

Java Type Inference Is Broken: Can We Fix It?

Daniel Smith

Department of Computer Science
Rice University
Houston, Texas, USA
dsmith@rice.edu

Robert Cartwright

Department of Computer Science
Rice University
Houston, Texas, USA
cork@rice.edu

Abstract

Java 5, the most recent major update to the Java Programming Language, introduced a number of sophisticated features, including a major extension to the type system. While the technical details of these new features are complex, much of this complexity is hidden from the typical Java developer by an ambitious type inference mechanism. Unfortunately, the extensions to the Java 5 type system were so novel that their technical details had not yet been thoroughly investigated in the research literature. As a result, the Java 5 compiler includes a pragmatic but flawed type inference algorithm that is, by design, neither sound nor locally complete. The language specification points out that neither of these failures is catastrophic: the correctness of potentially-unsound results must be verified during type checking; and incompleteness can usually be worked around by manually providing the method type parameter bindings for a given call site.

This paper dissects the type inference algorithm of Java 5 and proposes a significant revision that is sound and able to calculate correct results where the Java 5 algorithm fails. The new algorithm is locally complete with the exception of a difficult corner case. Moreover, the new algorithm demonstrates that several arbitrary restrictions in the Java type system—most notably the ban on lower-bounded type parameter declarations and the limited expressibility of intersection types—are unnecessary. We hope that this work will spur the evolution of a more coherent, more comprehensive generic type system for Java.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.2 [*Programming Languages*]: Language Classifications—Java, Object-oriented languages; D.3.3 [*Programming Lan-*

guages]: Language Constructs and Features—Classes and objects, Polymorphism

General Terms Design, Languages

1. Introduction

Java 5¹, the most recent major update to the Java Programming Language, introduced a number of sophisticated features, including a major extension to the type system. While the technical details of these new features are complex, much of this complexity is hidden from the typical Java developer by an ambitious type inference mechanism.

Prior to the release of Java 5, there was no type inference in Java. According to the Java language culture, the type of every variable, method, and dynamically allocated object must be explicitly declared by the programmer. When generics (classes and methods parameterized by type) were introduced in Java 5, the language retained this requirement for variables, methods, and allocations. But the introduction of polymorphic methods (parameterized by type) dictated that either (i) the programmer provide the method type arguments at every polymorphic method call site or (ii) the language support the inference of method type arguments. To avoid creating an additional clerical burden for programmers, the designers of Java 5 elected to perform type inference to determine the type arguments for polymorphic method calls.

Since generics constituted a major technical addition to the language, the Java language designers, with input from the Java Community Process [15], spent several years carefully evaluating potential generic extensions and their technical implications. Nevertheless, the final design included a novel mechanism for declaring special union types called *wildcards* supporting covariant and contravariant subtyping. The use of wildcards also necessitated adding some support for *intersection types* in the language. The inclusion of wildcards helped support the accurate parameteric typing of

This is the authors' printing of the work. It is posted here by permission of the ACM for personal use. Not for redistribution. The definitive version is available at <http://doi.acm.org/10.1145/1449764.1449804>.

OOPSLA'08 October 19–23, 2008, Nashville, Tennessee, USA.

Copyright © 2008 ACM 978-1-60558-215-3/08/10...\$5.00

Reprinted from OOPSLA'08, Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 19–23, 2008, Nashville, Tennessee, USA., pp. 505–524.

¹ Throughout this paper, we use “Java 5” to refer to the language update coinciding with the release of Java SE 5.0 and specified by the 3rd edition of the *Java Language Specification (JLS)* [3]. The current Java platform version, Java SE 6.0, is consistent with this specification and makes no relevant language changes.

many existing Java library methods, including the reflection library, which is an impressive achievement. On the other hand, this feature was so novel that its technical foundations, particularly with regard to type inference, had not yet been thoroughly investigated in the research literature.

Java wildcards are loosely based on virtual types, initially presented in the context of generic classes by Thorup and Torgersen [11] and subsequently refined by Igarashi and Viroli [5]. But none of this supporting research focused on a core subset of the actual Java 5 design. Moreover, none of this prior work covered all of the technical problems that arise during type argument inference. As a result, the Java 5 compiler includes a pragmatic but flawed type inference algorithm. While it essentially follows Odersky’s algorithm for GJ [1] (an academic forerunner of Java 5 that did not include wildcards), the additional complexity introduced by wildcards and intersections led to an algorithm that is, by design, neither sound nor locally complete.² The Java Language Specification (*JLS*) [3] points out that neither of these failures is catastrophic: the correctness of potentially-unsound results must be verified during type checking; and incompleteness can usually be worked around by manually providing the method type parameter bindings for a given call site.³

This paper dissects the type inference algorithm of Java 5 and proposes a significant revision that is sound and able to calculate correct results where the Java 5 algorithm fails. The new algorithm is locally complete with the exception of a difficult corner case. Moreover, the new algorithm demonstrates that several arbitrary restrictions in the Java type system—most notably the ban on lower-bounded type parameter declarations and the limited expressibility of intersection types—are unnecessary. We hope that this work will spur the evolution of a more coherent, more comprehensive generic type system for Java.

To motivate the need for changes in the algorithm, we first enumerate a number of bugs buried in the specification. Next, we discuss ways in which, even after these bugs have been addressed, the algorithm falls short—either because it fails to correctly type a typeable program (requiring the programmer to insert explicit annotations), or because it unduly limits expressiveness of the language. Among these shortcomings is an incorrect *join* function and the language restrictions noted above.

Next, we formally specify an improved inference algorithm that addresses many of the shortcomings in the Java 5 algorithm. This specification includes definitions of the language’s core type operations, such as subtyping and wildcard capture.

² Soundness requires that inferred type arguments not violate the typing rules; completeness requires that the algorithm produces a result where some valid choice of type arguments exists.

³ Note, however, that some types, such as wildcard capture variables, are inexpressible and thus can’t be used as explicit type arguments.

Finally, we discuss the implications for backwards compatibility arising out of changes to the inference algorithm.

2. Java 5 Inference

2.1 Overview

The Java 5 type argument inference algorithm produces type arguments for use at a specific call site of a polymorphic method. For example, consider the following method declaration:

```
class Util {
    static <T> Iterable<T>
        compose(Iterable<? extends T> list, T elt) {
        ...
    }
}
```

where `compose` adds a new element to an `Iterable` collection. As a running example, we’ll refer to the following type hierarchy:

```
interface Animal { ... }
interface Herbivore extends Animal { ... }
interface Carnivore extends Animal { ... }
```

Given an `Iterable<Herbivore>` named `hs`, an `Herbivore` `h`, and a `Carnivore` `c`, a client may invoke `compose` with an explicit type argument:

```
Iterable<Herbivore> hs2 =
    Util.<Herbivore>compose(hs, h);
Iterable<Animal> hs3 =
    Util.<Animal>compose(hs, c);
```

or, more typically, elide the type argument and rely on type inference to choose a value based on the types of the arguments:

```
Iterable<Herbivore> hs2 = Util.compose(hs, h);
Iterable<Animal> hs3 = Util.compose(hs, c);
```

The Java 5 inference algorithm frames the problem as a heuristic attempt to satisfy a set of subtyping constraints [3, 15.12.2.7-8].⁴ Let $P_1 \dots P_n$ be the method’s type parameters. The notation $\lceil P_1 \rceil$ represents the declared upper bound of P_1 (this notation extends to arbitrary type variables; later, we’ll use $\lfloor X \rfloor$ to refer to variable X ’s lower bound). Given $A_1 \dots A_m$ as the types of the method invocation’s arguments and $F_1 \dots F_m$ as the corresponding method parameters’ declared types, we must find a substitution σ binding $P_1 \dots P_n$ that satisfies the following constraints (for all valid choices of i —there are $m + n$ such constraints to satisfy):

$$A_i <: \sigma F_i$$

$$\sigma P_i <: \sigma \lceil P_i \rceil$$

In certain circumstances, the algorithm also takes into account the type expected by the method invocation’s context.

⁴ In the following discussion, we depart from the notation used by the *JLS* in order to maintain a consistent presentation throughout this paper. However, the ideas expressed by the notation are consistent with the specification.

That is, where R is the method’s declared return type and E is the expected type,

$$\sigma R <: E$$

is added to the set of subtyping constraints. The type E is only defined, and thus this additional constraint is only used in inference, where the method invocation appears as the value of a `return` statement, the initializer of a variable declaration, or the right-hand side of an assignment. That is, only in contexts in which the programmer has provided an explicit value for E . (And even where E is defined, the Java 5 algorithm often ignores it.)

Note that these constraints describe exactly the conditions under which an invocation of a non-overloaded method with well-typed arguments is well-typed (and, if the constraint involving E is used, the conditions under which the enclosing expression is well-typed): if the constraints are satisfiable, there exists a choice of type arguments that makes the expression well-typed; if the constraints are unsatisfiable, the expression is ill-typed for *all* choices of type arguments.

The Java 5 algorithm generates this set of constraints for each polymorphic method call site, and then the algorithm attempts to choose a value for σ satisfying the constraints. In the last `compose` invocation above—`Util.compose(hs, c)`—the relevant constraints are:

$$\begin{aligned} \text{Iterable<Herbivore>} &<: \sigma \text{Iterable<? extends T>} \\ \text{Carnivore} &<: \sigma T \\ \sigma T &<: \text{Object} \end{aligned}$$

The first two constraints are derived from the invocation argument types, and the third from the (trivial) bound of T . The inferred substitution is $[T := \text{Animal}]$.

Because the algorithm makes no guarantees about its results—it is, by design, neither sound nor complete—type checking handles the inferred arguments as if they were explicitly provided by the user: first checking their correctness, and then using them to determine the type of the method invocation expression.

2.2 Constraint Solving

Internally, the Java 5 algorithm is a two-step process. First, argument–value pairs (A_i and F_i for all $i \leq m$) are reduced to a conjunction of bounding constraints on the instantiations of $P_1 \dots P_n$. Each constraint is an assertion that σP_i , for some i , is a subtype or supertype of a given bound. Second, a type satisfying these bounds is chosen for each type argument.

The reduction in the first phase is achieved with three mutually recursive functions $<:_{\sigma}$, $:>_{\sigma}$, and $=_{\sigma}$. Each function \odot takes two arguments, A (derived from an argument type) and F (derived from a formal parameter type), and attempts to produce constraints describing the circumstances under which $A \odot \sigma F$ is true. In the base case, where $F = P_i$, the result is a bound on the corresponding argument, σP_i .

For example, in the last `compose` invocation, the following two constraints must be reduced ($\sigma T <: \text{Object}$, derived from the declared bound on T , is ignored until the second phase):

$$\begin{aligned} \text{Iterable<Herbivore>} &<: \sigma \text{Iterable<? extends T>} \\ \text{Carnivore} &<: \sigma T \end{aligned}$$

Bounds for the second constraint are trivially produced:

$$\text{Carnivore} <:_{\sigma} T = \{\sigma T :> \text{Carnivore}\}$$

Handling the first constraint is slightly more complex:

$$\begin{aligned} \text{Iterable<Herbivore>} &<:_{\sigma} \text{Iterable<? extends T>} = \\ \text{Herbivore} &<:_{\sigma} T = \{\sigma T :> \text{Herbivore}\} \end{aligned}$$

Thus, the full set of bounds for the `compose` invocation is

$$\{\sigma T :> \text{Herbivore}, \sigma T :> \text{Carnivore}\}$$

In the second phase, the lower bounds of σP_i are combined with a *join* operation to produce a single type.⁵ The *join* function is also implemented heuristically: ideally, the invocation $\text{join}(S, T)$ produces a most specific type J such that $S <: J$ and $T <: J$. By *most specific* we mean that any other common supertype J' of S and T is also a supertype of J : $\forall J', J <: J'$. The Java 5 *join* function produces a common supertype, but in some cases it is not the most specific.

