RICE UNIVERSITY

Completing the Java Type System

by

Daniel Smith

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Robert Cartwright, Chair
Professor of Computer Science

Walid Taha
Assistant Professor of Computer Science

Alan Cox
Associate Professor of CS and ECE

Houston, Texas

November, 2007

Abstract

Completing the Java Type System

by

Daniel Smith

The recent introduction of type variables and wildcards to the Java language, while greatly enriching the expressivity of the type system, comes with prohibitions against a variety of natural and useful expressions. Principal among these is the declaration of type variables with lower (“`super`”) bounds, naturally motivated by the support for such bounds on wildcards. We describe two variations on the Java type system that enhance the current language specification with support for these features. These variations must address the inference of method type arguments, and in so doing improve the inference algorithm so that it is both sound and complete. The first, simpler variation makes use of *union types*; the second more closely matches the current Java type system and formalizes its notion of “infinite types,” as produced by static analysis.

Acknowledgments

Rice is a great place for research. I appreciate the opportunities to broaden my thinking and experiment with my ideas that the Computer Science program has offered. Particularly, Robert "Corky" Cartwright has been a solid supporter and adviser, helping me to navigate uncharted territory and introducing me to powerful new ideas; Walid Taha has encouraged my growing interest in the logic and mathematics underlying Computer Science; and a number of peers, including Mathias Ricken, Moez Abdel-Gawad, James Sasitorn, Seth Nielson, Seth Fogarty, and Elspeth Simpson have provided interesting and rewarding discussions. In this particular work, I appreciate Corky, Walid, and Alan Cox's suggestions for improving the thesis's structure and focus, and guidance in the direction of future work. It is always gratifying to be surrounded by smart people.

Of course, I could never put off facing the real world (where people have real jobs, quiet neighborhoods, and even affordable family health care) like this without the kind support of my family. Tara has happily sacrificed quite a lot, and Madison makes smiling habitual.

Contents

Abstract ii

Acknowledgments iii

List of Tables vii

1 Introduction 1

2 Java 5 Type System 4

2.1 Key Features 4

2.1.1 Bounded Type Variables 4

2.1.2 Parameterized Class Types 5

2.1.3 Bounded Wildcards 8

2.1.4 Intersection Types 12

2.2 Historical Evolution and Related Work 13

3 Motivation 17

3.1 Lower Bounds on Type Variables 17

3.2 First-Class Intersection Types 20

3.3 Wildcards and Variables with Upper and Lower Bounds 22

3.4 The Null Type 23

3.5 Array Bounds on Type Variables 24

4 Union-based Type System 26

4.1 Fundamentals 26

4.1.1 Types 26

4.1.2 Bounds 29

4.1.3 Structural Well-formedness 29

4.1.4 Substitution 30

4.1.5 Wildcard Capture 31

4.1.6 Direct Supertype 31

4.1.7 Subtype Relation 32

4.1.8 Bounds Checking 35

4.1.9 Semantic Well-formedness 35

4.2 Type Argument Inference 38

4.2.1 Overview 38

4.2.2 Constraint Formulas 39

4.2.3 Subtype Inference 42

4.2.4 Supertype Inference 44

4.2.5 Inference Algorithm 44

4.2.6 Special Cases 47

5 Join-based Type System 50

5.1 Fundamentals 50

5.1.1 Types 50

5.1.2 Structural Well-formedness 51

5.1.3 Wildcard Capture 53

5.1.4 Subtyping 55

5.1.5 Join Function 58

5.2 Type Argument Inference 61

5.2.1 Constraint Formulas 61

5.2.2 Subtype Inference 61

5.2.3 Supertype Inference 62

6 Comparison of Type System Variations 64

6.1 Comparison of Type Operations 65

6.1.1 Fundamentals 65

6.1.2 Type Argument Inference 68

6.2 Backwards Compatibility 73

7 Conclusion 77

A Summary of Bugs in the JLS 79

A.1 Errors 79

A.2 Simple Missing Features 82

A.3 Confusing Content 84

B Summary of `javac` Bugs 88

B.1 Present in Versions 5.0 and 6.0 88

B.2 Fixed in Version 6.0 89

Bibliography 91

Tables

4.1 Rules for subtyping 33

4.2 Rules for subtype inference 43

4.3 Rules for supertype inference 45

5.1 $join(S,T)$ where S and T are not directly related 59

6.1 Major differences in type system variations 65

Chapter 1

Introduction

Version 3 of the Java Language Specification [2] introduces significant new concepts to the Java type system. These new features, present in Java 5.0 and later, include bounded type variables, parameterized class types, bounded wildcards as type arguments, and intersection types.* Each of these constructs provides opportunities for greater expressivity in program code. There are, however, natural ways in which these types might be used that are prohibited by the specification. In particular:

- While wildcards and type variables produced by wildcard capture may have upper (“`extends`”) and lower (“`super`”) bounds, declared type variables may only have upper bounds.
- An intersection type may only be used in program code as the upper bound of a declared type variable (and, implicitly, as the supertype of a declared class or interface). It may not be used as the type of a variable or parameter, the return type of a method, or even as a bound on a wildcard.
- A wildcard may not have *both* an upper and a lower bound.
- The null type (that is, the type containing only the value `null`) cannot be

* While a class or interface declaration’s list of superinterfaces in previous versions could be considered an intersection type, the concept was not made explicit.

expressed in any context, including as a type argument.

- A declared type variable may not be bounded by an array type.

Eliminating these restrictions is desirable from both a theoretical and a practical standpoint. Theoretically, doing so merely interprets the current features at their logical extent, providing a simpler yet more complete type system. Additionally, the type argument inference algorithm can be defined correctly, rather than as a heuristic analysis (as it is defined in Java 5[†]), only with improvements that allow for more expressive types. Practically, there are useful kinds of programs that cannot be accurately expressed under the current language rules.

This thesis begins with an overview in Chapter 2 of the relevant features of the Java 5 type system, followed by a review of the evolution of these features. Chapter 3 provides a number of small examples to demonstrate the practical need for removing the restrictions listed above. Subsequently, the key contribution of this work is presented: the formal definition of two type system variations. Both are powerful enough to handle the more general language that lacks the above restrictions. Chapter 4 describes the first variation, the *union-based* system, which is simpler, but relies on *union types*, a construct that is not part of Java 5. Chapter 5 covers the second, the *join-based* system. This variation more closely matches Java 5, but is more complex and restricted than the union-based alternative. For both variations,

[†] Here and subsequently, *Java 5* is used informally to refer to the language defined in version 3 of the Java Language Specification [2] and implemented in platform releases J2SE 5.0 and Java SE 6.

we describe an improved algorithm for local inference of method type arguments that is both sound and complete. Finally, in Chapter 6 we compare Java 5 and the two variations, and discuss how enhancing Java 5 would affect backwards compatibility.

Chapter 2

Java 5 Type System

The following briefly presents the relevant features of the Java 5 type system. Readers unfamiliar with parameterized types, wildcards, or intersection types in Java should find it particularly useful. This is followed by a brief summary of the historical evolution of these features and references to other related work.

2.1 Key Features

2.1.1 Bounded Type Variables

Type variables allow abstraction over types in Java classes and methods, much as term variables allow abstraction over values. Java has always had *subtype polymorphism*—the ability to describe the type of a variable in terms of some shared supertype of its anticipated instances. Type variables provide additional expressivity via *parametric polymorphism*, allowing a type to be described in terms of some unknown type. For example, a type variable can be used to precisely describe the return type of a method in terms of its parameters' types:

```
<T> T pick(T t1, T t2) {
  int x = new Random().nextInt();
  if (x % 2 == 0) return t1;
  else return t2;
}
```

The invocation `pick(12, 23)` has static type `Integer`, while `pick("hello", "mom")` has type `String`. Type analysis determines these typings by inferring an argument that can safely instantiate the parameter `T` in each case. Often, the choice of `T` is not so clear—we would expect `pick(12, 3.14)` to have type `Number`, for example, while the type for `pick(12, "hello")` may simply be `Object`.*

Type variables can be made more useful by declaring an *upper bound* for the variable. All instances of the parameter then must by subtypes of that bound. For example, the following variation on `pick` will only accept `Number`s as arguments:

```
<T extends Number> T pickN(T t1, T t2) {
  if (t1.doubleValue() < t2.doubleValue()) return t1;
  else return t2;
}
```

Within the scope of the type variable `T`, type analysis may assume that the unknown choice of `T` is a subtype of `Number`. Thus the expression `t1.doubleValue()` is valid, as it refers to the `doubleValue()` method of the `Number` class.

2.1.2 Parameterized Class Types

In addition to methods, class and interface declarations may be parameterized by type variables. In this case, the class or interface no longer corresponds to a single type; rather, it is a *type constructor* describing an infinite number of types.

* In fact, due to the more complex structure of the inheritance tree in the `java.lang` package, and given other features in the type system, more accurate type arguments can be inferred in each of these cases.

We'll use a mutable box class as a standard example:

```
interface Box<T> {
  T get();
  void set(T v);
}
```

Given a `Box<String>`, we can invoke `set()` with a `String` argument to mutate the box, and access the wrapped value—known by type analysis to have type `String`—by invoking `get()`.

Parameterized interfaces may be implemented by providing an instantiation of the interface type in the `implements` clause:

```
class StringBox implements Box<String> {
  private String val;
  public String get() { return val; }
  public void set(String v) { val = v; }
}
```

Not surprisingly, type arguments in the `implements` clause may be expressed in terms of *other* type variables:

```
class GenericBox<S> implements Box<S> {
  private S val;
  public S get() { return val; }
  public void set(S v) { val = v; }
}
```

Note, however, that there need not be a one-to-one correspondence between variables declared in a subclass and those of the superclass or superinterface:

```
class NumListBox<N extends Number> implements Box<List<N>> {
  public List<N> get() { ... }
  public void set(List<N> v) { ... }
}
```

Relationships between parameterized class types are determined by performing substitution on the declared supertypes. `StringBox` is a subtype of `Box<String>`; `GenericBox<Float>` is a subtype of `Box<Float>`; and `NumListBox<Float>` is a subtype of `Box<List<Float>>`. Note that, for the sake of type soundness, there is no relationship between different parameterizations of the same class: a `Box<Float>` is *not* a `Box<Number>`—while we can safely `get()` `Number`s from a `Box<Float>`, we cannot safely `set()` its contents to an `Integer`. For simplicity, this rule holds for all classes, even where such relationships would make sense. An `Iterator<Float>` cannot be treated as an `Iterator<Number>`, nor can a `Predicate<Number>` be used where a `Predicate<Float>` is required.

To ease the migration from legacy Java code, the language supports the use of a class name, without parameters, as a type. Thus the type `Box` is an approximation to how the class might be declared in the absence of type variables, and all parameterizations are considered subtypes of `Box`. These improper types, termed *raw types*, are best avoided in new code; we will generally ignore them here, except in later formal treatment of the type system.

In the presence of parameterized types, bounds on type variables can be much more complex. For example, the bound of a variable `S` may be described in terms of a variable `T`. These relationships may even be recursive:

```
<T extends Comparable<T>> void method1(T a, T b);

<T1 extends Convertible<T2>, T2 extends Convertible<T1>>
  void method2(T1 a, T2 b);
```

Such bounds are quite useful at times, as these examples suggest. However, they present significant challenges for type analysis.

2.1.3 Bounded Wildcards

Wildcards provide a means to describe a variety of instantiations of the same type constructor, much as interfaces provide a means to describe a variety of different concrete classes. The symbol `?` represents an unknown type argument, possibly constrained to fall beneath some upper bound. The `add()` method, below, is a simple example of wildcard usage:

```
int add(Box<? extends Number> b1, Box<? extends Number> b2) {
  return b1.get().intValue() + b2.get().intValue();
}
```

Each argument to `add` can have type `Box<Integer>`, `Box<Float>`, or `Box<Number>`, and the method is able to handle it appropriately. Of course, we could accomplish the same thing *without* wildcards by using type variables:

```
<T1 extends Number, T2 extends Number>
int add(Box<T1> b1, Box<T2> b2) {
  return b1.get().intValue() + b2.get().intValue();
}
```

This variation is a bit more verbose, and perhaps less natural to typical programmers. More importantly, in many circumstances, wildcards allow for fundamentally more expressive types. There is no way to eliminate a wildcard from a mutable field, for example:

```
class ClassBox implements Box<Class<?>> {
  private Class<?> c;
  public Class<?> get() { return c; }
  public void set(Class<?> cl) { c = cl; }
}
```

If we attempt to remove the wildcard by introducing a type variable, we get a very different class:

```
class ClassBoxT<T> implements Box<Class<T>> {
  private Class<T> c;
  public Class<T> get() { return c; }
  public void set(Class<T> cl) { c = cl; }
}
```

Over the course of its lifetime, an instance of `ClassBox` may hold a `Class<String>` at one point, and later a `Class<Number>`. A `ClassBoxT`, on the other hand, has `T` fixed at creation time. If we choose `T` to be `String`, the object can never wrap a `Class<Number>`.

