Taming Wildcards in Java's Type System [*]

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 [18].

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 [6, 16–18], 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 [20]. 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-

[*] 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.

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.

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 [2, 8, 13, 20]. In response to these challenges, researchers have explored several ways of fixing wildcards. The work by Smith and Cartwright [13] 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 [18]. `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 [16] 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 [2, 3, 6, 8, 17, 18, 20]. The wildcard `?` represents an existentially quantified type variable, so that `List<?>` is shorthand for the existentially quantified type `∃X. List<X>`. Existential quantification is dual to universal quantification; a `∀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 `∃X. List<X>` by instantiating `X` to `String`. In particular, any list can be used as an `∃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 `∃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 [7, 9, 10]). For example, `∃X extends Error. List<X>` subsumes `∃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 [5, 18] and is actually part of the specification for subtyping [5: Chapter 4.10.2].

2.3 Implicit Constraints

While users can explicitly constrain wildcards via the `extends` and `super` clauses, Java also imposes implicit constraints on wildcards to make them more convenient. For example, consider the following interfaces specializing `List` via F-bounded polymorphism [4]:

```
interface Numbers<P extends Number> extends List<P> {}
interface Errors<P extends Error> extends List<P> {}
```

If a user uses the type `Numbers<?>`, Java implicitly constrains the wildcard to be a subtype of `Number` [5: Chapter 5.1.10], saving the user the effort of expressing a constraint that always holds. The same is done for `Errors<?>`, so that `Errors<?>` is a subtype of `List<? extends Error>` [5: Chapter 4.10.2]. (Note that we will reuse these interfaces throughout this paper.)

Implicit constraints on wildcards are more than just a syntactic convenience though; they can express constraints that users cannot express explicitly. Consider the following interface declaration:

```
interface SortedList<P extends Comparable<P>> extends List<P> {}
```

If one uses the type `SortedList<?>`, Java implicitly constrains the wildcard, call it `X`, so that `X` is a subtype of `Comparable<X>` [5: Chapter 5.1.10]. However, it is impossible to state this constraint explicitly on the wildcard argument to `SortedList`. This is because the

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

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

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

Figure 1. Example of cyclic dependency in subtyping [8]

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

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

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

Figure 2. Example of acyclic proofs in subtyping [8]

explicit constraint would need to make a recursive reference to the wildcard, but wildcards are unnamed so there is no way to reference a wildcard in its own constraint. One might be tempted to encode the constraint explicitly as `SortedList<? extends Comparable<?>>`, but this does not have the intended semantics because Java will not correlate the wildcard for `Comparable` with the one for `SortedList`.

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 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<`$\exists$`X. List<X>>`, whereas it would represent simply $\exists$`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 [8] 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 [8, 19] 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 [5: 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 [20]. 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.

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 $\exists$`X` : `X` $<::$ `Infinite<?>`. `Infinite<X>`, but then the nested `Infinite<?>` needs to be recursively translated which

Java Wildcards

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

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

Steps of Proof
`C` $<\!\!\cdot\!:$ `Super<?,?>`
`Super<Super<?,?>,C>` $<\!\!\cdot\!:$ `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` $<\!\!\cdot\!:$ $\exists$`X′,Y′:Y′` $<::$ `X′. Super<X′,Y′>`
`Super<`$\exists$`X,Y:Y` $<::$ `X. Super<X,Y>,C>` $<\!\!\cdot\!:$ $\exists$`X′,Y′:Y′` $<::$ `X′. Super<X′,Y′>`
`C` $<\!\!\cdot\!:$ $\exists$`X,Y:Y` $<::$ `X. Super<X,Y>`
(repeats forever)

Figure 3. Example of an implicitly infinite subtyping proof

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 constraints 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

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

Using Declaration 2
`D` $<\!\!\cdot\!:$ `List<? super C<D>>`
`List<C<? extends List<? super C<D>>>>` $<\!\!\cdot\!:$ `List<? super C<D>>`
`C<D>` $<\!\!\cdot\!:$ `C<? extends List<? super C<D>>>`
`D` $<\!\!\cdot\!:$ `List<? super C<D>>`
Rejected (implicit assumption above is explicit here)

