a goal affects the design of type-argument inference. However, we do not present an actual algorithm since the techniques we present need to be built upon further to produce an algorithm which prevents ambiguity but is also powerful enough to be practical.

Typical inference algorithms work by collecting a set of constraints and then attempting to determine a solution to those constraints. If those constraints are not guaranteed to be sufficient, then any solution is verified to be a correct typing of the expression (in this case the generic-method invocation). Both `javac` [6: Chapters 15.12.2.7 and 15.12.2.8] and Smith and Cartwright [6] use this approach. Smith and Cartwright actually collect a set of sets of constraints, with each set of constraints guaranteed to be sufficient.

However, to prevent ambiguity due to type-argument inference, necessity of constraints is important rather than sufficiency. For the ambiguous program in Figure 13, each of the solutions we described in Section 5.3 was sufficient; however, none of them were necessary, which was the source of ambiguity. Unfortunately, Smith and Cartwright's algorithm is specialized to find sufficient rather than necessary sets of constraints. This is why their algorithm results in two separate solutions for Figure 13. However, their algorithm could be altered to sacrifice sufficiency for sake of necessity by producing a less precise but necessary constraint at each point where they would currently introduce a disjunction of constraints, which actually simplifies the algorithm since it no longer has to propagate disjunctions.

After a necessary set of constraints has been determined, one then needs to determine a solution. Some constraints will suggest that it is necessary for a type argument to be a specific type, in which case one just checks that the specific type satisfies the other constraints on that type argument. However, other type arguments will only be constrained above and/or below by other types so that there can be many types satisfying the constraints. In order to prevent ambiguity, one cannot simply choose solutions for these type arguments arbitrarily. For example, if the parameterized return type of the method is covariant (and not bivariant) with respect to a type parameter, then the solution for the corresponding type argument must be the join of all its lower bounds, ensuring the inferred return type is the most precise possible. Fortunately, since such joins would occur covariantly in the return type, it is safe to use the construction described in Section 6.1.

Unfortunately, requiring the inferred return type to be the most precise possible seems too restrictive to be practical. Consider the `singleton` method in Figure 13. Under this restriction, type-argument inference would never be permitted for any invocation of `singleton` (without an expected return type) even though the inferred types of most such invocations would not affect the semantics of the program. In light of this, we believe the unambiguous-inference challenge should be addressed by combining the above techniques with an ability to determine when choices can actually affect the semantics of the program. We have had promising findings on this front, but more thorough proofs and evaluations need to be done, so we leave this to future work.

6.3 Removing Intermediate Types

The processes above introduce new kinds of types, namely lazy existential types. Ideally these types need not be a part of the actual type system but rather just be an algorithmic intermediary. Fortunately this is the case for lazy existential types. By examining how the lazy existential type is used while type checking the rest of the program, one can determine how to replace it with an existential type which may be less precise but with which the program will still type check. This is done by tracking the pairs $v <: \tau'$ and $\tau <: v$, where v is lazily constrained, that are checked and found to hold using the modified SUB-VAR rules. After type checking has completed, the lazy existential type can be replaced by an existential type using only the tracked constraints (or slight variations thereof to prevent escaping variables). Proof-tracking techniques can also be used to eliminate intersection types, important for addressing the non-determinism issues we will discuss in Section 7.2, as well as our specialized union types.

7. Challenges of Type Checking

Wildcards pose difficulties for type checking in addition to the subtyping and inference challenges we have discussed so far. Here we identify undesirable aspects of Java's type system caused by these difficulties, and in Section 8 we present simple changes to create an improved type system.

7.1 Inferring Implicit Constraints

Java ensures that all types use type arguments satisfying the criteria of the corresponding type parameters. Without wildcards, enforcing this requirement on type arguments is fairly straightforward. Wildcards, however, complicate matters significantly because there may be a way to implicitly constrain wildcards so that the type arguments satisfy their requirements. For example, consider the following interface declaration:

```
interface SubList<P extends List<? extends Q>, Q> {}
```

Java accepts the type `SubList<?,Number>` because the wildcard can be implicitly constrained to be a subtype of `List<? extends Number>` with which the requirements of the type parameters are satisfied. However, Java rejects the type `SubList<List<Integer>,?>` even though the wildcard can be implicitly constrained to be a supertype of `Integer` with which the requirements of the type parameters are satisfied (we formalize design requirements for implicit constraints in Section C.2). Thus, Java's implicit-constraint inference is incomplete and as a consequence types that could be valid are nonetheless rejected by Java.

This raises the possibility of extending Java to use complete implicit-constraint inference (assuming the problem is decidable). However, we have determined that this would cause significant algorithmic problems. In particular, complete implicit-constraint inference would enable users to express types that have an implicitly infinite body rather than just implicitly infinite constraints. Consider the following class declaration:

```
class C<P extends List<Q>, Q> extends List<C<C<?,?>,?>> {}
```

For the type `C<C<?,?>,?>` to be valid, `C<?,?>` must be a subtype of `List<X>` where `X` is the last wildcard of `C<C<?,?>,?>`. Since `C<?,?>` is a subtype of `List<C<C<?,?>,?>>`, this implies `X` must be equal to `C<C<?,?>,?>`, and with this implicit constraint the type arguments satisfy the requirements of the corresponding type parameters. Now, if we expand the implicit equality on the last wildcard in `C<C<?,?>,?>` we get the type `C<C<?,?>,C<C<?,?>,?>>`, which in turn contains the type `C<C<?,?>,?>` so that we can continually expand to get the infinite type `C<C<?,?>,C<C<?,?>,...>>`. As one might suspect, infinite types of this form cause non-termination problems for many algorithms.

Another problem from a user's perspective is that the reasons for when a type is valid or invalid using complete implicit-constraint inference can be rather complex. To illustrate this, consider the following interface declaration:

```
interface MegaList<P extends List<? extends P>> extends List<P> {}
```

The type `SubList<MegaList<?>,?>` would be valid if we allowed infinite types or existential types with explicit recursive constraints, both of which make non-termination even more of a problem. In particular, the implicit constraint on the second wildcard would be that it is a supertype of either the infinite type `List<? extends List<? extends ...>>` or the existential type $\exists$`X`:`X` $<::$ `List<? extends X>`. `List<X>`. If the parameter `P` for `MegaList` were constrained by `List<? extends Number>` instead, then the implicit lower bound could simply be `Number`. On the other hand, if the parameter `P` for `SubList` were constrained by `List<P>` instead, then no implicit constraint is possible. Thus slight changes to class declarations or to the type system can affect whether a type is valid often for unintuitive reasons, making the type system fragile and reducing changeability of the code base.

In light of these observations, we will propose using implicit-constraint inference slightly stronger than Java's in order to address a slight asymmetry in Java's algorithm while still being user friendly as well as compatible with all algorithms in this paper.

7.2 Non-Deterministic Type Checking

The type checker in `javac` is currently non-deterministic from the user's perspective. Consider the following interface declaration:

```
interface Maps<P extends Map<?,String>> extends List<P> {}
```

`javac` allows one to declare a program variable `m` to have type `Maps<? extends Map<String,?>>`. The type of `m`, then, has a wildcard which is constrained to be a subtype of both `Map<?,String>` and `Map<String,?>`. This means that `m.get(0).entrySet()` has two types, essentially $\exists$`X. Set<Entry<X,String>>` and $\exists$`Y. Set<Entry<String,Y>>`, neither of which is a subtype of the other. However, the type-checking algorithm for `javac` is designed under the assumption that this will never happen, and as such `javac` only checks whether one of the two options is sufficient for type checking the rest of the program, which is the source of non-determinism.

`javac` makes this assumption because Java imposes single-instantiation inheritance, meaning a class (or interface) can extend C`<`τ_1`, ..., `τ_n`>` and C`<`τ'_1`, ..., `τ'_n`>` only if each τ_i equals τ'_i [6: Chapter 8.1.5] (in other words, prohibiting multiple-instantiation inheritance [9]). However, it is not clear what single-instantiation inheritance should mean in the presence of wildcards. The Java language specification is ambiguous in this regard [6: Chapter 4.4], and `javac`'s enforcement is too weak for the assumptions made by its algorithms, as demonstrated above.

Thus, we need to reconsider single-instantiation inheritance in detail with wildcards in mind. There are two ways to address this: restrict types in some way, or infer from two constraints a stronger constraint that is consistent with single-instantiation inheritance. We consider the latter first since it is the more expressive option.

Knowing that the wildcard in `m`'s type above is a subtype of both `Map<?,String>` and `Map<String,?>`, single-instantiation inheritance suggests that the wildcard is actually a subtype of `Map<String,String>`. With this more precise constraint, we can determine that the type of `m.get(0).entrySet()` is `Set<Entry<String,String>>`, which is a subtype of the two alternatives mentioned earlier. For this strategy to work, given two upper bounds on a wildcard we have to be able to determine their meet: the most general common subtype consistent with single-instantiation inheritance. Interestingly, the meet of two types may not be expressible by the user. For example, the meet of `List<?>` and `Set<?>` is $\exists$`X. List<X> & Set<X>`.

Unfortunately, meets encounter many of the same problems of complete implicit-constraint inference that we discussed in Section 7.1. Assuming meets can always be computed, predicting when two types have a meet can be quite challenging. Furthermore, meets pose algorithmic challenges, such as for equivalence checking since with them `Maps<? extends Map<String,?>>` is equivalent to `Maps<? extends Map<String,String>>` even though neither explicit constraint is redundant.

This problem is not specific to combining implicit and explicit constraints on wildcards. Java allows type parameters to be constrained by intersections of types: `<P extends `τ_1` & ... & `τ_n`>`. Although Java imposes restrictions on these intersections [6: Chapter 4.4], when wildcards are involved the same problems arise as with combining implicit and explicit constraints. So, while `javac` rejects the intersection `Map<?,String> & Map<String,?>`, `javac` does permit the intersection `Numbers<?> & Errors<?>`. Should `P` be constrained by this intersection, then due to the implicit constraints on the wildcards `P` is a subtype of both `List<? extends Number>` and `List<? extends Error>`, which once again introduces non-determinism into `javac`'s type checking.

As a more severe alternative, one might consider throwing out single-instantiation inheritance altogether and redesigning the type checker for multiple-instantiation inheritance, especially if Java decided to also throw out type erasure. However, multiple-instantiation inheritance in the presence of wildcards can actually lead to ambiguity in program semantics. Suppose an object has an implementation of both `List<String>` and `List<Integer>`. That object can be passed as a `List<?>`, but which `List` implementation is passed depends on whether the wildcard was instantiated with `String` or `Integer`. Thus an invocation of `get(0)` to get an `Object` from the `List<?>` (which is valid since the wildcard implicitly extends `Object`) would return different results depending on the subtyping proof that was constructed (non-deterministically from the user's perspective). Thus a language with wildcards would either need to use single-instantiation inheritance or statically determine when subtyping can ambiguously affect semantics.

After careful consideration, our solution will be to restrict intersections and explicit constraints on wildcards so that they are consistent with single-instantiation inheritance adapted to wildcards.

```
class C implements List<D<? extends List<D<? extends C>>>> {}
class D<P extends C> {}
```

Is `D<? extends List<D<? extends C>>>`
equivalent to `D<? extends C>`**?**

Key Steps of Proof

`D<? extends List<D<? extends C>>>` $\overset{?}{\cong}$ `D<? extends C>`
(Checking $:>$) `D<? extends C>` $\lessdot:$ `D<? extends List<D<? extends C>>>`
`C` $\lessdot:$ `List<D<? extends C>>`
`List<D<? extends List<D<? extends C>>>>` $\lessdot:$ `List<D<? extends C>>`
`D<? extends List<D<? extends C>>>` $\overset{?}{\cong}$ `D<? extends C>`
(repeat forever)

Figure 14. Example of infinite proofs due to equivalence

7.3 Equivalence

`Numbers<?>` is equivalent to `Numbers<? extends Number>` because of the implicit constraint on the wildcard. As such, one would expect `List<Numbers<?>>` to be equivalent to `List<Numbers<? extends Number>>`. However, this is not the case according to the Java language specification [6: Chapters 4.5.1.1 and 4.10.2] and the formalization by Torgersen et al. [18] referenced therein (although `javac` makes some attempts to support this). The reason is that Java uses syntactic equality when comparing type arguments, reflected in our SUB-EXISTS rule by the use of $=$ in the sixth premise.

Ideally equivalent types could be used interchangeably. Thus, during subtyping Java should only require type arguments to be equivalent rather than strictly syntactically identical. The obvious way to implement this is to simply check that the type ar-

$$\dfrac{\begin{array}{c}\mathit{explicit}(\overset{?}{\tau}_1,\ldots,\overset{?}{\tau}_m) = \langle\Gamma;\Delta;\tau_1,\ldots,\tau_m\rangle \qquad \mathit{explicit}(\overset{?}{\tau}{}'_1,\ldots,\overset{?}{\tau}{}'_n) = \langle\Gamma';\Delta';\tau'_1,\ldots,\tau'_n\rangle \\ \text{for all}\left\{\begin{array}{c} C\langle P_1,\ldots,P_m\rangle \text{ is a subclass of } E\langle\bar{\tau}_1,\ldots,\bar{\tau}_k\rangle \\ \text{and} \\ D\langle P'_1,\ldots,P'_n\rangle \text{ is a subclass of } E\langle\bar{\tau}'_1,\ldots,\bar{\tau}'_k\rangle \end{array}\right\} \text{and } i \text{ in } 1 \text{ to } k,\ \bar{\tau}_i[P_1\mapsto\tau_1,\ldots,P_m\mapsto\tau_m] = \bar{\tau}'_i[P'_1\mapsto\tau'_1,\ldots,P'_n\mapsto\tau'_n]\end{array}}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash C\langle\overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_m\rangle \sqcup D\langle\overset{?}{\tau}{}'_1,\ \ldots,\ \overset{?}{\tau}{}'_n\rangle}$$

Figure 15. Definition of when two types join concretely

guments are subtypes of each other, as proposed by Smith and Cartwright [14]. Yet, to our surprise, this introduces another source of infinite proofs and potential for non-termination. We give one such example in Figure 14, and, as with prior examples, this example can be modified so that the infinite proof is acyclic. This example is particularly problematic since it satisfies both our inheritance and parameter restrictions. We will address this problem by canonicalizing types prior to syntactic comparison.

7.4 Inheritance Consistency

Lastly, for sake of completeness we discuss a problem which, although not officially addressed by the Java language specification, appears to already be addressed by `javac`. In particular, the type `Numbers<? super String>` poses an interesting problem. The wildcard is constrained explicitly below by `String` and implicitly above by `Number`. Should this type be opened, then transitivity would imply that `String` is a subtype of `Number`, which is inconsistent with the inheritance hierarchy. One might argue that this is sound because we are opening an uninhabitable type and so the code is unreachable anyways. However, this type is inhabitable because Java allows `null` to have any type. Fortunately, `javac` appears to already prohibit such types, preventing unsoundness in the language. Completeness of our subtyping algorithm actually assumes such types are rejected; we did not state this as an explicit requirement of our theorem because it already holds for Java as it exists in practice.

8. Improved Type System

Here we present a variety of slight changes to Java's type system regarding wildcards in order to rid it of the undesirable properties discussed in Section 7.

8.1 Implicit Lower Bounds

Although we showed in Section 7.1 that using complete implicit-constraint inference is problematic, we still believe Java should use a slightly stronger algorithm. In particular, consider the types `Super<Number,?>` and `Super<?,Integer>`. The former is accepted by Java whereas the latter is rejected. However, should Java permit type parameters to have lower-bound requirements, then the class `Super` might also be declared as

```
class Super<P super Q, Q> {}
```

Using Java's completely syntactic approach to implicit-constraint inference, under this declaration now `Super<Number,?>` would be rejected and `Super<?,Integer>` would be accepted. This is the opposite of before, even though the two class declarations are conceptually equivalent. In light of this, implicit-constraint inference should also infer implicit lower-bound constraints for any wildcard corresponding to a type parameter P with another type parameter Q constrained to extend P. This slight strengthening addresses the asymmetry in Java's syntactic approach while still having predictable behavior from a user's perspective and also being compatible with our algorithms even with the language extensions in Section 10.

8.2 Single-Instantiation Inheritance

We have determined an adaptation of single-instantiation inheritance to existential types, and consequently wildcards, which addresses the non-determinism issues raised in Section 7.2:

For all types τ and class or interface names C, if τ has a supertype of the form $\exists\Gamma:\Delta.\ C\langle\ldots\rangle$, then τ has a most precise supertype of that form.

Should this be ensured, whenever a variable of type τ is used as an instance of C the type checker can use the most precise supertype of τ with the appropriate form without having to worry about any alternative supertypes.

Java only ensures single-instantiation inheritance with wildcards when τ is a class or interface type, but not when τ is a type variable. Type variables can either be type parameters or captured wildcards, so we need to ensure single-instantiation inheritance in both cases. In order to do this, we introduce a concept we call concretely joining types, defined in Figure 15.

Conceptually, two types join concretely if they have no wildcards in common. More formally, for any common superclass or superinterface C, there is a most precise common supertype of the form $C\langle\tau_1,\ \ldots,\ \tau_n\rangle$ (i.e. none of the type arguments is a wildcard). In other words, their join is a (set of) concrete types.