In the running example, we have

$$\sigma T :> \text{join}(\text{Herbivore}, \text{Carnivore}) = \text{Animal}$$

In most cases, the result of joining the lower bounds of σP_i is then chosen as the binding of P_i . If, however, this is the type `null` (as is the case where there are no lower bounds), the declared bounds of $P_1 \dots P_n$ are incorporated; if the expected type of the call E is defined, the bounds produced by $E :>_{\sigma} R$ are also used. In this case, the *upper* bounds of σP_i are merged (by constructing an intersection) to produce the binding.

2.3 Bugs in the Java 5 Algorithm

The Java 5 algorithm produces useful results in most situations. However, there are a number of cases in which it fails, either because there exist choices for σ that it does not find, or because its choice of σ does not satisfy the relevant subtyping constraints. In both cases, this can lead to type errors or, where either the method to be invoked or an enclosing method call is overloaded, unexpected runtime behavior. It will not, fortunately, lead to violations of type safety, because type checking makes no assumptions about the correctness of the algorithm’s results. Some of the unsoundness and incompleteness properties of the algorithm arise from conscious engineering decisions. But in many cases the heuristic nature of the algorithm provides a cover for unintentional bugs. Some of these bugs are outlined below.

⁵The *join* function is called *lub* (for “least upper bound”) in the *JLS* [3, 15.12.2.7].

- The *join* function is defined incorrectly for some wildcard-parameterized types:

```
join(List<? extends A>, List<? super A>) = List<A>
```

This is clearly incorrect—the result is a supertype of neither *join* argument.

- The *join* function, because it discards some type information, is unnecessarily imprecise in some cases: the result cannot be a type variable (even if that variable is the common supertype of two other variables), nor can it be an array of a parameterized type (like `List<String>[]`).
- The inference algorithm does not correctly handle type variables. For example, where *A* is a variable and *F* is an array type, $A \text{ :> } F$ recurs on *A*'s upper bound rather than its lower bound.
- Default bounds on wildcards (`Object` and `null`) are incorrectly ignored by the algorithm. For example:

```
List<? super String> <: List<? extends T>
```

This ought to produce $\{\sigma T \text{ :> } Object\}$, rather than the specified $\{\}$.⁶

- The algorithm's use of a parameter's upper bound allows references to a parameter to leak into the calling context. For example, given the following method signature:

```
<T extends List<T>> T foo()
```

Inference may determine (depending on the enclosing context) that $\sigma T = List<T>$.

In addition to these bugs, the algorithm does not correctly handle the type `null`: $null \text{ :> } T$ produces no bounds when it ought to produce $\{T \text{ <: } null\}$. While clearly a mistake (the algorithm explicitly defines an incorrect result for `null`), this is not a problem as the language is currently defined, because no invocation involving `null` as a supertype ever occurs: `null` is inexpressible, is never chosen by the inference algorithm, and is ignored when it appears as a default bound. However, subtle changes to the language, including a fix for the wildcard-bound bug listed above, may violate this invariant.⁷

2.4 Additional Limitations

In addition to the mistakes described above, the Java 5 algorithm is limited by design in a number of ways. In some

⁶ This bug is of particular significance to the `javac` compiler, because it fails to verify the correctness of the inference results in this case, leading to a violation of type safety: code that compiles without error (or warning) will fail at runtime with an erasure-prompted `ClassCastException`. See Appendix A.1 for an example.

⁷ Note that the inference algorithm in Section 3 *does* produce `null`. Also, an expressible `null` type would be quite useful; there is an independent submission in Sun's Java bug database requesting it, with some accompanying discussion [17].

cases, these limitations lead to unsoundness or incompleteness; in others, they force restrictions on the rest of the language. Unlike the above bugs, which are clearly faults in the specification, language changes involving these items are open to debate. However, we argue that the community would be well served by amending the specification to address these limitations.

2.4.1 Correct Join

As mentioned previously, the Java 5 *join* function does not always produce a most specific bound.⁸ As a simple example, consider the following invocation:

```
join(List<Object>, List<String>)
```

The correct result in this case is `List<? super String>`; the Java 5 function, however, never produces wildcards with lower bounds, and will instead produce `List<?>`.

The correct definition in other cases is more subtle. Consider a similar invocation in which the two list element types are not directly related, but share a common supertype:

```
join(List<Herbivore>, List<Carnivore>)
```

The following is a tempting choice for the result (and is the result chosen by the Java 5 algorithm):

```
J1 = List<? extends Animal>
```

However, it is equally reasonable to choose a *lower* bound for the wildcard:

```
J2 = List<? super Herbivore & Carnivore>
```

Both J_1 and J_2 are supertypes of both `List<Herbivore>` and `List<Carnivore>`; yet neither is a subtype of the other. In practice, which type is more convenient depends on how the type is used (J_1 accommodates `get` operations, while J_2 accommodates `add` operations). A joint University of Aarhus–Sun Microsystems paper introducing wildcards makes note of this ambiguity [12, 3.1], but does not mention how it can be resolved—by either (i) producing a wildcard with *both* bounds:

```
List<? extends Animal super Herbivore & Carnivore>
```

or (ii) using a *union type* to represent the join:

```
List<Herbivore> | List<Carnivore>
```

Both of these types are subtypes of J_1 and J_2 , and both are optimal (the first is optimal in the absence of union types). But neither is valid in Java 5, so to accommodate either approach, the language would need to be extended.

A second problem with *join* is that it is recursive but not normalizing: the computation of $join(S, T)$ may depend on itself. For example, we may choose to define classes `C` and `D` as follows:

⁸ Thus the result is thus not really a *join* or *least upper bound* at all, as the terms are used in lattice theory.

```
class C implements Comparable<C> { ... }
class D implements Comparable<D> { ... }
```

Invoked with such types, the Java 5 *join* function has a circular dependency:

```
join(C,D) =
join(Comparable<C>, Comparable<D>) =
Comparable<? extends join(C,D)> =
...
```

This circular dependency is handled in the *JLS* by introducing recursive types. Informally, the result of the above *join* invocation is:

```
Comparable<? extends Comparable<? extends ...>>
```

Formally, this is the type

```
 $\mu X. Comparable<? extends X>$ 
```

Unfortunately, outside the context of the *join* function's definition, the specification makes no mention of such types. They are not included in the definition of types, and their subtyping relationships with other types are left unspecified.

Again, there are two alternatives. The first is to fully specify the behavior of all type operations (including subtyping, *join*, and inference) where recursive types are present. The second is to abandon recursive types and instead compute *join* using union types. This also requires adjusting the domain of all type operations, but has the advantage that algorithms involving unions are far simpler than those involving recursive types.

2.4.2 Analysis Using Wildcard Capture

In order to analyze subtyping relationships involving a wildcard-parameterized type, the Java 5 subtyping algorithm makes use of a *wildcard capture* operation (denoted $\|T\|$) that replaces the implicit “there exists” quantification expressed by a wildcard with a fresh type variable. Where class *C* declares parameter *P*,

$$\|C<? extends B>\| = C<Z>$$

where $[Z] = B \ \& \ [P := Z][P]$

For example, the class `Enum` has the following signature:

```
class Enum<E extends Enum<E>>
implements Comparable<E>
```

The assertion

```
Enum<? extends Runnable> <:
Comparable<? extends Enum<?>>
```

is true if and only if

$$\|Enum<? extends Runnable>\| = Enum<Z> <: Comparable<? extends Enum<?>>$$

where the fresh variable *Z* has both the wildcard's and the corresponding parameter's upper bounds:

$$[Z] = Runnable \ \& \ Enum<Z>$$

(In this particular example, the subtyping assertion is true, because $Z <: Enum<?>$.)

The Java 5 inference algorithm is inconsistent with the Java 5 subtyping algorithm in handling wildcards: rather than reasoning about wildcards by using capture, it simply recurs on the wildcard bound, ignoring the bound of the corresponding type parameter. The invocation

```
Enum<? extends Runnable> <: ? Comparable<? extends T>
```

produces $\{\sigma T := Runnable\}$. This constraint is too restrictive: the type `Enum<?>`, as we saw, is a valid choice for σT , but is not allowed by this inferred bound.

The apparent solution to this omission is to invoke capture in inference whenever it occurs in subtyping. However, this strategy forces us to generalize the inference algorithm to handle the broader domain of types generated by this rule.

There is an implicit assumption in the Java 5 algorithm that the constraints for all subtyping relationships with which it is presented can be expressed as a conjunction of simple bounds. Note, however, that the application $A :=_? F_1 \ \& \ F_2$ does not conform to this scheme: it can be satisfied by $A :=_? F_1$ or $A :=_? F_2$. In order to avoid the possibility that relevant information will be discarded, the Java 5 algorithm must guarantee that such applications will never occur.

In the absence of wildcard capture in type inference, we can make the following assertions about the arguments to $<:?$, $:=_?$, and $=_?$:

- The argument *F*, if it is a variable but not one of $P_1 \dots P_n$, cannot have bounds involving any of $P_1 \dots P_n$. This property holds because *F* is always derived from a type appearing in the method signature, and, due to scoping constraints, no variable appearing there can reference $P_1 \dots P_n$ at its declaration point.
- In $<:?$, where *A* is an intersection type and *F* is a class type, there is at most one class supertype of the intersection that has the same class name as the upper type; where *A* is an intersection and *F* is an array type, if the intersection has an array supertype, it also has a most specific array supertype.⁹
- The argument *F* cannot be an intersection type. Intersections can only be reached by the recursive application of the inference functions. However, variables, which might have intersections in their upper bounds, cannot involve any of $P_1 \dots P_n$, and so the recursion would never occur;

⁹As a technicality, the *JLS* does not describe how $A <: ? F$ should be handled where *A* is an intersection and *F* is an array, but the correct behavior follows directly from this assumption.

and for the reasons outlined in the previous point, there is no need to represent the supertype of a class type as an intersection.

Under these assumptions, all bounds produced by inference are cumulative. However, capture during inference violates the first assumption: it can produce new variables with bounds that refer to $P_1 \dots P_n$.

In order to extend the inference algorithm to handle invocations where one of a number of different constraint sets must hold, the representation of constraint sets can be extended to constraint *formulas*: simple bounds on $T_1 \dots T_n$ combined by boolean conjunction and disjunction. To reduce the complexity inherent in arbitrary boolean formulas, the domain of constraint formulas can be restricted to disjunctive normal form—a list of constraint sets (conjunctions), only one of which need be satisfied by the final choice of $T_1 \dots T_n$.¹⁰

A simpler but less complete solution is to use capture in restricted scenarios that do not necessitate the use of disjunction. We can perform capture on A , for example, without violating any of the above assumptions.

2.4.3 First-Class Intersection Types

As noted in the previous section, intersection types can introduce additional complexity to the inference algorithm. For this reason, their use in Java 5 is extremely limited: a programmer may only express an intersection in code when it appears as the upper bound of a type variable. (Programmers may be surprised to discover that the upper bound of a wildcard *cannot* be similarly expressed with an intersection.) If we are willing to extend the inference algorithm so that it handles disjunctive constraint formulas—as described above—it then becomes possible to support intersections as first-class citizens in the domain of types, admitting their usage anywhere an arbitrary type can appear.¹¹

As a simple motivating example, the Java API includes the interfaces `Flushable` and `Closeable`, implemented by streams that support a `flush` and a `close` operation, respectively. Taking advantage of these interfaces, it might be convenient to create a thread that occasionally flushes a stream, and at some point closes it. Such a thread would need to reference a variable of type `Flushable & Closeable`.