We described previously the *invariant* subtyping discipline used to relate instantiations of the same type—`Box<`S`>` is a subtype of `Box<`T`>` if and only if $S = T$. Wildcards provide a means to relax that constraint where needed. Thus, while `Box<Float>` is not a subtype of `Box<Number>`, it *is* a subtype of `Box<? extends Number>`. This relationship is called *covariant* subtyping. Naturally, there is a tradeoff: while we can `get()` a `Number` from a `Box<? extends Number>`, we cannot use `set()` to change its contents to a different `Number`.

What if we want to `set()` the contents of a `Box` to a `Number`, but don't care about the `get()` method? In that case, we would like to use *contravariant* subtyping, treating a `Box<Object>`, say, as a `Box<Number>`. Wildcards permit such relationships via *lower bounds*, prefixed by the keyword `super`:

```
void copy(Box<? extends Number> b1, Box<? super Number> b2) {
  b2.set(b1.get());
}
```

Type analysis of programs involving wildcards is achieved via a *wildcard capture* operation, which converts a parameterized type with wildcard arguments into a type in which these arguments are replaced with fresh type variables. Thus, to determine the method signatures in type `Box<? extends Number>`, we convert that type to `Box<`Z_1`>`, where variable Z_1 is bound by `Number`. We can then determine by substitution that `get()` returns a Z_1 and, similarly, `set()` accepts a Z_1. Since Z_1 is a fresh name, we cannot produce a value of that type to pass to `set()` (with the exception of the value `null`); on the other hand, given a $\texttt{Z}_1$ returned by `get()`, we can determine that the value is an instance of `Number`.

Lower bounds are handled similarly. The methods of `Box<? super Number>` are those of type $\texttt{Box<Z}_2\texttt{>}$, where variable $\texttt{Z}_2$ is a *supertype* of `Number`. Note that this requires a fundamental adjustment to our notion of type variables: they may now have either an upper or a *lower* bound. In general, a type variable has both bounds; by default, the lower bound is `null` and the upper is `Object`. The analysis can thus show that the `set()` method of $\texttt{Box<Z}_2\texttt{>}$ can be safely called with a `Number`, as `Number` is a subtype of $\texttt{Z}_2$.

Capture is more powerful than these examples demonstrate: the fresh variable is actually bounded both by the bounds of the wildcard *and* those of the corresponding parameter. Thus, a `NumListBox<?>` may be treated as a $\texttt{NumListBox<Z}_3\texttt{>}$, where $\texttt{Z}_3$ is a subtype of `Number`. A `NumListBox<? super Integer>` may be treated as a $\texttt{NumListBox<Z}_4\texttt{>}$, where $\texttt{Z}_4$ falls somewhere between lower bound `Integer` and upper bound `Number`. Where the wildcard and the parameter have unrelated upper bounds—`NumListBox<? extends Cloneable>`, for example—we are faced with a challenge: the capture of this type is $\texttt{NumListBox<Z}_5\texttt{>}$, but what is the upper bound of $\texttt{Z}_5$? It must be both a subtype of `Cloneable` and a subtype of `Number`. Intersection types, described in the next section, provide the expressiveness needed to describe this bound.

Finally, wildcards allow type argument inference to be more precise. Recall the method `pick`, which accepts two arguments of type `T` and returns a value of that same type. Given declarations `Box<Integer> bi` and `Box<Runnable> br`, what is the type of `pick(bi, br)`? Previously, we would have said `Object`. With wildcards, though, we can use `Box<?>` instead. Further, given declaration `Box<Number> bn`, we can express the type of `pick(bi, bn)` with a bounded wildcard: `Box<? extends Number>`. Note that this process is recursive—the upper bound is found by recursively determining the common supertype of the two type arguments. This recursion is not guaranteed to terminate, so the language has special facilities to support "infinite" wildcards, or wildcards defined in terms of themselves. For example, `Integer` implements `Comparable<Integer>`, while `String` implements `Comparable<String>`. Thus, `pick(23, "hello")` has type `Comparable<? extends Comparable<? extends Comparable<...>>>`. The existence of such types significantly complicates subtyping and other type operations.

2.1.4 Intersection Types

The last major feature of the Java 5 type system is intersection types. The type `A & B` describes all values that are instances of both `A` and `B`. For example, `String` is a subtype of `CharSequence & Serializable`, because the class implements both of these interfaces. The same is true for `StringBuilder`.

Intersections are only used in limited contexts. Programmers may describe the upper bound of a type variable with an intersection: `<T extends CharSequence & Serializable>`. As mentioned previously, intersections are also useful in describing the bound of a capture variable: the capture if `NumListBox<? extends Cloneable>` is `NumListBox<`Z_5`>`, where Z_5 is a subtype of `Cloneable & Number`. Type analysis also uses intersections to provide additional precision in type argument inference. In the `java.util` package, classes `LinkedList<T>` and `ArrayList<T>` both extend `AbstractList<T>`. They also both independently implement the `Serializable` and `Cloneable` interfaces. Thus, given declarations `LinkedList<String> ll` and `ArrayList<String> al`, the most precise type of `pick(ll, al)` is `AbstractList<String> & Serializable & Cloneable`.

2.2 Historical Evolution and Related Work

Type variables and parameterized types in Java 5 were inherited from the GJ language [1], an extension to Java designed to support generic programming. The original specification for GJ describes familiar concepts and design choices that are present in Java 5: a type variable may be bounded by a supertype; parameterized types follow an invariant subtyping discipline; all parameterized types for a specific class have a common "raw" supertype; variable and parameterized types are erased at run time; and method type arguments may be locally inferred at the call site.

Wildcards arose out of research to extend GJ and similar languages with covariant and contravariant subtyping. Thorup and Torgersen [7] 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 [4] extended this notion to include contravariance and established a connection to bounded existential types. Their work requires support for lower bounds on type variables, as described above, though these bounds are not made expressible in type variable declarations. A joint project between the University of Aarhus and Sun Microsystems [8] 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 this paper.

The 3rd edition of the Java Language Specification (hereafter JLS) [2] 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 capture 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 specification) was defined to produce self-referential wildcards, an approach that had been avoided in the Aarhus–Sun paper due to its complexity [8].

Torgersen, Ernst, and Hansen [9] complemented the specification with a formal discussion of wildcards as implemented in Java, and presented a core calculus extending Featherweight GJ [3] 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 [6] have demonstrated the *undecidability* of subtyping 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 the decidability results are not proven to extend to the full Java language, it seems likely that they will.

Finally, this thesis makes use of *union* types as a complement to intersections. These are explored in the context of object-oriented languages by Igarashi and Nagira [5]. 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 the 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.

Chapter 3
Motivation

Before presenting the formalisms that revise the Java 5 type system, we argue for the practical merit in improving on the language's deficiencies below.

3.1 Lower Bounds on Type Variables

Initially, 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 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 only 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> result = new LinkedList<E>();
  for (int i = 1; i <= n; i++) { result.add(i); }
  return result;
}
```

The `sequence` method is parameterized by `E`, a list element type. Depending on the instantiation of `E`, the method can be used to create lists of `Integer`s, 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.

Replacing `E` with a wildcard (eliminating the type variable declaration and returning a `List<? super Integer>`) 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 `set` method accepts only `Integers`. Another alternative is to define `E` without a bound, but then it would not be possible to add `Integers` to the list within the body of `sequence`.

Consider another example: the following implementation of `Set`, again not legal in Java 5, contains any string that starts with `"a"`, and allows clients to add additional elements.

```
class ASet<E super String> implements Set<E> {
  private Set<E> elts;
  ...
  public boolean add(E o) {
    boolean result = contains(o);
    elts.add(o);
    return result;
  }
  public boolean contains(Object o) {
    boolean result = o instanceof String &&
                     ((String) o).startsWith("a");
    return result || elts.contains(o);
  }
}
```

We could eliminate the lower bound on `E` (and the declaration of `E` altogether) by defining the class with supertype `Set<String>`, but this workaround would not allow us to use the class in a context that requires adding to, for example, a `Set<Object>`.

As a third example, The Aarhus–Sun paper [8] 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.[*] 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)`. This strategy is useful for unbounded and upper-bounded wildcards, but 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.

The JLS [2] indirectly provides some insight into the language designers' motivation for restricting type parameter declarations in this way. 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" (4.5.1). We will demonstrate that this reservation is unnecessary—type inference can be successfully defined in the presence of methods declaring type variables with lower bounds.

[*] 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.

There is a submission in Sun's Java bug database requesting this feature, with some accompanying discussion [11].

3.2 First-Class Intersection Types

Intersection types allow the behavior of two unrelated classes or interfaces to be described with a single type. In Java 5, an intersection is only expressible as the upper bound of a type variable. However, intersections are potentially quite useful wherever arbitrary types are allowed.

As a simple 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 with type `Flushable & Closeable`.

The following class, illegal in Java 5, demonstrates a variety of uses for intersection types. It defines a simple wrapper for `TreeSet` that insures its elements are comparable to each other (thus preventing any risk of a `ClassCastException` thrown by `TreeSet.add`, which occurs whenever incompatible objects are compared). The intersection type `T & Comparable<? super T>` is used as a type argument, a parameter type, a wildcard upper bound, and a local variable type.

```
public class SafeTreeSet<T> {
  private TreeSet<T & Comparable<? super T>> set;
  public SafeTreeSet() {
    set = new TreeSet<T & Comparable<? super T>>();
  }
  public void add(T & Comparable<? super T> elt) {
    set.add(elt);
  }
  public void addAll(Iterable<? extends T &
                             Comparable<? super T>> elts) {
    for (T & Comparable<? super T> elt : elts) { set.add(elt); }
  }
  public Iterator<? extends T> iterator() {
    return set.iterator();
  }
}
```

In Java 5, we could approximate this definition by defining `T` with an upper bound `Comparable<? super T>`. However, there is an important difference between the two approaches. In the latter case, `T` must be a subtype of `Comparable`. In the former case, that is not necessary—`T` could be `List<String>`, or even `Object`. In such cases, the set could contain values of *any* subtype of `T` that happened to implement `Comparable<? super T>`. For example, vendor A could define `IntListA` as a list of integers, comparable to all other integer lists; vendor B could define `IntListB` similarly. Both kinds of lists could safely coexist in a `SafeTreeSet<List<Integer>>`.

Alternatively, we could introduce a type variable `TC extends T & Comparable<? super T>` in each method declaration. This approach allows us to maintain the same semantics, but 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.

It is curious that intersections are supported in Java 5 as the upper bounds of type variables but not elsewhere, because it is not obvious that intersections are more useful in that particular context than any other. The lack of support for intersections as the bounds of wildcards is particularly surprising. (Chapter 6 explains how limitations of the type argument inference algorithm may have led to this restriction.)

Again, there is a submission in Sun's Java bug database requesting this feature, with some accompanying discussion [13].

3.3 Wildcards and Variables with Upper and Lower Bounds

As noted in Chapter 2, variables produced by capture, because they combine the bounds of a wildcard with the bounds of a corresponding type parameter, may have both nontrivial upper and lower bounds. Java implementations must thus allow for such variables, and it would be reasonable to make this functionality available to programmers, and extend it to wildcards.

The following method, illegal in Java 5, is a simple example demonstrating how such bounds might be useful. `appendSum` sums the values in a list and appends that value to the end of the list. It may operate on both lists of `Double`s and lists of `Number`s; an implementation without the two bounds would only be able to handle one or the other.