Using this new concept, we say that two types validly intersect each other if either is a subtype of the other or they join concretely. For Java specifically, we should impose additional requirements in Figure 15: C or D must be an interface to reflect single inheritance of classes [6: Chapter 8.1.4], and C and D cannot have any common methods with the same signature but different return types (after erasure) to reflect the fact that no class would be able to extend or implement both C and D [6: Chapter 8.1.5]. With this, we can impose our restriction ensuring single-instantiation inheritance for type parameters and captured wildcard type variables so that single-instantiation inheritance holds for the entire type system.

Intersection Restriction

For every syntactically occurring intersection τ_1 `&` … `&` τ_n, every τ_i must validly intersect with every other τ_j. For every explicit upper bound τ on a wildcard, τ must validly intersect with all other upper bounds on that wildcard.

This restriction has an ancillary benefit as well. Concretely joining types have the property that their meet, as discussed in Section 7.2, is simply the intersection of the types. This is not the case for `List<?>` with `Set<?>`, whose meet is `∃X. List<X> & Set<X>`. Our intersection restriction then implies that all intersections coincide with their meet, and so intersection *types* are actually unnecessary in our system. That is, the syntax `P extends` τ_1 `&` … `&` τ_n can simply be interpreted as `P extends` τ_i for each i in 1 to n without introducing an actual type τ_1 `&` … `&` τ_n. Thus our solution addresses the non-determinism issues discussed in Section 7.2 and simplifies the formal type system.

$$\frac{\vdash \tau \Mapsto \bar{\tau} \quad \vdash \tau' \Mapsto \bar{\tau}' \quad \bar{\tau} = \bar{\tau}'}{\mathbb{\Gamma} : \Delta \vdash \tau \cong \tau'}$$

$$\frac{\begin{array}{c} \textit{explicit}(\overset{?}{\tau}_1, \ldots, \overset{?}{\tau}_n) = \langle \Gamma; \Delta; \tau_1, \ldots, \tau_n \rangle \\ \textit{implicit}(\Gamma; C; \tau_1, \ldots, \tau_n) = \mathbb{\Delta} \\ \{v <:: \hat{\tau} \text{ in } \Delta \mid \text{for no } v <:: \hat{\tau}' \text{ in } \mathbb{\Delta},\ \vdash \hat{\tau}' \lessdot \hat{\tau}\} = \Delta_{<::} \\ \{v ::> \hat{\tau} \text{ in } \Delta \mid \text{for no } v ::> \hat{\tau}' \text{ in } \mathbb{\Delta},\ \vdash \hat{\tau} \lessdot \hat{\tau}'\} = \Delta_{::>} \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \vdash \tau_i \Mapsto \tau_i' \\ \{v <:: \hat{\tau}' \mid v <:: \hat{\tau} \text{ in } \Delta_{<::} \text{ and } \vdash \hat{\tau} \Mapsto \hat{\tau}'\} = \Delta'_{<::} \\ \{v ::> \hat{\tau}' \mid v ::> \hat{\tau} \text{ in } \Delta_{::>} \text{ and } \vdash \hat{\tau} \Mapsto \hat{\tau}'\} = \Delta'_{::>} \\ \langle \Gamma; \Delta'_{<::}, \Delta'_{::>}; \tau_1', \ldots, \tau_n' \rangle = \textit{explicit}(\overset{?}{\tau}{}'_1, \ldots, \overset{?}{\tau}{}'_n) \end{array}}{\vdash C\texttt{<}\overset{?}{\tau}_1\texttt{,}\ \ldots\texttt{,}\ \overset{?}{\tau}_n\texttt{>} \Mapsto C\texttt{<}\overset{?}{\tau}{}'_1\texttt{,}\ \ldots\texttt{,}\ \overset{?}{\tau}{}'_n\texttt{>}}$$

$$\frac{}{\vdash v \Mapsto v}$$

$$\frac{\begin{array}{c} \textit{explicit}(\overset{?}{\tau}_1, \ldots, \overset{?}{\tau}_n) = \langle \Gamma; \Delta; \tau_1, \ldots, \tau_n \rangle \\ \textit{explicit}(\overset{?}{\tau}{}'_1, \ldots, \overset{?}{\tau}{}'_n) = \langle \Gamma'; \Delta'; \tau_1', \ldots, \tau_n' \rangle \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \text{if } \tau_i' \text{ in } \Gamma' \text{ then } \tau_i = \tau_i'[\theta] \text{ else } \tau_i \text{ not in } \Gamma \\ \textit{implicit}(\Gamma; C; \tau_1, \ldots, \tau_n) = \mathbb{\Delta} \\ \text{for all } v <:: \hat{\tau} \text{ in } \Delta',\ \theta(v) <:: \hat{\tau}' \text{ in } \mathbb{\Delta}, \Delta \text{ with } \vdash \hat{\tau}' \lessdot \hat{\tau} \\ \text{for all } v ::> \hat{\tau} \text{ in } \Delta',\ \theta(v) ::> \hat{\tau}' \text{ in } \mathbb{\Delta}, \Delta \text{ with } \vdash \hat{\tau} \lessdot \hat{\tau}' \end{array}}{\vdash C\texttt{<}\overset{?}{\tau}_1\texttt{,}\ \ldots\texttt{,}\ \overset{?}{\tau}_n\texttt{>} \lessdot C\texttt{<}\overset{?}{\tau}{}'_1\texttt{,}\ \ldots\texttt{,}\ \overset{?}{\tau}{}'_n\texttt{>}}$$

$$\frac{}{\vdash v \lessdot v}$$

Figure 16. Rules for equivalence via canonicalization

8.3 Canonicalization

In order to support interchangeability of equivalent types we can apply canonicalization prior to all checks for syntactic equality. To enable this approach, we impose one last restriction.

Equivalence Restriction

For every explicit upper bound τ on a wildcard, τ must not be a strict supertype of any other upper bound on that wildcard. For every explicit lower bound τ on a wildcard, τ must be a supertype of every other lower bound on that wildcard.

With this restriction, we can canonicalize wildcard types by removing redundant explicit constraints under the assumption that the type is valid. By assuming type validity, we do not have to check equivalence of type arguments, enabling us to avoid the full challenges that subtyping faces. This means that the type validator must check original types rather than canonicalized types. Subtyping may be used inside these validity checks which may in turn use canonicalization possibly assuming type validity of the type being checked, but such indirect recursive assumptions are acceptable since our formalization permits implicitly infinite proofs.

Our canonicalization algorithm is formalized as the $\Mapsto$ operation in Figure 16. Its basic strategy is to identify and remove all redundant constraints. The primary tool is the $\lessdot$ relation, an implementation of subtyping specialized to be sound and complete between class and interface types only if τ is a supertype of τ' or they join concretely.

Equivalence Theorem. *Given the parameter restriction, the algorithm prescribed by the rules in Figure 16 terminates. Given the intersection and equivalence restrictions, the algorithm is furthermore a sound and nearly complete implementation of type equivalence provided all types are valid according to the Java language specification [6: Chapter 4.5].*

Proof. Here we only discuss how our restrictions enable soundness and completeness; the full proofs can be found in our technical report [16]. The equivalence restriction provides soundness and near completeness by ensuring the assumptions made by the $\lessdot$ relation hold. The intersection restriction provides completeness by ensuring that non-redundant explicit bounds are unique up to equivalence so that syntactic equality after recursively removing all redundant constraints is a complete means for determining equivalence. □

We say our algorithm is nearly complete because it is complete on class and interface types but not on type variables. Our algorithm will only determine that a type variable is equivalent to itself. While type parameters can only be equivalent to themselves, captured wildcard type variables can be equivalent to other types. Consider the type `Numbers<? super Number>` in which the wildcard is constrained explicitly below by `Number` and implicitly above by `Number` so that the wildcard is equivalent to `Number`. Using our algorithm, `Numbers<? super Number>` is not a subtype of `Numbers<Number>`, which would be subtypes should one use a complete equivalence algorithm. While from a theoretical perspective this seems to be a weakness, as Summers et al. have argued [15], from a practical perspective it is a strength since it forces programmers to use the more precise type whenever they actually rely on that extra precision rather than obscure it through implicit equivalences. Plus, our weaker notion of equivalence is still strong enough to achieve our goal of allowing equivalent types to be used interchangeably (provided they satisfy all applicable restrictions). As such, we consider our nearly complete equivalence algorithm to be sufficient and even preferable to a totally complete algorithm.

9. Evaluation of Restrictions

One might consider many of the examples in this paper to be contrived. Indeed, a significant contribution of our work is identifying restrictions that reject such contrived examples but still permit the Java code that actually occurs in practice. Before imposing our restrictions on Java, it is important to ensure that they are actually compatible with existing code bases and design patterns.

To this end, we conducted a large survey of open-source Java code. We examined a total of 10 projects, including NetBeans (3.9 MLOC), Eclipse (2.3 MLOC), OpenJDK 6 (2.1 MLOC), and Google Web Toolkit (0.4 MLOC). As one of these projects we included our own Java code from a prior research project because it made heavy use of generics and rather complex use of wildcards. Altogether the projects totalled 9.2 million lines of Java code with 3,041 generic classes and interfaces out of 94,781 total (ignoring anonymous classes). To examine our benchmark suite, we augmented the OpenJDK 6 compiler to collect statistics on the code it compiled. Here we present our findings.

To evaluate our inheritance restriction, we analyzed all declarations of direct superclasses and superinterfaces that occurred in our suite. In Figure 17, we present in logarithmic scale how many of the 118,918 declared superclasses and superinterfaces had type arguments and used wildcards and with what kind of constraints. If a class or interface declared multiple direct superclasses and superinterfaces, we counted each declaration separately. Out of all these declarations, *none* of them violated our inheritance restriction.

To evaluate our parameter restriction, we analyzed all constraints on type parameters for classes, interfaces, and methods that occurred in our suite. In Figure 18, we break down how the 2,003 parameter constraints used type arguments, wildcards, and constrained wildcards. Only 36 type-parameter constraints contained the syntax `? super`. We manually inspected these 36 cases and determined that out of all type-parameter constraints, *none* of them violated our parameter restriction. Interestingly, we found no case

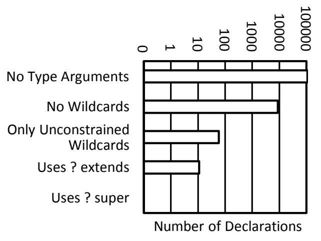

Figure 17. Wildcard usage in inheritance hierarchies

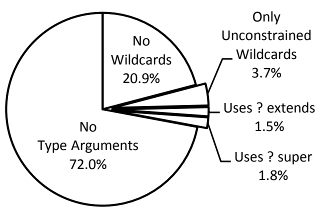

Figure 18. Wildcard usage in type-parameter constraints

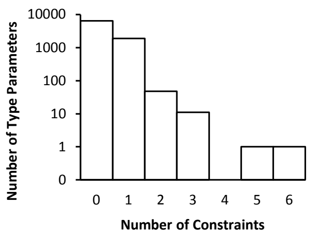

Figure 19. Number of constraints per type parameter

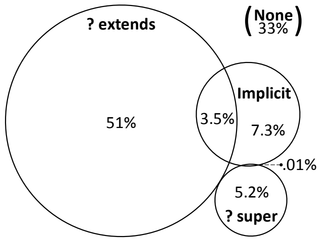

Figure 20. Distribution of constraints on wildcards

where the type with a `? super` type argument was nested inside the constraint; only the constraint itself ever had such a type argument.

To evaluate the first half of our intersection restriction, we examined all constraints on type parameters for classes, interfaces, and methods that occurred in our suite. In Figure 19 we indicate in logarithmic scale how many type parameters had no constraints, one constraint, or multiple constraints by using intersections. In our entire suite there were only 61 intersections. We manually inspected these 61 intersections and determined that, out of all these intersections, *none* of them violated our intersection restriction.

To evaluate the second half of our intersection restriction as well as our equivalence restriction, we examined all wildcards that occurred in our suite. In Figure 20 we break down the various ways the 19,018 wildcards were constrained. Only 3.5% of wildcards had both an implicit and an explicit upper bound. In all of those cases, the explicit upper bound was actually a subtype of the implicit upper bound (interestingly, though, for 35% of these cases the two bounds were actually equivalent). Thus out of all explicit bounds, *none* of them violated either our intersection restriction or our equivalence restriction. Also, in the entire suite there were only 2 wildcards that had both an explicit lower bound and an implicit upper bound, and in both cases the explicit lower bound was a strict subtype of the implicit upper bound.

In summary, *none* of our constraints were ever violated in our entire suite. This leads us to believe that the restrictions we impose will most likely have little negative impact on programmers.

We also manually investigated for implicitly infinite types, taking advantage of the syntactic classification of implicitly infinite types described in Section 9.1. We encountered only one example of an implicitly infinite type. It was actually the same as class `Infinite` in Section 3.2:

```
public interface Assumption<Self extends Assumption<?>> {...}
```

Surprisingly, this is precisely the same as the `Infinite` class in Section 3.2. However, the documentation of the above code suggested that the programmer was trying to encode the "self" type pattern [2, 3]. In fact, we replaced the wildcard above with the pertinent type parameter `Self`, which is actually a more faithful encoding of the "self" type pattern, and the entire code base still type checked. Thus, the additional flexibility of the wildcard in this case was not taken advantage of and was most likely a simple mistake.

This was the only implicitly infinite type we found, and we also did not find evidence of implicitly infinite subtyping proofs. These findings are significant because Cameron et al. proved soundness of wildcards assuming all wildcards and subtyping proofs translate to finite traditional existential types and subtyping proofs [5], and Summers et al. gave semantics to wildcards under the same assumptions [15], so although we have determined these assumptions do not hold theoretically our survey demonstrates that they do hold in practice. Nonetheless, we expect that Cameron et al. and Summers et al. would be able to adapt their work to implicitly infinite types and proofs now that the problem has been identified.

$$\frac{C\texttt{<}P_1\ \texttt{extends}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ P_n\ \texttt{extends}\ \tau_n\texttt{>} \qquad P_i \text{ occurs in } \tau_j}{C.i \rightsquigarrow C.j}$$

$$\frac{\begin{array}{c} C\texttt{<}P_1\ \texttt{extends}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ P_n\ \texttt{extends}\ \tau_n\texttt{>} \\ D\texttt{<}\overset{?}{\tau}_1\texttt{,}\ \ldots\texttt{,}\ \overset{?}{\tau}_m\texttt{>} \text{ occurs in } \tau_i \qquad \overset{?}{\tau}_j \text{ is a wildcard} \end{array}}{C.i \overset{?}{\rightarrow} D.j}$$

$$\frac{\begin{array}{c} C\texttt{<...>}\ \texttt{extends}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ \tau_n \\ D\texttt{<}\overset{?}{\tau}_1\texttt{,}\ \ldots\texttt{,}\ \overset{?}{\tau}_m\texttt{>} \text{ occurs in } \tau_i \qquad \overset{?}{\tau}_j \text{ is a wildcard} \end{array}}{C.0 \overset{?}{\rightarrow} D.j}$$

$$\frac{C.i \overset{?}{\rightarrow} C.j}{C.j \overset{?}{\rightsquigarrow} C.i} \qquad \frac{C.i \overset{?}{\rightarrow} D.j \qquad D.j \rightsquigarrow D.k}{C.i \overset{?}{\rightsquigarrow} D.k}$$

$$\frac{C\texttt{<}P_1\ \texttt{extends}\ \tau_1\texttt{,}\ \ldots\texttt{,}\ P_n\ \texttt{extends}\ \tau_n\texttt{>} \qquad D\texttt{<...>} \text{ occurs in } \tau_j}{C.i \overset{?}{\rightsquigarrow} D.0}$$

$$\frac{C\texttt{<...>}\ \texttt{extends}\ D_1\texttt{<...>,}\ \ldots\texttt{,}\ D_n\texttt{<...>}}{C.0 \overset{?}{\rightsquigarrow} D_i.0}$$

Figure 21. Definition of wildcard dependency

9.1 Preventing Implicitly Infinite Types

Implicitly infinite types arise when the implicit constraints on a wildcard in a type actually depend on that type in some way. For example, the implicit constraint on the wildcard in `Infinite<?>` comes from the constraint on the type parameter of `Infinite` which is itself `Infinite<?>`. Thus if one were to disallow such cyclic dependencies then one would prevent implicitly infinite types from happening.

Although unnecessary for our restrictions, we determined a way to statically detect such cyclic dependencies just be examining the class hierarchy and constraints on type parameters of generic classes. In Figure 21, we present simple rules for determining what we call wildcard dependencies. The judgement $C.i \overset{?}{\rightsquigarrow} D.j$ means that, the implicit constraints for wildcards in the constraint on the i^{th} parameter of C or (when i is 0) a direct superclass of C depend on the constraint on the j^{th} parameter of D or (when j is 0) the superclasses of D. If wildcard dependency is co-well-founded, then all wildcard types correspond to finite forms of our existential types, which is easy to prove by induction.

10. Extensions

Although in this paper we have focused on wildcards, our formalism and proofs are all phrased in terms of more general existential types described in Section A.1. This generality provides opportunity for extensions to the language. Here we offer a few such extensions which preliminary investigations suggest are compatible with our algorithms, although full proofs have not yet been developed.

10.1 Declaration-Site Variance

As Kennedy and Pierce mention [9], there is a simple translation from declaration-site variance to use-site variance which preserves and reflects subtyping. In short, except for a few cases, type arguments τ to covariant type parameters are translated to `? extends` τ, and type arguments τ to contravariant type parameters are translated to `? super` τ. Our restrictions on `? super` then translate to restrictions on contravariant type parameters. For example, our restrictions would require that, in each declared direct superclass and superinterface, only types at covariant locations can use classes or interfaces with contravariant type parameters. Interestingly, this restriction does not coincide with any of the restrictions presented by Kennedy and Pierce. Thus, we have found a new termination result for nominal subtyping with variance. It would be interesting to investigate existing code bases with declaration-site variance to determine if our restrictions might be more practical than prohibiting expansive inheritance.