Figure 4. Example of the inconsistency of rejecting infinite 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 termi-

```
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`

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

Figure 6. Grammar of our existential types (coinductive)

SUB-EXISTS

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

SUB-VAR

$$
\frac{}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v <: v}
$$

$$
\frac{v <:: \overset{\exists}{\tau} \text{ in } \mathbb{\Delta} \qquad \mathbb{\Gamma} : \mathbb{\Delta} \vdash \overset{\exists}{\tau} <: \overset{\exists}{\tau}'}{\mathbb{\Gamma} : \mathbb{\Delta} \vdash v <: \overset{\exists}{\tau}'}
$$

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

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

nates 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 our technical report [15].

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 $\exists\Gamma : \Delta(\dot{\Delta}).\ \overset{\exists}{\tau}$. The constraints Δ are the constraints corresponding to traditional existential types, combining both the implicit and explicit constraints on wildcards. The constraints $\dot{\Delta}$ 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 we may do structural recursion and still have termination. However, the combined constraints Δ 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 $\dot{\Delta}'$ in the supertype are checked, whereas the combined implicit and explicit constraints Δ 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 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

$$\textsc{Sub-Exists}$$

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

$$\textsc{Sub-Var}$$

$$\frac{}{\mathbb{\Gamma} : \Delta \vdash v <: v}$$

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

Figure 8. Subtyping rules specialized for wildcards

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 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 $\mathring{\Delta}$ and $\dot{\Delta}$ 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 [8]. Their restriction prohibits contravariance altogether, whereas we only restrict its usage. Furthermore, as Kennedy and Pierce mention [8], 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 $\langle P_1$ `extends` $\tau_1,\ \ldots,\ P_n$ `extends` $\tau_n\rangle$, *every syntactic occurrence in τ_i of a type* $C\langle\ldots,$ `? super` $\tau,\ \ldots\rangle$ *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

```
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

`C<?>` $\mathrel{<\!\!\overset{?}{:}}$ `List<? extends List<? super C<?>>>`
    `C<?>` $\mapsto$ `C<X>` with `X` $<::$ `List<List<? extends List<? super C<?>>>>`
`C<X>` $\mathrel{<\!\!\overset{?}{:}}$ `List<? extends List<? super C<?>>>`
`X` $\mathrel{<\!\!\overset{?}{:}}$ `List<? super C<?>>`
`List<List<? extends List<? super C<?>>>>` $\mathrel{<\!\!\overset{?}{:}}$ `List<? super C<?>>`
`C<?>` $\mathrel{<\!\!\overset{?}{:}}$ `List<? extends List<? super C<?>>>`
(repeats forever)

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

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 [5: 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 [5: Chapter 4.10.2] provided all types are valid according to the Java language specification [5: Chapter 4.5].*[‡]

Proof. Here we only discuss the reasons for our restrictions; the full proofs can be found in our technical report [15]. 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 [8] would provide a sound and complete subtyping algorithm [13]. This is because Kennedy and Pierce built off the work by Viroli [19] 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 [8], 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

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

```
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)
X_1 <:: X_0	C<X_0,X_1> $\overset{?}{<:}$ List<? super C<?,?>>
Y_1 <:: Y_0	C<Y_0,Y_1> $\overset{?}{<:}$ List<? super C<List<X_1>,?>>
X_2 <:: List<X_1>	C<List<X_1>,X_2> $\overset{?}{<:}$ List<? super C<List<Y_1>,?>>
Y_2 <:: List<Y_1>	C<List<Y_1>,Y_2> $\overset{?}{<:}$ List<? super C<List<X_2>,?>>
X_3 <:: List<X_2>	C<List<X_2>,X_3> $\overset{?}{<:}$ List<? super C<List<Y_2>,?>>
Y_3 <:: List<Y_2>	C<List<Y_2>,Y_3> $\overset{?}{<:}$ List<? super C<List<X_3>,?>>
X_4 <:: List<X_3>	C<List<X_3>,X_4> $\overset{?}{<:}$ List<? super C<List<Y_3>,?>>
	(continue forever)

Figure 10. Example of expansion through implicit constraints

```
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

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 and Pierce's algorithm to wildcards [15]. However, these restrictions are significantly more complex than ours, and the adapted algorithm would be strictly more complex than ours.

