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

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 with wildcards 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-
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<Integer>, yet operating on arbitrary lists was inconvenient because there was no form of variance. One had to define a method with a polymorphic variable X and a parameter of type List<X>, 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 to refer to all lists regardless of their elements. This can be especially limiting for parametric classes such as Class<X> 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<? extends Number> stands for a list whose elements have an arbitrary unknown static type. Types such as List<String>, List<Integer>, and 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 the contents and even get objects 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 Objects to a List<?> since it might represent a List<String> which only accepts Strings or a List<Integer> which only accepts Numbers.

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 Numbers. This is because one can get elements of the list and statically know they are Numbers, but one cannot add Numbers to the list since the list may actually represent a List<Integer> which does not accept arbitrary Numbers. Similarly, List<? super Number> is often used to indicate write-only lists. This time, one cannot get Numbers from the list since it may actually be a List<Object>, but one can add Numbers 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<? super Error>, but a List<? extends Error> can even be used as a List<Number> since Error is a subclass of Throwable. Similarly, a List<? super Throwable> can be used as a List<? super Throwable> which can be used as a List<Error> 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 subtypes 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<? super Number> represents an existential type list of Numbers which is often used to indicate read-only lists of Numbers, one cannot add arbitrary Numbers to a list since it might represent a List<String> which only accepts Strings or a List<Integer> which only accepts Numbers.

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 extends Number {
    List<? extends Number> extends List<?>(){
    }
    interface Errors extends Error{
        List<? extends Error> extends List<?>() {
    }
}
```

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 extends Comparable<<>> extends List<?>() {
    if (one uses the type SortedList<?>, Java implicitly constrains the wildcard, call it I, so that I is a subtype of Comparable<? extends List<?>() if one uses the type SortedList<?>, Java implicitly constrains the wildcard, call it I, so that I is a subtype of Comparable<? extends List<?>()
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 cor-
relate the wildcard for Comparable with the one for SortedList.

This means that implicit constraints offer more than just conve-
nience; they actually increase the expressiveness of wildcards, pro-
ducing a compromise between the friendly syntax of wildcards and
the expressive power of bounded existential types. However, this expres-
siveness 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 ∃Γ : Δ. τ. The set Γ is the set of bound
variables, and the set Δ is all constraints on those variables. Con-
straints in Δ have the form v :: τ' analogous to extends and
v :: τ analogous to super. Thus we denote the traditional exis-
tential type for the wildcard type SortedList<? extends Integer> as

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

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

We denote the traditional existential type for the wildcard type

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<? extends P> 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<? extends P>> stands for List< List< List<? extends P> > >, whereas it would represent simply
∃X : List<List<? extends P>> in a predicative system.

We have identified three sources of infinity due to impredica-
tivity of wildcards, in particular due to wildcards in the inheritance
hierarchy and in type-parameter constraints. The first source is in-
finitely many 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 a subtype of List<? extends P>?”
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<? extends P>. For this
to hold, C’s superclass List<List<? extends P>> must be a subtype of
List<? extends P> (Step 1). For this to hold, List<List<? extends P>>
must be List of some supertype of C, so C must be a subtype of
List<? extends P> (Step 2). This was our original question, though, so
whether C is a subtype of List<? extends P> actually depends on itself
non-trivially. This means that we can actually prove C is a subtype of
List<? extends P> provided we use an infinite proof.

The proof for Figure 1 repeats itself, but there are even subtype-
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 P<-> (which could just as easily be List<P> or Set<P>
without affecting the structure of the proof). This is known as ex-
pansive inheritance [8, 19] since the type parameter P expands to
the type argument P<-> corresponding to that same type parameter
P. Because of this expansion, the proof repeats itself every four
steps except with an extra P<-> layer so that the proof is acyclic.

Java rejects all infinite proofs [5: Chapter 4.10], and javac at-
ttempts to enforce this decision, rejecting the program in Figure 1
but suffering a stack overflow on the program in Figure 2. Thus,
either 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 inconsis-
tent 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 re-
cursive subtyping algorithm terminates because of this.

3.2 Implicitly Infinite Types
Wehr and Thiemann proved that subtyping of bounded impredica-
tive 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 deter-
mined that there are wildcard types that actually cannot be ex-
pressed 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 ex-
licit can actually result in an infinite traditional existential type.

Consider the following class declaration:

class Infinite<? extends Infinite<? extends Infinite<? extends Infinite<? extends Infinite<? extends Infinite<...>>>}> { }

The wildcard type Infinite<? extends Infinite<? extends Infinite<? extends Infinite<? extends Infinite<...>>>}> translates to an infinite traditional exis-
tential type because its implicit constraints must be made explicit.
In one step it translates to ∃X : I :: Infinite<? extends Infinite<? extends Infinite<...>>>, but
then the nested Infinite<? extends Infinite<? extends Infinite<...>>> needs to be recursively translated which
### Java Wildcards

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

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

Steps of Proof

- C <-> Super<?, ?>
- Super<? extends Super<?,?>, C> <-> Super<?, ?> (inheritance) (completes)

**Figure 3.** Example of an implicitly infinite subtyping proof

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 a valid Java uses subtyping which implicitly assumes all types involved are valid, potentially leading to an implicitly infinite proof. In Figure 3, `Super<Super<?, ?>, ?` is a valid type provided `?` is a subtype of `Super<?, ?>`. By inheritance, this reduces to whether `Super<Super<?, ?>, ?` is a subtype of `Super<?, ?>`. The implicit constraints on the wildcards in `Super<?, ?>` are not checked because Java implicitly assumes `Super<Super<?, ?>, ?` is a valid type. Thus the proof that `Super<Super<?, ?>, ?` is valid implicitly assumes that `Super<Super<?, ?>, ?` 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 repeats ad infinitum. Thus `Infinite<?>` is implicitly infinite. Interestingly, this type is actually inhabitable by the following class:

```java
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.

### Traditional Existential Types

```java
class C extends Super<∃ X, Y : Y -> X. Super<X, Y>, Super<X, Y>, C> {}
```

Is C a subtype of `∃ X, Y : Y -> X. Super<X, Y>, Super<X, Y>, C>`?

Steps of Proof

- C <-> ∃ X, Y : Y -> X. Super<X, Y>, Super<X, Y>, C
- Super<super<X, Y>, super<X, Y>, C> <-> Super<X, Y>, Super<X, Y>, C (inheritance)
  - (repeats forever)

**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 `D` is a valid type is finite. However, the second program is rejected because the proof that `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-
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 subtypeing rules for traditional existential types. Even without restrictions, our algorithm improves on the existing subtypeing 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 subtypeing 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 , in Figure 6.

Note that there are two sets of constraints so that we denote our existential types as : . The constraints are the constraints corresponding to traditional existential types, combining both the implicit and explicit constraints on wildcards. The constraints 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 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 means that is a subtype of in the context of type variables with constraints . 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 , since just applies assumed constraints on the type variable at hand. The first premise of examines the inheritance hierarchy to determine which, if any, invocations of that is a subclass of 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 in the supertype are checked, whereas the combined implicit and explicit constraints in the subtype are assumed. This separation is what enables termination and completeness.

We have no rule indicating that all types are subtypes of . 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 is a superclass of all classes and interfaces in Java and the fact that all variables in Java are implicitly constrained above by . 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 generating wildcards, we can specialize the algorithm to wildcards for a more direct solution. We present this specialization of our algorithm in Figure 8, with representing a Java type and representing a Java type argument which may be a (constrained) wildcard. The function 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, returns ( : Integer; ) returns . The function takes a list

Figure 5. Example of subtyping incorrectly rejected by javac

Figure 6. Grammar of our existential types (coinductive)

Figure 7. Subtyping rules for our existential types (inductive and coinductive definitions coincide given restrictions)
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(3, Numbers; 3) returns I ::= Number. Thus, applying explicit and then implicit accomplishes wildcard capture. Note that for the most part Δ and Δ combined act as Δ does in Figure 7.