Because declaration-site variance can be translated to wildcards, Java could use both forms of variance. A wildcard should not be used as a type argument for a variant type parameter since it is unclear what this would mean, although Java might consider interpreting the wildcard syntax slightly differently for variant type parameters for the sake of backwards compatibility.

10.2 Existential Types

The intersection restriction has the unfortunate consequence that constraints such as `List<?> & Set<?>` are not allowed. We can address this by allowing users to use existential types should they wish to. Then the user could express the constraint above using `exists X. List<X> & Set<X>`, which satisfies the intersection restriction. Users could also express potentially useful types such as `exists X. Pair<X,X>` and `exists X. List<List<X>>`.

Besides existentially quantified type variables, we have taken into consideration constraints, both explicit and implicit, and how to restrict them so that termination of subtyping is still guaranteed since the general case is known to be undecidable [21]. While all bound variables must occur somewhere in the body of the existential type, they cannot occur inside an explicit constraint occurring in the body. This both prevents troublesome implicit constraints and permits the join technique in Section 6.1. As for the explicit constraints on the variables, lower bounds cannot reference bound variables and upper bounds cannot have bound variables at covariant locations or in types at contravariant locations. This allows potentially useful types such as `exists X extends Enum<X>. List<X>`. As for implicit constraints, since a bound variable could be used in many locations, the implicit constraints on that variable are the accumulation of the implicit constraints for each location it occurs at. All upper bounds on a bound variable would have to validly intersect with each other, and each bound variable with multiple lower bounds would have to have a most general lower bound.

10.3 Lower-Bounded Type Parameters

Smith and Cartwright propose allowing type parameters to have lower-bound requirements (i.e. `super` clauses) [14], providing a simple application of this feature which we duplicate here.

```
<P super Integer> List<P> sequence(int n) {
  List<P> res = new LinkedList<P>();
  for (int i = 1; i <= n; i++)
    res.add(i);
  return res;
}
```

Our algorithms can support this feature provided lower-bound requirements do not have explicitly bound wildcard type arguments. Also, they should not be other type parameters in the same parameterization since that is better expressed by upper bounds on those type parameters.

10.4 Universal Types

Another extension that could be compatible with our algorithms is a restricted form of predicative universal types like `forall X. List<X>`. Although this extension is mostly speculative, we mention it here as a direction for future research since preliminary investigations suggest it is possible. Universal types would fulfill the main role that raw types play in Java besides convenience and backwards compatibility. In particular, for something like an immutable empty list one typically produces one instance of an anonymous class implementing raw `List` and then uses that instance as a `List` of any type they want. This way one avoids wastefully allocating a new empty list for each type. Adding universal types would eliminate this need for the back door provided by raw types.

11. Conclusion

Despite their conceptual simplicity, wildcards are formally complex, with impredicativity and implicit constraints being the primary causes. Although most often used in practice for use-site variance [7, 17–19], wildcards are best formalized as existential types [4, 5, 7, 9, 18, 19, 21], and more precisely as coinductive existential types with coinductive subtyping proofs, which is a new finding to the best of our knowledge.

In this paper we have addressed the problem of subtyping of wildcards, a problem suspected to be undecidable in its current form [9, 21]. Our solution imposes simple restrictions, which a survey of 9.2 million lines of open-source Java code demonstrates are already compatible with existing code. Furthermore, our restrictions are all local, allowing for informative user-friendly error messages should they ever be violated.

Because our formalization and proofs are in terms of a general-purpose variant of existential types, we have identified a number of extensions to Java that should be compatible with our algorithms. Amongst these are declaration-site variance and user-expressible existential types, which suggests that our algorithms and restrictions may be suited for Scala as well, for which subtyping is also suspected to be undecidable [9, 21]. Furthermore, it may be possible to support some form of universal types, which would remove a significant practical application of raw types so that they may be unnecessary should Java ever discard backwards compatibility as forebode in the language specification [6: Chapter 4.8].

While we have addressed subtyping, joins, and a number of other subtleties with wildcards, there is still plenty of opportunity for research to be done on wildcards. In particular, although we have provided techniques for improving type-argument inference, we believe it is important to identify a type-argument inference algorithm which is both complete in practice *and* provides guarantees regarding ambiguity of program semantics. Furthermore, an inference system would ideally inform users at declaration-site how inferable their method signature is, rather than having users find out at each use-site. We hope our explanation of the challenges helps guide future research on wildcards towards solving such problems.

Acknowledgements

We would like to thank Nicholas Cameron, Sophia Drossopoulou, Erik Ernst, Suresh Jagannathan, Andrew Kennedy, Jens Palsberg, Pat Rondon, and the anonymous reviewers for their many insightful suggestions and intriguing discussions.

$$\begin{aligned}\overset{\top}{\tau} &::= v \mid C\langle\overset{\top}{\tau},\ \ldots,\ \overset{\top}{\tau}\rangle \mid \exists\Gamma:\Delta.\,\overset{\top}{\tau}\\ \Gamma &::= \varnothing \mid v,\Gamma\\ \Delta &::= \varnothing \mid \Delta,\overset{\top}{\tau} <:: \overset{\top}{\tau}\end{aligned}$$

Figure 22. Grammar for traditional existential types (coinductive except for $\exists\Gamma:\Delta.\,\overset{\top}{\tau}$)

$$\frac{v \text{ in } \mathbb{\Gamma}}{\mathbb{\Gamma}\vdash v} \qquad \frac{\text{for all } i \text{ in } 1 \text{ to } n,\ \ \mathbb{\Gamma}\vdash\overset{\top}{\tau}_i}{\mathbb{\Gamma}\vdash C\langle\overset{\top}{\tau}_1,\ \ldots,\ \overset{\top}{\tau}_n\rangle} \qquad \frac{\begin{array}{c}\mathbb{\Gamma},\Gamma\vdash\Delta\\ \mathbb{\Gamma},\Gamma\vdash\overset{\top}{\tau}\end{array}}{\mathbb{\Gamma}\vdash\exists\Gamma:\Delta.\,\overset{\top}{\tau}}$$

$$\frac{}{\mathbb{\Gamma}\vdash\varnothing} \qquad \frac{\mathbb{\Gamma}\vdash\overset{\top}{\tau} \quad \mathbb{\Gamma}\vdash\overset{\top}{\tau}' \quad \mathbb{\Gamma}\vdash\Delta}{\mathbb{\Gamma}\vdash\Delta,\overset{\top}{\tau} <:: \overset{\top}{\tau}'}$$

Figure 23. Well-formedness for traditional existential types

$$(1)\quad \frac{\begin{array}{c}C\langle P_1,\ \ldots,\ P_m\rangle \text{ is a subclass of } D\langle\overset{\overline{\top}}{\tau}_1,\ \ldots,\ \overset{\overline{\top}}{\tau}_n\rangle\\ \text{for all } i \text{ in } 1 \text{ to } n, \mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\overline{\top}}{\tau}_i[P_1\mapsto\overset{\top}{\tau}_1,\ldots,P_m\mapsto\overset{\top}{\tau}_m]\cong\overset{\top}{\tau}'_i\end{array}}{\mathbb{\Gamma}:\mathbb{\Delta}\vdash C\langle\overset{\top}{\tau}_1,\ \ldots,\ \overset{\top}{\tau}_m\rangle <: D\langle\overset{\top}{\tau}'_1,\ \ldots,\ \overset{\top}{\tau}'_n\rangle}$$

$$(2)\quad \frac{\mathbb{\Gamma},\Gamma:\mathbb{\Delta},\Delta\vdash\overset{\top}{\tau} <: \overset{\top}{\tau}'}{\mathbb{\Gamma}:\mathbb{\Delta}\vdash\exists\Gamma:\Delta.\,\overset{\top}{\tau} <: \overset{\top}{\tau}'}$$

$$(3)\quad \frac{\begin{array}{c}\text{for all } v \text{ in } \Gamma,\ \ \mathbb{\Gamma}\vdash\theta(v)\\ \text{for all } \overset{\overline{\top}}{\tau} <:: \overset{\hat{\top}}{\tau} \text{ in } \Delta,\ \ \mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\overline{\top}}{\tau}[\theta] <: \overset{\hat{\top}}{\tau}[\theta]\\ \mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau} <: \overset{\top}{\tau}'[\theta]\end{array}}{\mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau} <: \exists\Gamma:\Delta.\,\overset{\top}{\tau}'}$$

$$(4)\quad \frac{\overset{\top}{\tau} <:: \overset{\top}{\tau}' \text{ in } \mathbb{\Delta}}{\mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau} <: \overset{\top}{\tau}'}$$

$$(5)\quad \frac{}{\mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau} <: \overset{\top}{\tau}}$$

$$(6)\quad \frac{\mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau} <: \overset{\top}{\tau}' \quad \mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau}' <: \overset{\top}{\tau}''}{\mathbb{\Gamma}:\mathbb{\Delta}\vdash\overset{\top}{\tau} <: \overset{\top}{\tau}''}$$

Figure 24. Traditional subtyping rules for existential types (coinductive except for transitivity)

A. Type Systems

Here we formalize traditional existential types and our existential types along with the subtyping rules for both systems. We will also formalize a simplified formalism of wildcards; we do not claim this is *the* formalism of wildcards since wildcards are not ours to formalize, but this formalism captures the key concepts and challenges of wildcards. Note that, throughout our formalism, definitions are coinductive by default, and problems with name hiding is assumed to be handled behind the scenes for simplicity.

A.1 Traditional Existential Types

In Figure 22, we show the grammar for (a form of) traditional existential types. Note that the constructor $\exists\Gamma:\Delta.\,\overset{\top}{\tau}$ is inductive, meaning there cannot be an infinite number of sequential applications of that constructor; it may only be used a finite number of times between applications of coinductive constructors. In Figure 23, we show the rules for determining when a traditional existential type is well-formed, essentially just ensuring that type variables are in scope.

$$\frac{v \text{ in } \mathbb{\Gamma}}{\mathbb{\Gamma}\vdash v} \qquad \frac{\begin{array}{c}\mathbb{\Gamma}\vdash\Gamma:\Delta\\ \mathbb{\Gamma}\vdash\Gamma:\dot{\Delta}\\ \text{for all } i \text{ in } 1 \text{ to } n,\ \ \mathbb{\Gamma},\Gamma\vdash\overset{\exists}{\tau}_i\end{array}}{\mathbb{\Gamma}\vdash\exists\Gamma:\Delta(\dot{\Delta}).\,C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle}$$

$$\frac{}{\mathbb{\Gamma}\vdash\Gamma:\varnothing} \qquad \frac{\begin{array}{c}v \text{ in } \Gamma\\ \mathbb{\Gamma},\Gamma\vdash\overset{\exists}{\tau}\\ \mathbb{\Gamma}\vdash\Gamma:\Delta\end{array}}{\mathbb{\Gamma}\vdash\Gamma:\Delta,v <:: \overset{\exists}{\tau}} \qquad \frac{\begin{array}{c}v \text{ in } \Gamma\\ \mathbb{\Gamma},\Gamma\vdash\overset{\exists}{\tau}\\ \mathbb{\Gamma}\vdash\Gamma:\Delta\end{array}}{\mathbb{\Gamma}\vdash\Gamma:\Delta,v ::> \overset{\exists}{\tau}}$$

Figure 25. Well-formedness for our existential types

Subtyping We present the traditional rules for subtyping in Figure 24. Rule 1 incorporates the class hierarchy into the type system, using some externally specified notion of equivalence such as syntactic equality or being subtypes of each other; this notion of equivalence must be reflexive, transitive, and preserved by constraint-satisfying substitutions. Rule 2 opens an existential type, whereas Rule 3 packs an existential type. Opening grants access to the existentially quantified types and proofs in the subtype by adding them as variables in the environment. Packing specifies the types and proofs to be existentially quantified in the supertype be instantiating the bound variables to valid types in the current environment and constructing the required proofs within the current environment. Rule 4 grants access to the proofs provided by the environment. Rule 5 ensures reflexivity, and Rule 6 ensures transitivity.

Note that transitivity is not coinductive. If transitivity were coinductive, we could prove $\overset{\top}{\tau} <: \overset{\top}{\tau}'$ for any $\overset{\top}{\tau}$ and $\overset{\top}{\tau}'$: first apply transitivity with $\overset{\top}{\tau}$ as the middle type, apply reflexivity to prove the left side, then the right goal is $\overset{\top}{\tau} <: \overset{\top}{\tau}'$ again so we can repeat this process using transitivity ad infinitum. So, although a proof we have an infinite number of uses of transitivity, there cannot be infinite consecutive uses of transitivity.

A.2 Our Existential Types

We already presented the grammar of our existential types in Figure 6. In Figure 25 we formalize basic well-formedness of our existential types; throughout this paper we will implicitly assume all types are well-formed. This well-formedness simply ensures type variables are used appropriately. The major ways our existential types differ from traditional existential types are the prevention of types like $\exists v.v$, the extra set of constraints $\dot{\Delta}$, and the fact that constraints can only be lower or upper bounds of variables in Γ.

We also already presented our subtyping rules in Figure 7. However, although we will show that for wildcards all proofs are finite, in order to separate termination/finiteness from soundness we make SUB-VAR inductive in the more general case. Otherwise, an assumption of the form $v <:: v$ would allow one to prove v is a subtype of anything by simply applying SUB-VAR forever. This corresponds to the restriction on transitivity in traditional subtyping, since SUB-VAR essentially combines transitivity with assumption. Also, as with traditional existential types, the externally specified notion of equivalence must be reflexive, transitive, and preserved by substitution.

In Figure 26 we present the definition of our adaptation of unification. This adaptation prevents variables from escaping and is designed to enable equivalence rather than syntactic equality. The judgement $\mathbb{\Gamma} \xleftarrow{\cancel{\mathbb{\Gamma}}} \mathbb{\Gamma}' \vdash \overset{\exists}{\tau} \approx_\theta \overset{\exists}{\tau}'$ has a lot of parts. $\mathbb{\Gamma}'$ are the type variables to unify; they only occur in $\overset{\exists}{\tau}'$. $\cancel{\mathbb{\Gamma}}$ are type variables which should be ignored; they only occur in $\overset{\exists}{\tau}'$. $\mathbb{\Gamma}$ are the type variables that from the general context; they may occur in $\overset{\exists}{\tau}$ and $\overset{\exists}{\tau}'$. The sets $\mathbb{\Gamma}$, $\mathbb{\Gamma}'$, and $\cancel{\mathbb{\Gamma}}$ are disjoint. The judgement holds if for each variable v in $\mathbb{\Gamma}'$ and free in $\overset{\exists}{\tau}'$ that variable is mapped by θ to some subexpression of $\overset{\exists}{\tau}$ corresponding to at least one (but not

$$\frac{v' \text{ in } \mathbb{\Gamma}' \qquad \text{for all } v \text{ free in } \overset{\exists}{\tau},\ v \text{ in } \mathbb{\Gamma}}{\mathbb{\Gamma} \xleftarrow{\not{\mathbb{\Gamma}}} \mathbb{\Gamma}' \vdash \overset{\exists}{\tau} \approx_{\{v' \mapsto \overset{\exists}{\tau}\}} v'} \qquad \frac{v \text{ in } \not{\mathbb{\Gamma}}}{\mathbb{\Gamma} \xleftarrow{\not{\mathbb{\Gamma}}} \mathbb{\Gamma}' \vdash \overset{\exists}{\tau} \approx_{\varnothing} v} \qquad \frac{v \text{ in } \mathbb{\Gamma}}{\mathbb{\Gamma} \xleftarrow{\not{\mathbb{\Gamma}}} \mathbb{\Gamma}' \vdash v \approx_{\varnothing} v}$$

$$\frac{\begin{array}{c}\text{for all } i \text{ in } 1 \text{ to } n,\ \mathbb{\Gamma} \xleftarrow{\not{\mathbb{\Gamma}},\Gamma'} \mathbb{\Gamma}' \vdash \overset{\exists}{\tau}_i \approx_{\theta_i} \overset{\exists}{\tau}'_i \\ \theta \subseteq \bigcup_{i \text{ in } 1 \text{ to } n} \theta_i \\ \text{for all } v' \text{ in } \Gamma',\ (\text{exists } i \text{ in } 1 \text{ to } n \text{ with } \theta_i \text{ defined on } v') \text{ implies } \theta \text{ defined on } v'\end{array}}{\mathbb{\Gamma} \xleftarrow{\not{\mathbb{\Gamma}}} \mathbb{\Gamma}' \vdash \exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle \approx_\theta \exists\Gamma' : \Delta'(\mathbb{\Delta}').\ C\langle\overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle}$$

Figure 26. Unification adapted to existential types with equivalence

```
$\textit{translate}(\overset{\exists}{\tau})$
 $= \textbf{case } \overset{\exists}{\tau} \textbf{ of}$
    $v \mapsto v$
    $\exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle$
      $\mapsto \exists\Gamma : \textit{translate-cxt}(\Delta).\ C\langle\textit{translate}(\overset{\exists}{\tau}_1), \ldots, \textit{translate}(\overset{\exists}{\tau}_n)\rangle$

$\textit{translate-cxt}(\Delta)$
 $= \textbf{case } \Delta \textbf{ of}$
    $\varnothing \mapsto \varnothing$
    $\Delta, v <:: \overset{\exists}{\tau} \mapsto \textit{translate-cxt}(\Delta), v <:: \textit{translate}(\overset{\exists}{\tau})$
    $\Delta, v ::> \overset{\exists}{\tau} \mapsto \textit{translate-cxt}(\Delta), \textit{translate}(\overset{\exists}{\tau}) <:: v$
```

Figure 27. Translation from our existential types to traditional existential types

$$\begin{array}{l}\tau ::= v \mid C\langle\overset{?}{\tau},\ \ldots,\ \overset{?}{\tau}\rangle \\ \overset{?}{\tau} ::= \tau \mid \texttt{?} \mid \texttt{? extends } \tau \mid \texttt{? super } \tau\end{array}$$

Figure 28. Grammar for wildcard types (inductive)

$$\frac{v \text{ in } \mathbb{\Gamma}}{\mathbb{\Gamma} \vdash v} \qquad \frac{\text{for all } i \text{ in } 1 \text{ to } n,\ \mathbb{\Gamma} \vdash \overset{?}{\tau}_i}{\mathbb{\Gamma} \vdash C\langle\overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_n\rangle}$$

$$\frac{}{\mathbb{\Gamma} \vdash \texttt{?}} \qquad \frac{\mathbb{\Gamma} \vdash \tau}{\mathbb{\Gamma} \vdash \texttt{? extends } \tau} \qquad \frac{\mathbb{\Gamma} \vdash \tau}{\mathbb{\Gamma} \vdash \texttt{? super } \tau}$$

Figure 29. Well-formedness for wildcard types

necessarily all) of the locations of v in $\overset{\exists}{\tau}'$ whose free variables are contained $\mathbb{\Gamma}$. Standard unification would require the subexpression to correspond to all locations of v in $\overset{\exists}{\tau}'$, but we relax this to permit equivalence rather than syntactic equality; the subsequent premises in SUB-EXISTS make up for this relaxation among others.