Another example appears in Appendix A.2. By using intersections in method signatures and implementation code,

¹⁰ Normalization to disjunctive normal form, as with other methods for solving arbitrary logical formulas, can potentially take an intractable amount of time and space for large problem sizes. However, in this particular application, problem sizes are always quite small: programmers almost never use more than four or five type parameters, for example, and we generally expect the number of distinct, incompatible bounds inferred for those variables to be relatively small. Also note that the complexity encoded by multiple disjuncts is discarded once inference at a particular call site determines a solution—multiple nested polymorphic method calls will not lead to exponential growth in the size of constraint formulas.

¹¹ There is an independent submission in Sun’s Java bug database requesting this feature, with some accompanying discussion [18].

it is possible to define an ordered set that ensures that its elements are comparable to each other without requiring the element type `T` to explicitly implement `Comparable`.

In Java 5, it is often possible to approximate the first-class use of an intersection by introducing a type variable `T` with an intersection upper bound, and replacing all instances of the intersection with references to `T`. However, this approach is inconvenient for the same reason that writing programs without wildcards is inconvenient—it results in a proliferation of variable declarations that are irrelevant to the public interface of a class or method. Further, such a conversion is not possible in general: a mutable field, for example, may hold values with *different* types, all compatible with the intersection, over the course of its lifetime. There may be no non-intersection choice for `T` that covers the domain of these values.

Support for first-class intersections, combined with the ability to make full use of wildcard capture during inference, provides a compelling motivation for extending the inference algorithm with disjunctive reasoning.

2.4.4 Recursively-Bounded Type Parameters

Java 5 supports F-bounded polymorphism: a type parameter may appear recursively within its own bound; mutual recursion among parameter bounds is also allowed. As noted in the list of specification bugs (Section 2.3), where the inference algorithm attempts to incorporate the upper bounds of $P_1 \dots P_n$ before choosing σ , it does so incorrectly and allows these recursive references to leak into the calling context.

Notice that the subtyping constraint involving parameter bounds make reference to σ on both sides of the constraint:

$$\sigma P_i <: \sigma [P_i]$$

Thus, the techniques used to solve other subtyping constraints are difficult to apply here.

If we’re interested in simply patching the specification bug, the workaround is for inference to give up in cases that will produce out-of-scope results. A more useful solution is to simply ignore upper bounds and always choose the inferred lower bound (even if it is `null`). In practice, where there are multiple correct choices for σ , choosing the most *specific* instantiations—the inferred lower bounds—is usually most useful to programmers.

However, this solution is incomplete. Consider a simplified case in which there is only one parameter, P . Inference may determine that $\sigma P >: T$, for some T . Yet where $T \not<: [P := T][P]$ it is possible that there exists some S such that $T <: S <: [P := S][P]$. The choice of σ changes the bound that must be satisfied—the types $[P := T][P]$ and $[P := S][P]$ need not be related.

As an example, consider a type `MyString` which extends the standard `String` class (which in turn extends

`Comparable<String>`).¹² Now assume we pass two `MyStrings` to a method with the following signature:

```
<T extends Comparable<T>> T min(T x, T y)
```

The best choice for σ is `[T := String]`. But the inferred lower bound is `MyString`, not `String`. Thus, inference fails to produce a correct result, and the user must provide an explicit parameter.¹³

To improve on the “lower bound” strategy, what is needed is a way to describe a type that falls within a number of bounds, some of which are in scope in the calling context, and some of which are parameterized by $P_1 \dots P_n$. Interestingly, *wildcard capture* has already been defined for exactly this purpose! It combines the bound of a wildcard with the potentially-recursive bound of a type parameter.

Unfortunately, if we examine the results of wildcard capture applied to lower-bounded wildcards, we find that it, too, is incapable of managing some recursive bounds. For example, if class `Foo` has a parameter `T extends Comparable<T>`, we have the following:

```
||Foo<? super MyString>|| = Foo<Z>
  where [Z] = Comparable<Z>
  and [Z] = MyString
```

In this case, the capture variable `Z` (and thus the wildcard itself) is malformed, because its bounds are inconsistent: `MyString $\not\prec$ Comparable<Z>`.¹⁴

It is not clear how best to proceed. If wildcard capture could be improved in some way so that this class of seemingly reasonable type expressions were well-formed, it would be a useful fall-back for constraint solving in inference: where a set of inferred lower bounds do not satisfy the declared bounds of a method’s parameters, a set of properly-bounded capture variables could be chosen instead. But describing such an improvement remains an open research question.

Another alternative is to use trial and error, walking a depth-first traversal of the type hierarchy from the lower bound to `Object`. But it is not clear that the satisfying type `S`, if it exists, will always appear in this enumeration.

Barring a universal solution to this problem, any Java type argument inference algorithm will be incomplete.

¹² `String` cannot actually be extended, because it is declared `final`. But it is a familiar class that implements `Comparable`, so we use it here.

¹³ Another workaround in this case is to use a wildcard in the bound: `Comparable<? super T>`. That works for this simple method, but would not if a more complex class were being used. See Appendix A.3 for a similar, but more realistic and complex, example.

¹⁴ While the specification is not clear about the conditions under which a wildcard is malformed, it would violate transitivity of subtyping if capture were allowed to produce a variable with a lower bound that was not a subtype of its upper bound. Thus, such wildcards must be malformed.

2.4.5 Lower-Bounded Type Parameters

While wildcards may be bounded from either above or below, type parameters are not given this flexibility: only an upper bound is expressible. It’s natural to wonder whether this inconsistency is necessary (especially given that variables produced by wildcard capture can have both upper and lower bounds).¹⁵ In fact, the limitation is closely tied to the type argument inference algorithm, and improvements to the algorithm would make this restriction unnecessary.

At first glance, it may seem that a lower bound on a type variable provides no useful information for the programmer. For example, if `T` has a lower bound `Integer` and a method declares a parameter of type `T`, the method programmer must assume that, in the most general case, `T` represents the type `Object`, and thus has none of the methods specific to `Integer`.

This intuition, however, is superficial. When the type `T` is nested, both upper and lower bounds of the variable may be useful. The following method definition, not legal in Java 5, demonstrates one reasonable use of a variable with a lower bound:

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

Depending on the instantiation of `E`, the `sequence` method can be used to create lists of `Integers`, lists of `Numbers`, or lists of `Objects` (among other things). In each case, the method will add some number of `Integers` to the list before returning it.

This example could be roughly translated into legal Java by replacing `E` with a wildcard (eliminating the type variable declaration and returning a `List<? super Integer>`). But this is not a satisfactory alternative: a client may, for example, need to read from and write to a `List<Number>`, while a `List<? super Integer>`’s `get` method returns `Objects` and its `add` method accepts only `Integers`.

In addition to the practical argument, support for lower-bounded variable declarations is mandated by an appeal to uniformity: these bounds directly complement similar bounds on wildcards. The Aarhus–Sun paper notes that, where a name representing a wildcard is needed, the equivalent of an existential-type *open* operation may be performed by invoking a polymorphic method [12, 3.3].¹⁶ For example, it is possible to shuffle (both read from and write to) a `List<?>` by invoking a method with signature

```
<E> void shuffle(List<E> l)
```

¹⁵ This feature has been independently requested and discussed in Sun’s Java bug database: [16].

¹⁶ Existential types are traditionally used to define an API in terms of a private, unnamed type. Clients of the API can use it by invoking an *open* operation, declaring a type variable as a stand-in for the private type.

Similarly, we can sort a wildcard-parameterized list, as long as the wildcard is bounded by something we know how to sort (a `List<? extends Number>`, say):

```
<E extends Number> void sort(List<E> l)
```

Unfortunately, this strategy cannot work for lower-bounded wildcards, since Java 5 prohibits the declaration of a corresponding lower-bounded type variable. This problem is a fundamental deficiency in the language’s support for wildcards: a “handle” or witness for certain wildcards is simply inexpressible without the loss of information about the wildcards’ bounds. (See Appendix A.4 for a pseudo-real-world example in which such a handle is needed.)

Given the significance of lower-bounded parameters, why are they prohibited? The *JLS* indirectly provides some insight into the language designers’ motivation for making this restriction. While discussing lower bounds on wildcards, it implies that allowing lower bounds on method type parameters would make type inference for these methods impossible: “Unlike ordinary type variables declared in a method signature, no type inference is required when using a wildcard. Consequently, it is permissible to declare lower bounds on a wildcard” [3, 4.5.1].

It’s easy to see that where inference consistently chooses the inferred lower bound as a parameter’s instantiation, that bound will often violate the parameter’s declared lower bound.

Where the declared lower bound is not defined in terms of $P_1 \dots P_n$, the solution is trivial: simply *join* the two bounds. But for interdependent lower bounds (probably rare in practice, if they serve any purpose at all), we can encounter the same problems as described in the previous section. Fortunately, wildcard capture can easily be extended to handle lower-bounded type parameters. And since many typical method invocations do not lead to inferred *upper* bounds, the capture-based strategy described in Section 2.4.4 will often produce useful, well-formed results. The only cases that can’t be handled, as in the previous section, are those in which a capture variable appearing in an interdependent bound makes the type incompatible with the opposite inferred bound. In practice, such occurrences involving interdependent lower bounds are probably quite rare.

2.4.6 Broader Inference Locality

Finally, perhaps the most visible shortcoming of Java 5 inference is its limited local scope. As has been discussed, the choice for σ at a particular call site is almost always determined exclusively by the types of the invocation’s arguments and the corresponding method parameters. But there are often a number of valid local choices for σ ; choosing the *best* one from a broader perspective depends on how the return type of the method is used in a wider context.

As a simple example, where a method with type parameter T returns values of type `List<T>`, an assignment to

a variable of type `List<Cloneable>` will *only* be valid if $\sigma = [T := Cloneable]$. Any more specific choice for T will render the program incorrect. Similarly, if we nest the expression in another method invocation, the best choice for σ depends on the corresponding formal parameter type. And if the outer method is overloaded or parametric, determining a choice for σ that will render the entire expression well-typed is quite complex.

This potential for complexity highlights a tension between two important language design goals. On the one hand, we do not want to force programmers to insert explicit type annotations where those annotations appear obvious and redundant. On the other hand, we do not want the complexity of inference to lead to programmer confusion, where programmers cannot easily predict what the results of inference (and, where overloading is involved, the behavior of the program) will be.

We can balance these concerns and still improve the algorithm by making more use of the bounds arising from an easily-determined expected type. As has been discussed, these bounds are derived from the following constraint:

$$\sigma R <: E$$

While, strictly speaking, any use of the expected type E allows context to influence the type of an expression, this is not a concern in practice because E is only defined where its value is obvious to programmers (specifically, where the method invocation is the value of a `return` statement, the initializer of a variable declaration, or the right-hand side of an assignment). The Java 5 algorithm makes use of these bounds in limited cases, but it would be simple to extend the algorithm to *always* use them where E is defined.

It might also be practical to define E in additional contexts, such as where the invocation is an argument to another method invocation, and the outer method is neither overloaded nor parametric.