```
public void appendSum(List<? extends Number super Double> vals) {
  double result = 0.0;
  for (Number n : vals) { result += n.doubleValue(); }
  vals.add(result);
}
```

In addition to being occasionally convenient for programmers, we will later demonstrate that wildcards with both upper and lower bounds allow us to improve on the *join* function (*lub* in the JLS) by providing a tightly-bound wildcard encompassing two different parameterizations of the same class. It is noted in the Aarhus–Sun paper that the "best" bound of a wildcard produced by *join* may be ambiguous—either an upper or a lower bound could be useful, depending on how the type will be used [8]. By allowing wildcards with *both* bounds, we remove that ambiguity.

3.4 The Null Type

Since its inception, the Java language has had a `null` value without a corresponding expressible null type (the type is written here as `null`, and is distinguishable from the *value* `null` by context). In the absence of type variables and parameterized classes, the null type arguably has little utility in program code. However, in combination with the Java 5 features, it is quite useful.

For example, a singleton "empty" value can often only be expressed with a null type. The following class, illegal in Java 5, defines a singleton representing an immutable empty list, implementing the `List<null>` interface:

```
public class EmptyList extends AbstractList<null> {
  public static final EmptyList INSTANCE = new EmptyList();
  private EmptyList() {}
  public null get(int i) {
    throw new IndexOutOfBoundsException();
  }
  public int size() { return 0; }
}
```

Clients can use wildcards to reference this list in whatever context is needed:

```
List<? extends Number> l1 = EmptyList.INSTANCE;
List<? extends String> l2 = EmptyList.INSTANCE;
```

Without using the type `null`, there is no way to define a similar singleton that can be safely typed wherever it is needed. Instead, developers must either perform an unchecked cast or create a new object for each instantiation of the element type.

There is a submission in Sun's Java bug database requesting this feature, with some accompanying discussion [12].

3.5 Array Bounds on Type Variables

The Java 5 design choice prohibiting array types as the upper bounds of type variables, yet permitting them as the upper bounds of wildcards, seems arbitrary. While allowing the bounds would arguably make no difference to the current language—the declaration `T extends Foo[]` can probably always be rewritten `T extends Foo`, with all references to `T` replaced by `T[]`—the enhancements suggested here would be incomplete without allowing array types in variable bounds. In particular, an array type in a lower bound cannot always be eliminated: while any useful subtype of `Foo[]` *must* be an array type (or a variable or intersection defined in terms of an array type), the supertype of `Foo[]` could be, for example, `Serializable`.

Chapter 4
Union-based Type System

We now present the first of two revisions to the Java 5 type system. This *union-based* system is characterized by its use of *union types* in wildcard capture and type argument inference. Rather than simply describing changes to the operations defined in the JLS [2], we redefine them fully in clear, formal terms. We thus do not presuppose significant familiarity with the JLS, and are able to avoid a variety of bugs and ambiguities that appear there (these are outlined in Appendix A).

4.1 Fundamentals

4.1.1 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`<`$T_1 \ldots T_n$`>`, where C is a name and, for all i, T_i is a type.
- A *wildcard-parameterized class type* C`<`$W_1 \ldots W_n$`>`, where C is a name and, for all i, W_i is a *type argument*, which is one of:
  - A type.
  - A *wildcard* `? extends` T_u `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 `&` ... `&` T_n, where, for all i, T_i is a type.
- A *union type* T_1 `|` ... `|` T_n, where, for all i, 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.*

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`<>` (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.† For example, if class `Foo` declares inner class `Bar`, the expression

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

* This 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.

† This 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`.

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 thesis. Igarashi and Nagira [5] 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.

4.1.2 Bounds

Type variables are always bounded—a valid instantiation of a variable must be a subtype of its *upper bound*, and a 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 $\lceil X \rceil$ is shorthand for the application of *upper* to X, producing X's upper bound; similarly, $\lfloor X \rfloor$ produces X's lower bound.

4.1.3 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 $T = C\texttt{<}T_1 \ldots T_n\texttt{>}$, the class named C exists and has n type parameters.
- Where $T = C\texttt{<}W_1 \ldots W_n\texttt{>}$, 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 chapter. 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.

4.1.4 Substitution

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

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

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-paramterized 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.

4.1.5 Wildcard Capture

Wildcard capture is an operation on type arguments (either types or wildcards, $W_1 \ldots W_n$) and their corresponding type parameters ($P_1 \ldots P_n$), 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.

$\mathit{capture}(W_1 \ldots W_n, P_1 \ldots P_n) = T_1 \ldots T_n$, 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:
  - $\lceil Z_i \rceil = W_{iu}$ `&` $\lceil P_i \rceil [P_1 := T_1 \; \ldots \; P_n := T_n]$.
  - $\lfloor Z_i \rfloor = W_{il}$ `|` $\lfloor P_i \rfloor [P_1 := T_1 \; \ldots \; P_n := T_n]$.

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

4.1.6 Direct Supertype

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

- Where $T = C\texttt{<}T_1 \ldots T_n\texttt{>}$,
  - If $C = \texttt{Object}$, $T\!\uparrow$ is undefined.
  - If C declares no supertypes, $T\!\uparrow = \texttt{Object}$.
  - If C declares supertypes $S_1 \ldots S_m$ and type parameters $P_1 \ldots P_n$, let $S_i' = S_i[P_1 := T_1 \ \ldots \ P_n := T_n]$; $T\!\uparrow = S_1' \ \texttt{\&} \ \ldots \ \texttt{\&} \ S_m'$.
- Where $T = C\texttt{<}W_1 \ldots W_n\texttt{>}$, $T\!\uparrow = \|C\texttt{<}W_1 \ldots W_n\texttt{>}\|\!\uparrow$.
- Where $T = C$,
  - If C declares no supertypes, $T\!\uparrow = \texttt{Object}$.
  - If C declares supertypes $S_1 \ldots S_m$, $T\!\uparrow = |S_1| \ \texttt{\&} \ \ldots \ \texttt{\&} \ |S_m|$.

$|S_i|$, 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.

4.1.7 Subtype Relation

The type S is a *subtype* of T, denoted $S <: T$, if and only if the assertion in the corresponding cell of Table 4.1 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 $false$.) If $S <: T$, then equivalently T is a *supertype* of S (denoted $T :> S$). Where two types are mutual subtypes of each other—that is, $S <: T$ and $T <: S$—we say that they are equivalent, denoted $S \cong T$.

	`null`	$C_t\texttt{<}T_1\ldots T_m\texttt{>}$	$C_t\texttt{<}W_1\ldots W_m\texttt{>}$	C_t
`null`	*true*	*true*	*true*	*true*
$C_s\texttt{<}S_1\ldots S_n\texttt{>}$	-	[1]	[2]	[3]
$C_s\texttt{<}W_1\ldots W_n\texttt{>}$	-	$\Vert S\Vert <: T$	$\Vert S\Vert <: T$	$\Vert S\Vert <: T$
C_s	-	$S\!\uparrow <: T$	$S\!\uparrow <: T$	[3]
$p_s\texttt{[]}$	-	[4]	[4]	[4]
$S'\texttt{[]}$	-	[4]	[4]	[4]
X_s	$\lceil S\rceil <: T$	$\lceil S\rceil <: T$	$\lceil S\rceil <: T$	$\lceil S\rceil <: T$
$S_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ S_n$	$\exists i, S_i <: T$	$\exists i, S_i <: T$	$\exists i, S_i <: T$	$\exists i, S_i <: T$
$S_1\ \vert\ \ldots\ \vert\ S_n$	$\forall i, S_i <: T$	$\forall i, S_i <: T$	$\forall i, S_i <: T$	$\forall i, S_i <: T$

	$p_t\texttt{[]}$	$T'\texttt{[]}$	X_t	$T_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T_m$	$T_1\ \vert\ \ldots\ \vert\ T_m$
`null`	*true*	*true*	*true*	*true*	*true*
$C_s\texttt{<}S_1\ldots S_n\texttt{>}$	-	-	$S <: \lfloor T\rfloor$	$\forall i, S <: T_i$	$\exists i, S <: T_i$
$C_s\texttt{<}W_1\ldots W_n\texttt{>}$	-	-	$S <: \lfloor T\rfloor$	$\forall i, S <: T_i$	$\exists i, S <: T_i$
C_s	-	-	$S <: \lfloor T\rfloor$	$\forall i, S <: T_i$	$\exists i, S <: T_i$
$p_s\texttt{[]}$	$p_s = p_t$	-	$S <: \lfloor T\rfloor$	$\forall i, S <: T_i$	$\exists i, S <: T_i$
$S'\texttt{[]}$	-	$S' <: T'$	$S <: \lfloor T\rfloor$	$\forall i, S <: T_i$	$\exists i, S <: T_i$
X_s	$\lceil S\rceil <: T$	$\lceil S\rceil <: T$	[5]	$\forall i, S <: T_i$	[6]
$S_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ S_n$	$\exists i, S_i <: T$	$\exists i, S_i <: T$	[7]	$\forall i, S <: T_i$	$\exists i, S_i <: T$
$S_1\ \vert\ \ldots\ \vert\ S_n$	$\forall i, S_i <: T$	$\forall i, S_i <: T$	$\forall i, S_i <: T$	$\forall i, S_i <: T$	$\forall i, S_i <: T$

[1]: If $C_s = C_t$, $\forall i, S_i \cong T_i$; otherwise, $S\!\uparrow <: T$
[2]: If $C_s = C_t$, for all i:

- If W_i is a type, $S_i \cong W_i$
- If W_i is a wildcard `? extends` T_{iu} `super` T_{il}, $S_i <: T_{iu} \wedge T_{il} <: S_i$

Otherwise, $S\!\uparrow <: T$
[3]: Either $C_s = C_t$ or $S\!\uparrow <: T$
[4]: `Cloneable & Serializable` $<: T$
[5]: $X_s = X_t$ or $\lceil S\rceil <: T$or $S <: \lfloor T\rfloor$
[6]: $\lceil S\rceil <: T$or $\exists i, S <: T_i$
[7]: $\exists i, S_i <: T$or $S <: \lfloor T\rfloor$

Table 4.1 : Rules for subtyping

In addition to the assumption of structurally well-formed arguments, we require the following of the subtyping arguments:[‡]

- No variable is bounded by itself—that is, $\lceil X \rceil^{+} \neq X$ and $\lfloor X \rfloor^{+} \neq X$.
- The class table is acyclic: $C\texttt{<}T_1 \ldots T_n\texttt{>}{\uparrow}^{+} \neq C\texttt{<}T'_1 \ldots T'_n\texttt{>}$ for any choice of $T'_1 \ldots T'_n$.
- The class table does not exhibit *expansive inheritance*, as defined by Kennedy and Pierce [6].

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

These subtyping rules are syntax-directed; defining an algorithm in terms of this definition is a straightforward process. As demonstrated by Kennedy and Pierce [6], however, such an algorithm must be extended to keep track of "in-process" invocations and terminate whenever a subtyping invocation depends on itself.

The interactions between variables, intersections, and unions are tricky. Where $S = X_s$ and $T = T_1 \texttt{ | } \ldots \texttt{ | } T_m$, for example, it is possible that $T_i = S$ for some i, so we must decompose T and not S; on the other hand, it is possible that $\lceil S \rceil$ is T, so we must also recur on S's upper bound *without* decomposing T. In general, it is always safe to check for both conditions as in cases [6] and [7], while in some cases,

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

one of the checks is provably unnecessary.

4.1.8 Bounds Checking

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

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

- For all i, $T_i <: \lceil P_i \rceil [P_1 := T_1 \ \ldots \ P_n := T_n]$.
- For all i, $\lfloor P_i \rfloor [P_1 := T_1 \ \ldots \ P_n := T_n] <: T_i$.

Bounds checking is complicated by the fact that the bound for a specific argument may depend on that or other arguments. So substitution must be used to instantiate each bound before it is checked.

4.1.9 Semantic Well-formedness

We do not provide in this thesis any formal proof of the correctness of the above operations. However, the consistent use of case analysis provides a framework for such a proof. In addition, we describe here a notion of *semantic* well-formedness and assert, for each operation, a variety of properties that hold when processing semantically well-formed types. These assertions would be essential to a larger proof of type safety, and they provide a standard by which to informally verify correctness.

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 $T = C\texttt{<}T_1 \dots T_n\texttt{>}$, and class C has parameters $P_1 \dots P_n$, $inBounds(T_1 \dots T_n, P_1 \dots P_n)$.
- Where $T = C\texttt{<}W_1 \dots W_n\texttt{>}$, $\|C\texttt{<}W_1 \dots W_n\texttt{>}\|$ 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 the 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.[§]

Substitution. If the following are true, we can guarantee that the result of the substitution $T[P_1 := T_1 \ \dots \ P_n := T_n]$ is well-formed.

- T, $P_1 \dots P_n$, and $T_1 \dots T_n$ are well-formed.
- $inBounds(T_1 \dots T_n, P_1 \dots P_n)$ holds.

[§] 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.

Under these conditions, substitution has the following properties, which must be true in order for well-formedness to be preserved:

- $T_1 <: T_2 \Rightarrow T_1[P_1 := T_1 \ \ldots \ P_n := T_n] <: T_2[P_1 := T_1 \ \ldots \ P_n := T_n]$

- $\mathit{inBounds}(S_1 \ldots S_m, Q_1 \ldots Q_m) \Rightarrow \mathit{inBounds}((S_1 \ldots S_m)[P_1 := T_1 \ \ldots \ P_n := T_n],$ $Q_1 \ldots Q_m)$ (assuming $Q_1 \ldots Q_m$ do not involve $P_1 \ldots P_n$)

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. Where all the arguments are wildcards, however, we can make the following claim: if the types $\mathit{capture}(W_1 \ldots W_n, P_1 \ldots P_n)$ are well-formed,

$$\mathit{inBounds}(\mathit{capture}(W_1 \ldots W_n, P_1 \ldots P_n), P_1 \ldots P_n)$$

Subtyping. Where the inputs are well-formed, the subtype relation is reflexive and transitive. It also guarantees, for all well-formed T, T', etc., that:

- $\texttt{null} <: T <: \texttt{Object}$
- $\lceil X \rceil = T \Rightarrow X <: T$
- $\lfloor X \rfloor = T \Rightarrow T <: X$
- For all i, $T_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ T_n <: T_i$
- For all i, $T_i <: T_1 \ \texttt{|} \ \ldots \ \texttt{|} \ T_n$
- Where $T'_1 \ldots T'_m \subseteq T_1 \ldots T_n$ (allowing for arbitrary permutations),
  - $T_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ T_n <: T'_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ T'_m$
  - $T'_1 \ \texttt{|} \ \ldots \ \texttt{|} \ T'_m <: T_1 \ \texttt{|} \ \ldots \ \texttt{|} \ T_n$

- For all X such that $inBounds(T_1, X)$ and $inBounds(T_2, X)$, $T_1 \cong T_2 \Rightarrow T[X := T_1] \cong T[X := T_2]$

The final assertion of equivalence preservation implies that implementations are free to replace any type with a simplified, but equivalent, form whenever it is convenient. For example, the intersection `String & Object` could be safely replaced with the simple class type `String`.¶ This fact is particularly useful when creating intersections and unions, which can easily be simplified when one of the component types is a subtype of another. However, the intersections and unions created by *capture* may only be simplified with care: the *upper* and *lower* functions involving the fresh variable Z_i cannot be defined until *after* the bounds of Z_i have been determined; yet Z_i may appear in its own bound. So any algorithm that attempts, for example, to simplify the intersection produced by *capture* must not attempt to access Z_i's bounds before they have been defined.

4.2 Type Argument Inference

4.2.1 Overview

A type argument inference algorithm provides an instantiation of a set of method type parameters (we use $T_1 \ldots T_n$ to denote the arguments and $P_1 \ldots P_n$ to denote the parameters) for a specific call site. In Java 5, these types are inferred based solely on type information available locally at the method call. Thus, the result

¶ Such a transformation may change the result of erasure: $T \cong T' \not\Rightarrow |T| \cong |T'|$, but erasure is only applied to types expressed in code, so it is not relevant to this discussion.

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:

- $\forall j, A_j <: F_j[P_1 := T_1 \ \dots \ P_n := T_n]$.
- $R[P_1 := T_1 \ \dots \ P_n := T_n] <: E$.
- $\mathit{inBounds}(T_1 \dots T_n, P_1 \dots P_n)$.

Here we present such an algorithm. We proceed by first producing a set of constraints required to satisfy the first two conditions, and then using *capture* to produce types that both meet these constraints and fall within bounds specified by $P_1 \dots P_n$. The algorithm is both sound and complete: sound in the sense that the results, $T_1 \dots T_n$, satisfy the three above equations; and complete in the sense that a result is produced whenever a solution exists.

The algorithm is expressed in terms of two functions, $<:_?$ and $:>_?$. $A <:_? F$ produces a minimal set of constraints on $T_1 \dots T_n$ required to satisfy $A <: F[P_1 := T_1 \ \dots \ P_n := T_n]$; $A :>_? F$ similarly produces the constraints satisfying $A :> F[P_1 := T_1 \ \dots \ P_n := T_n]$. The constraints are expressed as logical formulas, combined with the operations $\wedge_{cf}$ and $\vee_{cf}$ as outlined below. For convenience, a third inference function, $\cong_?$, is a shortcut for $\wedge_{cf}(A <:_? F, A :>_? F)$.