A.2.1 Translation to Traditional Existential Types

Our existential types are designed to be a bridge into traditional existential types. As such, there is a simple translation from them to traditional existential types, shown in Figure 27, which basically drops Δ.

A.3 Wildcard Types

We present the grammar of wildcard types in Figure 28 and basic well-formedness in Figure 29. Note that well-formedness is not type validity; it only checks that variables are in the appropriate scope. Type validity turns out to not be important for termination, only for soundness and completeness. In fact, it is important that type validity is not necessary for termination of subtyping since

```
$\textit{convert}(\tau)$
 $= \textbf{case } \tau \textbf{ of}$
    $v \mapsto v$
    $C\langle\overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_n\rangle$
      $\mapsto$ $\textbf{let } \langle\Gamma; \Delta; \tau_1, \ldots, \tau_n\rangle = \textit{explicit}(\overset{?}{\tau}_1, \ldots, \overset{?}{\tau}_n)$
             $\mathbb{\Delta} = \textit{implicit}(\Gamma; C; \tau_1, \ldots, \tau_n)$
             $\Delta' = \textit{convert-cxt}(\Delta)$
             $\mathbb{\Delta}' = \textit{convert-cxt}(\mathbb{\Delta})$
         $\textbf{in } \exists\Gamma : \mathbb{\Delta}', \Delta'(\Delta').\ C\langle\textit{convert}(\tau_1),\ \ldots,\ \textit{convert}(\tau_n)\rangle$

$\textit{convert-cxt}(\Delta)$
 $= \textbf{case } \Delta \textbf{ of}$
    $\varnothing \mapsto \varnothing$
    $\Delta, v <:: \tau \mapsto \textit{convert-cxt}(\Delta), v <:: \textit{convert}(\tau)$
    $\Delta, v ::> \tau \mapsto \textit{convert-cxt}(\Delta), v ::> \textit{convert}(\tau)$
```

Figure 31. Translation from wildcard types to our existential types

Java's type validity rules rely on subtyping. As such, we will discuss type validity when it becomes relevant.

The subtyping rules for wildcard types are shown in Figure 30. We have tried to formalize the informal rules in the Java language specification [6: Chapters 4.4, 4.10.2, and 5.1.10] as faithfully as possible, although we did take the liberty of simplifying away intersection types and the null type by allowing type variables to have multiple constraints and different kinds of constraints. Notice that these are the first subtyping rules to discuss requirements on type parameters. This is because these requirements do not affect subtyping except for how they affect the constraints on the type variables. Thus in the translation from wildcard types to our existential types shown in Figure 31, the constraints resulting from these requirements are included in Δ and hence the subtyping between our existential types and consequently traditional existential types. We will discuss this more as it becomes relevant.

B. Termination

While termination for subtyping even restricted forms of traditional existential types is known to be undecidable [21], here we show that under certain conditions subtyping for both our existential types and wildcard types is decidable. We present a number of requirements on our existential types and show how the restrictions discussed in Section 4.4 ensure that the translation from wildcards to our existential types satisfies these requirements.

B.1 Termination for Our Existential Types

Here we define requirements on our existential types and prove that under these requirements the subtyping algorithm in Figure 7 is guaranteed to terminate.

$$\frac{C\langle P_1,\ \ldots,\ P_m\rangle \text{ is a subclass of } D\langle\bar{\tau}_1,\ \ldots,\ \bar{\tau}_n\rangle}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash C\langle\tau_1,\ \ldots,\ \tau_m\rangle <: D\langle\bar{\tau}_1[P_1\mapsto\tau_1,\ldots,P_m\mapsto\tau_m],\ \ldots,\ \bar{\tau}_n[P_1\mapsto\tau_1,\ldots,P_m\mapsto\tau_m]\rangle}$$

$$\frac{\text{for all } i \text{ in 1 to } n,\ \ \mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau_i \in \overset{?}{\tau}_i}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash C\langle\tau_1,\ \ldots,\ \tau_n\rangle <: C\langle\overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_n\rangle}$$

$$\frac{\begin{array}{c}\textit{explicit}(\overset{?}{\tau}_1,\ldots,\overset{?}{\tau}_n) = \langle\Gamma;\dot{\Delta};\tau_1,\ldots,\tau_n\rangle\\ \textit{implicit}(\Gamma;C;\tau_1,\ldots,\tau_n) = \mathring{\Delta}\\ \mathbb{\Gamma},\Gamma : \mathbf{\Delta},\mathring{\Delta},\dot{\Delta} \vdash C\langle\tau_1,\ \ldots,\ \tau_n\rangle <: \tau'\end{array}}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash C\langle\overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_n\rangle <: \tau'}$$

$$\frac{v <:: \tau \text{ in } \mathbf{\Delta}}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash v <: \tau} \qquad \frac{v ::> \tau \text{ in } \mathbf{\Delta}}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau <: v}$$

$$\frac{}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau <: \tau} \qquad \frac{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau <: \tau' \quad \mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau' <: \tau''}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau <: \tau''}$$

$$\frac{}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau \in \tau} \qquad \frac{}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau \in \texttt{?}} \qquad \frac{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau <: \tau'}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau \in \texttt{? extends } \tau'} \qquad \frac{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau' <: \tau}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau \in \texttt{? super } \tau'}$$

```
$\textit{explicit}(\overset{?}{\tau}_1,\ldots,\overset{?}{\tau}_n)$
 $=$ **case** $\overset{?}{\tau}_i$ **of**
    $\tau$                     $\mapsto \langle\tau;\varnothing;\varnothing\rangle$
    $\texttt{?}$                   $\mapsto \langle v_i;v_i;\varnothing\rangle$
    $\texttt{? extends } \tau$   $\mapsto \langle v_i;v_i;v_i <:: \tau\rangle$
    $\texttt{? super } \tau$     $\mapsto \langle v_i;v_i;v_i ::> \tau\rangle$
```

```
$\textit{implicit}(\Gamma;C;\tau_1,\ldots,\tau_n)$
 $=$ **for** $i$ **in** $1$ **to** $n$**,** $C\langle P_1,\ \ldots,\ P_n\rangle$ requires $P_1$ to extend $\bar{\tau}^i_j$ for $j$ in 1 to $k_i$
   **in for** $i$ **in** $1$ **to** $n$**, let** $\mathring{\Delta}_i =$ **if** $\tau_i$ in $\Gamma$ as $v$
                                  **then if** $k_i = 0$
                                      **then** $v <:: \texttt{Object}$
                                      **else** $v <:: \bar{\tau}^i_1[P_1\mapsto\tau_1,\ldots,P_n\mapsto\tau_n],\ldots,v <:: \bar{\tau}^i_{k_i}[P_1\mapsto\tau_1,\ldots,P_n\mapsto\tau_n]$
                                  **else** $\varnothing$
     **in let** $\mathring{\Delta} = \mathring{\Delta}_1,\ldots,\mathring{\Delta}_n$
       **in** $\mathring{\Delta}, \{v' ::> v \mid v <:: v' \text{ in } \mathring{\Delta} \text{ and } v' \text{ in } \Gamma\}$
```

Figure 30. Subtyping for wildcard types (inductive)

```
$\textit{explicit}(\overset{\exists}{\tau})$
 $=$ **case** $\overset{\exists}{\tau}$ **of**
    $v \mapsto v$
    $\exists\Gamma:\Delta(\dot{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle$
      $\mapsto \exists\Gamma:\bullet\,(\textit{explicit}(\dot{\Delta})).\ C\langle\textit{explicit}(\overset{\exists}{\tau}_1),\ \ldots,\ \textit{explicit}(\overset{\exists}{\tau}_n)\rangle$

$\textit{explicit}(\dot{\Delta})$
 $=$ **case** $\dot{\Delta}$ **of**
    $\varnothing \mapsto \varnothing$
    $\dot{\Delta}', v <:: \overset{\exists}{\tau} \mapsto \textit{explicit}(\dot{\Delta}'), v <:: \textit{explicit}(\overset{\exists}{\tau})$
    $\dot{\Delta}', v ::> \overset{\exists}{\tau} \mapsto \textit{explicit}(\dot{\Delta}'), v ::> \textit{explicit}(\overset{\exists}{\tau})$
```

Figure 32. Explicit projection

B.1.1 Projections

We define a number of projections that we apply to our existential types for sake of reasoning. These projections replace components of an existential type with $\bullet$ to indicate that those components are somehow not relevant. For example, the *explicit* projection in Figure 32 recursively conceals all the constraints that do not need to be checked. The components left over are then essentially the components explicitly stated by the user.

In Figure 33 we present a variety of accessibility projections. The projection $\textit{access-sub}_i(\overset{\exists}{\tau})$ creates a type whose free variables are the variables which, supposing τ were the subtype, could eventually be either the subtype or supertype after i or more swaps. The projection $\textit{access-sup}_i(\overset{\exists}{\tau})$ does the same but supposing τ were the supertype. The projection $\textit{access}_i(\overset{\exists}{\tau})$ is the union of $\textit{access-sub}_i(\overset{\exists}{\tau})$ and $\textit{access-sup}_i(\overset{\exists}{\tau})$. Note that $i = 0$ does not imply all free variables in $\overset{\exists}{\tau}$ remain free after any of the projections because some free variables may never eventually be either the subtype or supertype.

The $\textit{unifies}(\overset{\exists}{\tau})$ projection in Figure 34 creates a type whose free variables are guaranteed to be unified should $\overset{\exists}{\tau}$ be unified with another existential type.

B.1.2 Properties for Termination

We say a type $\overset{\exists}{\tau} = \exists\Gamma:\Delta(\dot{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle$ is terminating if it and all types contained within it satisfy the following termination requirements (using the coinductive definition of this recursively defined property).

Algorithm Requirement 1. *explicit*$(\overset{\exists}{\tau})$ results in a finite type and any variable that is free in $\overset{\exists}{\tau}$ is also free in *explicit*$(\overset{\exists}{\tau})$.

$$\begin{array}{l}
\textit{access-var}_\phi(v) \\
\;= \textbf{if } \phi(0) \\
\quad \textbf{then } v \\
\quad \textbf{else } \bullet \\[1em]
\textit{access-sub}_\phi(\overset{\exists}{\tau}) \\
\;= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto \textit{access-var}_\phi(v) \\
\quad \exists\Gamma : \Delta(\dot{\Delta}).\ C\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n \rangle \\
\qquad \mapsto \exists\Gamma : \textit{access-cxt}_\phi(\Delta)(\bullet). \\
\qquad\qquad\qquad\qquad \bullet\langle \textit{access}_\phi(\overset{\exists}{\tau}_1),\ \ldots,\ \textit{access}_\phi(\overset{\exists}{\tau}_n) \rangle \\[1em]
\textit{access-cxt}_\phi(\Delta) \\
\;= \textbf{case } \Delta \textbf{ of} \\
\quad \varnothing \mapsto \varnothing \\
\quad \Delta', v <:: \overset{\exists}{\tau} \mapsto \textit{access-cxt}_\phi(\Delta'), v <:: \textit{access-sub}_\phi(\overset{\exists}{\tau}) \\
\quad \Delta', v ::> \overset{\exists}{\tau} \mapsto \textit{access-cxt}_\phi(\Delta'), v ::> \textit{access-sup}_\phi(\overset{\exists}{\tau}) \\[1em]
\textit{access-sup}_\phi(\overset{\exists}{\tau}) \\
\;= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto \textit{access-var}_\phi(v) \\
\quad \exists\Gamma : \Delta(\dot{\Delta}).\ C\langle \ldots \rangle \mapsto \exists\Gamma : \bullet\,(\textit{access-exp}_\phi(\dot{\Delta})).\ \bullet \\[1em]
\textit{access-exp}_\phi(\dot{\Delta}) \\
\;= \textbf{case } \dot{\Delta} \textbf{ of} \\
\quad \varnothing \mapsto \varnothing \\
\quad \dot{\Delta}', v <:: \overset{\exists}{\tau} \mapsto \textit{access-exp}_\phi(\dot{\Delta}'), v <:: \textit{access-sup}_\phi(\overset{\exists}{\tau}) \\
\quad \dot{\Delta}', v ::> \overset{\exists}{\tau} \mapsto \textit{access-exp}_\phi(\dot{\Delta}'), v ::> \textit{access-sub}_{\lambda i.\phi(1+i)}(\overset{\exists}{\tau}) \\[1em]
\textit{access}_\phi(\overset{\exists}{\tau}) \\
\;= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto \textit{access-var}_\phi(v) \\
\quad \exists\Gamma : \Delta(\dot{\Delta}).\ C\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n \rangle \\
\qquad \mapsto \exists\Gamma : \textit{access-cxt}_\phi(\Delta)(\textit{access-exp}_\phi(\dot{\Delta})). \\
\qquad\qquad\qquad\qquad \bullet\langle \textit{access}_\phi(\overset{\exists}{\tau}_1),\ \ldots,\ \textit{access}_\phi(\overset{\exists}{\tau}_n) \rangle
\end{array}$$

Figure 33. Accessibility projections

$$\begin{array}{l}
\textit{unifies}(\overset{\exists}{\tau}) \\
\;= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto v \\
\quad \exists\Gamma : \Delta(\dot{\Delta}).\ C\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n \rangle \\
\qquad \mapsto \exists\Gamma : \bullet\,(\bullet).\ \bullet\langle \textit{unifies}(\overset{\exists}{\tau}_1),\ \ldots,\ \textit{unifies}(\overset{\exists}{\tau}_n) \rangle
\end{array}$$

Figure 34. Unifies projection

This requirement indicates that, while implicit information can be infinite, types should still be expressible finitely.

Algorithm Requirement 2. For all v in Γ, v is free in *unifies*$(\overset{\exists}{\tau})$.

This requirement indicates that every variable bound in an existential type should occur at a location which guarantees a specific assignment during unification. This prevents types like $\exists$`X` : ... (`X` <:: `List<X>`). `String` for which determining whether it is a supertype of `String` would require searching through the entire class hierarchy for a class C which extends `List<`C`>`, ruining efficiency and modularity.

Algorithm Requirement 3. No variable in Γ occurs free in *access-exp*$_{\textbf{true}}(\dot{\Delta})$.

It turns out that even very simply forms of explicit constraints referencing bound variables can cause subtyping to not terminate.

$$\begin{array}{l}
\textit{height-sub}(\overset{\exists}{\tau}) \\
\;= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto 0 \\
\quad \exists\Gamma : \Delta(\dot{\Delta}).\ C\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n \rangle \mapsto \bigsqcup \left\{ \begin{array}{ll}
\displaystyle\bigsqcup_{i \text{ in } 1 \text{ to } n} & \left\{ \begin{array}{l} \textit{height-sub}(\overset{\exists}{\tau}_i) \\ \textit{height-sup}(\overset{\exists}{\tau}_i) \end{array} \right. \\
\displaystyle\bigsqcup_{v <:: \overset{\exists}{\tau}' \text{ in } \Delta} & \textit{height-sub}(\overset{\exists}{\tau}') \\
\displaystyle\bigsqcup_{v ::> \overset{\exists}{\tau}' \text{ in } \Delta} & \textit{height-sup}(\overset{\exists}{\tau}')
\end{array} \right. \\[2em]
\textit{height-sup}(\overset{\exists}{\tau}) \\
\;= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto 0 \\
\quad \exists\Gamma : \Delta(\dot{\Delta}).\ C\langle \ldots \rangle \mapsto \bigsqcup \left\{ \begin{array}{l}
\displaystyle\bigsqcup_{v <:: \overset{\exists}{\tau}' \text{ in } \Delta} \textit{height-sup}(\overset{\exists}{\tau}') \\
1 + \displaystyle\bigsqcup_{v ::> \overset{\exists}{\tau}' \text{ in } \Delta} \textit{height-sub}(\overset{\exists}{\tau}')
\end{array} \right.
\end{array}$$

Figure 35. Definition of useful height functions

For example, if we define $\overset{\exists}{\tau}$ as $\exists$`X,Y` : ... (`X` <:: `Y`). `Pair<X,Y>`, representing pairs where the first component is a subtype of the second component, and if type parameter `P` is constrained to be a subtype of `Pair<P,`$\overset{\exists}{\tau}$`>` then determining whether `P` is a subtype of $\overset{\exists}{\tau}$ can run forever. Algorithmically, the problem is that if a bound variable is accessible in an explicit constraint then unification of that variable allows the accessible portion of a type to grow which can lead to non-termination.