4.4 Termination

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

Inheritance Restriction

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

Note that the programs leading to infinite proofs in Section 3.1 (and in the upcoming Section 4.5) violate our inheritance restriction. This restriction is most similar to a significant relaxation of the contravariance restriction that Kennedy and Pierce showed is the type argument corresponding to the explicit cast in the recursive calls. There are only two ways in which it...
class C<P, Q extends P> implements List<List<X>> super C<List<Q>>
{
    ...
}

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

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Subtyping (wildcard capture done automatically)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₁ &lt;&lt; Y₀</td>
<td>C&lt;X₁, Y₀&gt; &lt;&lt; List&lt;? super C&lt;List&lt;X₁&gt;,?&gt;,?&gt;&gt;</td>
</tr>
<tr>
<td>X₂ &lt;&lt; List&lt;X₁&gt;</td>
<td>C&lt;List&lt;X₁&gt;, Y₀&gt; &lt;&lt; List?&lt; super C&lt;List&lt;X₁&gt;,?&gt;,?&gt;&gt;</td>
</tr>
<tr>
<td>X₃ &lt;&lt; List&lt;X₂&gt;</td>
<td>C&lt;List&lt;X₂&gt;, Y₀&gt; &lt;&lt; List?&lt; super C&lt;List&lt;X₂&gt;,?&gt;,?&gt;&gt;</td>
</tr>
<tr>
<td>X₄ &lt;&lt; List&lt;X₃&gt;</td>
<td>C&lt;List&lt;X₃&gt;, Y₀&gt; &lt;&lt; List?&lt; super C&lt;List&lt;X₃&gt;,?&gt;,?&gt;&gt;</td>
</tr>
</tbody>
</table>

(continue forever)

Figure 10. Example of expansion through implicit constraints

```java
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 restric-
tions are significantly more complex than ours, and the adapted al-
gorithm 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(τ, τ’) function which
returns the most precise common supertype of τ and τ’. Unfor-
unately, 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<?>. 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 class-
room when it failed to type check the code in Figure 11. This sim-
ple program fails to type check because javac determines that the
type of (mValue ? trues : falses) is List<?> rather than the obvious

5.2 Capture Conversion

Java has generic methods as well as generic classes [5: Chap-
ter 8.4.4]. For example, the method getFirst in Figure 12 is generic
with respect to P. Java attempts to infer type arguments for invoca-
tions 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<? extends Number> and List<Integer> is just List<? super C<List<Integer>>>.
However, our investigations suggest that it may be impossible for an existen-
tial 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.

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. τ | τ’ 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
However, their approach is not strictly

Figure 12. Examples of capture conversion

List<? super Var>. In particular, javac’s algorithm may even fail to
return τ when both arguments are the same type τ.

Smith and Cartwright have already shown that
the method getFirst returns the most precise common supertype of
both arguments which we explained in Section 5.1. Their approach to type in-
ference improves on Java’s approach, their approach is not strictly
better than Java’s. Consider the method getFirstNonEmpty in Figure 12. javac accepts getFirstNonEmpty, combining List<String> and List<Object> into List? and then instantiating ? 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 ? so that List<String> 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 typelane, which is the return type of foo which depends on the inferred type arguments for the generic methods foo and singleton. Using java’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 ? 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 τ < τ', 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 ? : 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 (⊔) of captured wildcard types Π:Γ, C<τ1, ..., τn> and ∃v:Γ', C<τ1, ..., τn> and ∀v:Γ, C<τ1, ..., τn> and ∃v:Γ', C<τ1, ..., τn> with new fresh variables Γ, and assignments θ and θ' such that each ⌈v|θ⌉ equals ⌈v|θ'⟩, ..., ⌈v|θ'⟩ and each ⌈v|θ'⟩ equals ⌈v|θ⟩, ..., ⌈v|θ⟩. For example, the anti-unification of the types Map<String,String> and Map<Integer,Integer> is Map<e,θ> with assignments v → String and v → Integer. The join, then, is the lazy existential type

\[
\exists \Gamma : \theta \rightarrow \Gamma : \theta' \rightarrow \Gamma' : \theta'' \rightarrow \Gamma''
\]

The lazy constraint (θ → Γ : Δ; θ' → Γ' : Δ') indicates that the constraints on Π are the constraints that hold in context Γ : Δ after substituting with θ and in context Γ' : Δ' 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 τ and τ' 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 (θ → Γ : Δ; θ' → Γ' : Δ') 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 <: τ' with v constrained by a lazy constraint rather than standard constraints, one checks that both θ(v) <: τ'[θ] holds and θ'(v) <: τ'[θ'] holds, applying the substitutions to relevant constraints in the context as well. A similar adaptation is also made for τ <: 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

\[
\exists : \{\text{I} \rightarrow \text{String}\} \rightarrow \text{I} : \{\text{I} \rightarrow \text{Integer}\} \rightarrow \text{I} : \{\text{I} \rightarrow \text{Object}\}
\]

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 I <: Comparable 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 $\tau_{\nu_1}(v_1 | \ldots | v_n)$ or $\tau_{\nu_2}(\tau | v_1 | \ldots | v_n)$ where each $v_i$ is not lazily constrained and $\tau_{\nu}$ is a supertype of some wildcard capture of each upper bound of each type variable (and of $\tau$ if present) with the property that any other non-variable $\tau'$ which is a supertype of each $v_i$ (and $\tau$) is also a supertype of $\tau_{\nu}$. A type $\tau'$ is a supertype of this specialized union type if it is a supertype of $\tau_{\nu}$ or of each $v_i$ (and $\tau$). Note that $\tau_{\nu}$ might not be a supertype of any $v_i$ or of $\tau$ and may instead be the join of the upper bounds of each $v_i$ (plus $\tau$) 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 $v \prec \tau$ and $\tau \preceq v$, 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:

```java
interface SubList<? extends List<? extends T>, T> {}
```

Java accepts the type `SubList<String, Number>` because the wildcard can be implicitly constrained to be a subtype of `List<String>` with which the requirements of the type parameters are satisfied. However, `Java` rejects the type `SubList<List<String>, Integer>` even though the wildcard can be implicitly constrained to be a supertype of `List<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:

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

For the type `C<?, ?>` to be valid, `C<?, ?>` must be a subtype of `List<?>` where `X` is the last wildcard of `C<?, ?>`. Since `C<X>` is a subtype of `List<C<X>, ?>`, this implies `X` must be equal to `C<X, ?>`, 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?<X, ?>, ?>, we get the type `C<X, ?>, C<X, ?>, ?>, which in turn contains the type `C<X, ?>, ?>, ?> so that we can continually expand to get the infinite type `C<X, ?>, ?>, ?>, ...`. 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:

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

`javac` allows one to declare a program variable `X` to have type `Maps<P>` using `Map<String,?>`. The type of `X`, then, has a wildcard which is constrained to be a subtype of both `Map<?,String>` and `List<String>`. This means that `get(0).entrySet()` has two types, essentially `Set<Entry<X,String>>` and `Set<Entry<String,X>>`, 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<X1, ..., Xn>` only if each `Xi` equals `Xj` [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. 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 `n`’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,?>`. With this more precise constraint, we can determine that the type of `X.get(0).entrySet()` is `Set<Entry<String,X>>`, 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<X>` and `Set<X>` is `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<X>` extends `Map<String,?>` is equivalent to `class C implements List<X> extends List<X> extends C>() {}`.

#### Key Steps of Proof

- `D<X>` extends `List<X>` extends `C()` equivalent to `D<X>` extends `C()`
- `(Checking :>) D<X>` extends `C` << `List<X>` extends `C`
- `List<X>` extends `List<X>` extends `D<X>` << `List<X>` extends `C`
- `D<X>` extends `List<X>` extends `C()` equivalent to `D<X>` extends `C()` (repeat forever)

![Figure 14](https://via.placeholder.com/150)

Example of infinite proofs due to equivalence.
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 `Super<P super Q, Q>` 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 so 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

```java
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 \( \tau \) and class or interface names \( C \), if \( \tau \) has a supertype of the form \( \exists P : \Delta \) or \( \tau \) with the appropriate form without having to worry about any alternative supertypes.