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.

```
<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

Smith and Cartwright take a different approach to joining types. They extend the type system with union types [13]. 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.

5.2 Capture Conversion

Java has generic methods as well as generic classes [5: 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 [5: 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 [5: 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 [13]. 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 [13], 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

```
<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

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 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 [13], `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 [5: 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 [11, 12] 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

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

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` [5: Chapters 15.12.2.7 and 15.12.2.8] and Smith and Cartwright [5] 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 (in our technical report we formalize when a wildcard can be implicitly constrained [15]). 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 addition to making it difficult for users to predict which types will be accepted or rejected as illustrated in our technical report [15]). 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.

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 `∃X. Set<Entry<X,String>>` and `∃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 [5: Chapter 8.1.5] (in other words, prohibiting multiple-instantiation inheritance [8]). 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 [5: 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 `∃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

```
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>` $<\!\!\cdot^{?}\!:$ `D<? extends List<D<? extends C>>>`
`C` $<\!\!\cdot^{?}\!:$ `List<D<? extends C>>`
`List<D<? extends List<D<? extends C>>>>` $<\!\!\cdot^{?}\!:$ `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

`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 [5: 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.

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 [5: Chapters 4.5.1.1 and 4.10.2] and the formalization by Torgersen et al. [17] 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-

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

Figure 15. Definition of when two types join concretely

guments are subtypes of each other, as proposed by Smith and Cartwright [13]. 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 $\exists\Gamma:\Delta.\ C\texttt{<}\ldots\texttt{>}$,
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 $C\texttt{<}\tau_1,\ \ldots,\ \tau_n\texttt{>}$ (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 [5: 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 [5: 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.

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

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

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

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

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

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 [5: 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 [15]. 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 [14], 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

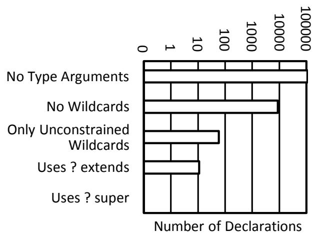

Figure 17. Wildcard usage in inheritance hierarchies

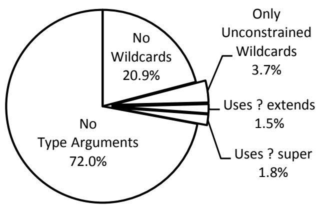

Figure 18. Wildcard usage in type-parameter constraints

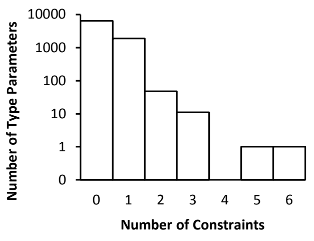

Figure 19. Number of constraints per type parameter

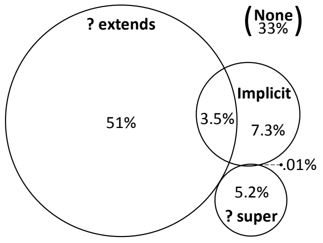

Figure 20. Distribution of constraints on wildcards

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 the technical report [15]. We encountered only one example of an implicitly infinite type. It was actually the same as class `Infinite` in Section 3.2; however, we investigated this further and determined this was actually an error which we easily corrected [15]. We also did a manual investigation for implicitly infinite proofs and found none. 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 [3], and Summers et al. gave semantics to wildcards under the same assumptions [14], 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.

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 [15]. 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 [8], 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 [20]. 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) [13], 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 [6, 16–18], wildcards are best formalized as existential types [2, 3, 6, 8, 17, 18, 20], and more precisely as coinductive existential types with coinductive subtyping proofs [15], 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 [8, 20]. 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 [15], 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 [8, 20]. 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 [5: 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.

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

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

[4] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, 1989.

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

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

[7] 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.

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

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

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

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

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

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

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

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

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

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

[18] 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.

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

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