3. Improved Algorithm

Having considered a variety of solutions to shortcomings in the Java 5 type argument inference algorithm, we now combine and formally present some of these solutions in a full algorithm definition.¹⁷ As was discussed, there are a number of ways to handle many of the algorithm’s limitations (including simply accepting them). Here we present one approach, motivated by a desire to address most of the above concerns while minimizing language changes. All the listed bugs have been fixed; the possibility of other bugs is minimized by defining subtyping in a syntax-directed manner, and by specifying subtype inference to correspond directly

¹⁷ This definition was originally published as one of two type system variations by Smith [10]. This version supersedes the first variation, and includes both some technical corrections and some improvements to the presentation. The second variation is not included here, but may be independently useful: it formalizes the *JLS*’s usage of recursive types.

with subtyping. The *join* operation is replaced with *union types*; recursive types are not present. Inference is able to handle disjunctive constraints by using *constraint formulas*, represented in disjunctive normal form. First-class intersection types and lower bounds on declared type variables are supported; wildcards and type variables are generalized to allow a single variable or wildcard to have both an upper and a lower bound. Inference still occurs on a per-invocation basis, but the expected return type is always used when it is available.

Unlike core calculi like Featherweight Java [4], this definition is intended to cover the full scope of the Java language. On the other hand, it is not a full language definition, but rather limited to a definition of types and the tools needed to analyze them.

3.1 Fundamentals

3.1.1 Types

A *type* is one of the following:

- The *null type* (denoted `null` here, but distinguishable by context from the value `null`).
- A *ground parameterized class type* $C\langle T_1 \dots T_n \rangle$, where C is a name and, for all i , T_i is a type.
- A *wildcard-parameterized class type* $C\langle W_1 \dots W_n \rangle$, where C is a name and, for all i , W_i is a *type argument*, which is one of:
 - A type.
 - A *wildcard* $? \text{ extends } T_u \text{ super } T_l$, where T_u and T_l are types. (A wildcard is *not* a type.)
- A *raw class type* C , where C is a name.
- A *primitive array type* $p[]$, where p is a primitive type name (`int`, `char`, etc.)
- A *reference array type* $T[]$, where T is a type.
- A *type variable* X , where X is a name.
- An *intersection type* $T_1 \ \& \ \dots \ \& \ T_n$, where, for all $i \leq n$, T_i is a type.
- A *union type* $T_1 \ | \ \dots \ | \ T_n$, where, for all $i \leq n$, T_i is a type.

For simplicity, we have ignored primitive types. Generally, primitives can be handled by implementations separately before deferring to the type operations defined here. We also ignore any distinction between classes and interfaces—hereafter, the word “class” means either a class or an interface.

The *names* referred to in the definition are assumed to be globally-unique identifiers. Every class and type variable declared by a program must have exactly one such name.¹⁸

¹⁸This is essentially what is meant by *canonical names*, as defined in the *JLS*. However, that definition does not apply to local classes and interfaces, nor to type variables.

All lists in this definition may be of any length, including 0. The type of a class C with no declared parameters is the ground parameterized class type $C\langle \rangle$ (but may informally be written C).

Types of nested classes do not appear explicitly in this definition. Instead, these are just treated like top-level classes. We follow the convention that a class’s list of parameters includes all type variables available from outer declarations.¹⁹ For example, if class `Foo` declares inner class `Bar`, the expression

```
new Foo<String, Object>().new Bar<Cloneable>()
```

has type

```
Foo.Bar<String, Object, Cloneable>
```

In Java code, we would instead write

```
Foo<String, Object>.Bar<Cloneable>
```

Intersections represent the most general type for which each of T_i is a supertype. If an intersection consists of some number of interface names, for example, any class that implements all the listed interfaces is a subtype of the intersection.

Complementing this notion, unions represent the least general type for which each of T_i is a subtype. Any common supertype of these types is also a supertype of the union. Because unions are not currently part of Java, certain operations involving these types, such as method lookup and erasure, are defined neither in the *JLS* nor in this paper. Igarashi and Nagira [6] develop object-based union types in depth and present possible definitions for these missing pieces.

In the notation that follows, we maintain the following conventions:

- X , Y , Z , P , and Q represent type variables (P and Q usually represent declared type parameters).
- C represents a class name.
- W represents a type argument—either a type or a wildcard.
- All other capital letters represent arbitrary types.

To simplify the definition of structurally-recursive functions, we will refer to the types of which a type is directly composed as its *component types*. The wildcard bounds of a wildcard-parameterized class type are among its component types.

3.1.2 Bounds

Type variables are always bounded—a valid instantiation of a variable must be a subtype of its *upper bound*, and a

¹⁹This list does *not* extend beyond a local scope—if a class is defined inside a method, its parameters do not include those of the method or of the enclosing class; the list also excludes variables that are not available because a class is declared `static`.

supertype of its *lower bound*. This information is provided by the source code, and where it is elided, `Object` is the default upper bound and `null` is the default lower bound.

The functions *upper* and *lower*, which map variables to their bounds, are implicit parameters (for conciseness) to most of the operations that follow. Additionally, the *capture* function may produce *new* variables, and thus new instances of *upper* and *lower*. These updated bound functions are implicitly threaded through all subsequent operations on the types produced by capture.

The expression $[X]$ is shorthand for the application of *upper* to X , producing X 's upper bound; similarly, $[X]$ produces X 's lower bound.

3.1.3 Structural Well-formedness

A type T is *structurally well-formed* (in the context of a set of class definitions) if and only if all of its component types are structurally well-formed and it violates none of the following assertions:

- Where $T = C\langle T_1 \dots T_n \rangle$, the class named C exists and has n type parameters.
- Where $T = C\langle W_1 \dots W_n \rangle$, the class named C exists and has n type parameters, and there exists some i such that W_i is a wildcard (thus $n \geq 1$).
- Where $T = C$, the class named C exists and has at least one type parameter.

Except where noted, all type operations defined below assume a domain of structurally well-formed types (this includes types passed as implicit arguments, such as a class's parameters or a variable's bounds). The safety of the type system relies on a stronger notion of *semantic* well-formedness, defined later in this section. This distinction is necessary because semantic well-formedness relies on subtyping and other type operations; we cannot in general guarantee the semantic well-formedness of the operations' arguments, and instead must settle for the structural checks defined here.

3.1.4 Substitution

Substitution instantiates a set of type variables, and is denoted σT or, more explicitly, $[P_1 := T_1 \dots P_n := T_n]T$. The types involved need not be well-formed. It is defined as the structurally-recursive application of the following rule:

$$[P_1 := T_1 \dots P_n := T_n]X = T_i \text{ if, for some } i, X = P_i; \text{ otherwise } [P_1 := T_1 \dots P_n := T_n]X = X.$$

By *structurally-recursive* we mean that, for arbitrary T , the substitution is applied to each T 's component types, and a new type is constructed from these modified types.

The bounds of a wildcard within a wildcard-parameterized type are components of that type; the bounds of a variable are not. Thus, substitution cannot be used directly to instantiate the bounds of a variable.

3.1.5 Wildcard Capture

Wildcard capture is an operation on type arguments (either types or wildcards, $W_1 \dots W_n$) and their corresponding type parameters ($P_1 \dots P_n$), producing a globally-unique variable for each wildcard. Each new variable has the same bounds as the wildcard, combined with the (instantiated) bounds of the corresponding type parameter.

$capture(W_1 \dots W_n, P_1 \dots P_n) = T_1 \dots T_n$, where, for all i :

- If W_i is a type, $T_i = W_i$.
- If W_i is the wildcard `? extends W_{iu} super W_{il}` , $T_i = Z_i$ for a fresh name Z_i , where:
 - $[Z_i] = W_{iu} \ \& \ [P_1 := T_1 \dots P_n := T_n][P_i]$.
 - $[Z_i] = W_{il} \ | \ [P_1 := T_1 \dots P_n := T_n][P_i]$.

Capture is principally used to convert a wildcard-parameterized class type to a ground parameterized class type. We use the notation $\|C\langle W_1 \dots W_n \rangle\|$ to represent such a conversion: where $P_1 \dots P_n$ are the type parameters of class C and $capture(W_1 \dots W_n, P_1 \dots P_n) = T_1 \dots T_n$, we have $\|C\langle W_1 \dots W_n \rangle\| = C\langle T_1 \dots T_n \rangle$.

3.1.6 Direct Supertype

Our subtyping definition relies on determining the *direct supertype* of a class type, denoted $T\uparrow$. This is defined as follows:

- Where $T = C\langle T_1 \dots T_n \rangle$,
 - If $C = \text{Object}$, $T\uparrow$ is undefined.
 - Else if C declares no supertypes, $T\uparrow = \text{Object}$.
 - Else C declares supertypes $S_1 \dots S_m$ and type parameters $P_1 \dots P_n$; let $\sigma = [P_1 := T_1 \dots P_n := T_n]$; $T\uparrow = \sigma S_1 \ \& \ \dots \ \& \ \sigma S_m$.
- Where $T = C\langle W_1 \dots W_n \rangle$, $T\uparrow = \|C\langle W_1 \dots W_n \rangle\|\uparrow$.
- Where $T = C$,
 - If C declares no supertypes, $T\uparrow = \text{Object}$.
 - If C declares supertypes $S_1 \dots S_m$, $T\uparrow = |S_1| \ \& \ \dots \ \& \ |S_m|$.

$|S_i|$, used in the raw case, denotes the *erasure* of the given type, as defined in the *JLS* (4.6).

The direct supertype operation is implicitly parameterized by a class table which contains the supertype declarations defined in the source code. All operations that depend on direct supertypes are similarly parameterized.

3.1.7 Subtype Relation

The type S is a *subtype* of T , denoted $S <: T$, if and only if this relationship can be demonstrated with the following set of inference rules:

$T <: T$ (REFLEX)

$$\frac{S <: S' \quad S' <: T}{S <: T} \text{ (TRANS)}$$

$\text{null} <: T$ (NULL)

$$\frac{\forall i \leq n, S_i \cong T_i}{C \langle S_1 \dots S_n \rangle <: C \langle T_1 \dots T_n \rangle} \text{ (CLASS-EQUIV)}$$

$$\frac{\forall i \leq n, S_i \in W_i}{C \langle S_1 \dots S_n \rangle <: C \langle W_1 \dots W_n \rangle} \text{ (CLASS-CONTAIN)}$$

$$\frac{\|C \langle W_1 \dots W_n \rangle\| <: T}{C \langle W_1 \dots W_n \rangle <: T} \text{ (CLASS-CAPT)}$$

$C \langle T_1 \dots T_n \rangle <: C$ (CLASS-ERASE)

$T <: T^\uparrow$ (CLASS-SUP)

$p[] <: \text{Cloneable} \ \& \ \text{Serializable}$ (PRIM-ARR-CLASS)

$T[] <: \text{Cloneable} \ \& \ \text{Serializable}$ (ARR-CLASS)

$$\frac{S <: T}{S[] <: T[]} \text{ (ARR-COVAR)}$$

$X <: [X]$ (VAR-SUP)

$[X] <: X$ (VAR-SUB)

$T_1 \ \& \ \dots \ \& \ T_n <: T_i$ (INTER-SUP)

$$\frac{\forall i \leq n, S <: T_i}{S <: T_1 \ \& \ \dots \ \& \ T_n} \text{ (INTER-SUB)}$$

$$\frac{\forall i \leq n, S_i <: T}{S_1 \ | \ \dots \ | \ S_n <: T} \text{ (UNION-SUP)}$$

$T_i <: T_1 \ | \ \dots \ | \ T_n$ (UNION-SUB)

If $S <: T$, then equivalently T is a *supertype* of S (denoted $T \succ S$). Where two types are mutual subtypes of each other—that is, $S <: T$ and $T <: S$ —we say that they are equivalent, denoted $S \cong T$.²⁰

The expression $S \in W$, used to express containment by a type argument (either a type or a wildcard) in CLASS-CONTAIN, is shorthand for the following:

- Where W is a type T , $S \cong T$.
- Where W is a wildcard $? \text{ extends } T_u \text{ super } T_l$, $S <: T_u \wedge T_l <: S$.