Java only ensures single-instantiation inheritance with wildcards when \( \tau \) is a class or interface type, but not when \( \tau \) 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 \( C<\bar{\tau}_1, \ldots, \bar{\tau}_m> \) (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 inheri-
tance 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 \( \bar{\tau}_1 \& \ldots \& \bar{\tau}_m \), every \( \tau_i \) must validly intersect with every other \( \tau_i \). For every explicit upper bound \( \tau \) on a wildcard, \( \tau \) 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 \( \exists X. \text{List}\langle X \rangle \& \text{Set}\langle X \rangle \). 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 \( ? \) extends \( \tau \) \& \ldots \& \bar{\tau}_m \) can simply be interpreted as \( ? \) extends \( \tau_i \) for each \( i \) in 1 to \( m \) without introducing an actual type \( \tau_i \) \& \ldots \& \bar{\tau}_m \). Thus our solution addresses the non-determinism issues discussed in Section 7.2 and simplifies the formal type system.
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 \( \tau \) on a wildcard, \( \tau \) must not be a strict supertyp of any other upper bound on that wildcard.

For every explicit lower bound \( \tau \) on a wildcard, \( \tau \) 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 \( \Rightarrow \) operation in Figure 16. Its basic strategy is to identify and remove all redundant constraints. The primary tool is the \( \prec \) relation, an implementation of subtyping specialized to be sound and complete between class and interface types only if \( \tau \) is a supertype of \( \tau' \) 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 \( \prec \) relation hold. The intersection restriction provides completeness by ensuring that non-redundant explicit bounds are unique up to equivalence so that syntactic equality can recur 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 `Number<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

\[
\frac{\vdash \tau \Rightarrow \tilde{\tau}}{\vdash \Delta \vdash \tau \Rightarrow \tilde{\tau}}
\]

\[
\begin{array}{ll}
\text{explicit}(\tilde{\tau}_1, \ldots, \tilde{\tau}_n) = (\Gamma; \Delta; \tau_1, \ldots, \tau_n) \\
\text{implicit}(\hat{\tau}_1, \ldots, \hat{\tau}_n) = (\Gamma; \Delta; \tau_1, \ldots, \tau_n) \\
\end{array}
\]

\[
\frac{\forall v :: \tilde{\tau} \in \Delta \mid \text{for no } v :: \tilde{\tau}' \in \Delta, \vdash \tilde{\tau} \prec \tilde{\tau}'}{\forall v :: \tilde{\tau} \in \Delta \mid \text{for no } v :: \tilde{\tau}' \in \Delta, \vdash \tilde{\tau} \prec \tilde{\tau}'}
\]

\[
\frac{\forall i \in 1 \text{ to } n, \tau_i \Rightarrow \tilde{\tau}_i}{\forall i \in 1 \text{ to } n, \tau_i \Rightarrow \tilde{\tau}_i}
\]

\[
\begin{array}{ll}
\frac{\exists C \tilde{\tau}_1, \ldots, \tilde{\tau}_n \Rightarrow C\tilde{\tau}_1, \ldots, \tilde{\tau}_n}{\vdash v \Rightarrow v}
\end{array}
\]

**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.

\[
\frac{\vdash \tau \Rightarrow \tilde{\tau}' \quad \vdash \tau' \Rightarrow \tilde{\tau}'}{\Gamma : \Delta \vdash \tau \equiv \tau'}
\]
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 \( \tau \) to covariant type parameters are translated to \(? extends \tau\), and type arguments \( \tau \) to contravariant type parameters are translated to \(? super \tau\). 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 \(\text{List}\langle ? \rangle \& \text{Set}\langle ? \rangle\) 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 \(\exists X. \text{List}\langle X \rangle \& \text{Set}\langle X \rangle\), which satisfies the intersection restriction. Users could also express potentially useful types such as \(\exists X. \text{Pair}\langle X, X \rangle\) and \(\exists X. \text{List}\langle \text{List}\langle X \rangle \rangle\).

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 \(\exists X. \text{Extends} \langle \text{Enum}\langle X \rangle, \text{List}\langle X \rangle \rangle\). 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.

```java
<? super Integer> List<? extends Integer> sequence(int n) {
    List<Integer> res = new LinkedList<Integer>();
    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 \(\forall X. \text{List}<X>\). Although this form of 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 in- ferable 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