4.2.2 Constraint Formulas

A *constraint formula* is a formula in first-order logic expressing upper and lower bounds on our choices for types $T_1 \dots T_n$. We restrict the form of a constraint formula as follows, modeled after disjunctive normal form:

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

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 `null` $<: T_1 <:$ `Object` $\wedge \ldots \wedge$ `null` $<: T_n <:$ `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`.

Constraint formulas may be combined by *and*-ing or *or*-ing them together. $\wedge_{cf}$ is used where multiple formulas must *all* be satisfied; $\vee_{cf}$ is used where just one of a number of formulas must be satisfied.

Let $SC_1 \ldots SC_m$ be simple constraint formulas. Let T_{ijl} refer to the lower bound of T_i in SC_j, and T_{iju} refer to the corresponding upper bound. Then $\wedge_{cf}(SC_1 \ldots SC_m)$ has value

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

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 SC_j is *true* (and $m > 1$), that formula may be discarded.

In the general case, we define $\wedge_{cf}$ by merging each possible combination of simple constraint formulas. Let $C_1 \ldots C_m$ be constraint formulas. Each of of these formulas can be treated as a set of *simple* formulas; the cross product of these sets, $C_1 \times \ldots \times C_m$, produces m' tuples of the form $(SC_1 \ldots SC_m)$. Applying $\wedge_{cf}$ to each of these tuples (as defined above for simple formulas), we produce the set of simple formulas $SC'_1 \ldots SC'_{m'}$. Then we have

$$\wedge_{cf}(C_1 \ldots C_m) = \vee_{cf}(SC'_1 \ldots SC'_{m'})$$

Again, we note that if any of C_j is $true$, that set may be discarded; if any of C_j is $false$, the result will also be $false$ (because $m' = 0$).

The $\vee_{cf}$ operation could be defined to simply concatenate its arguments together. However, we wish to ensure that all formulas we produce are in a minimal form. To do so, we first need a notion of constraint-formula implication. A simple constraint formula SC_1 *models* SC_2 (denoted $SC_1 \models SC_2$) if and only if, for all i, $T_{i1u} <: T_{i2u}$ and $T_{i1l} :> T_{i2l}$. In other words, if $T_1 \ldots T_n$ satisfies SC_1, we can guarantee that $T_1 \ldots T_n$ satisfies SC_2.

Now, we define $\vee_{cf}(C_1 \ldots C_m)$ as follows. Again treating these formulas as sets of simple formulas, let $SC_1 \ldots SC_k$ be the union $C_1 \cup \ldots \cup C_m$. We can compute a minimal equivalent subset of $SC_1 \ldots SC_k$, $SC'_1 \ldots SC'_{k'}$, where *minimal* means that $\forall i, \exists j, SC_i \models SC'_j$. Now we have

$$\vee_{cf}(C_1 \ldots C_m) = \bigvee_{i=1}^{k'} SC'_i$$

Again note that we've preserved important logical properties: if any of $C_1 \ldots C_m$ is $false$, it may be ignored; if any of $C_1 \ldots C_m$ is $true$, it will be the *only* member of the minimal subset, and the result will be $true$.

4.2.3 Subtype Inference

The invocation $A <:_? F|_\mu$ produces a constraint formula supporting the assumption that $A <: F$. The parameter μ is a set of previous invocations of $<:_?$ and $:>_?$. 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 $<:_?$, $:>_?$, and $\cong_?$), the previous invocation is accumulated in μ.

- If the invocation $A <:_? F \in \mu$, the result is $false$.
- Else if, for some i, $F = P_i$, the result is $A <: T_i <: \texttt{Object}$.
- Else if F involves none of $P_1 \ldots P_n$, the result is $A <: F$ (treating the boolean result of $<:$ as a trivial constraint formula).
- Otherwise, the result is given in Table 4.2. (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 4.2 to Table 4.1. Notice that the rules for inference follow directly from subtyping. The only changes replace boolean operations with their analogs: $<:$ becomes $<:_?$; "and" and "or" become $\wedge_{cf}$ and $\vee_{cf}$.

	$C_f\texttt{<}F_1 \ldots F_m\texttt{>}$	$C_f\texttt{<}W_1 \ldots W_m\texttt{>}$	$F'\texttt{[]}$
`null`	*true*	*true*	*true*
$C_a\texttt{<}A_1 \ldots A_n\texttt{>}$	[1]	[2]	-
$C_a\texttt{<}W_1 \ldots W_n\texttt{>}$	$\lVert A \rVert <:_? F$	$\lVert A \rVert <:_? F$	-
C_a	$A\!\uparrow\, <:_? F$	$A\!\uparrow\, <:_? F$	-
$p_s\texttt{[]}$	[4]	[4]	-
$A'\texttt{[]}$	[4]	[4]	$A' <:_? F'$
X_a	$\lceil A \rceil <:_? F$	$\lceil A \rceil <:_? F$	$\lceil A \rceil <:_? F$
$A_1 \texttt{ \& } \ldots \texttt{ \& } A_n$	$\vee_{cf}(A_i <:_? F)$	$\vee_{cf}(A_i <:_? F)$	$\vee_{cf}(A_i <:_? F)$
$A_1 \texttt{ } \vert \texttt{ } \ldots \texttt{ } \vert \texttt{ } A_n$	$\wedge_{cf}(A_i <:_? F)$	$\wedge_{cf}(A_i <:_? F)$	$\wedge_{cf}(A_i <:_? F)$

	X_f	$F_1 \texttt{ \& } \ldots \texttt{ \& } F_m$	$F_1 \texttt{ } \vert \texttt{ } \ldots \texttt{ } \vert \texttt{ } F_m$
`null`	*true*	*true*	*true*
$C_a\texttt{<}A_1 \ldots A_n\texttt{>}$	$A <:_? \lfloor F \rfloor$	$\wedge_{cf}(A <:_? F_i)$	$\vee_{cf}(A <:_? F_i)$
$C_a\texttt{<}W_1 \ldots W_n\texttt{>}$	$A <:_? \lfloor F \rfloor$	$\wedge_{cf}(A <:_? F_i)$	$\vee_{cf}(A <:_? F_i)$
C_a	$A <:_? \lfloor F \rfloor$	$\wedge_{cf}(A <:_? F_i)$	$\vee_{cf}(A <:_? F_i)$
$p_s\texttt{[]}$	$A <:_? \lfloor F \rfloor$	$\wedge_{cf}(A <:_? F_i)$	$\vee_{cf}(A <:_? F_i)$
$A'\texttt{[]}$	$A <:_? \lfloor F \rfloor$	$\wedge_{cf}(A <:_? F_i)$	$\vee_{cf}(A <:_? F_i)$
X_a	[5]	$\wedge_{cf}(A <:_? F_i)$	[6]
$A_1 \texttt{ \& } \ldots \texttt{ \& } A_n$	[7]	$\wedge_{cf}(A <:_? F_i)$	$\vee_{cf}(A <:_? F_i)$
$A_1 \texttt{ } \vert \texttt{ } \ldots \texttt{ } \vert \texttt{ } A_n$	$\wedge_{cf}(A_i <:_? F)$	$\wedge_{cf}(A_i <:_? F)$	$\wedge_{cf}(A_i <:_? F)$

[1]: There are two cases:

- If $C_a = C_f$, $\wedge_{cf}(A_1 \cong_? F_1 \ldots A_n \cong_? F_n)$
- Otherwise, $A\!\uparrow\, <:_? F$

[2]: There are two cases:

- If $C_a = C_f$, $\wedge_{cf}(CF_1 \ldots CF_n)$ where, for all i:
  - If W_i is a type, $CF_i = A_i \cong_? W_i$
  - If W_i is a wildcard `? extends` F_{iu} `super` F_{il}, $CF_i = \wedge_{cf}(A_i <:_? F_{iu}, A_i :>_? F_{il})$
- Otherwise, $A\!\uparrow\, <:_? F$

[4]: `Cloneable & Serializable` $<:_? F$

[5]: $\vee_{cf}(\lceil A \rceil <:_? F, A <:_? \lfloor F \rfloor)$

[6]: $\vee_{cf}(\lceil A \rceil <:_? F, A <:_? F_1 \ldots A <:_? F_m)$

[7]: $\vee_{cf}(A_1 <:_? F \ldots A_n <:_? F, A <:_? \lfloor F \rfloor)$

Table 4.2 : Rules for subtype inference

4.2.4 Supertype Inference

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

- If the invocation $A :>_? F \in \mu$, the result is $false$.
- If, for some i, $F = P_i$, the result is `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 4.3. (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 4.2, Table 4.3 follows directly from the subtyping rules (this is slightly less apparent, because the table has been transposed, allowing the "upper" argument to appear on the left).

4.2.5 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 types $T_1 \dots T_n$—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

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

	C_f`<`$F_1 \ldots F_m$`>`	C_f`<`$W_1 \ldots W_m$`>`	F'`[]`
`null`	-	-	-
C_a`<`$A_1 \ldots A_n$`>`	[1]	$A :>_? \|F\|$	[4]
C_a`<`$W_1 \ldots W_n$`>`	[2]	$A :>_? \|F\|$	[4]
C_a	[3]	$A :>_? \|F\|$	[4]
p_s`[]`	-	-	-
A'`[]`	-	-	$A' :>_? F'$
X_a	$\lfloor A \rfloor :>_? F$	$\lfloor A \rfloor :>_? F$	$\lfloor A \rfloor :>_? F$
A_1 `&` ... `&` A_n	$\wedge_{cf}(A_i :>_? F)$	$\wedge_{cf}(A_i :>_? F)$	$\wedge_{cf}(A_i :>_? F)$
A_1 `|` ... `|` A_n	$\vee_{cf}(A_i :>_? F)$	$\vee_{cf}(A_i :>_? F)$	$\vee_{cf}(A_i :>_? F)$

	X_f	F_1 `&` ... `&` F_m	F_1 `|` ... `|` F_m
`null`	$A :>_? \lceil F \rceil$	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$
C_a`<`$A_1 \ldots A_n$`>`	$A :>_? \lceil F \rceil$	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$
C_a`<`$W_1 \ldots W_n$`>`	$A :>_? \lceil F \rceil$	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$
C_a	$A :>_? \lceil F \rceil$	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$
p_s`[]`	$A :>_? \lceil F \rceil$	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$
A'`[]`	$A :>_? \lceil F \rceil$	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$
X_a	[5]	[7]	$\wedge_{cf}(A :>_? F_i)$
A_1 `&` ... `&` A_n	$\wedge_{cf}(A_i :>_? F)$	$\wedge_{cf}(A_i :>_? F)$	$\wedge_{cf}(A :>_? F_i)$
A_1 `|` ... `|` A_n	[6]	$\vee_{cf}(A :>_? F_i)$	$\wedge_{cf}(A :>_? F_i)$

[1]: There are two cases:

- If $C_a = C_f$, $\wedge_{cf}(A_1 \cong_? F_1 \ldots A_n \cong_? F_n)$
- Otherwise, $A :>_? F\uparrow$

[2]: There are two cases:

- If $C_a = C_f$, $\wedge_{cf}(CF_1 \ldots CF_n)$ where, for all i:
  - If W_i is a type, $CF_i = W_i \cong_? F_i$
  - If W_i is a wildcard `? extends` A_{iu} `super` A_{il}, $CF_i = \wedge_{cf}(A_{iu} :>_? F_i, A_{il} <:_? F_i)$
- Otherwise, $A :>_? F\uparrow$

[3]: If $C_a = C_f$, *true*; otherwise, $A :>_? F\uparrow$

[4]: $A :>_?$ `Cloneable & Serializable`

[5]: $\vee_{cf}(A :>_? \lceil F \rceil, \lfloor A \rfloor :>_? F)$

[6]: $\vee_{cf}(A :>_? \lceil F \rceil, A_1 :>_? F \ldots A_n :>_? F)$

[7]: $\vee_{cf}(A :>_? F_1 \ldots A :>_? F_m, \lfloor A \rfloor :>_? F)$

Table 4.3 : Rules for supertype inference

Given the formula CF, we must choose types for $T_1 \ldots T_n$ satisfying the bounds of the corresponding parameters:

$$\mathit{inBounds}(T_1 \ldots T_n, P_1 \ldots P_n)$$

If we treat each assertion $T_{il} <: T_i <: T_{iu}$ in a constraint formula as a wildcard—`? extends` T_{iu} `super` T_{il}—we can represent CF as follows:

$$CF = \bigvee_{j=1}^{m} W_{1j} \ldots W_{nj}$$

To satisfy the bounds, we let

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

If the result is not well-formed, the given constraints are not satisfiable. We continue with $W_{12} \ldots W_{n2}$, etc., until a well-formed solution is found. If no well-formed solution can be found by this process, none exists, and the algorithm reports failure.

Where a well-formed result *is* found, we have produced a satisfactory inference result. In addition, $T_1 \ldots T_n$ are capture variables representing the *entire range* of possible results: for all $T'_1 \ldots T'_n$, where $T'_1 \ldots T'_n$ are valid choices for instantiation types, $\mathit{inBounds}(T'_1 \ldots T'_n, T_1 \ldots T_n)$ holds.

Without defining additional kinds of types or introducing more contextual information into the inference algorithm, it is impossible to predict *which* of the valid choices for $T'_1 \ldots T'_n$ is most useful to the programmer. In many typical circumstances, however, the capture variables $T_1 \ldots T_n$ are an inconvenient choice. A good best guess at the programmer's intent is the lower bound of these capture variables. We thus choose, for all i, to return the following as instantiation types:

$$T'_i = \lfloor T_i \rfloor [T_1 := T'_1 \;\ldots\; T_n := T'_n]$$

In most cases, this substitution eliminates all mention of the fresh capture variables from the inferred arguments. Note, however, that this is not always the case—if one of P_i appears in its own lower bound, for example, the corresponding capture variable will (harmlessly) remain in the inference result.

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` can be satisfied with either `T` = `String` or `T` = `Integer`, 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`); 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.

4.2.6 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—that is, we may not wish to determine what type is "expected" in the invocation's context. Then there are no constraints on the return type, and we do not include $E :>_? R$ in the result.

- R may be `void`. Again, there is no constraint for the return type and we do not incorporate $E :>_? R$.

- 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 $T_1 \ldots T_n$. So we can assume here that $A_1 \ldots A_m$ 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 $F_1 \ldots F_j$, and F_j is the array type F'_j`[]`. The constraint calculation must then contain $A_1 <:_? F_1 \ldots A_{j-1} <:_? F_{j-1}$ and, if $m \geq j$, $A_j <:_? F'_j \ldots A_m <:_? F'_j$.

- The type parameters of a class enclosing the method declaration may appear in $F_1 \ldots F_m$, R, or the bounds of $P_1 \ldots P_n$. This can be handled in one of two ways (where the class parameters are $Q_1 \ldots Q_k$ and the instantations are $S_1 \ldots S_k$):

– Perform substitution to instantiate the class variables. This is straightforward for $F_1 \ldots F_m$ and R, but more difficult for the method type parameters, because substitution does not recur into the bounds of variables (doing so would redefine the bounds of the given variable in the *upper* function, with potentially unexpected consequences). We can work around this problem by defining and performing inference with fresh parameters $P'_1 \ldots P'_n$ such that

$$\lceil P'_i \rceil = \lceil P_i \rceil [P_1 := P'_1 \ \ldots \ P_n := P'_n][Q_1 := S_1 \ \ldots \ Q_k := S_k]$$

$$\lfloor P'_i \rfloor = \lfloor P_i \rfloor [P_1 := P'_1 \ \ldots \ P_n := P'_n][Q_1 := S_1 \ \ldots \ Q_k := S_k]$$

– Include the class type parameters in the list of parameters to be inferred (that is, $P_1 \ldots P_k = Q_1 \ldots Q_k$). In addition to the bounds produced by the inference algorithm, we constrain each of $T_1 \ldots T_k$ to be bound above and below by $S_1 \ldots S_k$. We thus would not actually infer new types for these parameters, but we *would* get the necessary substitutions without any extra effort.

Chapter 5
Join-based Type System

We define the *join-based* type system by modifying the union-based version, eliminating union types from the definition and substituting the use of a *join* function in *capture* and $\wedge_{cf}$. The goal of the *join* function is to determine a non-union type that has the same properties as the union—all common supertypes are also supertypes of the join. In order to correctly define such a function in the presence of wildcards, we introduce *wildcard references*, which provide a mechanism for describing "infinite" wildcard bounds.

5.1 Fundamentals

5.1.1 Types

We revise the definition of *type argument* as follows. A type argument is one of:

- A type.
- A *wildcard* `? extends` T_u `super` T_l, where T_u and T_l are types.
- A *wildcard reference* `?`$_n$ where n is a natural number.

Wildcard references use de-Bruijn–style indexing to identify an enclosing wildcard to which the reference is treated as equivalent. For example, the informal type `List<? extends List<? extends List<...>>>` is expressed as `List<? extends`

`List<?`$_0$`>>`; `0` is the index of the directly enclosing wildcard. As a more complex example, both references in the type `C<? extends C<?`$_0$`> & D<? extends C<?`$_1$`>[]>>` point to the left-most wildcard. In examples we will sometimes, for the sake of readability, use an equivalent representation giving names to both wildcards and references: `C<?' extends C<?'> & D<? extends C<?'>[]>>`.

To correctly interpret wildcard references, some type operations must be parameterized by a wildcard context, Γ, which is simply a list of enclosing wildcards. The reference value, n, is an index into this list. We define the constant Γ_0 as the empty wildcard context; Γ_0 is *always* the parameter of an invocation of any operation (either external or recursive) unless a different parameter is explicitly provided. In operations parameterized by a wildcard context, the assumption that its parameters are structurally well-formed is weakened to allow types that are only structurally well-formed under the given context.