²⁰We use \cong in subtyping where Java 5 uses $=$. This is independent of the main concerns addressed in this paper, but is included as a convenience to programmers, since there exist types that are \cong , and can thus be used interchangeably, but that are not $=$.

Although it is sound, we do *not* include the following distribution rule for intersections and unions:²¹

$S \ \& \ (T_1 \ | \ \dots \ | \ T_n) <: (S \ \& \ T_1) \ | \ \dots \ | \ (S \ \& \ T_n)$ (DIST)

Table 1 reexpresses subtyping algorithmically—for any pair of types, subtyping is defined to hold if and only if a rule referenced in the corresponding cell of the table holds (S matches one of the cases in the left column; T matches one of the cases in the top row. A “-” in the table represents a result that is trivially *false*). The correspondence between the above declarative rules and the algorithmic rules in Table 1 is expressed formally in Section 3.1.9.

Note that certain rules may be applicable to an S – T pair but not appear in the corresponding table cell. For example, REFLEX is frequently elided; VAR-SUB* is not used where S is a union. In such cases, the elided rule is provably redundant. On the other hand, there are times (like the variable-variable case) where every applicable rule must be tested.

Implementing a subtyping algorithm in terms of these algorithmic rules is a straightforward process. In order to guarantee termination, however, we require the following of the subtyping arguments in addition to the assumption that the types be structurally well-formed:²²

- No variable is bounded by itself—that is, $[X]^+ \neq X$ and $[X]^+ \neq X$.
- The class table is acyclic: $C \langle T_1 \dots T_n \rangle^{\uparrow+} \neq C \langle T'_1 \dots T'_n \rangle$ for any choice of $T'_1 \dots T'_n$.
- The class table does not exhibit *expansive inheritance*, as defined by Kennedy and Pierce [7].

Unlike the *JLS*, we do *not* prohibit multiple-instantiation inheritance: we might have $C \langle T_1 \dots T_n \rangle^{\uparrow+} = D \langle S_1 \dots S_m \rangle$ and $C \langle T_1 \dots T_n \rangle^{\uparrow+} = D \langle S'_1 \dots S'_m \rangle$ where, for some i , $S_i \neq S'_i$.

Even with these limitations, certain subtyping invocations may depend on themselves (Kennedy and Pierce [7] provide some examples). So the algorithm must keep track of “in-process” invocations and terminate (negatively) whenever a subtyping invocation depends on itself.

3.1.8 Bounds Checking

We use *inBounds* to assert that type arguments ($T_1 \dots T_n$) do not violate the bound assertions of their corresponding type parameters ($P_1 \dots P_n$).

$\text{inBounds}(T_1 \dots T_n, P_1 \dots P_n)$ is defined for structurally well-formed types as follows:

- For all i , $T_i <: [P_1 := T_1 \ \dots \ P_n := T_n][P_i]$.
- For all i , $[P_1 := T_1 \ \dots \ P_n := T_n][P_i] <: T_i$.

²¹A distribution rule could (and ought to) be added as an extension to the current presentation, although defining a straightforward subtyping algorithm under such a rule requires tedious normalization steps.

²²The use of transitive closure here is intended to also permit the decomposition of intersection and union types.

		T :				
		null	$C_t \langle T_1 \dots T_m \rangle$	$C_t \langle W_1 \dots W_m \rangle$	C_t	
		true	true	true	true	
S :	$C_s \langle S_1 \dots S_n \rangle$	-	CLASS-EQUIV, CLASS-SUP*	CLASS-CONTAIN, CLASS-SUP*	CLASS-ERASE*, CLASS-SUP*	
	$C_s \langle W_1 \dots W_n \rangle$	-	CLASS-CAPT*	CLASS-CAPT*	CLASS-CAPT*	
	C_s	-	CLASS-SUP*	CLASS-SUP*	REFLEX, CLASS-SUP*	
	$p_s \square$	-	PRIM-ARR-CLASS*	PRIM-ARR-CLASS*	PRIM-ARR-CLASS*	
	$S' \square$	-	ARR-CLASS*	ARR-CLASS*	ARR-CLASS*	
	X_s	VAR-SUP*	VAR-SUP*	VAR-SUP*	VAR-SUP*	
	$S_1 \ \& \dots \ \& \ S_n$	INTER-SUP*	INTER-SUP*	INTER-SUP*	INTER-SUP*	
	$S_1 \ \dots \ \ S_n$	UNION-SUP	UNION-SUP	UNION-SUP	UNION-SUP	
		T :				
		$p_t \square$	$T' \square$	X_t	$T_1 \ \& \dots \ \& \ T_m$	$T_1 \ \dots \ \ T_m$
		true	true	true	true	true
S :	$C_s \langle S_1 \dots S_n \rangle$	-	-	VAR-SUB*	INTER-SUB	UNION-SUB*
	$C_s \langle W_1 \dots W_n \rangle$	-	-	VAR-SUB*	INTER-SUB	UNION-SUB*
	C_s	-	-	VAR-SUB*	INTER-SUB	UNION-SUB*
	$p_s \square$	REFLEX	-	VAR-SUB*	INTER-SUB	UNION-SUB*
	$S' \square$	-	ARR-COVAR	VAR-SUB*	INTER-SUB	UNION-SUB*
	X_s	VAR-SUP*	VAR-SUP*	REFLEX, VAR-SUP*, VAR-SUB*	INTER-SUB	VAR-SUP*, UNION-SUB*
	$S_1 \ \& \dots \ \& \ S_n$	INTER-SUP*	INTER-SUP*	INTER-SUP*, VAR-SUB*	INTER-SUB	INTER-SUP*
$S_1 \ \dots \ \ S_n$	UNION-SUP	UNION-SUP	UNION-SUP	UNION-SUP	UNION-SUP	

$T <: T$ (REFLEX)

$$\frac{\forall i, S_i \cong T_i}{C \langle S_1 \dots S_n \rangle <: C \langle T_1 \dots T_n \rangle} \text{ (CLASS-EQUIV)}$$

$$\frac{\forall i, S_i \in W_i}{C \langle S_1 \dots S_n \rangle <: C \langle W_1 \dots W_n \rangle} \text{ (CLASS-CONTAIN)}$$

$$\frac{\|C \langle W_1 \dots W_n \rangle\| <: T}{C \langle W_1 \dots W_n \rangle <: T} \text{ (CLASS-CAPT*)}$$

$$\frac{C <: T}{C \langle S_1 \dots S_m \rangle <: T} \text{ (CLASS-ERASE*)}$$

$$\frac{S \uparrow <: T}{S <: T} \text{ (CLASS-SUP*)}$$

$$\frac{\text{Cloneable \& Serializable } <: T}{p_s \square <: T} \text{ (PRIM-ARR-CLASS*)}$$

$$\frac{\text{Cloneable \& Serializable } <: T}{S \square <: T} \text{ (ARR-CLASS*)}$$

$$\frac{S <: T}{S \square <: T \square} \text{ (ARR-COVAR)}$$

$$\frac{[X] <: T}{X_s <: T} \text{ (VAR-SUP*)}$$

$$\frac{S <: [X_t]}{S <: X_t} \text{ (VAR-SUB*)}$$

$$\frac{\exists i, S_i <: T}{S_1 \ \& \dots \ \& \ S_n <: T} \text{ (INTER-SUP*)}$$

$$\frac{\forall i, S <: T_i}{S <: T_1 \ \& \dots \ \& \ T_n} \text{ (INTER-SUB)}$$

$$\frac{\forall i, S_i <: T}{S_1 \ | \dots \ | \ S_n <: T} \text{ (UNION-SUP)}$$

$$\frac{\exists i, S <: T_i}{S <: T_1 \ | \dots \ | \ T_m} \text{ (UNION-SUB*)}$$

Table 1. Algorithmic rules for subtyping

3.1.9 Semantic Well-formedness

In order to make useful assertions about the correctness of the above operations, a stronger notion of well-formedness is needed. Thus, a type is *semantically well-formed* (in the context of a set of class definitions) if and only if it is structurally well-formed, all of its component types are semantically well-formed, and it violates none of the following assertions:

- Where $T = C\langle T_1 \dots T_n \rangle$, and class C has parameters $P_1 \dots P_n$, $\text{inBounds}(T_1 \dots T_n, P_1 \dots P_n)$.
- Where $T = C\langle W_1 \dots W_n \rangle$, $\|C\langle W_1 \dots W_n \rangle\|$ is semantically well-formed.
- Where $T = X$, $\lfloor X \rfloor <: \lceil X \rceil$.

“Well-formed,” when used without qualification, refers to semantic well-formedness. In the context of a full language definition, all types expressed in code should be well-formed, and type analysis must only produce new types that are well-formed.

Note the use of capture in validating the arguments of a wildcard-parameterized class type. It is tempting to try to avoid capture conversion here, and in many situations its use can be eliminated. However, in general, we must use capture to insure two important conditions: first, that the variables generated by capture are not malformed—each variable’s lower bound is a subtype of its upper bound; and second, that non-wildcard arguments are within their bounds. Bounds in both cases may be defined in terms of capture variables and other type arguments, so the bounds must be instantiated before they are checked.²³

Given this stronger well-formedness notion, we can make a few important assertions about substitution, wildcard capture, and subtyping. (We do not provide in this paper proofs for these assertions. They do, however, provide a standard by which to informally verify correctness.)

Substitution. The notion of well-formedness allows us to make the following definition and assertion regarding substitution:

Definition. An invocation $[P_1 := T_1 \dots P_n := T_n]T$ is well-formed if and only if T , $P_1 \dots P_n$, and $T_1 \dots T_n$ are well-formed and $\text{inBounds}(T_1 \dots T_n, P_1 \dots P_n)$.

Theorem. Where the substitution invocation σT is well-formed, its result is also well-formed.

The following lemmas help to demonstrate this result:

Lemma. Where the substitution invocations are well-formed, $T_1 <: T_2 \Rightarrow \sigma T_1 <: \sigma T_2$.

Lemma. Where the substitution invocations are well-formed and the parameters $Q_1 \dots Q_m$ do not have bounds that in-

²³ We do not need to check the inBounds condition for capture variables, since these variables are guaranteed to be in bounds, but we don’t complicate the definition with this fact here.

volve the parameters of σ , $\text{inBounds}(S_1 \dots S_m, Q_1 \dots Q_m) \Rightarrow \text{inBounds}(\sigma S_1 \dots \sigma S_m, Q_1 \dots Q_m)$.

Wildcard capture. In general, capture may produce malformed types from well-formed arguments—this is why the rules for semantic well-formedness check that the type is well-formed *after* capture. We can, however, make the following claim:

Theorem. Where $W_1 \dots W_n$ are all wildcards and the types $\text{capture}(W_1 \dots W_n, P_1 \dots P_n)$ are well-formed,

$$\text{inBounds}(\text{capture}(W_1 \dots W_n, P_1 \dots P_n), P_1 \dots P_n)$$

Subtyping. Well-formedness allows us to make the following soundness and completeness claim about the subtyping algorithm:

Theorem. Where S , T , and all types in the implicit environment (variable bounds and a class table) are well-formed, the assertion $S <: T$ is true according to algorithmic subtyping (defined in Table 1) if and only if it is derivable by the declarative subtyping inference rules.