Algorithm Requirement 4. Δ has finite length.

This simply ensures that we can enumerate all the constraints in Δ when checking assumptions on opened type variables.

Algorithm Requirement 5. $<::$ in Δ is co-well-founded on Γ, no variable in Γ occurs free in *access-cxt*$_{>0}(\Delta)$, and the relation "there exists $\overset{\hat{\exists}}{\tau}$ with $v' ::> \overset{\hat{\exists}}{\tau}$ in Δ and v free in *access-sup*$_{\textbf{true}}(\overset{\hat{\exists}}{\tau})$" is well-founded on Γ.

The first component ensures that the transitive closure of $<::$ is anti-symmetric on type variables, preventing the algorithm from cycling through assumptions forever. The second component prevents assumed constraints from simulating the same problems that inheritance causes as we discussed in Sections 3.1 and 4.4. The third component ensures that as we process assumed lower bounds on type variables we slowly dwindle the type variables that are reachable in a sense.

Algorithm Requirement 6. *height-sub*$(\overset{\exists}{\tau})$ is finite.

Although we can handle implicitly infinite types, certain implicitly infinite types can cause infinite recursion. This requirement prevents problematic types such as in Figure 9. Note that, since this requirement must hold for all subexpressions of $\overset{\exists}{\tau}$ as well, it also implies that *height-sup*$(\overset{\exists}{\tau})$ is finite.

Algorithm Requirement 7. No variable in Γ occurs free in *access*$_{>0}(\overset{\exists}{\tau}_i)$ for any i in 1 to n.

Constraints on type variables and uses of nested existential types referencing such type variables can cause an indirect form of problematically implicitly infinite types. For example, the type $\overset{\exists}{\tau}$ defined as $\exists v : v <:: \overset{\exists}{\tau}(\varnothing)$. `List<List<? super List<? extends `$v$`>>>` has *height-sub* of 1 but the `? extends` v in the body is essentially an indirect means towards `? extends` τ because of the constraint on v and if this replacement were done ad infinitum then *height-sub*

$$\begin{array}{l}
\mathit{swaps\text{-}sub}(\overset{\exists}{\tau}) \\
= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto \{\mathit{sub}(v) \mapsto 0\} \\
\quad \exists\Gamma:\Delta(\Delta).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle \\
\qquad \mapsto \textbf{fix } v \textbf{ from } \Gamma \\
\qquad\qquad \textbf{using} \begin{cases} \mathit{sub}(v) \mapsto \displaystyle\bigsqcup_{v<::\overset{\exists}{\tau}' \text{ in } \Delta} \mathit{swaps\text{-}sub}(\overset{\exists}{\tau}') \\ \mathit{sup}(v) \mapsto \displaystyle\bigsqcup_{v::>\overset{\exists}{\tau}' \text{ in } \Delta} \mathit{swaps\text{-}sup}(\overset{\exists}{\tau}') \end{cases} \\
\qquad\qquad \textbf{in} \displaystyle\bigsqcup_{i \text{ in } 1 \text{ to } n} \begin{cases} \mathit{swaps\text{-}sub}(\overset{\exists}{\tau}_i) \\ \mathit{swaps\text{-}sup}(\overset{\exists}{\tau}_i) \\ \{\mathit{const} \mapsto 0\} \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathit{swaps\text{-}sup}(\overset{\exists}{\tau}) \\
= \textbf{case } \overset{\exists}{\tau} \textbf{ of} \\
\quad v \mapsto \{\mathit{sup}(v) \mapsto 0\} \\
\quad \exists\Gamma:\Delta(\Delta).\ C\langle\ldots\rangle \mapsto \textbf{replace } v \textbf{ from } \Gamma \\
\qquad\qquad\qquad \textbf{using} \begin{cases} \mathit{sub}(v) \mapsto \{\mathit{const} \mapsto \infty\} \\ \mathit{sup}(v) \mapsto \{\mathit{const} \mapsto \infty\} \end{cases} \\
\qquad\qquad\qquad \textbf{in} \bigsqcup \begin{cases} \displaystyle\bigsqcup_{v<::\overset{\exists}{\tau}' \text{ in } \Delta} \mathit{swaps\text{-}sup}(\overset{\exists}{\tau}') \\ 1 + \displaystyle\bigsqcup_{v::>\overset{\exists}{\tau}' \text{ in } \Delta} \mathit{swaps\text{-}sub}(\overset{\exists}{\tau}') \end{cases}
\end{array}$$

Figure 36. Definition of the key height functions

would be ∞. Thus this requirement prevents types which have an indirectly infinite height.

B.1.3 Height Functions

In order to prove termination, we define two key mutually recursive height functions shown in Figure 36. They operate on the lattice of partial functions to the coinductive variant of natural numbers (i.e. extended with ∞ where $1 + \infty = \infty$). The domain of the partial function for the height of $\overset{\exists}{\tau}$ is a subset of $\mathit{sub}(v)$ and $\mathit{sup}(v)$ for v free in $\overset{\exists}{\tau}$ as well as *const*. Note that the use of partial maps is significant since 1 plus the completely undefined map results in the undefined map whereas 1 plus $\{\mathit{sub}(v) \mapsto 0\}$ is $\{\mathit{sub}(v) \mapsto 1\}$. This means that $\mathit{map} + (n + \mathit{map}')$ might not equal $n + (\mathit{map} + \mathit{map}')$ since the two partial maps may be defined on different domain elements. Using this lattice, the function $\mathit{swaps\text{-}sub}(\overset{\exists}{\tau})$ indicates the height of $\overset{\exists}{\tau}$ should it be the subtype, while $\mathit{swaps\text{-}sup}(\overset{\exists}{\tau})$ indicates the height of $\overset{\exists}{\tau}$ should it be the supertype. The functions are mutually recursive because each time a swap occurs the role of $\overset{\exists}{\tau}$ switches.

The partial map indicates the largest number of swaps in $\overset{\exists}{\tau}$ that can take place such that, for $\mathit{sub}(v)$, the subtype becomes v, for $\mathit{sup}(v)$, the supertype becomes v, and for *const*, the subtype becomes a non-variable. For example, the extended natural number for *const* of *swaps-sup*(`List<? super List<List<? super Integer>>>`) is 1 greater than that of *swaps-sub*(`List<List<? super Integer>>`) which is at least greater than that of *swaps-sup*(`List<? super Integer>`) which is 1 greater than that of *swaps-sub*(`Integer`) which is 0. Thus, if the supertype is `List<? super List<List<? super Integer>>>` then the subtype can become a non-variable provided at most 2 swaps in the type have been processed. If on the other hand `Map<? super Integer,? super Integer>` is the supertype then the subtype can become a non-variable after at most 1 swap in the type since the two `? supers` occur side-by-side rather than nested.

The operation $\textbf{replace } d \textbf{ from } \mathcal{D} \textbf{ using } d \mapsto \mathit{map}_d \textbf{ in } \mathit{map}'$ results in $\mathit{map}' \bigsqcup_{d \text{ in } \mathcal{D}} (\mathit{map}'(d) + \mathit{map}_d)$, where an undefined value plus anything results in the completely undefined partial mapping. This acts somewhat like substitution. For example, $\textbf{replace } \mathit{sub}(\mathtt{X}) \textbf{ using } \{\mathit{sub}(\mathtt{Y}) \mapsto 2\} \textbf{ in } \{\mathit{sub}(\mathtt{X}) \mapsto 1, \mathit{sub}(\mathtt{Y}) \mapsto 1\}$ results in $\{\mathit{sub}(\mathtt{Y}) \mapsto 3\}$. The idea is that the **using** clause indicates that $\mathit{sub}(\mathtt{X})$ is at most $2 + \mathit{sub}(\mathtt{Y})$ and the **in** clause indicates the result is at most $1 + \mathit{sub}(\mathtt{X})$ or $1 + \mathit{sub}(\mathtt{Y})$, from which we deduce the result is at most $3 + \mathit{sub}(\mathtt{Y})$.

The operation $\textbf{fix } d \textbf{ from } \mathcal{D} \textbf{ using } d \mapsto \mathit{map}_d \textbf{ in } \mathit{map}'$ determines the least $\{\overline{\mathit{map}}_d\}_{d \text{ in } \mathcal{D}}$ with each $\overline{\mathit{map}}_d$ greater than or equal to $\textbf{replace } d' \textbf{ from } \mathcal{D} \textbf{ using } d' \mapsto \overline{\mathit{map}}_{d'} \textbf{ in } \mathit{map}_d$ and returns $\textbf{replace } d \textbf{ from } \mathcal{D} \textbf{ using } d \mapsto \overline{\mathit{map}}_d \textbf{ in } \mathit{map}'$. Thus, the **fix** operation produces the least fixpoint for the **replace** operation.

Lemma 1. *Given two sets of domain elements $\mathcal{D}$ and $\mathcal{D}'$ and two families of partial maps $\{\mathit{map}_d\}_{d \text{ in } \mathcal{D}}$ and $\{\mathit{map}'_{d'}\}_{d' \text{ in } \mathcal{D}'}$ and any partial map $\overline{\mathit{map}}$, if no map_d maps any domain element d' in $\mathcal{D}'$ to an extended natural number, then*

$$\begin{array}{ll}
\textbf{fix} \begin{cases} d \textbf{ from } \mathcal{D} \\ d' \textbf{ from } \mathcal{D}' \end{cases} & \textbf{fix } d \textbf{ from } \mathcal{D} \\
 & \textbf{using } d \mapsto \mathit{map}_d \\
\textbf{using} \begin{cases} d \mapsto \mathit{map}_d \\ d' \mapsto \mathit{map}'_{d'} \end{cases} = & \textbf{in fix } d' \textbf{ from } \mathcal{D}' \\
 & \quad \textbf{using } d' \mapsto \mathit{map}'_{d'} \\
\textbf{in } \overline{\mathit{map}} & \quad \textbf{in } \overline{\mathit{map}}
\end{array}$$

Proof. Follows from the fact that, under the assumption, the following can be easily shown to hold for any $\widehat{\mathit{map}}$:

$$\begin{array}{ll}
\textbf{replace} \begin{cases} d \textbf{ from } \mathcal{D} \\ d' \textbf{ from } \mathcal{D}' \end{cases} & \textbf{replace } d \textbf{ from } \mathcal{D} \\
 & \textbf{using } d \mapsto \mathit{map}_d \\
\textbf{using} \begin{cases} d \mapsto \mathit{map}_d \\ d' \mapsto \textbf{replace } d \textbf{ from } \mathcal{D} \\ \qquad \textbf{using } d \mapsto \mathit{map}_d \\ \qquad \textbf{in } \mathit{map}'_{d'} \end{cases} = & \textbf{in replace } d' \textbf{ from } \mathcal{D}' \\
 & \quad \textbf{using } d' \mapsto \mathit{map}'_{d'} \\
\textbf{in } \widehat{\mathit{map}} & \quad \textbf{in } \widehat{\mathit{map}}
\end{array}$$

□

Lemma 2. *Given a set of domain elements $\mathcal{D}$ and a family of partial maps $\{\mathit{map}_d\}_{d \text{ in } \mathcal{D}}$, if the preorder reflection of the relation "map_d maps d' to 0" holds for all d and d' in $\mathcal{D}$ and no map_d maps a d' to a non-zero extended natural number for d and d' in $\mathcal{D}$, then for any partial map map', the following holds:*

$$\begin{array}{ll}
 & \textbf{replace } d \textbf{ from } \mathcal{D} \\
\textbf{fix } d \textbf{ from } \mathcal{D} & \textbf{using } d \mapsto \textbf{replace } d' \textbf{ from } \mathcal{D} \\
\textbf{using } d \mapsto \mathit{map}_d = & \qquad \textbf{using } d' \mapsto \varnothing \\
\textbf{in } \mathit{map}' & \qquad \textbf{in } \displaystyle\bigsqcup_{d'' \text{ in } \mathcal{D}} \mathit{map}_{d''} \\
 & \textbf{in } \mathit{map}'
\end{array}$$

Proof. Easily shown through tedious calculation. □

Lemma 3. *Given a finite set of domain elements $\mathcal{D}$ and a family of partial maps $\{\mathit{map}_d\}_{d \text{ in } \mathcal{D}}$, define $d \ll^{\phi}_{\mathit{map}} d'$ as "$\mathit{map}_{d'}$ maps d to an extended natural number i such that $\phi(i)$ holds". If $\ll^{>0}_{\mathit{map}}$ is well-founded on $\mathcal{D}$ modulo the equivalence coreflection of the preorder reflection of $\ll^{\mathbf{true}}_{\mathit{map}}$ then $\textbf{fix } d \textbf{ from } \mathcal{D} \textbf{ using } d \mapsto \mathit{map}_d \textbf{ in } \mathit{map}'$ is fix_n where fix is defined inductively by the following:*

$$\begin{array}{ll}
 & \textbf{replace } d \textbf{ from } \mathcal{E}_{i+1} \\
 & \textbf{using } d \mapsto \textbf{replace } d' \textbf{ from } \mathcal{E}_{i+1} \\
\mathit{fix}_0 = \mathit{map}' \quad \mathit{fix}_{i+1} = & \qquad \textbf{using } d' \mapsto \varnothing \\
 & \qquad \textbf{in } \displaystyle\bigsqcup_{d'' \text{ in } \mathcal{E}_{i+1}} \mathit{map}_{d''} \\
 & \textbf{in } \mathit{fix}_i
\end{array}$$

where $\{\mathcal{E}_i\}_{i \text{ in } 1 \text{ to } n}$ is any toposort (ignoring self-loops) according to $\ll^{\mathbf{true}}_{\mathit{map}}$ of the equivalence classes of the equivalence coreflection of the preorder reflection of $\ll^{\textbf{true}}_{map}$ (i.e. the strongly connected components of $\ll^{\textbf{true}}_{map}$).

Proof. A toposort (ignoring self-loops) exists because $\ll^{>0}_{map}$ is assumed to be well-founded modulo the equivalence coreflection of the preorder reflection of $\ll^{\textbf{true}}_{map}$. So what has to be shown is that the fixpoint is equivalent to the inductively defined value.

We prove by induction on i that the following holds:

$$\mathit{fix}_i = \textbf{fix } d \textbf{ from } \bigcup_{j \text{ in } 1 \text{ to } i} \mathcal{E}_j \textbf{ using } d \mapsto \mathit{map}_d \textbf{ in } \mathit{map}'$$

This obviously holds for 0 since the set $\bigcup_{j \text{ in } 1 \text{ to } 0} \mathcal{E}_j$ is empty and fix_0 is defined to be map'.

For $i+1$ we rely on our assumptions and our lemmas. First, by definition of $\ll^{\textbf{true}}_{map}$ and toposort, for any domain element d in $\mathcal{E}_{i+1}$, map_d maps no domain element in $\bigcup_{j \text{ in } 1 \text{ to } i} \mathcal{E}_j$ to an extended natural number. Thus by Lemma 1 and then by inductive assumption we have

$$\begin{array}{l}\textbf{fix } d \textbf{ from } \displaystyle\bigcup_{j \text{ in } 1 \text{ to } i+1} \mathcal{E}_j \\ \textbf{using } d \mapsto \mathit{map}_d \\ \textbf{in } \mathit{map}'\end{array} = \begin{array}{l}\textbf{fix } d \textbf{ from } \mathcal{E}_{i+1} \\ \textbf{using } d' \mapsto \mathit{map}_d \\ \textbf{in fix } d' \textbf{ from } \displaystyle\bigcup_{j \text{ in } 1 \text{ to } i} \mathcal{E}_j \\ \quad \textbf{using } d \mapsto \mathit{map}_{d'} \\ \quad \textbf{in } \mathit{map}'\end{array} = \begin{array}{l}\textbf{fix } d \textbf{ from } \mathcal{E}_{i+1} \\ \textbf{using } d \mapsto \mathit{map}_d \\ \textbf{in } \mathit{fix}_i\end{array}$$

Note that we are trying to prove the left value is equal to fix_{i+1}, which we can now do by proving the right value is equal to fix_{i+1}. Since $\mathcal{E}_{i+1}$ is defined to be an equivalence class of the equivalence coreflection of the preorder reflection of $\ll^{\textbf{true}}_{map}$, and since our well-foundedness assumption implies $d \ll^{>0}_{map} d'$ does not hold for any d and d' in $\mathcal{E}_{i+1}$, we can apply Lemma 2 to conclude that

$$\begin{array}{l}\textbf{fix } d \textbf{ from } \mathcal{E}_{i+1} \\ \textbf{using } d \mapsto \mathit{map}_d \\ \textbf{in } \mathit{fix}_i\end{array} = \begin{array}{l}\textbf{replace } d \textbf{ from } \mathcal{E}_{i+1} \\ \textbf{using } d \mapsto \textbf{replace } d' \textbf{ from } \mathcal{E}_{i+1} \\ \qquad \textbf{using } d' \mapsto \varnothing \\ \qquad \textbf{in } \displaystyle\bigsqcup_{d'' \text{ in } \mathcal{E}_{i+1}} \mathit{map}_{d''} \\ \textbf{in } \mathit{fix}_i\end{array} = \mathit{fix}_{i+1}$$

Lastly, since $\mathcal{D}$ equals $\bigcup_{i \text{ in } 1 \text{ to } n} \mathcal{E}_i$, we have that

$$\textbf{fix } d \textbf{ from } \mathcal{D} \textbf{ using } d \mapsto \mathit{map}_d \textbf{ in } \mathit{map}' = \mathit{fix}_n$$

□

B.1.4 Inheritance

Our algorithm assumes the presence of an inheritance table. This inheritance must satisfy a requirement in order for our algorithm to terminate.