5.1.2 Structural Well-formedness

To simplify the scope of operations like subtyping in the presence of wildcard references, we only allow references to occur in specific contexts. A wildcard may only be referenced in its upper bound, and then only nested within some combination of intersections, wildcard upper bounds in parameterized types, and array types. For example, types such as `C<? extends D<? super C<?`$_1$`>>>` and `C<? extends D<C<?`$_0$`>>>` are not structurally well-formed.

It is also necessary, to prevent problems in the *capture* function, to prohibit the lower bound of a variable from being defined in terms of itself.

These stipulations are made explicit in the extended structural well-formedness definition:

- A wildcard-parameterized class type C`<`$W_1 \dots W_n$`>` is structurally well-formed in context Γ if and only if:
  - The class or interface named C exists and has n type parameters.
  - There exists some i such that W_i is a wildcard or wildcard reference.
  - For all i, if W_i is a type, it is structurally well-formed under Γ_0; otherwise, it is structurally well-formed under Γ.
- An array type T'`[]` is structurally well-formed in context Γ if and only if T' is structurally well-formed under Γ.
- A variable type X is structurally well-formed if and only if X is not *reachable* from $\lfloor X \rfloor$.
- An intersection type T_1 `&` $\dots$ `&` T_n is structurally well-formed in context Γ if and only if, for all i, T_i is structurally well-formed under Γ.
- A wildcard $W =$ `? extends` T_u `super` T_l is structurally well-formed in context Γ if and only if T_u is structurally well-formed under $W :: \Gamma$ and T_l is structurally well-formed under Γ_0.
- A wildcard reference `?`$_n$ is structurally well-formed under Γ if and only if $\Gamma[n]$ is defined.

Note that the predicate's domain is extended to include all type arguments.

To clarify the meaning of the variable case, a variable X is *reachable* from a type T if and only if $X = T$, X is reachable from one of T's component types, or T is a variable and X is reachable from $\lceil T \rceil$ or $\lfloor T \rfloor$.

5.1.3 Wildcard Capture

Because capture places the upper bound of a wildcard in a context in which that wildcard no longer encloses its bound, the operation must eliminate any free references, thus avoiding producing malformed types. This is done with the *unroll* operation.

$\mathit{unroll}(T)|_\Gamma = T$, with the following exceptions:

- $\mathit{unroll}(C\texttt{<}W_1 \ldots W_n\texttt{>})|_\Gamma = C\texttt{<}W'_1 \ldots W'_n\texttt{>}$ where, for all i:
  - If W_i is a wildcard `? extends` T_{iu} `super` T_{il},
    $W'_i =$ `? extends` $\mathit{unroll}(T_{iu})|_{W_i::\Gamma}$ `super` T_{il}.
  - If W_i is a wildcard reference $\texttt{?}_m$ and $|\Gamma| = m$, $W'_i = \Gamma[m]$.
  - Otherwise, $W'_i = W_i$.
- $\mathit{unroll}(T'\texttt{[]})|_\Gamma = \mathit{unroll}(T')|_\Gamma\texttt{[]}$.
- $\mathit{unroll}(T_1 \texttt{ \& } \ldots \texttt{ \& } T_n)|_\Gamma = T'_1 \texttt{ \& } \ldots \texttt{ \& } T'_n$ where, for all i, $T'_i = \mathit{unroll}(T_i)|_\Gamma$.

Capture requires that the type arguments $W_1 \ldots W_n$ be structurally well-formed. Thus, there is no i such that W_i is a wildcard reference. On the other hand, there may be some i such that W_i is a wildcard with an upper bound that contains free wildcard references. For example, the type $\texttt{C<?}_0\texttt{>}$ is not well-formed, but the wildcard `? extends` $\texttt{C<?}_0\texttt{>}$ is, and may appear in the type argument list.

To facilitate subtyping and *join*, *capture* consumes and produces an implicit parameter *source* which, like *upper* and *lower*, is threaded through subsequent operations. The *source* parameter is a function that maps capture variables to the wildcard–variable pair from which they were created.

We redefine capture as follows: $\mathit{capture}(W_1 \ldots W_n, P_1 \ldots P_n) = T_1 \ldots T_n$, 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:
  - $\lceil Z_i \rceil = \mathit{unroll}(W_{iu})|_{W_i}$ `&` $\lceil P_i \rceil[P_1 := T_1 \ \ldots \ P_n := T_n]$.
  - $\lfloor Z_i \rfloor = \mathit{join}(W_{il}, \lfloor P_i \rfloor[P_1 := T_1 \ \ldots \ P_n := T_n])$.
  - $\mathit{source}(Z_i) = (W_i, P_i)$.

The use of *join* in the lower bound of Z_i is problematic. The *join* function operates by comparing (via $<:$) and decomposing its arguments. But if Z_i were to occur within the type $\lfloor P_i \rfloor[P_1 := T_1 \ \ldots \ P_n := T_n]$, the function might need to know the lower bound of Z_i before it has been defined! More formally, where *capture* consumes bound function *lower* and extends it to produce $\mathit{lower}_1 \ldots \mathit{lower}_n$ (ultimately returning lower_n), lower_i cannot be passed to *join* when defining $\lfloor Z_i \rfloor$, because the definition of lower_i depends on that *join* invocation. Thus, we have stipulated that no variable may be reachable from its lower bound.

Even with this restriction, it is possible for a naive ordering of $\mathit{lower}_1 \ldots \mathit{lower}_n$ to cause problems: if Z_3 is reachable from $\lfloor Z_1 \rfloor$, then the definition of lower_1 depends on lower_3, and $\lfloor Z_3 \rfloor$ must be resolved before $\lfloor Z_1 \rfloor$. Implementations must determine a safe sequence of *join* invocations that avoids all such problems.

5.1.4 Subtyping

The subtype relation is parameterized by a wildcard context, Γ, and a list of types, γ. The lengths of these two lists must be equal; γ conveys the fact that the current invocation of subtyping arose out of a test that the type $\gamma[i]$ is within the upper bound of $\Gamma[i]$. By default, γ is the constant γ_0, an empty list.

We modify the rules for subtyping, $S <: T|_{\Gamma,\gamma}$, as follows:

- If $S = C_s$`<`$S_1 \ldots S_n$`>`, $T = C_t$`<`$W_1 \ldots W_m$`>`, and $C_s = C_t$, then for all i:
  - If W_i is a type T_i, $S_i \cong T_i$.
  - If W_i is a wildcard `? extends` T_{iu} `super` T_{il}, $S_i <: T_{iu}|_{W_i::\Gamma,S_i::\gamma}$ and $T_{il} <: S_i|_{\Gamma_0,\gamma_0}$.
  - If W_i is a wildcard reference `?`$_k$, either $\exists j, \Gamma[j] = \Gamma[k] \wedge (\gamma[j] = S_i \vee \mathit{source}(\gamma[j]) = \mathit{source}(S_i))$; or the wildcard case holds for $\Gamma[k]$.
- The parameters Γ and γ are propagated into other recursive invocations of subtyping (rather than the defaults Γ_0 and γ_0) whenever:
  - The second parameter to the recursive invocation is T, unmodified.
  - $T = T'$`[]`.
  - $T = T_1$ `&` $\ldots$ `&` T_n.

The parameters Γ and γ are used to accomplish two things. First, as discussed previously, the wildcard context allows us to determine the meaning of wildcard references appearing in T. (Were we to recur on a wildcard upper bound appearing in S, we would need *two* wildcard contexts, one for S and one for T. But we always perform capture where S is wildcard-parameterized.) Second, we must be able to recognize combinations of S and T in which S *should* be a subtype of T, but, in the absence of γ, no finite application of the rules can show it.

This second point requires some discussion. Consider the type $T =$ `C<?' extends C<?'>>`. Assume the class `D` is declared as `D extends C<D>`. Then in order to show that `D` $<:$ T, we might follow these steps:

$$\begin{aligned}
\texttt{D} &<: \texttt{C<?' extends C<?'>>} \\
\texttt{C<D>} &<: \texttt{C<?' extends C<?'>>} \\
\texttt{D} &<: \texttt{C<?'>}\text{ where }\texttt{?' extends C<?'>} \\
\texttt{C<D>} &<: \texttt{C<?'>}\text{ where }\texttt{?' extends C<?'>} \\
\texttt{D} &<: \texttt{C<?'>}\text{ where }\texttt{?' extends C<?'>} \\
&\ldots
\end{aligned}$$

The subtype relation must be defined to somehow "catch" this infinite regress and determine that it constitutes a valid subtyping.*

To formalize our understanding of subtyping in the presence of wildcard references, we define a notion of *finite expansion* that removes references to certain wildcards from a type.

The *kth-degree finite expansion* of a type, $\langle T\rangle^k|_\Gamma$, is defined as follows:

- $\langle C\texttt{<}W_1 \ldots W_n\texttt{>}\rangle^k|_\Gamma = C\texttt{<}W'_1 \ldots W'_n\texttt{>}$ where, for all i:
  - If W_i is a wildcard `? extends` T_{iu} `super` T_{il}, $W'_i =$ `? extends` $\langle T_{iu}\rangle^k|_{W_i::\Gamma}$ `super` T_{il}.
  - If W_i is a wildcard reference $\texttt{?}_m$, $k = 0$, and $|\Gamma| = m$, $W'_i =$ `? extends Object super null`.
  - If W_i is a wildcard reference $\texttt{?}_m$, $k > 0$, $|\Gamma| = m$, and $\Gamma[m] =$ `? extends` T_{iu} `super` T_{il}, $W'_i =$ `? extends` $\langle T_{iu}\rangle^{k-1}|_{\Gamma_0}$ `super` T_{il}.
  - Otherwise, $W'_i = W_i$.
- $\langle T'\texttt{[]}\rangle^k|_\Gamma = \langle T'\rangle^k|_\Gamma\texttt{[]}$.
- $\langle T_1$ `&` $\ldots$ `&` $T_n\rangle^k|_\Gamma = T'_1$ `&` $\ldots$ `&` T'_n where, for all i, $T'_i = \langle T_i\rangle^k|_\Gamma$.

* There are other forms of infinite regress, some of which are highlighted by Kennedy and Pierce [6], that must *not* constitute a valid subtyping.

- In all other cases, $\langle T\rangle^k|_\Gamma = T$.

For example:

$\langle$`C<?' extends C<?'>>`$\rangle^0 =$ `C<? extends C<?>>`
$\langle$`C<?' extends C<?'>>`$\rangle^1 =$ `C<? extends C<? extends C<?>>>`
$\langle$`C<?' extends C<?'>>`$\rangle^2 =$ `C<? extends C<? extends C<? extends C<?>>>>`

Finite expansion is defined such that, for all T, $\langle T\rangle^0 :> \langle T\rangle^1 :> \langle T\rangle^2 :> \ldots :> T$. The type T can be viewed as the limit of these finite expansions as the degree approaches ∞. This leads us to the conclusion, based on transitivity of subtyping, that the following property must hold in the subtype relation:

Assume there exists some k such that, for all n, $S <: \langle T\rangle^{kn}$. Then $S <: T$.

With this understanding, it makes sense that the recursion in subtyping halts in the wildcard reference case where S_i is already being compared to $\Gamma[k]$. We're essentially converting the wildcard reference to `? extends Object super null`, as in finite expansion. We thus transform an attempt to prove that $S_i <: T$ into a proof that, for some k, $S_i <: \langle T\rangle^k$. Further, the demonstration provides evidence for the inductive case: assuming $S_i <: \langle T\rangle^{kn}$ for some n, $S_i <: \langle T\rangle^{k(n+1)}$.

The subtyping clause for detecting infinite regress is more complex than simply asserting that $\gamma[k] = S_i$. First, it is possible for the same wildcard to appear multiple times in Γ before being compared twice to the type S_i. If, for example, the wildcard reference appears 5 levels deep (that is, it is the reference `?`$_5$), while S_i is similarly defined in terms of itself, but at 3 levels deep, Γ will have size 15 before S_i is compared to the same wildcard; at that point, Γ will contain 3 instances of the wildcard. So the index j may be chosen for any position at which $\Gamma[j] = \Gamma[k]$. Second, because capture produces unique variables with each application, two capture variables that were produced from the same wildcard–variable pair must be considered equivalent. Otherwise, not even the reflexive assertion `C<?' extends C<?'>>` $<:$ `C<?' extends C<?'>>` can be demonstrated!

5.1.5 Join Function

The function $join(S, T)$ produces a tight upper bound on S and T. That is, for all U such that $S <: U$ and $T <: U$, $join(S, T) <: U$. The use of $join$ extends to an arbitrary number of arguments, where $join() =$ `null`, $join(T) = T$, and $join(T_1 \dots T_n) = join(\dots join(join(T_1, T_2), T_3), \dots, T_n)$.

An additional parameter, θ, is a list of pairs of types. It is used to detect contexts in which a wildcard reference must be created. By default, external invocations provide an empty list (θ_0); recursive invocations propagate the given θ.

$join(S, T)|_\theta$ is defined as follows:

- If $T <: S$, $join(S, T)|_\theta = S$.
- Else if $S <: T$, $join(S, T)|_\theta = T$.
- Otherwise, the result is given in Table 5.1. (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 case that cannot occur.)

Where $S \cong T$, the choice of S as the result over T is arbitrary and inconsequential, since we can freely substitute T for S.

	`null`	C_t<$T_1 \dots T_m$>	C_t<$W_1 \dots W_m$>	C_t
`null`	-	-	-	-
C_s<$S_1 \dots S_n$>	-	[1]	$join(S, \lVert T \rVert)$	$join(S, T\uparrow)$
C_s<$W_1 \dots W_n$>	-	$join(\lVert S \rVert, T)$	$join(\lVert S \rVert, \lVert T \rVert)$	$join(S, T\uparrow)$
C_s	-	$join(S\uparrow, T)$	$join(S\uparrow, T)$	$join(S\uparrow, T\uparrow)$
p_s`[]`	-	[2a]	[2a]	[2a]
S'`[]`	-	[2a]	[2a]	[2a]
X_s	-	$join(\lceil S \rceil, T)$	$join(\lceil S \rceil, T)$	$join(\lceil S \rceil, T)$
S_1 `&` ... `&` S_n	-	[4a]	[4a]	[4a]

	p_t`[]`	T'`[]`	X_t	T_1 `&` ... `&` T_m
`null`	-	-	-	-
C_s<$S_1 \dots S_n$>	[2b]	[2b]	$join(S, \lceil T \rceil)$	[4b]
C_s<$W_1 \dots W_n$>	[2b]	[2b]	$join(S, \lceil T \rceil)$	[4b]
C_s	[2b]	[2b]	$join(S, \lceil T \rceil)$	[4b]
p_s`[]`	[3]	[3]	$join(S, \lceil T \rceil)$	[4b]
S'`[]`	[3]	$join(S', T')$`[]`	$join(S, \lceil T \rceil)$	[4b]
X_s	$join(\lceil S \rceil, T)$	$join(\lceil S \rceil, T)$	$join(\lceil S \rceil, \lceil T \rceil)$	$join(\lceil S \rceil, T)$
S_1 `&` ... `&` S_n	[4a]	[4a]	$join(S, \lceil T \rceil)$	[5]

[1]: There are two cases:

- If $C_s = C_t$, C_s<$J_1 \dots J_n$>, where for all i:
  - If $S_i \cong T_i$, $J_i = S_i$
  - If, for some k, $\theta[k] = (S'_i, T'_i)$ where $S_i = S'_i \lor source(S_i) = source(S'_i)$ and $T_i = T'_i \lor source(T_i) = source(T'_i)$, $J_i = ?_k$
  - Otherwise, $J_i =$ `? extends` $join(S_i, T_i)|_{(S_i,T_i)::\theta}$ `super` S_i `&` T_i
- If $C_s \neq C_t$, $join(S\uparrow, T)$ `&` $join(S, T\uparrow)$.

[2a]: $join($`Cloneable & Serializable`$, T)$

[2b]: $join(S,$ `Cloneable & Serializable`$)$

[3]: `Cloneable & Serializable`

[4a]: $join(S_1, T)$ `&` ... `&` $join(S_n, T)$

[4b]: $join(S, T_1)$ `&` ... `&` $join(S, T_m)$

[5]: $join(S_1, T_1)$ `&` ... `&` $join(S_1, T_m)$ `&` ... `&` $join(S_n, T_1)$ `&` ... `&` $join(S_n, T_m)$

Table 5.1 : $join(S, T)$ where S and T are not directly related

The rules for computing *join* in Table 5.1 are derived directly from the subtype relation (Table 4.1), under the assumptions that $S \not<: T$ and $T \not<: S$. As we would expect, the table is symmetric, corresponding to the symmetry of the *join* function.

Case [1] is of particular interest. By allowing wildcards to have both upper and lower bounds, we are able to produce a tight bound on two different parameterizations of the same class. For example, where we have interface declarations `A`, `B extends A`, and `C extends A`, the result of *join*(`List<B>, List<C>`) is `List<? extends A super B & C>`. Without both bounds, we would be forced, as observed in the Aarhus–Sun paper [8], to choose between `List<? extends A>` and `List<? super B & C>`, both of which are valid results. Since neither result is a subtype of the other, a type satisfying the join property would not exist.

Wildcard references are produced in case [1] where a recursive invocation of *join* would otherwise produce the same result as the invocation represented by $\theta[k]$. As an example, the type `Integer` implements `Comparable<Integer>`, while the type `Double` implements `Comparable<Double>`; both are `Number`s. So we have (abbreviating the names):