3.2 Type Argument Inference

3.2.1 Overview

The type argument inference algorithm produces an instantiation $\sigma = [P_1 := T_1 \dots P_n := T_n]$ of a set of method type parameters for a specific call site. The result is a function of the types of the formal parameters ($F_1 \dots F_m$), the types of the invocation’s arguments ($A_1 \dots A_m$), the method’s return type (R), and the type expected in the call site’s context (E). An inference result must satisfy the following:

$$\begin{aligned} \forall i, A_i <: \sigma F_i \\ \sigma R <: E \\ \text{inBounds}(T_1 \dots T_n, P_1 \dots P_n) \end{aligned}$$

We proceed by first producing a set of bounding constraints satisfying first two conditions, and then choosing types that both meet these constraints and fall within bounds specified by $P_1 \dots P_n$. The algorithm is sound, but not complete: the results will always satisfy the three above constraints, but where $P_1 \dots P_n$ are referenced within their own bounds, it may fail to produce a result where one exists.²⁴ In all other cases, it is complete.

Bounding constraints on the type arguments are determined by two functions, $<:_{\text{?}}$ and $:>_{\text{?}}$. $A <:_{\text{?}} F$ produces a minimal set of constraints on $T_1 \dots T_n$ required to satisfy $A <: \sigma F$; $A :>_{\text{?}} F$ similarly produces the constraints satisfying $A :> \sigma F$. The constraints are expressed as logical formulas, combined and normalized with the operations \wedge_{cf} and \vee_{cf} as outlined below. For convenience, a third inference function, $\cong_{\text{?}}$, is a shortcut for $\wedge_{\text{cf}}(A <:_{\text{?}} F, A :>_{\text{?}} F)$.

²⁴ See Section 2.4.4 for discussion regarding this special case.

3.2.2 Constraint Formulas

A *constraint formula* is a formula in first-order logic expressing upper and lower bounds on our choices for types $T_1 \dots T_n$. Where a certain instantiation contains types that fall within the bounds expressed by a formula ϕ , that instantiation *satisfies* the formula: $\sigma \models \phi$. We also use this notation for implication: $\phi \models \rho$ means $\forall \sigma, (\sigma \models \phi) \Rightarrow (\sigma \models \rho)$.

In the inference algorithm, we restrict the form of all constraint formulas as follows, modeled after disjunctive normal form:

$$\bigvee_{j=1}^m T_{1jl} <: T_1 <: T_{1ju} \wedge \dots \wedge T_{njl} <: T_n <: T_{nju}$$

The value of m may be any natural number. We will use *false* as an abbreviation for the formula in which $m = 0$. If $m = 1$, the formula is a *simple constraint formula*; we use *true* to represent the simple formula

$$\text{null} <: T_1 <: \text{Object} \wedge \dots \wedge \text{null} <: T_n <: \text{Object}$$

Finally, an expression such as $C <: T_1 <: D$ is taken as an abbreviation for a simple constraint formula in which the given parameter has the specified bounds, and all other parameters are bounded by the unconstraining `null` and `Object`.

It will be necessary to produce conjunctions and disjunctions of constraint formulas. The operations \wedge_{cf} and \vee_{cf} serve this purpose, while maintaining the invariant normalized form. These operations, defined below, are sound and complete with respect to simple conjunction and disjunction:

Lemma. $\sigma \models (\phi \wedge \rho)$ if and only if $\sigma \models \wedge_{\text{cf}}(\phi, \rho)$.

Lemma. $\sigma \models (\phi \vee \rho)$ if and only if $\sigma \models \vee_{\text{cf}}(\phi, \rho)$.

Conjunction. Let $\rho_1 \dots \rho_m$ be simple constraint formulas. Let T_{ijl} refer to the lower bound of T_i in ρ_j , and T_{iju} refer to the upper bound. Then $\wedge_{\text{cf}}(\rho_1 \dots \rho_m)$ has value

$$\bigwedge_{i=1}^n (T_{i1l} \mid \dots \mid T_{iml}) <: T_i <: (T_{i1u} \ \& \ \dots \ \& \ T_{imu})$$

The construction of unions and intersections here is required to eliminate redundant entries: `String & Object` reduces to `String`, for example. If the result is unsatisfiable—that is, for some T_i , the lower bound is not a subtype of the upper bound—it is simplified to *false*. Note also that if any of ρ_j is *true* (and $m > 1$), that formula is automatically discarded (because its bounds are always redundant).

In the general case—where the arguments $\phi_1 \dots \phi_m$ are not simple—we define \wedge_{cf} by merging each possible combination of simple constraint formulas. In this case, each of $\phi_1 \dots \phi_m$ can be treated as a *set* of simple formulas; the cross product of these sets, $\phi_1 \times \dots \times \phi_m$, produces k m -tuples of the form $(\rho_1 \dots \rho_m)$. Applying \wedge_{cf} to each of these tuples (as defined above for simple formulas), we produce the set of simple formulas $\rho'_1 \dots \rho'_k$. Then we have

$$\wedge_{\text{cf}}(\phi_1 \dots \phi_m) = \vee_{\text{cf}}(\rho'_1 \dots \rho'_k)$$

Again, we note that if any of ϕ_j is *true*, that set will be discarded; if any of ϕ_j is *false*, the result will also be *false* (because $k = 0$).

Disjunction. The \vee_{cf} operation would be correct to simply concatenate its arguments together. However, we wish to ensure that all formulas we produce are minimal. We can use the following to help eliminate redundant simple formulas:

Theorem. For simple constraint formulas ρ_1 and ρ_2 , $\rho_1 \models \rho_2$ if and only if, for all i , $T_{i1u} <: T_{i2u}$ and $T_{i1l} >: T_{i2l}$.

Now, we define $\vee_{\text{cf}}(\phi_1 \dots \phi_m)$ as follows. Again treating these formulas as sets of simple formulas, let $\rho_1 \dots \rho_k$ be the union $\phi_1 \cup \dots \cup \phi_m$. We can compute a minimal equivalent subset of $\rho_1 \dots \rho_k$, $\rho'_1 \dots \rho'_k$, where *minimal* means that $\forall i, \exists j, \rho_i \models \rho'_j$ (the intuition is that if one formula implies another, the first is more constraining on σ ; the only way a choice for σ will satisfy *neither* formula is if it does not satisfy the less constraining one). Now we have

$$\vee_{\text{cf}}(\phi_1 \dots \phi_m) = \bigvee_{i=1}^{k'} \rho'_i$$

Again note that the trivial cases are handled correctly: if any of $\phi_1 \dots \phi_m$ is *false*, it will be ignored; if any of $\phi_1 \dots \phi_m$ is *true*, it will be the *only* member of the minimal subset, and the result will be *true*.

3.2.3 Subtype Inference

The invocation $A <:_{\text{?}} F \mid_{\mu}$ produces a constraint formula supporting the assumption that $A <: \sigma F$. The parameter μ is a set of previous invocations of $<:_{\text{?}}$ and $>:_{\text{?}}$. For brevity, we do not express μ explicitly; it is always empty on external invocations, and wherever one of these operations is recursively invoked (including mutual recursion between $<:_{\text{?}}$, $>:_{\text{?}}$, and $\cong_{\text{?}}$), the previous invocation is accumulated in μ .

- If the invocation $A <:_{\text{?}} F \in \mu$, the result is *false*.
- Else if, for some i , $F = P_i$, the result is $A <: T_i <: \text{Object}$.
- Else if F involves none of $P_1 \dots P_n$, the result is $A <: F$ (treating the boolean result of $<:_{\text{?}}$ as a trivial constraint formula).
- Otherwise, the result is given in Table 2. (A matches one of the cases in the left column; F matches one of the cases in the top row. A “-” in the table represents the formula *false*.)

Compare Table 2 to Table 1. Notice that the rules for inference follow directly from subtyping. The only changes replace boolean operations with their analogs: $<:_{\text{?}}$ becomes $<:_{\text{?}}$; “and” and “or” become \wedge_{cf} and \vee_{cf} .

3.2.4 Supertype Inference

The invocation $A >:_{\text{?}} F \mid_{\mu}$ produces a constraint formula supporting the assumption that $A >: \sigma F$. The parameter μ is as described in the previous section.

		F :		
		$C_f \langle F_1 \dots F_m \rangle$	$C_f \langle W_1 \dots W_m \rangle$	$F' \square$
	null	<i>true</i>	<i>true</i>	<i>true</i>
	$C_a \langle A_1 \dots A_n \rangle$	[1]	[2]	-
	$C_a \langle W_1 \dots W_n \rangle$	$\ A\ <{:?} F$	$\ A\ <{:?} F$	-
	C_a	$A \uparrow <{:?} F$	$A \uparrow <{:?} F$	-
A:	$p_s \square$	[4]	[4]	-
	$A' \square$	[4]	[4]	$A' <{:?} F'$
	X_a	$[A] <{:?} F$	$[A] <{:?} F$	$[A] <{:?} F$
	$A_1 \ \& \dots \ \& \ A_n$	$\vee_{\mathit{cf}}(A_i <{:?} F)$	$\vee_{\mathit{cf}}(A_i <{:?} F)$	$\vee_{\mathit{cf}}(A_i <{:?} F)$
	$A_1 \ \dots \ \ A_n$	$\wedge_{\mathit{cf}}(A_i <{:?} F)$	$\wedge_{\mathit{cf}}(A_i <{:?} F)$	$\wedge_{\mathit{cf}}(A_i <{:?} F)$

		F :		
		X_f	$F_1 \ \& \dots \ \& \ F_m$	$F_1 \ \dots \ \ F_m$
	null	<i>true</i>	<i>true</i>	<i>true</i>
	$C_a \langle A_1 \dots A_n \rangle$	$A <{:?} [F]$	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	$\vee_{\mathit{cf}}(A <{:?} F_i)$
	$C_a \langle W_1 \dots W_n \rangle$	$A <{:?} [F]$	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	$\vee_{\mathit{cf}}(A <{:?} F_i)$
	C_a	$A <{:?} [F]$	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	$\vee_{\mathit{cf}}(A <{:?} F_i)$
A:	$p_s \square$	$A <{:?} [F]$	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	$\vee_{\mathit{cf}}(A <{:?} F_i)$
	$A' \square$	$A <{:?} [F]$	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	$\vee_{\mathit{cf}}(A <{:?} F_i)$
	X_a	[5]	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	[6]
	$A_1 \ \& \dots \ \& \ A_n$	[7]	$\wedge_{\mathit{cf}}(A <{:?} F_i)$	$\vee_{\mathit{cf}}(A <{:?} F_i)$
	$A_1 \ \dots \ \ A_n$	$\wedge_{\mathit{cf}}(A_i <{:?} F)$	$\wedge_{\mathit{cf}}(A_i <{:?} F)$	$\wedge_{\mathit{cf}}(A_i <{:?} F)$

[1]: There are two cases:

- If $C_a = C_f, \wedge_{\mathit{cf}}(A_1 \cong? F_1, \dots, A_n \cong? F_n)$
- Otherwise, $A \uparrow <{:?} F$

[2]: There are two cases:

- If $C_a = C_f, \wedge_{\mathit{cf}}(\phi_1 \dots \phi_n)$ where, for all i :
 - If W_i is a type, $\phi_i = A_i \cong? W_i$
 - If W_i is a wildcard $? \text{ extends } F_{iu} \text{ super } F_{il}$, $\phi_i = \wedge_{\mathit{cf}}(A_i <{:?} F_{iu}, A_i :>? F_{il})$
- Otherwise, $A \uparrow <{:?} F$

[4]: Cloneable & Serializable $<{:?} F$

[5]: $\vee_{\mathit{cf}}([A] <{:?} F, A <{:?} [F])$

[6]: $\vee_{\mathit{cf}}([A] <{:?} F, A <{:?} F_1 \dots A <{:?} F_m)$

[7]: $\vee_{\mathit{cf}}(A_1 <{:?} F \dots A_n <{:?} F, A <{:?} [F])$

Table 2. Rules for subtype inference

- If the invocation $A :>? F \in \mu$, the result is *false*.
- If, for some i , $F = P_i$, the result is $\text{null} <: T_i <: A$.
- If F involves none of $P_1 \dots P_n$, the result is $F <: A$ (treating the boolean result of $<:$ as a trivial constraint formula).
- Otherwise, the result is given in Table 3. (A matches one of the cases in the left column; F matches one of the cases in the top row. A “-” in the table represents the formula *false*.)