Inheritance Requirement. For all classes C and D, whenever C<P_1, …, P_m> is a subclass of D<$\overset{\exists}{\tau}_1$, …, $\overset{\exists}{\tau}_n$> and $\exists\Gamma : \Delta(\Delta)$. $\overset{\exists}{\tau}$ is a subexpression of $\overset{\exists}{\tau}_i$ for some i in 1 to n, then Δ must not contain any constraints of the form $v ::> \overset{\exists}{\tau}'$.

Lemma 4. *Given the Inheritance Requirement, for all classes C and D, if C<P_1, …, P_m> is a subclass of D<$\overset{\exists}{\tau}_1$, …, $\overset{\exists}{\tau}_n$> then the following holds:*

$$\bigsqcup_{i \text{ in } 1 \text{ to } n} \begin{cases}\mathit{swaps\text{-}sub}(\overset{\exists}{\tau}_i) \\ \mathit{swaps\text{-}sup}(\overset{\exists}{\tau}_i)\end{cases} \sqsubseteq \{\mathit{const} \mapsto 0\} \bigsqcup_{i \text{ in } 1 \text{ to } m} \begin{Bmatrix}\mathit{sub}(P_i) \mapsto 0 \\ \mathit{sup}(P_i) \mapsto 0\end{Bmatrix}$$

Proof. Easily proven by coinduction using the more general property that if $\overset{\exists}{\tau}$ satisfies the restriction in the Inheritance Requirement then

$$\mathit{swaps\text{-}sub}(\overset{\exists}{\tau}) \sqcup \mathit{swaps\text{-}sup}(\overset{\exists}{\tau}) \sqsubseteq \{\mathit{const} \mapsto 0\} \bigsqcup_{v \text{ free in } \overset{\exists}{\tau}} \begin{Bmatrix}\mathit{sub}(v) \mapsto 0 \\ \mathit{sup}(v) \mapsto 0\end{Bmatrix}$$

□

B.1.5 Recursion

Here we formalize the recursive behavior of our subtyping algorithm. For this, we define a four-part lexicographic ordering on judgements $\mathbb{\Gamma} : \Delta \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'$. Note that any many of the components use extended natural numbers, and for these we use the relation $1 + m \leq n$ (which in particular means ∞ is less than ∞ with respect to these metrics, which we deal with in Section B.1.7).

1. $$\begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \begin{cases}\mathit{sub}(v) \mapsto \displaystyle\bigsqcup_{v <:: \hat{\overset{\exists}{\tau}} \text{ in } \Delta} \mathit{swaps\text{-}sub}(\hat{\overset{\exists}{\tau}}) \\ \mathit{sup}(v) \mapsto \displaystyle\bigsqcup_{v ::> \hat{\overset{\exists}{\tau}} \text{ in } \Delta} \mathit{swaps\text{-}sup}(\hat{\overset{\exists}{\tau}})\end{cases} \\ \textbf{in } \mathit{swaps\text{-}sub}(\overset{\exists}{\tau}) + \mathit{swaps\text{-}sup}(\overset{\exists}{\tau}')\end{array}$$
   Note that $+$ on partial maps is defined such that it is commutative and distributes through **replace** and **fix**. For example, $\{x \mapsto 3, y \mapsto 5\} + \{y \mapsto 2, z \mapsto 1\}$ equals $\{x \mapsto 0, y \mapsto 7, z \mapsto 1\}$ rather than $\{y \mapsto 7\}$.
2. maximum depth of the free variables in $\mathit{access\text{-}sup}_{=0}(\overset{\exists}{\tau}')$ with respect to the relation "there exists $\hat{\overset{\exists}{\tau}}$ with $v' ::> \hat{\overset{\exists}{\tau}}$ in Δ and v free in $\mathit{access\text{-}sup}_{=0}(\hat{\overset{\exists}{\tau}})$"
3. subexpression on $\mathit{access\text{-}sup}_{=0}(\mathit{explicit}(\overset{\exists}{\tau}'))$
4. if $\overset{\exists}{\tau}$ is a variable v, then depth of v with respect to the reverse of the relation $<::$ in Δ on $\mathbb{\Gamma}$; else -1

Theorem 1. *Provided the Algorithm Requirements and Inheritance Requirement hold, each recursive premise of our subtyping algorithm is strictly less than the conclusion with respect to the above ordering.*

Proof. The proof is given in Figure 37. It makes use of the properties proven below. □

Lemma 5. *For all* $\overset{\exists}{\tau}$ *and* θ,

$$\mathit{swaps\text{-}sub}(\overset{\exists}{\tau}[\theta]) = \begin{array}{l}\textbf{replace } v \textbf{ from domain of } \theta \\ \textbf{using} \begin{cases}\mathit{sub}(v) \mapsto \mathit{swaps\text{-}sub}(\theta(v)) \\ \mathit{sup}(v) \mapsto \mathit{swaps\text{-}sup}(\theta(v))\end{cases} \\ \textbf{in } \mathit{swaps\text{-}sub}(\overset{\exists}{\tau})\end{array}$$

and

$$\mathit{swaps\text{-}sup}(\overset{\exists}{\tau}[\theta]) = \begin{array}{l}\textbf{replace } v \textbf{ from domain of } \theta \\ \textbf{using} \begin{cases}\mathit{sub}(v) \mapsto \mathit{swaps\text{-}sub}(\theta(v)) \\ \mathit{sup}(v) \mapsto \mathit{swaps\text{-}sub}(\theta(v))\end{cases} \\ \textbf{in } \mathit{swaps\text{-}sup}(\overset{\exists}{\tau})\end{array}$$

Proof. Easily proved from the definitions of *swaps-sub* and *swaps-sup* by coinduction and tedious calculation. □

Lemma 6. *Suppose all of the Algorithm Requirements hold. If swaps-sub*($\overset{\exists}{\tau}$) *maps sub*(v) *or sup*(v) *to i such that* $\phi(i)$ *holds, then* v *must be free in access-sub*$_\phi(\overset{\exists}{\tau})$. *Likewise, if swaps-sup*($\overset{\exists}{\tau}$) *maps sub*(v) *or sup*(v) *to i such that* $\phi(i)$ *holds, then* v *must be free in access-sup*$_\phi(\overset{\exists}{\tau})$.

Proof. We prove by coinduction (since freeness of type variables is a coinductive property when dealing with coinductive types). If $\overset{\exists}{\tau}$ is v, then *swaps-sub*($\overset{\exists}{\tau}$) maps *sub*(v) to 0 and *swaps-sup*($\overset{\exists}{\tau}$) maps *sup*(v) to 0 so that in either case i is 0, and *access-sub*($\overset{\exists}{\tau}$) and *access-sup*($\overset{\exists}{\tau}$) both equal *access-var*(v) which is v since by assumption ϕ holds for i which is 0 in both cases.

If on the other hand $\overset{\exists}{\tau}$ is $\exists\Gamma : \Delta(\Delta)$. C<$\overset{\exists}{\tau}_1$, …, $\overset{\exists}{\tau}_n$> then *swaps-sub* and *access-sub* recurse to the same subexpressions of $\overset{\exists}{\tau}$ and

There are four kinds of recursive premises and each component of the above metric is corresponds to one of these kinds:

1. Decreased by the recursive premise in SUB-EXISTS checking explicit lower bounds after substitution.
2. Decreased by the recursive premise in SUB-VAR checking assumed lower bounds on type variables.
3. Decreased by the recursive premise in SUB-EXISTS checking explicit upper bounds after substitution.
4. Decreased by the recursive premise in SUB-VAR checking assumed upper bounds on type variables.

For each recursive premise we have to prove it decreases the corresponding metric component and preserves or decreases all earlier metric components:

1. The conclusion at hand is $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_m\rangle <: \exists\Gamma' : \Delta'(\mathbb{\Delta}').\ D\langle\overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle$. From Lemma 1 we know that the following holds for any partial map *map* since type variables in Γ cannot be free in $\mathbb{\Delta}$ and since Δ cannot impose constraints on type variables in $\mathbb{\Gamma}$:

$$\begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma}, \Gamma \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{map} \end{array} = \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in fix } v' \textbf{ from } \Gamma \\ \quad \textbf{using} \left\{\begin{array}{lll} sub(v') & \mapsto & \displaystyle\bigsqcup_{v'<::\overset{\hat{\exists}}{\tau} \text{ in } \Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v') & \mapsto & \displaystyle\bigsqcup_{v'::>\overset{\hat{\exists}}{\tau} \text{ in } \Delta} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \quad \textbf{in } \textit{map} \end{array}$$

   Because of Algorithm Requirement 2, the second premise is guaranteed to construct an assignment θ of all variables in Γ' to subexpressions of any of the $\overset{\exists}{\tau}_i$ without any escaping type variables. Furthermore, from the definition of $\approx_\theta$ and the definition of *swaps-sub*, for each v in Γ' we know *swaps-sub*$(\theta(v)) \sqcup$ *swaps-sup*$(\theta(v))$ is less than or equal to *swaps-sub*$(\overset{\bar{\exists}}{\tau}_i[P_1{\mapsto}\overset{\exists}{\tau}_1, \ldots, P_m{\mapsto}\overset{\exists}{\tau}_m]) \sqcup$ *swaps-sup*$(\overset{\bar{\exists}}{\tau}_i[P_1{\mapsto}\overset{\exists}{\tau}_1, \ldots, P_m{\mapsto}\overset{\exists}{\tau}_m])$ for some i in 1 to n. Lemmas 4 and 5 and monotonicity of *swaps-sub* and *swaps-sup* then inform us that under our Inheritance Requirement, for each v in Γ', *swaps-sub*$(\theta(v)) \sqcup$ *swaps-sup*$(\theta(v))$ must be less than or equal to *swaps-sub*$(\overset{\exists}{\tau}_i) \sqcup$ *swaps-sup*$(\overset{\exists}{\tau}_i)$ for some i in 1 to m. So, from the above observations we can deduce that the following must hold for any v in Γ':

$$\begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma}, \Gamma \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{swaps-sub}(\theta(v)) \sqcup \textit{swaps-sup}(\theta(v)) \end{array} \sqsubseteq \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{swaps-sub}(\exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_m\rangle) \end{array}$$

   Since for this case we are processing an explicit lower-bound constraint, we only care that the above holds for *swaps-sup*$(\theta(v))$; knowing it also holds for *swaps-sub*$(\theta(v))$ will be useful for another case.
   For a specific lower-bound constraint v ::> $\overset{\bar{\exists}}{\tau}$ in Δ' being processed in SUB-EXISTS, we also have to show the following:

$$1 + \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma}, \Gamma \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{swaps-sub}(\overset{\bar{\exists}}{\tau}[\theta]) \end{array} \sqsubseteq \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{swaps-sup}(\exists\Gamma' : \Delta'(\mathbb{\Delta}').\ D\langle\overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle) \end{array}$$

   This follows from the first equality above and the definition of *swaps-sup* (specifically its treatment of explicit lower bounds).
   From the above two inequalities and the fact that $+$ on partial maps is commutative we get the following proving that the metric decreases in this case:

$$1 + \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma}, \Gamma \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta},\Delta} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{swaps-sub}(\overset{\bar{\exists}}{\tau}[\theta]) + \textit{swaps-sup}(\theta(v)) \end{array} \sqsubseteq \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \left\{\begin{array}{lll} sub(v) & \mapsto & \displaystyle\bigsqcup_{v<::\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ sup(v) & \mapsto & \displaystyle\bigsqcup_{v::>\overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{array}\right. \\ \textbf{in } \textit{swaps-sub}(\exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_m\rangle) + \textit{swaps-sup}(\exists\Gamma' : \Delta'(\mathbb{\Delta}').\ D\langle\overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle) \end{array}$$

Figure 37. Proof that the metric decreases during recursion in our subtyping algorithm

2. Since $\mathbb{\Gamma}$, $\mathbb{\Delta}$, and $\overset{\exists}{\tau}$ are the same for the conclusion and the recursive premise at hand, the first component of the metric stays the same or decreases due to the fixpoint property of **fix**.
   The second component decreases by its definition.
3. For the same reasons as earlier, we can deduce that the following must hold for any v in Γ':

$$\begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma}, \Gamma \\ \textbf{using} \begin{cases} \textit{sub}(v) \mapsto \bigsqcup_{v <:: \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}, \Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ \textit{sup}(v) \mapsto \bigsqcup_{v ::> \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}, \Delta} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{cases} \\ \textbf{in } \textit{swaps-sub}(\theta(v)) \end{array} \sqsubseteq \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \begin{cases} \textit{sub}(v) \mapsto \bigsqcup_{v <:: \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ \textit{sup}(v) \mapsto \bigsqcup_{v ::> \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{cases} \\ \textbf{in } \textit{swaps-sub}(\exists\Gamma : \Delta(\mathbb{\Delta}).\ C\langle \overset{\exists}{\tau}_1, \ldots, \overset{\exists}{\tau}_m \rangle) \end{array}$$

   For similar reasons, for any explicit upper bound $v <:: \overset{\bar{\exists}}{\tau}$ in Δ' we can deduce the following due to the definition of *swaps-sup* (specifically its treatment of explicit upper bounds):

$$\begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma}, \Gamma \\ \textbf{using} \begin{cases} \textit{sub}(v) \mapsto \bigsqcup_{v <:: \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}, \Delta} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ \textit{sup}(v) \mapsto \bigsqcup_{v ::> \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}, \Delta} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{cases} \\ \textbf{in } \textit{swaps-sup}(\overset{\bar{\exists}}{\tau}[\theta]) \end{array} \sqsubseteq \begin{array}{l}\textbf{fix } v \textbf{ from } \mathbb{\Gamma} \\ \textbf{using} \begin{cases} \textit{sub}(v) \mapsto \bigsqcup_{v <:: \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sub}(\overset{\hat{\exists}}{\tau}) \\ \textit{sup}(v) \mapsto \bigsqcup_{v ::> \overset{\hat{\exists}}{\tau} \text{ in } \mathbb{\Delta}} \textit{swaps-sup}(\overset{\hat{\exists}}{\tau}) \end{cases} \\ \textbf{in } \textit{swaps-sup}(\exists\Gamma' : \Delta'(\mathbb{\Delta}').\ D\langle \overset{\exists}{\tau}'_1, \ldots, \overset{\exists}{\tau}'_n \rangle) \end{array}$$

   Consequently the first component of the metric stays the same or decreases.
   The second component stays the same or decreases by its definition and the definition of *access-sup* (specifically its treatment of explicit upper bounds).
   The third component decreases by its definition and the definitions of *explicit* and *access-sup* (specifically their treatment of explicit upper bounds).
4. Since $\mathbb{\Gamma}$, $\mathbb{\Delta}$, and $\overset{\exists}{\tau}'$ are the same for the conclusion and the recursive premise at hand, the first component of the metric stays the same or decreases due to the fixpoint property of **fix**.
   The second component stays the same by its definition since $\mathbb{\Gamma}$, $\mathbb{\Delta}$, and $\overset{\exists}{\tau}'$ are the same for the conclusion and the recursive premise at hand.
   The third component stays the same by its definition since $\overset{\exists}{\tau}'$ is the same for the conclusion and the recursive premise at hand.
   The fourth component decreases by its definition since the assumed upper bound must either be a variable so the depth is decreased or be an existential type so this component decreases since -1 is less than any depth.

Figure 37. Proof that the metric decreases during recursion in our subtyping algorithm (continued)

likewise for *swaps-sup* and *access-sup* (note that *access* has more free variables than both *access-sub* and *access-sup*). For *swaps-sub* and *access-sub*, all recursions are with the same ϕ and the mapping for one of them must be i since Algorithm Requirement 7 enables us to assume both *swaps-sub*$(\overset{\exists}{\tau}_j)$ and *swaps-sup*$(\overset{\exists}{\tau}_j)$ maps *sub*(v') and *sup*(v') to 0 if anything for v' in Γ. As for *swaps-sup*, i must either come from a recursion for explicit upper bounds or from 1 plus a recursion for explicit lower bounds. In the former case, ϕ is the same for the corresponding recursion in *access-sup*. In the latter case, the corresponding recursion in *access-sup* uses $\lambda j.\phi(1+j)$ which accounts for the 1 plus in *swaps-sub* so that the coinductive invariant is maintained.

Thus by coinduction we have shown that free variables in *access* projections reflect mappings in swap heights assuming Algorithm Requirement 7 holds. □

Corollary 1. *Given $\overset{\exists}{\tau}$, if v is not free in access-sub$_{\textit{true}}(\overset{\exists}{\tau})$ then swaps-sub$(\overset{\exists}{\tau})$ does not map sub(v) or sup(v) to anything, and if v is not free in access-sup$_{\textit{true}}(\overset{\exists}{\tau})$ then swaps-sup$(\overset{\exists}{\tau})$ does not map sub(v) or sup(v) to anything.*

B.1.6 Context

As we pointed out in the paper, bounds on type variables in the context can produce the same problems as inheritance. As such, we require the initial context $\mathbb{\Gamma} : \mathbb{\Delta}$ for a subtyping invocation to satisfy our Context Requirement.