$$join(\texttt{Int}, \texttt{Dbl}) = \texttt{Nbr \& Cm<?' extends Nbr \& Cm<?'> super Int \& Dbl>}$$

Case [5] produces an intersection of arity $n \times m$. While the size of the result is daunting, a good implementation ought to optimize the operation—for example, it is quite likely that some of the results of *join* would be redundant, and the implementation could then replace the result with a simpler, equivalent type.

5.2 Type Argument Inference

The two core functions in type argument inference, $<:_?$ and $:>_?$, must be updated to reflect the changes that have been made to the subtype relation. In addition, $\wedge_{cf}$ must be defined in terms of *join* instead of union types. The rest of the algorithm is unchanged.

5.2.1 Constraint Formulas

We revise $\wedge_{cf}$ as applied to simple constraint formulas as follows. Let $SC_1 \ldots SC_m$ be simple constraint formulas. Let T_{ijl} refer to the lower bound of T_i in SC_j, and T_{iju} refer to the corresponding upper bound. Then $\wedge_{cf}(SC_1 \ldots SC_m)$ has value

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

5.2.2 Subtype Inference

In addition to μ, $<:_?$ is now parameterized by wildcard context Γ and list of types γ. As in subtyping, these parameters are by default Γ_0 and γ_0, but are propagated into recursive invocations where F is unchanged, F is an array, or F is an intersection.

We also modify the definition of $A <:_? F|_{\Gamma,\gamma,\mu}$ in the following ways:

- Where F *and* the wildcards to which F contains references involve none of $P_1 \ldots P_n$, the result is $A <: F|_{\Gamma,\gamma}$ (treating the boolean result of $<:$ as a trivial constraint formula).
- If $A = C_a$<$A_1 \ldots A_n$>, $F = C_f$<$W_1 \ldots W_m$>, and $C_a = C_f$, the result is $\wedge_{cf}(CF_1 \ldots CF_n)$ where, for all i:
  - If W_i is a type, $CF_i = A_i \cong_? W_i$.

  - If W_i is a wildcard `? extends` F_{iu} `super` F_{il},
    $CF_i = \wedge_{cf}(A_i <:_? F_{iu}|_{W_i::\Gamma, A_i::\gamma}, A_i :>_? F_{il}|_{\Gamma_0,\gamma_0})$.
  - If W_i is a wildcard reference `?`$_k$ and $\exists j, \Gamma[j] = \Gamma[k] \wedge (\gamma[j] = A_i \vee \mathit{source}(\gamma[j]) = \mathit{source}(A_i))$ then $CF_i = \mathit{true}$; otherwise, CF_i is the result of the wildcard case for $\Gamma[k]$.

It is possibly the case that $A <:_? F$ will never be invoked where F contains wildcard references—given the ways in which the inference algorithm uses this function (and the language specification uses the inference algorithm), F arises out of either the declared type of a formal parameter or the declared return type of a method, neither of which are types produced by *join*. If that is true, these extensions to $<:_?$ are unnecessary. However, A in $A :>_? F$ certainly *can* contain wildcard references, so there is little benefit to assuming they are not present here, and we prefer not to make such a guarantee.

5.2.3 Supertype Inference

The changes to the subtype relation are similarly reflected in the definition of $:>_?$. We add parameters Γ and γ. These are by default Γ_0 and γ_0, except when propagated into recursive invocations when A is unchanged, A is an array, or A is an intersection.

Similar changes to the other rules must also be made:

- Where F involves none of $P_1 \dots P_n$, the result is $F <: A|_{\Gamma,\gamma}$ (treating the boolean result of $<:$ as a trivial constraint formula).

- If $A = C_a$`<`$W_1 \ldots W_n$`>`, $F = C_f$`<`$F_1 \ldots F_m$`>`, and $C_a = C_f$, the result is $\wedge_{cf}(CF_1 \ldots CF_n)$ where, for all i:
  - If W_i is a type, $CF_i = W_i \cong_? F_i$.
  - If W_i is a wildcard `? extends` A_{iu} `super` A_{il}, $CF_i = \wedge_{cf}(A_{iu} :>_? F_i|_{W_i::\Gamma, F_i::\gamma}, A_{il} <:_? F_i|_{\Gamma_0,\gamma_0})$.
  - If W_i is a wildcard reference `?`$_k$ and $\exists j, \Gamma[j] = \Gamma[k] \wedge (\gamma[j] = F_i \vee$ $source(\gamma[j]) = source(F_i))$ then $CF_i = true$; otherwise, CF_i is the result of the wildcard case for $\Gamma[k]$.

Chapter 6

Comparison of Type System Variations

Now that we have three variations on the Java type system—union-based, join-based, and the original (as defined by the JLS [2])—we can discuss the comparative merits of each. Table 6.1 highlights the most important differences.

The greatest strength of the JLS system is, of course, that it is the current standard—changes to that standard have far-reaching impact, and must not be made lightly. On the other hand, there are clear problems with the JLS that need to be addressed, and as long as these changes are being made, it would make sense to consider the benefits of the other variations.

In addition to supporting the extended features described earlier, both the union-based and join-based variations address one shortcoming of the JLS: type argument inference is an inflexible, heuristic procedure, making no guarantees about soundness or completeness.* While sharing a strong inference algorithm, the two other systems are distinguished by the presence of union types on the one hand and wildcard references on the other. The union-based system has a clear advantage in this regard, due to its relative simplicity.

Below, we describe how the join- and union-based systems differ from the JLS in specific operations. Any major shortcomings of the JLS are highlighted, while

* To deal with potential unsoundness, the results of inference are checked for correctness before being accepted by the type checker.

	JLS	*Union-based*	*Join-based*
Unique features	Self-referencing wildcards (but their handling in subtyping is unspecified).	Union types.	Self-referencing wildcards.
Inference	Unsound, incomplete, contains bugs; no disjunctions.	Sound and complete; requires reasoning about disjunctions.	
Necessary restrictions	Variables with lower bounds are inexpressible, as are intersections in most contexts; no multiple-instantiation inheritance.	None.	Variables cannot have self-referencing lower bounds.
Other	Wildcards can't have both bounds; type `null` is inexpressible; variable bounds can't be arrays.	Subtyping uses equivalence, not equality; inference always uses the expected type where available.	

Table 6.1 : Major differences in type system variations

additional "bugs" and confusing content in the JLS are described in Appendix A. We conclude by discussing how defining the specification in terms of one of these systems would affect backwards compatibility.

6.1 Comparison of Type Operations

6.1.1 Fundamentals

Types. While wildcard references are not explicitly described in the JLS definition of types (4.3-4.5, 4.8-4.9), it requires in the definition of *join* (15.12.2.7) that some form of "infinite types" exist. Thus types in the JLS are like types in the join-based system, with the exception that an intersection in the JLS is only permitted to appear as the upper bound of a variable. The JLS definition also excludes wildcards with *both* upper and lower bounds.

Wildcard capture. The lower bound of a capture variable in the JLS is simply the lower bound of the wildcard—the corresponding parameter must always have a `null` lower bound. Extending the language to support lower bounds on declared variables requires the use of unions or *join* in capture.

Subtyping. The most significant difference between the union- and join-based subtyping rules and the JLS rules is the use of *equivalence* when comparing arguments of ground parameterized types rather than *equality*. This change is not strictly necessary, but it provides some convenience to programmers, and gives language specifiers and implementors far greater flexibility. The order of the types in an intersection produced by inference, for example, need not be made explicit, because `C<A & B>` is equivalent to `C<B & A>`. And the implementor is free to simplify types to their equivalents without such optimizations being explicitly specified. The use of equivalence has positive implications for backwards compatibility as well, as described below.

The JLS fails to describe how subtyping works in the presence of wildcard references. Without specific rules for demonstrating membership in the relation, a variety of valid subtypings are implicitly prohibited. This failure is addressed by the join-based system.

Well-formedness. In most respects, the union- and join-based systems have less restrictive well-formedness rules than the JLS. Most significantly, the class table and intersections expressed in code are prohibited by the JLS from exhibiting multiple-instantiation inheritance—no type may be a subtype of two distinct parameterizations of the same class. Further, the form of intersections is restricted so that they can have only one most specific array or non-interface class supertype; and when expressed in code, intersections are restricted such that a non-interface class type may only appear as the first component type. Some of these restrictions help to simplify type argument inference (as it is defined in the JLS), but they are otherwise unnecessary.

If convenience to programmers is the goal of limitations on intersections, it might be more useful to instead check that an intersection expressed in code is not trivially uninhabited (except by `null`)—for example, only the value `null` has type `Integer & Float` or `String & Cloneable` (`String` is declared `final` and does not implement `Cloneable`). We do not specify such an analysis here.

One restriction that *should* be present in the JLS but is not is the requirement that a variable's lower bound be a subtype of its upper bound. Variables with both bounds may only be produced in the JLS system by wildcard capture, but it is possible for users to provide lower bounds on wildcards that lead to these inconsistent variables.†

Join. In the JLS, *join* is called *lub* (15.12.2.7). The function sometimes produces a tight result, as does the join-based system, but in many instances is less precise. As discussed in the join-based definition, wildcards with both upper and

† The `javac` compiler does insure that wildcard lower bounds are consistent with the corresponding parameters' upper bounds, despite the absence of such an assertion in the JLS.

lower bounds are essential to defining a tight *join* function. The JLS version is thus inherently limited. In addition, it does not produce a best bound in some situations in which such a bound *is* expressible: *join*(`List<Object>`, `List<String>`), for example, produces `List<?>`, not `List<? super String>`; variables are not produced by the function under any circumstances; arrays are not handled correctly—they must always be erased; and, in a case that is simply incorrect, invocations like *join*(`List<? extends Number>`, `List<? super Number>`) produce results like `List<Number>`.[‡]

The JLS version of *join* does not perform capture when handling wildcard-parameterized class types in some cases. Since wildcards carry with them the assumption that instantiations will fall within the parameter's declared bounds, this is a valid approach to take. Capture must be used to determine a direct supertype, however, so we prefer the brevity of consistently using capture in the join-based system.

6.1.2 Type Argument Inference

The type argument inference algorithms in the union-based and join-based systems are essentially equivalent, with slight adjustments made to support different subtype relations. The inference algorithm defined in the JLS (15.12.2.7) is also similar, at a high level, to these algorithms. It relies on analogs to $<:_?$ and $:>_?$, which produce

[‡] Fortunately (for the sake of type safety), `javac` does not faithfully implement the JLS in this last case.

bounding constraints on the inferred types; the inferred lower bounds are combined via *join* to determine a satisfactory type. There are significant differences between the algorithms, however.

First, constraint formulas in the JLS are constraint *sets*, capable of representing conjunctions but not disjunctions (in other words, they can only represent simple constraint formulas). The JLS algorithm avoids the need to describe disjunctions based on the following simplifying assumptions:

- Neither A nor F is a union type.
- F, if it is a variable but not one of $P_1 \ldots P_n$, cannot involve any of $P_1 \ldots P_n$. This is because F is always derived from a type appearing in the method signature, no *declared* variable appearing there can have $P_1 \ldots P_n$ in scope at its declaration point, and *capture* variables are never produced by the JLS algorithm.
- In $<:_?$, where A is an intersection 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, there is at most one most specific array supertype of the intersection.[§]
- F cannot be an intersection. Intersections can only be reached recursively by the inference functions. However, variables, which might have intersections in

[§] As 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.

their upper bounds, cannot involve any of $P_1 \ldots P_n$, and so the recursion would never occur; and for the reasons outlined in the previous point, there is no need to represent the supertype of a class type as an intersection.

In the presence of these simplifications, it's clear from Tables 4.2 and 4.3 that $\vee_{cf}$ need never be invoked. So the only useful constraint formula that is inexpressible in the JLS representation is $false$ (the handling of this problem is described below). However, if we want to allow arbitrary intersection types, use *capture* on F to improve the algorithm, or relax any of these assumptions, a more versatile constraint representation is necessary.

Second, the JLS inference algorithm is (intentionally, in some cases) unsound. The following limitations lead to unsoundness:

- The $false$ constraint formula is not representable, so situations that lead trivially to unsatisfiability are merely ignored, as are inconsistent combinations of lower and upper bounds on any of $T_1 \ldots T_n$.

- Default bounds on wildcards are incorrectly ignored. For example, `List<?>` $<:_?$ `List<? extends T>` should produce `Object` $<:$ `T` $<:$ `Object`; the JLS algorithm produces $true$ instead.

- If A in $:>_?$ is `null`, the result is $true$ (rather than, for example, `null` $<:$ `T` $<:$ `null`).

- If A in $:>_?$ is a variable, it is not handled correctly: if its *upper* bound is an array, the algorithm recurs on that bound; otherwise, it produces $true$. The correct behavior is to recur on the *lower* bound.

- In situations in which one of the P_is' upper bound is used to produce T_i, references to $P_1 \dots P_n$ in that bound may leak into the calling context. For example, given method signature `<T extends List<T>> T method()`, inference may determine (depending on the value of E) that `T` = `List<T>`.

These problems don't lead to an unsound *type system*, because the JLS requires an additional check that the inferred types are satisfactory before using them. They *can*, however, lead to unnecessary failure of the algorithm and encourage implementation bugs. The following program, for example, is compiled by `javac` without warning, but throws a `ClassCastException` at run time. The cause of the bug seems to be the third item above, compounded by the fact that the compiler does not verify its inference results in this case.