Similarly to Table 2, Table 3 follows directly from the subtyping rules (this is slightly less apparent, because the ta-

ble has been transposed, allowing the “upper” type to appear on the left).

3.2.5 Inference Algorithm

Building on these definitions, we now describe the full algorithm for type argument inference.

The first two conditions to be satisfied by the instantiation σ —that the invocation’s arguments are subtypes of their corresponding formal parameters, and that the return type is a subtype of the expected type—are described by the formula

$$\phi = \wedge_{\mathit{cf}}(A_1 <{:?} F_1 \dots A_m <{:?} F_m, E :>? R)$$

		$F:$		
		$C_f \langle F_1 \dots F_m \rangle$	$C_f \langle W_1 \dots W_m \rangle$	$F' \square$
	null	-	-	-
	$C_a \langle A_1 \dots A_n \rangle$	[1]	$A :>? \ F\ $	[4]
	$C_a \langle W_1 \dots W_n \rangle$	[2]	$A :>? \ F\ $	[4]
	C_a	[3]	$A :>? \ F\ $	[4]
A:	$p_s \square$	-	-	-
	$A' \square$	-	-	$A' :>? F'$
	X_a	$[A] :>? F$	$[A] :>? F$	$[A] :>? F$
	$A_1 \ \& \ \dots \ \& \ A_n$	$\bigwedge_{cf}(A_i :>? F)$	$\bigwedge_{cf}(A_i :>? F)$	$\bigwedge_{cf}(A_i :>? F)$
	$A_1 \ \ \dots \ \ A_n$	$\bigvee_{cf}(A_i :>? F)$	$\bigvee_{cf}(A_i :>? F)$	$\bigvee_{cf}(A_i :>? F)$

		$F:$		
		X_f	$F_1 \ \& \ \dots \ \& \ F_m$	$F_1 \ \ \dots \ \ F_m$
	null	$A :>? [F]$	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$
	$C_a \langle A_1 \dots A_n \rangle$	$A :>? [F]$	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$
	$C_a \langle W_1 \dots W_n \rangle$	$A :>? [F]$	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$
	C_a	$A :>? [F]$	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$
A:	$p_s \square$	$A :>? [F]$	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$
	$A' \square$	$A :>? [F]$	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$
	X_a	[5]	[7]	$\bigwedge_{cf}(A :>? F_i)$
	$A_1 \ \& \ \dots \ \& \ A_n$	$\bigwedge_{cf}(A_i :>? F)$	$\bigwedge_{cf}(A_i :>? F)$	$\bigwedge_{cf}(A :>? F_i)$
	$A_1 \ \ \dots \ \ A_n$	[6]	$\bigvee_{cf}(A :>? F_i)$	$\bigwedge_{cf}(A :>? F_i)$

[1]: There are two cases:

- If $C_a = C_f, \bigwedge_{cf}(A_1 \cong? F_1, \dots, A_n \cong? F_n)$
- Otherwise, $A :>? F \uparrow$

[2]: There are two cases:

- If $C_a = C_f, \bigwedge_{cf}(\phi_1 \dots \phi_n)$ where, for all i :
 - If W_i is a type, $\phi_i = W_i \cong? F_i$
 - If W_i is a wildcard $? \text{ extends } A_{iu} \text{ super } A_{il}, \phi_i = \bigwedge_{cf}(A_{iu} :>? F_i, A_{il} <:? F_i)$
- Otherwise, $A :>? F \uparrow$

[3]: If $C_a = C_f, \text{ true}$; otherwise, $A :>? F \uparrow$

[4]: $A :>? \text{ Cloneable \& Serializable}$

[5]: $\bigvee_{cf}(A :>? [F], [A] :>? F)$

[6]: $\bigvee_{cf}(A :>? [F], A_1 :>? F \dots A_n :>? F)$

[7]: $\bigvee_{cf}(A :>? F_1 \dots A :>? F_m, [A] :>? F)$

Table 3. Rules for supertype inference

Given the formula ϕ , we must choose types for $T_1 \dots T_n$ satisfying the bounds of the corresponding parameters: $\text{inBounds}(T_1 \dots T_n, P_1 \dots P_n)$.

We first choose values for $T_1 \dots T_n$ based on the inferred lower bounds: for each conjunction in ϕ of the form

$$T_{il} <: T_1 <: T_{1u} \wedge \dots \wedge T_{nl} <: T_n <: T_{nu}$$

we choose $T_i = T_{il}$. If this choice satisfies the inBounds condition, that is the result. Otherwise, the next disjunct in ϕ is used.

If no solution is found using the inferred lower bounds, we instead use capture to produce the results. If we treat each assertion $T_{il} <: T_i <: T_{iu}$ in a constraint formula as a

wildcard— $? \text{ extends } T_{iu} \text{ super } T_{il}$ —we can represent ϕ as follows:

$$\phi = \bigvee_{j=1}^m W_{1j} \dots W_{nj}$$

To satisfy the bounds, we let

$$T_1 \dots T_n = \text{capture}(W_{11} \dots W_{n1}, P_1 \dots P_n)$$

If the resulting capture variables are well-formed, these are the choice for $T_1 \dots T_n$. Otherwise, the next disjunct in ϕ is used. If no results are found in this way, the algorithm reports failure.

Note that there is a nondeterminacy present in the above algorithm: where more than one simple constraint formula

in the final constraints is satisfiable, the choice of *which* formula to use depends on the order in which they are enumerated. This nondeterminacy is inherent in the inference problem: if the constraints on T_1 can be satisfied with either $T_1 = \text{String}$ or $T_1 = \text{Integer}$ (but not with $T_1 = \text{null}$), the algorithm must arbitrarily choose one or the other. Clearly, such nondeterminacy must be avoided in a full specification—two different implementations must not choose different types for T_1 . To do so, the specification would need to extend the treatment of formula operations in terms of sets to preserve a well-defined order of elements.

3.2.6 Correctness and Complexity

We do not make a formal analysis of the correctness or complexity of the inference algorithm here. We do, however, state some useful properties that we expect to hold and informally discuss the algorithm’s efficiency.

First, termination is closely tied to the termination of subtyping.

Lemma. *The type argument inference algorithm terminates if the corresponding subtyping algorithm terminates.*

The correctness of the algorithm is also closely tied to that of subtyping.

Lemma. *The results of $<:_{\tau}$ and $:>_{\tau}$ are sound and complete with respect to the subtyping rules: $\sigma \models A <:_{\tau} F$ if and only if $\sigma A <: \sigma F$ (and the equivalent for $:>_{\tau}$).*

Given this assertion, soundness is straightforward.

Theorem. *If type argument inference produces a result $\sigma = [P_1 := T_1 \dots P_n := T_n]$, the corresponding method invocation is well-typed (as is the enclosing expression, where E is defined and the expression is otherwise correct):*

- $\forall i, A_i <: \sigma F_i$
- $\sigma R <: E$
- $\text{inBounds}(T_1 \dots T_n, P_1 \dots P_n)$

The completeness of $<:_{\tau}$ and $:>_{\tau}$ is similarly important in understanding the algorithm’s limitations: the only source of incompleteness is in choosing an instantiation that satisfies both the inferred and the declared bounds. In the typical, simple case in which parameter bounds are not recursive or interdependent, the algorithm can be expected to produce valid results when they exist.

To address efficiency, note that the dominating source of complexity (both in time and space) is in the manipulation of constraint formulas: \wedge_{cf} , in particular, calculates a cross product that may produce up to $m \times n$ disjuncts when given arguments with m and n disjuncts, respectively. Union and intersection types, as constructed by \wedge_{cf} , also have the potential to grow to intractable sizes. However, these are consistently minimized to eliminate redundancy; and, as noted in Section 2.4.2, we expect the sizes of the algorithm’s inputs to be quite small in practice.

3.2.7 Special Cases

When the above inference algorithm is used in the context of the full Java language, a variety of subtleties must be addressed:

- E may be undefined, or R may be `void`. Then there are no constraints on the return type, and we do not include $E :>_{\tau} R$ in the result.
- The types involved may be primitives. The inference operations can be easily extended to handle both primitive subtyping and reference subtyping, as appropriate.
- Boxing or unboxing of the arguments or return value may be allowed. Determining whether these conversions should occur is always possible without knowing σ . So we can assume here that $A_1 \dots A_m$ and E represent the types *after* any necessary conversions.
- Variable-length arguments may be used. In this case, the method signature provides formal parameter types $F_1 \dots F_j$, and F_j is the array type $F'_j \square$. The constraint formula calculation must then contain $A_1 <:_{\tau} F_1, \dots, A_{j-1} <:_{\tau} F_{j-1}$, and, if $m \geq j$, $A_j <:_{\tau} F'_j, \dots, A_m <:_{\tau} F'_j$.
- The type parameters of a class enclosing the method declaration may appear in $F_1 \dots F_m, R$, or the bounds of $P_1 \dots P_n$. Substitution can be used to remove these from $F_1 \dots F_m$ and R ; but in order to handle any references in $P_1 \dots P_n$, we must define new variables $P'_1 \dots P'_n$ with bounds defined by the class parameter instantiations. Alternately, the class parameters can be included in the list of parameters to be “inferred,” but be constrained so that the only valid choice to instantiate a class parameter is the one that has already been provided.

4. Backwards Compatibility

Enhancements to the Java language are generally made in a backwards-compatible fashion: the revised language is a superset of the previous version, and the behavior of previous programs is preserved. Unfortunately, changes to the current specification that affect *join* and type argument inference are almost impossible to make without rendering some programs incorrect, and changing the behavior of others.

Consider, for example, the signature of the method `java.util.Arrays.asList`:

```
static <T> List<T> asList(T... ts)
```

If this method is invoked in a context in which the expected type E is unknown—as an argument to another method, for example—invariant subtyping can easily cause a correct program to become incorrect with only slight modifications to the inference algorithm. That is, where the original algorithm produces $T_1 = U$ and the context of the invocation requires a `List<U>`, an algorithm that produces

a *better* but different type V will lead to an assertion that `List<V> <: List<U>`, which is false.

More troubling is the possibility that a change to *join* or the inference algorithm, while not invalidating a certain previously well-formed program, will change the *meaning* of that program. This is possible because overloading resolution is dependent on the types produced by type checking. The value of the `test` method below, for example, depends on the sophistication of the inference algorithm used:

```
interface NumBox<T extends Number> {
    T get();
}
static <T> T unwrap(NumBox<? extends T> arg) {
    return arg.get();
}
static int f(Object o) { return 0; }
static int f(Number n) { return 1; }

static int test(NumBox<?> arg) {
    return f(unwrap(arg));
}
```

A system with an inference algorithm that uses capture (or otherwise incorporates the declared bounds of a wildcard's corresponding parameter) can determine that the `f(Number)` function is applicable in the body of `test`; one that does not will instead resolve `f` to the `f(Object)` function.