Define $v \ll^{\phi}_{\mathbb{\Delta}} v'$ as "there exists $\overset{\exists}{\tau}$ with either $v' <:: \overset{\exists}{\tau}$ in $\mathbb{\Delta}$ and v free in *access-sub*$_\phi(\overset{\exists}{\tau})$ or $v' ::> \overset{\exists}{\tau}$ in $\mathbb{\Delta}$ and v free in *access-sup*$_\phi(\overset{\exists}{\tau})$".

Context Requirement. $\mathbb{\Gamma}$ and $\mathbb{\Delta}$ must have finite length, $<::$ in $\mathbb{\Delta}$ must be co-well-founded on $\mathbb{\Gamma}$, the relation "there exists $\overset{\exists}{\tau}$ with $v' ::> \overset{\exists}{\tau}$ in $\mathbb{\Delta}$ and v free in *access-sup*$_{\textbf{true}}(\overset{\exists}{\tau})$" is well-founded on $\mathbb{\Gamma}$, and $\ll^{<0}_{\mathbb{\Delta}}$ is well-founded on $\mathbb{\Gamma}$ modulo the equivalence coreflection of the preorder reflection of $\ll^{\textbf{true}}_{\mathbb{\Delta}}$ on $\mathbb{\Gamma}$.

B.1.7 Well-foundedness

In Section B.1.5 we identified a metric which decreases as our subtyping algorithm recurses. However, this does not guarantee termination. For that, we need to prove that the metric is well-founded at least for the contexts and existential types we apply our algorithm to. We do so in this section.

Theorem 2. *Provided the Algorithm Requirements and Context Requirement hold, the metric presented in Section B.1.5 is well-founded.*

Proof. Since we used a lexicographic ordering, we can prove well-foundedness of the metric by proving well-foundedness of each component.

1. This component is well-founded provided the partial map maps *const* to a natural number (i.e. something besides ∞). It is possible for *const* to be mapped to nothing at all, but by the

definition of *swaps-sub* this implies $\overset{\exists}{\tau}$ is a type variable and in this case this component is always preserved exactly so that termination is still guaranteed. Thus the key problem is proving that *const* does not map to ∞.
First we prove that, due to the Algorithm Requirements, for any $\overset{\exists}{\tau}$ the partial maps resulting from the height functions *swaps-sub*$(\overset{\exists}{\tau})$ and *swaps-sup*$(\overset{\exists}{\tau})$ never map any domain element to ∞. In fact, we will show that *swaps-sub*$(\overset{\exists}{\tau})$ and *swaps-sup*$(\overset{\exists}{\tau})$ never map any domain element to values greater than *height-sub*$(\overset{\exists}{\tau})$ and *height-sup*$(\overset{\exists}{\tau})$ respectively. Algorithm Requirement 6 then guarantees these values are not ∞ (since finiteness of *height-sub*$(\overset{\exists}{\tau})$ implies finiteness of *height-sup*$(\overset{\exists}{\tau})$ from the definition of *height-sup*). We do so by coinduction.
The cases when $\overset{\exists}{\tau}$ is a variable are obvious, so we only need to concern ourselves with when $\overset{\exists}{\tau}$ is of the form $\exists\Gamma : \Delta(\Delta).\ C\texttt{<}\overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\texttt{>}$. For *swaps-sub*, Algorithm Requirements 1, 5 and 7 and Lemma 3 allow us to replace the fixpoint with the join of the various partial maps after replacing *sub*(v) and *sup*(v) with $\varnothing$ so that *swaps-sub* coincides with *height-sub*. For *swaps-sup*, the only concern is the replacements with $\{const \mapsto \infty\}$; however from Algorithm Requirement 3 and Corollary 1 we know *sub*(v) and *sup*(v) are not mapped to anything and so the replacement does nothing. Thus *height-sub* and *height-sup* are upper bounds of *swaps-sub* and *swaps-sup* respectively.
Lastly we need to prove that the fixpoint over Γ and Δ is finite. Here we apply Lemmas 6 and 3 using the Context Requirement to express this fixpoint in terms of a finite number of replacements of finite quantities which ensures the result is finite. Thus this component is well-founded.

2. The invariant that "there exists $\overset{\hat{\exists}}{\tau}$ with $v' ::> \overset{\hat{\exists}}{\tau}$ in Δ and v free in *access-sup*$_{=0}(\overset{\hat{\exists}}{\tau})$" is well-founded on Γ holds initially due to the Context Requirement and is preserved through recursion due to Algorithm Requirement 5. In light of this, this component is clearly well-founded.
3. This component is well-founded due to Algorithm Requirement 1 and the definition of *access-sup*.
4. The invariant that $<::$ in Δ is co-well-founded on Γ holds initially due to the Context Requirement and is preserved through recursion due to Algorithm Requirement 5, and so this component is well-founded.

□

Note that most of the Algorithm Requirements are used solely to prove well-foundedness of this metric. In particular, many are used to prove that the first component is well-founded. We have provided one set of requirements that ensure this property, but other sets would work as well. For example, requiring all types to be finite allows Algorithm Requirement 5 to be relaxed to require well-foundedness rather than complete absence of free variables in *access-cxt*(Δ). However, our set of requirements is weak enough to suit many purposes including wildcards in particular.

B.1.8 Termination for Subtyping Our Existential Types

We have finally built up the set of tools to prove that our subtyping algorithm terminates under our requirements. Now we simply need to put the pieces together.

Theorem 3. *Given the Algorithm Requirements, Termination Requirement, and Context Requirement, the algorithm prescribed by the rules in Figure 7 terminates.*

Proof. By Theorems 1 and 2 we have that the recursion in the algorithm is well-founded, thus we simply need to ensure that all the intermediate steps terminate. The only intermediate step in SUB-VAR is the enumeration through the constraints in Δ which terminates since Δ is always finite due to the Context Requirement and Algorithm Requirement 4. As for SUB-EXISTS, the construction of θ terminates since $\approx_\theta$ recurses on the structure of *unifies*$(\overset{\exists}{\tau}'_i)$ for each i in 1 to n which is always finite since *explicit*$(\overset{\exists}{\tau}')$ is finite by Algorithm Requirement 1, and due to that same requirement there are only a finite number of recursive calls to check explicit constraints in Δ'. Thus the subtyping algorithm terminates. □

B.2 Termination for Wildcard Types

We just laid out a number of requirements on our existential types in order to ensure termination of subtyping, and in Section A.3 we showed how to translate wildcard types to our existential types. Here we prove that the existential types resulting from this translation satisfies our requirements supposing the Inheritance Restriction and Parameter Restriction from Section 4.4 both hold. The rules in Figure 8 are the result of combining the translation from wildcards to our existential types with (a specialized simplification of) the subtyping rules for our existential types, so this section will prove that the algorithm prescribed by these rules always terminates. Later we will demonstrate that these rules are sound and complete with respect to the rules in Figure 30 corresponding to the Java language specification.

Algorithm Requirement 4 holds because wildcard types are finite and the explicit portion of a translated wildcard type corresponds directly with the syntax of the original wildcard type.

Algorithm Requirement 2 holds because any bound variable in a translated wildcard type corresponds to a wildcard in the wildcard type and so will occur as a type argument to the class in the body.

Algorithm Requirement 3 holds because explicit constraints on wildcard cannot reference the type variable represented by that wildcard or any other wildcard.

Algorithm Requirement 4 holds because the set of constraints is simply the union of the explicit and implicit constraints on the wildcards, both of which are finite.

Algorithm Requirement 5 holds because Java does not allow type parameters to be bounded above by other type parameters in a cyclic manner, the Parameter Restriction ensures that any type variable occurring free in *access-cxt*(Δ) must be from another type argument to the relevant class and so must not be a bound type variable, and lastly because Java does not allow implicit lower bounds on wildcards and explicit lower bounds cannot reference the type variables represented by wildcards.

Algorithm Requirement 6 holds because the Parameter Restriction ensures the height is not increased by implicit constraints.

Algorithm Requirement 7 holds because types in the body cannot reference type variables corresponding to wildcards unless the type is exactly one such type variable.

The Inheritance Requirement is ensured by the Inheritance Restriction, and the Context Requirement is ensured by the Parameter Restriction.

C. Soundness

Here we give an informal argument of soundness of traditional existential types and subtyping. Then we show formally that subtyping is preserved by translation from our existential types to traditional existential types under new requirements. Lastly we show that subtyping is preserved by translation from wildcard types to our existential types and how wildcard types fulfill the new requirements, thus suggesting that subtyping of wildcards is sound. Note that this last aspect actually proves that our subtyping algorithm for wildcard types is complete with respect to the Java language specification; later we will show it is sound with respect to the Java language specification.

C.1 Soundness for Traditional Existential Types

Although we offer no proof that our application of traditional existential types is sound, we do offer our intuition for why we expect this to be true. In particular, existing proofs of soundness have assumed finite versions of existential types and proofs [5], so the key challenge is adapting these proofs to the coinductive variation we have presented here.

Suppose a value of static type $\overset{\top}{\tau}$ is a tuple $\langle v, \overset{\top}{\tau}_v, p\rangle$ where v is an instance of runtime type $\overset{\top}{\tau}_v$ (which is a class) and p is a proof of $\overset{\top}{\tau}_v <: \overset{\top}{\tau}$. Of course, since this is Java all the types erase, but for showing soundness its convenient to think of the types and proofs as usable entities. For example, when invoking a method on a value $\langle v, \overset{\top}{\tau}_v, p\rangle$ of static type C<$\overset{\overline{\top}}{\tau}_1$, …, $\overset{\overline{\top}}{\tau}_n$>, the execution needs to know that the v-table for v has the expected C method; in particular, it needs to use the fact that $\overset{\top}{\tau}_v$ is a subclass of C<$\overset{\overline{\top}}{\tau}_1$, …, $\overset{\overline{\top}}{\tau}_n$>. Supposing we put the subtyping rules for assumption, reflexivity, and transitivity aside for now, then the proof p can only exist if the class $\overset{\top}{\tau}_v$ is a subclass of C<$\overset{\overline{\top}}{\tau}_1$, …, $\overset{\overline{\top}}{\tau}_n$> since Rule 1 is the only rule that can apply when both the subtype and supertype are classes. Thus, having such a proof p ensures that the method invocation is sound.

Another major operation the language needs to do is open existential types (i.e. wildcard capture). Thus, given a value $\langle v, \overset{\top}{\tau}_v, p\rangle$ of static type $\exists\Gamma : \Delta.\ \overset{\top}{\tau}$, the language needs to be able to get values for the bound variables Γ and proofs for the properties in Δ. Again, the proof p provides this information. Supposing we put the subtyping rules for assumption, reflexivity, and transitivity aside once more, then the proof p can only exist if there is an instantiation θ of the bound variables Γ, satisfying the constraints in Δ, since Rule 3 is the only rule that can apply when the subtype is a class and the supertype is an existential type. Thus, by using this rule the proof p can provide the instantiations of the variables and the proofs for the properties, ensuring that the opening (or wildcard capture) is sound.

Notice that, in order to prove soundness of the fundamental operations, we keep destructing p; that is, we break p down into smaller parts such as a subclassing proof or instantiations of variables and subproofs for those variables. Since soundness of any individual operation never actually needs to do induction on the entire proof, it is sound for the proof to be infinite. Note that this is because execution in Java is a coinductive process (i.e. non-termination is implicit), so this line of reasoning does not apply to a language in which would like to have termination guarantees.

Now we return to the subtyping rules for assumption, reflexivity, and transitivity. First, we can put aside the subtyping rules for assumption because we can interpret any proof of $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\top}{\tau} <: \overset{\top}{\tau}'$ as a function from instantiations θ of $\mathbb{\Gamma}$ and proofs of $\mathbb{\Delta}[\theta]$ to proofs of $\varnothing : \varnothing \vdash \overset{\top}{\tau}[\theta] <: \overset{\top}{\tau}'[\theta]$. Second, we can put aside reflexivity, since for any type $\overset{\top}{\tau}$ well-formed in $\varnothing : \varnothing$ one can construct a proof of $\varnothing : \varnothing \vdash \overset{\top}{\tau} <: \overset{\top}{\tau}$ without using reflexivity (except for variables). Third, we can put aside transitivity, since, if one has proofs of $\varnothing : \varnothing \vdash \overset{\top}{\tau} <: \overset{\top}{\tau}'$ and $\varnothing : \varnothing \vdash \overset{\top}{\tau}' <: \overset{\top}{\tau}'$ not using assumption, reflexivity for non-variables, or transitivity, then one can construct a proof of $\varnothing \vdash \overset{\top}{\tau} <: \overset{\top}{\tau}''$ not using assumption, reflexivity for non-variables, or transitivity. Thus, the subtyping rules for assumption, reflexivity, and transitivity correspond to operations on canonical proofs rather than constructors on canonical proofs. This is why they cannot be used coinductively (although this restriction is only visible for transitivity).

C.1.1 Consistency and Null References

The above discussion does not account for the possibility of null references. Null references are problematic because when an existential type is opened its constraints are assumed to hold because we assume the existential type is inhabited, but this last assumption may not hold because the inhabitant at hand may just be a null reference and not contain any proofs that the constraints hold. Thus, after opening the wildcard type `Numbers<? super String>` we henceforth can prove that `String` is a subtype of `Number`, which enables a variety of unsound code since opening a wildcard type may not trigger a `NullPointerException` when the reference is actually `null`.

To address this, we introduce the notion of consistency of a traditional existential type (beyond just well formed) in a given context, formalized in Figure 38. The key is the use of the premise $\mathbb{\Gamma} : \mathbb{\Delta}(\Delta) \vdash \overset{\top}{\tau} <: \overset{\top}{\tau}'$ which is like normal subtyping except that the assumptions in Δ may only be used after an application of Rule 1. This prevents assumptions from being able to break the class hierarchy or add new constraints to variables already in $\mathbb{\Gamma}$ so that, in particular, any pair of types already well-formed in $\mathbb{\Gamma} : \mathbb{\Delta}$ that are subtypes after opening a consistent existential type are guaranteed to have be subtypes before opening that existential type. In this way, even if an inhabitant of a valid existential type is null, opening it cannot lead to an unsound subtypings since its constraints do not affect subtyping between types from the outside world.

C.2 Preserving Our Subtyping

For the most part it is clear how proofs of subtyping for our existential types can be translated to proofs of traditional subtyping between the translated types. The key challenge is that our subtyping proofs check the constraints in $\dot{\Delta}$ whereas traditional proofs check the constraints in Δ. This is where implicit constraints come into play, which is possibly the most interesting aspect of this problem from a type-design perspective. Again, we address this with the concept of validity for our existential types in a given context.

We do not define validity, rather we assume there is a definition of validity which satisfies certain requirements in addition to consistency as described in Section C.2.1. There are many definitions of validity that satisfy these requirements, each one corresponding to a different implicit-constraint system. This degree of flexibility allows our existential types to be applied to type systems besides wildcards as well as variations on wildcards.

The requirements are fairly simple. First, validity must be closed under opening: if $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \exists\Gamma : \Delta(\dot{\Delta}).\ C$<$\overset{\exists}{\tau}_1$, …, $\overset{\exists}{\tau}_n$> holds, then $\mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \overset{\exists}{\tau}_i$ must hold for each i in 1 to n and similarly each bound in either Δ or $\dot{\Delta}$ must be valid. Second, validity must be closed under substitution: if $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau}$ holds and θ is an assignment of the variables in $\mathbb{\Gamma}$ to types valid in $\mathbb{\Gamma}' : \mathbb{\Delta}'$ such that each constraint in $\mathbb{\Delta}[\theta]$ holds in $\mathbb{\Gamma}' : \mathbb{\Delta}'$, then $\mathbb{\Gamma}' : \mathbb{\Delta}' \vdash \overset{\exists}{\tau}[\theta]$ must hold as well. Lastly, but conceptually the most important, validity must ensure that implicit constraints hold provided explicit constraints hold: if $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \exists\Gamma : \Delta(\dot{\Delta}).\ C$<$\overset{\exists}{\tau}_1$, …, $\overset{\exists}{\tau}_m$> and $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \exists\Gamma' : \Delta'(\dot{\Delta}').\ D$<$\overset{\exists}{\tau}'_1$, …, $\overset{\exists}{\tau}'_n$> hold with C<P_1, …, P_m> a subclass of D<$\overset{\overline{\exists}}{\tau}_1$, …, $\overset{\overline{\exists}}{\tau}_n$>, then if θ assigns variables in Γ' to types valid in $\mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta$ with $\mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \overset{\overline{\exists}}{\tau}_i[P_1 \mapsto \overset{\exists}{\tau}_1, \ldots, P_m \mapsto \overset{\exists}{\tau}_m] \cong \overset{\exists}{\tau}'_i[\theta]$ such that $\dot{\Delta}'[\theta]$ holds in $\mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta$ then $\Delta'[\theta]$ holds in $\mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta$. Also, all types used in the inheritance hierarchy must be valid.

This last condition essentially says that any assignment of bound type variables that can occur during subtyping between valid types ensures that the constraints in Δ will hold whenever the constraints in $\dot{\Delta}$ hold and so do not need to be checked explicitly. As such, subtyping between valid types is preserved by translation to traditional existential types.

This last requirement can be satisfied in a variety of ways. The simplest is to say that a type is valid if $\dot{\Delta}$ is exactly the same as Δ, in which case our subtyping corresponds exactly to traditional subtyping. If a class hierarchy has the property that all classes are subclasses of `Object<>`, then a slightly more complex

$$\frac{}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v} \qquad \frac{\text{for all } i \text{ in } 1 \text{ to } n,\ \ \mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\top}{\tau}_i}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash C\langle\overset{\top}{\tau}_1,\ \ldots,\ \overset{\top}{\tau}_n\rangle} \qquad \frac{\begin{array}{c}\text{for all } \overset{\top}{\tau} <:: \overset{\top}{\tau}' \text{ in } \Delta',\ \overset{\top}{\tau} \text{ in } \Gamma \text{ or } \overset{\top}{\tau}' \text{ in } \Gamma \\ \text{for all } \overset{\top}{\tau} <::^{+} \overset{\top}{\tau}' \text{ in } \Delta, \overset{\top}{\tau} \text{ not in } \Gamma \text{ and } \overset{\top}{\tau}' \text{ not in } \Gamma \text{ implies } \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}(\Delta') \vdash \overset{\top}{\tau} <: \overset{\top}{\tau}' \\ \text{for all } \overset{\top}{\tau} <:: \overset{\top}{\tau}' \text{ in } \Delta,\ \ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \overset{\top}{\tau} \text{ and } \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \overset{\top}{\tau}' \\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \Delta \vdash \overset{\top}{\tau}\end{array}}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash \exists\Gamma : \Delta.\ \overset{\top}{\tau}}$$

Figure 38. Consistency for traditional existential types (in addition to well-formedness)

system could say a type is valid if Δ is Δ plus v <:: `Object<>` for each type variable v in Γ. Theoretically, since `String` has no strict subclasses, a system could also constrain any type variable explicitly extending `String` to also implicitly be a supertype of `String`, although this causes problems for completeness as we will discuss later. Nonetheless, there is a lot of flexibility here and the choices made can significantly affect the type system. Of course, at present we are concerned with Java, and so we will examine how this last requirement is ensured by wildcards in Section C.3.1.

C.2.1 Consistency

As with traditional existential types, our existential types also have a notion of consistency shown in Figure 39, although ours is a little bit stronger in order to address the more restricted set of subtyping rules. First, consistency requires that the constraints in Δ are implied by the constraints in Δ, which will be important in Section D.1. Second, consistency requires that types which can be bridged together by assumptions on variables are actually subtypes of each other. Because subtyping rules for our existential types are more restrictive, this in enough to ensure that consistency is preserved by translation to traditional existential types. This second requirement will also be important for transitivity as we will show in Section C.3.2. Lastly, consistency must hold recursively as one would expect.

C.3 Preserving Wildcard Subtyping

Now we show that the translation from wildcard types to our existential types preserves subtyping, which in particular implies that our algorithm is complete but also implies that subtyping of wildcards is sound supposing our informal reasoning for soundness of traditional existential types holds. We do this in two phases: first show that our subtyping rules in Figure 8 is preserved by translation, and then show that our rules subsumes those in the Java language specification as formalized in Figure 30. Note that here the notion of equivalence that we use is simply syntactic equality.

C.3.1 Preservation

In fact, preservation of subtyping is fairly obvious. The only significant differences in the rules are some simplifications made since we are using syntactic equality for equivalence and since bound variables cannot occur in explicit constraints. However, remember that subtyping is preserved by the subsequent translation to traditional existential types provided this translation to our existential types produces only valid existential types for some notion of validity. Finally implicit constraints and validity of wildcard types comes into play.

We formalize validity of wildcard types in Figure 40. This formalization differs from the Java language specification [6] in two ways. First, it is weaker than the requirements of the Java language specification since it makes no effort to enforce properties such as single-instantiation inheritance; however, this difference is not significant since any strengthening of the validity we show here will still be sufficient for our needs. Second, we prevent problematic constraints as discussed in Section 7.4. This second difference is used simply to ensure that the corresponding traditional existential type is valid and so opening is sound even with the possibility of null references.

The key aspect of validity of wildcard types is the enforcement of type-parameter requirements. Since the implicit constraints are taken from these requirements (and the fact that all classes extend `Object`), implicit constraints hold automatically after instantiation in subtyping whenever the subtype is a valid wildcard types. Since type validity uses subtyping without first ensuring type validity of the types being compared, it is possible to finitely construct what are essentially infinite proofs as discussed in Section 3.3.

The last check done in Figure 40 for types with wildcard type arguments is to make sure the combined implicit and explicit constraints are consistent with inheritance and the existing context. The judgement $\mathbb{\Gamma} : \mathbb{\Delta}(\Delta) \vdash \tau <: \tau'$ is like subtyping except that the assumptions in Δ may only be used after an application of rules for the judgement $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \tau \in \overset{?}{\tau}$. This ensures that valid wildcard types convert to consistent existential types.

C.3.2 Subsumption

Lastly we need to show that our subtyping rules subsume the official subtyping rules. For most of the official rules it should be clear how they can be translated to our rules since they are mostly subcomponents of our SUB-EXISTS and SUB-VAR rules. Reflexivity and transitivity are the only two difficult cases since we have no rules for reflexivity or transitivity.

Reflexivity can be shown for any of our existential types valid in any system by coinduction on the type. The case when the type is a variable is simply. The case when the type is existential uses the identity substitution and relies on the validity requirement that the constraints in $\mathbb{\Delta}$ must hold under the constraints in Δ.

Transitivity is difficult to prove since it has to be done concurrently with substitution. In particular, one must do coinduction on lists of subtyping proofs $\{\mathbb{\Gamma}_i : \mathbb{\Delta}_j \vdash \overset{\exists}{\tau}_i <: \overset{\exists}{\tau}'_i\}_{i \text{ in } 1 \text{ to } n}$ where each proof is accompanied with a list of constraint-satisfying substitutions $\{\theta^i_j : (\mathbb{\Gamma}^i_j : \mathbb{\Delta}^i_j) \to (\mathbb{\Gamma}^i_{j+1} : \mathbb{\Delta}^i_{j+1})\}_{j \text{ in } 1 \text{ to } k_i}$, with $(\mathbb{\Gamma}^i_1 : \mathbb{\Delta}^i_1) = (\mathbb{\Gamma}_i : \mathbb{\Delta}_i)$ and $(\mathbb{\Gamma}^i_{k_i} : \mathbb{\Delta}^i_{k_i}) = (\mathbb{\Gamma} : \mathbb{\Delta})$, such that $\overset{\exists}{\tau}'_i[\theta^i_1][\ldots][\theta^i_{k_i}] = \overset{\exists}{\tau}_{i+1}[\theta^{i+1}_1][\ldots][\theta^{i+1}_{k_{i+1}}]$ so that the proofs can be stitched together by substitution and transitivity to prove $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau}_1[\theta^1_1][\ldots][\theta^1_{k_1}] <: \overset{\exists}{\tau}'_n[\theta^n_1][\ldots][\theta^n_{k_n}]$. The recursive process requires a number of steps.

First, by induction we can turn a list of proofs with substitutions trying to show $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'$ into a path $\overset{\exists}{\tau} <::^{*} \overset{\overline{\exists}}{\tau}$ in $\mathbb{\Delta}$ followed by a (possibly empty) list of proofs with subtypings *between non-variables* trying to show $\mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\overline{\exists}}{\tau} <: \overset{\overline{\exists}}{\tau}'$ followed by a path $\overset{\overline{\exists}}{\tau}' ::>^{*} \overset{\exists}{\tau}'$ in $\mathbb{\Delta}$. A single proof can be turned into a path-proof-path by breaking it down into a finite number of applications of SUB-VAR followed by SUB-EXISTS or reflexivity on a variable. Given a list of path-proof-paths, we can use consistency to turn a $\overset{\exists}{\tau} <::^{+} v$ path followed by a $v ::>^{+} \overset{\exists}{\tau}'$ path into a proof of $\overset{\exists}{\tau} <: \overset{\exists}{\tau}'$ and repeat this process until the only $<::^{+}$ and $::>^{+}$ paths are at the extremities. Given a path-proof-path and a constraint-satisfying substitution, simply replace each assumption with the corresponding proof from the constraint-satisfying substitution and push this substitution onto the stack of substitutions for each of the

$$\frac{}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash v} \qquad \frac{\begin{array}{c}\text{for all } v <:: \overset{\exists}{\tau} \text{ in } \mathbf{\Delta},\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash v <: \overset{\exists}{\tau} \qquad \text{for all } v ::> \overset{\exists}{\tau} \text{ in } \mathbf{\Delta},\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau} <: v \\ \text{for all } v ::>^{+} \overset{\exists}{\tau} \text{ and } v <::^{+} \overset{\exists}{\tau}' \text{ in } \Delta,\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}' \\ \text{for all } v <:: \overset{\exists}{\tau} \text{ in } \Delta,\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau} \qquad \text{for all } v ::> \overset{\exists}{\tau} \text{ in } \Delta,\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau} \\ \text{for all } v <:: \overset{\exists}{\tau} \text{ in } \mathbf{\Delta},\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau} \qquad \text{for all } v ::> \overset{\exists}{\tau} \text{ in } \mathbf{\Delta},\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau} \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \mathbb{\Gamma}, \Gamma : \mathbb{\Delta}, \mathbf{\Delta} \vdash \overset{\exists}{\tau}_i\end{array}}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \exists \Gamma : \Delta(\mathbf{\Delta}).\ C\langle \overset{\top}{\tau}_1,\ \ldots,\ \overset{\top}{\tau}_n \rangle}$$

Figure 39. Consistency for our existential types (in addition to well-formedness)

$$\frac{}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash v} \qquad \frac{\begin{array}{c}\text{for all } i \text{ in } 1 \text{ to } n,\ C\langle P_1,\ \ldots,\ P_n \rangle \text{ requires } P_i \text{ to extend } \bar{\tau}^{i}_{j} \text{ for } j \text{ in } 1 \text{ to } k_i \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \mathbb{\Gamma} : \mathbf{\Delta} \vdash \overset{?}{\tau}_i \\ \textit{explicit}(\overset{?}{\tau}_1, \ldots, \overset{?}{\tau}_n) = \langle \Gamma; \mathbf{\Delta}; \tau_1, \ldots, \tau_n \rangle \\ \textit{implicit}(\Gamma; C; \tau_1, \ldots, \tau_n) = \mathbb{\Delta} \\ \text{for all } i \text{ in } 1 \text{ to } n \text{ and } j \text{ in } 1 \text{ to } k_i,\ \mathbb{\Gamma}, \Gamma : \mathbf{\Delta}, \mathbb{\Delta}, \mathbf{\Delta} \vdash \tau_i <: \bar{\tau}^{i}_{j}[P_1 \mapsto \tau_1, \ldots, P_n \mapsto \tau_n] \\ \text{for all } v ::> \hat{\tau} \text{ in } \mathbf{\Delta} \text{ and } v <::^{+} \hat{\tau}' \text{ in } \mathbb{\Delta} \text{ with } \hat{\tau}' \text{ not in } \Gamma,\ \mathbb{\Gamma}, \Gamma : \Delta(\mathbb{\Delta}, \mathbf{\Delta}) \vdash \hat{\tau} <: \hat{\tau}'\end{array}}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash C\langle \overset{?}{\tau}_1,\ \ldots,\ \overset{?}{\tau}_n \rangle}$$

$$\frac{}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \texttt{?}} \qquad \frac{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \texttt{? extends } \tau} \qquad \frac{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \tau}{\mathbb{\Gamma} : \mathbf{\Delta} \vdash \texttt{? super } \tau}$$

Figure 40. Validity for wildcard types (in addition to well-formedness)

pre-existing proofs and then use the above techniques to convert the resulting sequence of proofs into a path-proof-path.

Second, we can use the above technique to coinductively turn lists of proofs with substitutions into subtyping proofs. Given such a list, first turn it into a path-proof-path. Apply SUB-VAR for each of the assumptions in the two paths (since the paths are finite, this is a finite number of applications of SUB-VAR). If the list of proofs is empty, then build the two paths must converge at some type $\overset{\exists}{\tau}$ and so simply build the reflexivity proof for $\overset{\exists}{\tau}$. Otherwise we have a list of proofs of subtypings between non-variable existential types, which implies that the head of each proof is an application of SUB-EXISTS with constraint-satisfying substitutions $\{\theta_i\}_{i \text{ in } 1 \text{ to } n}$. As such, we can apply SUB-EXISTS using the substitution $\theta_n; \theta^n_1; \ldots; \theta^n_{k_n}; \ldots; \theta_1; \theta^1_1; \ldots; \theta^1_{k_1}$. To prove the constraints are satisfied by this substitution, recurse for each constraint using the appropriate stack of substitutions.

Thus our subtyping is reflexive and transitive so any two valid wildcard types that are subtypes in Java will convert to consistent and appropriately valid types in our existential type system which are also subtypes, and those will translate to consistent types in the traditional existential type system which are also subtypes.

D. Completeness

Subtyping for traditional existential types is extremely powerful, even allowing infinite proofs. Here we show that the translation from our existential types to traditional existential types reflects subtyping, meaning if two types translate to subtypes then they must themselves be subtypes. Similarly, we show that the conversion from wildcard types to our existential types to our existential types also reflects subtyping. This suggests that we cannot hope for a more powerful subtyping system based on existential types without changing either our notion of validity or our notion of equivalence.

D.1 Reflection of Traditional Subtyping

In order to prove that the translation from our existential types to traditional existential types reflects subtyping, we first need to define a notion of consistency for contexts analogous to consistency

$$\frac{\text{for all } v ::>^{+} \overset{\exists}{\tau} \text{ and } v <::^{+} \overset{\exists}{\tau}' \text{ in } \Delta,\ \mathbb{\Gamma} : \mathbf{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'}{\vdash \mathbb{\Gamma} : \mathbf{\Delta}}$$

Figure 41. Consistency for our contexts (in addition to well-formedness)

for our existential types. We do so in Figure 41, although we also implicitly assume that all types involved are consistent in the appropriate context.

For reflection of traditional subtyping, we require all types involved to be consistent as well as the context. Because transitivity may only be used finitely, any subtyping proof for translated types can be turned into a list of non-transitive and non-reflexive subtyping proofs. We can use the consistency of the context to make sure that assumptions are only in the beginning and end of this list like we did in Section C.3.2. Because the constructor $\exists \Gamma : \Delta.\ \overset{\top}{\tau}$ is inductive, we can turn the proofs between existential types into constraint-satisfying substitutions. Also, since consistency of $\exists \Gamma : \Delta(\mathbf{\Delta}).\ C\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n \rangle$ requires Δ to imply $\mathbf{\Delta}$, then any substitution satisfying Δ also satisfies $\mathbf{\Delta}$. Thus the same techniques that we employed in Section C.3.2 can be employed here with slight adjustments to deal with the fact that not all traditional existential types involved are translations of our existential types, so we do not repeat the presentation of that complicated coinductive process here.

Note that there is one last subtle issue with equivalence. Our unification rule is designed so that every variable is assigned to some type which occurs in the target type. However, traditional existential types do not have this restriction. As such, if equivalence meant "subtypes of each other" then traditional existential types could determine that `List<∃X : X <:: String, X ::> String. Map<X,X>>` is a subtype of `∃Y : ∅. List<∃X : X <:: String, X ::> String. Map<X,Y>>` by assigning `Y` to `String` even though `String` does not occur at the location corresponding to `Y`. In our system, `Y` would be assigned to `X` which would be invalid since otherwise `X` would escape its bound. To prevent this problem, we must require that any two types deemed equivalent must have the same set of free variables.

D.2 Reflection of Our Subtyping

It should be clear that the conversion from wildcard types to our existential types reflects subtyping using the rules described in Figure 8. Thus the challenge is to show that the subtyping system described in Figure 30 subsumes that described in Figure 8, the converse of what we showed in Section C.3.2. It is clear that SUB-VAR corresponds to an application of transitivity and assumption (or reflexivity). Similarly, SUB-EXISTS corresponds to wildcard capture followed by a transitive application of inheritance with inhabitance ($\in$). Thus the conversion reflects subtyping.

References

[1] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, 1998.

[2] Kim B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. The MIT Press, 2002.

[3] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *TOPLAS*, 25:225–290, March 2003.

[4] Nicholas Cameron and Sophia Drossopoulou. On subtyping, wildcards, and existential types. In *FTfJP*, 2009.

[5] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *ECOOP*, 2008.

[6] James Gosling, Bill Joy, Guy Steel, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley Professional, third edition, June 2005.

[7] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ECOOP*, 2002.

[8] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2007.

[9] Andrew Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *FOOL*, 2007.

[10] Daan Leijen. HMF: Simple type inference for first-class polymorphism. In *ICFP*, 2008.

[11] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL*, 1996.

[12] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1969.

[13] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, pages 135–151. Edinburgh University Press, 1969.

[14] Daniel Smith and Robert Cartwright. Java type inference is broken: Can we fix it? In *OOPSLA*, 2008.

[15] Alexander J. Summers, Nicholas Cameron, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. Towards a semantic model for Java wildcards. In *FTfJP*, 2010.

[16] Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in Java's type system. Technical report, University of California, San Diego, March 2011.

[17] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP*, 1999.

[18] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *FOOL*, 2005.

[19] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC*, 2004.

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

[21] Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In *APLAS*, 2009.