Despite these incompatibilities, the bugs in the Java 5 specification (as described in Section 2.3), provide strong motivation for fixing these operations—even if the additional shortcomings of the inference algorithm are not addressed. So we are left with a problem: do we go to great lengths to enforce backwards compatibility with broken operations (perhaps by defining two inference algorithms, and using the second only when the first is unsuccessful), or relax this requirement in order to correct and clean up the specification? Complicating this question is the fact that the `javac` compiler, and presumably others, is not entirely consistent with the specification, especially in areas where the specification is incorrect. So it's not clear exactly which language any changes should seek to be backwards-compatible with.

We believe backwards-compatibility concerns can be mitigated in two ways. First, a new source-language compiler flag can be introduced, as was done in Java 1.4 when the `assert` keyword was added to the language. Second, a source-to-source tool can be developed that implements both the old and new inference algorithms, and inserts casts or explicit type arguments as necessary wherever the two conflict. In fact, because most programmers do not heavily exercise the language's generic features, it's quite likely problems would be rare enough that this diagnostic tool need not make any file modifications—it could simply identify a handful of problem sites and leave programmers to manually fix them.

Two properties of the algorithm specified in this paper soften the impact of the language change, minimizing the number of correct programs that would be rendered incorrect by the new system:

- Because type argument inference is defined in terms of the expected type E , changes to inference will rarely be problematic in contexts in which E is known (this includes assignments and return statements).
- Where inference produces a different result than the Java 5 algorithm, the new result is usually more specific; in practice, this is often safe, since type variables in method return types are frequently not nested.

5. Historical Evolution and Related Work

Algorithms for local type argument inference in languages with subtyping and bounded quantification was first explored by Cardelli [2] and later Pierce and Turner [9; 8]. Pierce and Turner noted the difficulty of performing inference for type parameters with interdependent bounds [8].

Type variables, parameterized types, and type argument inference in Java 5 were incorporated from the GJ language [1], an extension to Java designed to support generic programming. The original specification for GJ describes most of the features of Java 5, with the exception of wildcards and intersection types.

Wildcards arose out of research to extend GJ and similar languages with covariant and contravariant subtyping. Thorup and Torgersen [11] initially proposed what has become known as use-site covariance—allowing programmers to specify when a parameterized type is instantiated that a particular type parameter should be covariant. Igarashi and Viroli [5] extended this notion to include contravariance and established a connection to bounded existential types. Their work requires support for lower bounds on type variables, though these bounds are not expressible in type variable declarations. A joint project between the University of Aarhus and Sun Microsystems [12] extended these ideas and merged them with the rest of the Java language, describing in particular how wildcards affect type operations like type argument inference. Wildcard capture was first presented in a paper summarizing this project.

The 3rd edition of the *Java Language Specification* [3] enhanced this prior work in a number of ways. Wildcard capture was refined to produce variables whose bounds include both those of the wildcard and those of the corresponding type parameter. This enhancement produces a more useful capture variable, and may have been deemed necessary in order to guarantee that types produced by capture are well-formed (that is, the capture variable is within the declared parameter's bound). It has a number of interesting side effects: first, intersection types are required to express the bound of some capture variables; second, a capture variable may have *both* an upper and a lower bound; and third, a cap-

ture variable may appear in its own upper bound. Perhaps spurred by the requirement for intersections produced by capture, the language was also extended to allow intersection types as the bounds of declared type variables. In addition, the *join* operation (known as *lub* in the *JLS*) was defined to produce recursive types, an approach that was avoided in the Aarhus–Sun paper due to its complexity [12].

Torgersen, Ernst, and Hansen [13] complemented the specification with a formal discussion of wildcards as implemented in Java, and presented a core calculus extending Featherweight GJ [4] with wildcards. Their calculus, for the sake of generality, allows arbitrary combinations of upper and lower bounds on both declared type variables and wildcards. The paper, however, does not discuss how such generality might affect the full Java language, and type argument inference in particular; nor does it prove important properties of the calculus, such as type soundness or subtyping decidability.

In fact, Kennedy and Pierce [7] have demonstrated the *undecidability* of subtyping algorithms (and, by extension, subtype inference algorithms) for some object-oriented type systems that, like Java 5, contain contravariance. Their work is inconclusive on the question of whether Java 5 subtyping is decidable, but raises the possibility that it is not. A problem arises when recursive invocations of a subtyping algorithm are parameterized by increasingly larger types. Fortunately, Kennedy and Pierce’s work suggests a straightforward solution that can guarantee decidability in their simplified calculus: the class hierarchy must not exhibit a property termed *expansive inheritance*. Class declarations of this kind can be readily detected, and seem to serve no practical use, so it is reasonable to prohibit them. We follow this strategy here; while their decidability results are not proven to extend to the full Java language, it seems likely that they will.

Finally, this paper makes use of *union* types as a complement to intersections. These are explored in the context of object-oriented languages by Igarashi and Nagira [6]. While we do not argue here for first-class support for such types in the language—doing so would conflict with our goal of minimizing language changes—we do allow type analysis to produce them, and Igarashi and Nagira’s argument for full language support is worthy of consideration. Their work also suggests how the members—fields, methods, and nested classes—of union types might be determined, a topic which we do not explore here.²⁵

6. Conclusion

We have highlighted a number of bugs and limitations in the Java 5 type inference algorithm, and presented an improved version of the algorithm. The improved algorithm is sound,

²⁵Igarashi and Nagira interpret union types in a manner reminiscent of structural subtyping: the union contains a certain method if a method with that name is declared in each union element. If preferred, however, a more nominal approach could easily be developed.

and is able to produce correct results in a variety of cases in which the Java 5 algorithm falls short. It also minimizes the assumptions made by the algorithm, thus making possible extensions to the language like first-class intersection types and lower-bounded type variables.

The discussion of backwards compatibility in Section 4 addresses how changes to the language might be put into practice. Given the number of flaws in the current specification, an update to the inference algorithm seems inevitable, and that update will almost certainly violate backwards compatibility. Thus, addressing these bugs offers a good opportunity to make higher-level decisions about the inference algorithm, determining whether some non-essential improvements might also be made.

It would be useful to guide decisions about changes to the inference algorithm with an experimental study of their practical impact. An analysis tool might demonstrate, for example, that nearly all legacy Java programs would not be adversely affected by backwards compatibility problems under the improved algorithm proposed here.

The problems encountered when handling recursively-bounded type parameters provide another opportunity for future work. If a universal solution can be developed, it will be possible to create a locally-complete type argument inference algorithm.

The type system described in this paper has been implemented as a component of the DrJava IDE’s interactive interpreter [14]. This updated interpreter provides a useful demonstration of the improved inference algorithm.

A. Code Samples

A.1 javac Inference Failure with Wildcards

```
// This compiles in javac (1.5 & 1.6) without warn-
// ing but throws a ClassCastException at runtime.
<T> List<? super T> id(List<? super T> arg) {
    return arg;
}
void test() {
    List<Float> ln = new LinkedList<Float>();
    List<?> l = ln;
    List<? super String> ls = id(l);
    ls.add("hi");
    Float f = ln.get(0);
}
```

A.2 First-class Intersections

```
// For brevity, Comparable is abbreviated Cm
public class SafeTreeSet<T> {
    private TreeSet<T & Cm<? super T>> set;
    public SafeTreeSet() {
        set = new TreeSet<T & Cm<? super T>>();
    }
    public void add(T & Cm<? super T> elt) {
        set.add(elt);
    }
    public void addAll(Iterable<? extends T &
```

```

        Cm<? super T>> elts) {
    for (T & Cm<? super T> elt : elts) {
        set.add(elt);
    }
}
public Iterator<? extends T> iterator() {
    return colls.iterator();
}
}

```

A.3 Inference with Recursive Bounds

```

interface RecurBox<T extends RecurBox<T>> {
    T get();
    void set(T val);
}
interface Foo extends RecurBox<Foo> { }
// inherited: Foo get();
}
interface Bar extends Foo, Cloneable {
// inherited: Foo get();
}
<S extends RecurBox<S>, T extends S & Cloneable>
S unwrap(T arg) {
    return arg.get();
}

int typeToVal(Object o) { return 0; }
int typeToVal(Foo f) { return 1; }
int typeToVal(Bar b) { return 2; }
void test(Bar b) {
// For unwrap(b), S and T must satisfy:
// T <: S <: RecurBox<S>
// Bar <: T <: S & Cloneable
    assert typeToVal(unwrap(b)) == 1;
}

```

A.4 Existential Open with a Lower Bound

```

// Library code:
interface Processor<T> {
    /** Do some processing; true if successful. */
    boolean process(T arg);
}
interface ProcQueue<T>
    extends Queue<Processor<T>> { }

// Application code:
/** Pass each of vals to a Processor in the queue;
 * if processing a value is successful, increment
 * it. After all processing is complete, enqueue
 * the successful processors.
 */
static void runInts(ProcQueue<? super Integer> q,
    int[] vals) {
    return runIntsHelper(q, vals);
}

```

```

static <T super Integer>
void runIntsHelper(ProcQueue<T> q, int[] vs) {
    List<Processor<T>> keep =
        new LinkedList<Processor<T>>();
    for (int i = 0; i < vs.length; i++) {
        Processor<T> p = queue.remove();
        if (p.process(vs[i])) { keep.add(p); vs[i]++; }
    }
    queue.addAll(successful);
}

```

References

- [1] Gilad Bracha, Martin Odersky, David Stoutamire, & Philip Wadler. *Making the Future Safe for the Past: Adding Generativity to the Java Programming Language*. OOPSLA, 1998.
- [2] Luca Cardelli. *An Implementation of F_<*. Research report 97, DEC Systems Research Center, 1993.
- [3] James Gosling, Bill Joy, Guy Steele, & Gilad Bracha. *The Java Language Specification, Third Edition*. 2005.
- [4] Atshushi Igarashi, Benjamin Pierce, & Philip Wadler. *Featherweight Java: A Minimal Core Calculus for Java and GJ*. OOPSLA, 1999.
- [5] Atsushi Igarashi & Mirko Viroli. *On Variance-Based Subtyping for Parameteric Types*. ECOOP, 2002.
- [6] Atsushi Igarashi & Hideshi Nagira. *Union Types for Object-Oriented Programming*. Journal of Object Technology, vol. 6, no. 2, February 2007.
- [7] Andrew J. Kennedy & Benjamin C. Pierce. *On Decidability of Nominal Subtyping with Variance*. FOOL/WOOD, 2007.
- [8] Benjamin C. Pierce & David N. Turner. *Local Type Argument Synthesis with Bounded Quantification*. Technical report TR495, Indiana University, 1997.
- [9] Benjamin C. Pierce & David N. Turner. *Local Type Inference*. POPL, 1998.
- [10] Daniel Smith. *Completing the Java Type System*. Master's thesis, Rice University, 2007.
- [11] Kresten Krab Thorup & Mads Torgersen. *Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes*. Lecture Notes in Computer Science, 1999.
- [12] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, & Neal Gafter. *Adding Wildcards to the Java Programming Language*. SAC, 2004.
- [13] Mads Torgersen, Erik Ernst, & Christian Plesner Hansen. *Wild FJ*. FOOL, 2005.
- [14] DrJava IDE. <http://drjava.org>.
- [15] Java Community Process. <http://jcp.org>.
- [16] "Type variables should have lower/super bounds." Java Request for Enhancement. http://bugs.sun.com/view_bug.do?bug_id=5052956.
- [17] "Please introduce a name for the 'null' type." Java Request for Enhancement. http://bugs.sun.com/view_bug.do?bug_id=5060259.
- [18] "Multiply-bounded reference type expressions." Java Request for Enhancement. http://bugs.sun.com/view_bug.do?bug_id=6350706.