```
<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);
}
```

Third, the JLS inference algorithm is incomplete. This is due in part to the limitations of the *join* function, as discussed previously. In addition, the algorithm never invokes *capture*. As noted above, using capture in some cases would require support for disjunctive constraint formulas (performing capture on A, however, has no such effect). But a consequence of *not* using capture is that types that could be produced from the bounds of parameter declarations are ignored, and, consequently, the inferred constraints may be too restrictive. Consider, for example, a class declaration `Foo<X extends A>` (let `A` and `B` be interface names). The invocation `Foo<? extends B>` $<:_?$ `Foo<? extends T>` produces `Z` $<:$ `T` $<:$ `Object` where $\lceil$`Z`$\rceil$ = `A & B`; the JLS algorithm produces the more restrictive `B` $<:$ `T` $<:$ `Object`. The type `A` meets the constraints on `T` in the first case, but not the second.

Finally, the initial framing of the problem and final process of choosing inferred types are slightly different in the JLS algorithm. We must use *capture* in the union- and join-based algorithms to satisfy the *inBounds* condition where variables may have lower bounds; this is not necessary where there are no declared lower bounds (although it would solve the problem described above of parameters in *upper* bounds appearing out of context).

The JLS also differs in not incorporating the method return type and expected type (R and E) into the algorithm unless the lower bound produced for a type variable is `null`. Incorporating such contextual information into type checking is a significant (and perhaps undesirable) paradigm shift, but since the JLS already requires this information, there is no reason not to take advantage of it in all cases.

In addition, where a variable has lower bound `null` even *after* considering the return type, the JLS chooses its *upper* bound as the result. It is an open question whether such a choice is more useful to programmers in general; as we discussed previously, the choice of specific types for $T_1 \ldots T_n$ once the bounds on the variables have been established is essentially arbitrary.

6.2 Backwards Compatibility

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

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` $= T_1$ and the context of the invocation requires a `List<`T_1`>`, an algorithm that produces a *better* but different type T_2 will lead to an assertion that `List<`T_2`>` $<:$ `List<`T_1`>`, 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 previous section's outline of problems in the JLS's *join* function and type argument inference algorithm should provide motivation for fixing these operations, even if the other recommendations in this thesis are not incorporated. 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 JLS, especially in areas where the JLS is incorrect. So it's not clear exactly which language any changes should seek to be backwards-compatible with.

The most attractive path is to introduce a new source language version that is not entirely backwards compatible. This is analogous the the situation in which a `1.4` source language version was specified in order to add the keyword `assert` to the language, breaking any previous programs that happened to use `assert` as a variable name. A source-to-source tool could be developed that implemented both the old and new type systems, and inserted casts or explicit type arguments as necessary wherever the two conflicted.

A variety of properties of the union- and join-based type systems 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, and because the inference algorithm is complete, changes to inference will *never* be problematic in contexts in which E is known (this includes assignments and return statements).

- The results of $join$ are always subtypes of the results produced by the JLS (except where the JLS is incorrect); in practice, inferring a subtype for a parameter is often safe, since type variables in method return types are frequently not nested.

- The use of equivalence in subtyping allows changes that produce different but equivalent types to be permitted.

Assuming one of the type systems defined in this thesis were to be adopted, which one would be more appropriate? The union-based system has some clear advantages over the join-based version. The lack of wildcard references greatly simplifies the specification and implementation. And it makes no restriction of self-referencing lower variable bounds. Its biggest drawback is that it is less backwards-compatible. Assume classes `B` and `C` are subtypes of `A`. In a context in which the expected type, E, is unknown, the invocation `Arrays.asList(new B(), new C())` has type `List<A>` under the join-based system and `List<B|C>` in the union-based system. The first type is backwards-compatible, while the second is not. In general, where E is unknown, the arguments to a method imply more than one bound on a parameter, and the parameter appears in an invariant context in the return type, the union-based system will often be incompatible because it will give the parameter a union type. The join-based system will be compatible in many of these instances. While this situation probably does not arise frequently, the difference is worth some consideration. The benefits of the union-based system, however, may be enough to outweigh these compatibility concerns.

Chapter 7
Conclusion

We have defined two variations on the Java type system able to support lower bounds on declared type variables, wildcards with both upper and lower bounds, and intersections as first-class types. We have demonstrated that the operations the Java language performs on types can be extended to correctly handle these types without artificial restrictions. In addition, we have demonstrated that the proper application of these extensions makes possible the definition of a sound and complete type argument inference algorithm.

An immediate goal of this work is to incorporate one of the two variations into a future version of the Java language; as discussed when comparing the improved type systems to the current specification, a number of flaws in the JLS strengthen this cause, since a revision that is not backwards-compatible probably ought to be made anyway to fix those bugs. The argument for these changes could be further strengthened by an experimental study of their practical impact. It would be useful to demonstrate, for example, that many current Java programs would not be adversely affected by backwards-compatibility problems under one or both of the type system variations.

The described type systems are being implemented as components of the DrJava IDE's interactive interpreter [10]. When completed, the updated interpreter will provide a useful demonstration of these modified typing rules.

Another potential path for future work is to complement the various assertions of safety, well-formedness, soundness, etc., made throughout this thesis with formal proofs. The scope of this work—a full, practical object-oriented language rather than a theoretical calculus such as Featherweight Java [3]—makes such a task significant, but clearly the support provided by formal proof is important in establishing that these type systems are not just convenient, but ideal in important ways.

Finally, it may be useful to explore how some of the changes introduced in these type systems might support future features of the language. Union types, for example, could be quite useful to users when combined with a pattern-matching construct, as discussed by Igarashi and Nagira [5]. And tightly-bounded variables, such as `LN extends List<Number> super List<Number>`, combined with support for equivalence in subtyping, provide a convenient framework for supporting type aliases—the variable `LN` could be used wherever `List<Number>` was required.

Appendix A
Summary of Bugs in the JLS

This thesis notes in passing a variety of specification bugs in the JLS. A comprehensive list of the bugs we have encountered in working with the JLS follows, including many not noted elsewhere.

A.1 Errors

- The textual definition of reference types (4.3) lists class types, interface types, and array types, but not variable types; the formal definition *does* include variable types. Variable types are also ignored in 4.3.4. (It would also make sense to include the null type and intersection types in this definition, despite the fact that they are not expressible in program code, because they *are* reference types.)

- The definition of "type variable" does not include a lower bound (4.4), but subtyping (4.10.2) and capture conversion (5.1.10) require variables with lower bounds. There is no compile-time restriction on the relationship between a lower and an upper bound (analogous to the compile-time restrictions that apply to intersections produced by capture conversion (4.9)), so it is not clear that a static error should occur when `class Foo<T extends Integer>` is instantiated as `Foo<? super Number>` (`javac` treats this as an error, and it would be unwise not to do so).

- The definition of subtyping involving intersections (4.10) is incomplete. It does not allow an intersection to appear as the direct supertype of *any* type, implying that an intersection is only a supertype of itself. This is clearly inconsistent with the definition of intersection types (4.9). The correct subtyping definition, as presented here, is nontrivial when an intersection is compared to a variable or to another intersection.

- The definition of subtyping does not cover infinite types. For example, the fact that `X extends List<X>` $<:$ `List<? extends List<...>>` cannot be arrived at by any number of applications of the *direct supertype* and *containment* relations.

- The type argument inference algorithm (15.12.2.7) does not correctly handle the null type. `null` $>>$ `T` ought to produce `T` $<:$ `null`.

- The type argument inference algorithm (15.12.2.7) does not correctly handle variables. $A << F$, where A is a variable with an intersection upper bound and F is an array, produces no constraints. (This is incorrect if one of the members of the intersection is an array type.) $A >> F$ or $A = F$, where A is a variable and F is an array type, incorrectly recurs on A's upper bound (this is only valid for $A << F$). $A >> F$, where A is a variable and F is *not* an array type, produces no constraints; it ought to recur on A's lower bound.

- Trivial bounds on wildcards are incorrectly ignored by the type argument inference algorithm (15.12.2.7). The fact that the wildcard in `List<?>` has upper bound `Object` and lower bound `null` ought to be exploited where possible; bounded wildcards *also* have trivial bounds that can be exploited. For example, `List<? super String>` $<<$ `List<? extends T>` ought to produce `T` $:>$ `Object`.

- The *lub* function (a.k.a. *join*, 15.12.2.7) is defined incorrectly for some wildcard-parameterized types: *lub*(`List<? extends A>`, `List<? super A>`) would produce `List<A>`, which is clearly incorrect. `javac` fixes the bug.

- The process that produces an infinite type is not formally specified by *lub* (a.k.a. *join*, 15.12.2.7). A strict interpretation leads to the conclusion that a *lub* invocation that *ought* to produce an infinite type actually has an undefined result.

- The inference algorithm's handling of unresolved type arguments (15.12.2.8) allows references to an unresolved parameter in the upper bound of a parameter to leak into the calling context. For example, given method signature `<T extends List<T>> T method()`, inference may determine (depending on the assignment context) that `T` $=$ `List<T>`.

A.2 Simple Missing Features

- The implicit use of equality in the subtype relation (4.10) when comparing type arguments is based on syntactic equality ("the same," 4.3.4). Thus, the following code is illegal, because a tightly-bound variable is not syntactically equal to its bounds. `javac` does not report an error, and we favor its approach, defining "equals" in terms of mutual subtypes.

  ```
  class NumBox<T extends Number> { ... }
  NumBox<? super Number> nb = ...;
  NumBox<Number> nb2 = nb;
  ```

- Unboxing conversion (5.1.8) should be defined to explicitly support *subtypes* of the wrapper types, such as a variable with bound `Integer`. It should be possible, for example, to unbox the members of a `List<? extends Integer>`. Alternately, *all* contexts that allow an unboxing conversion (such as numeric promotion (5.6)) should allow a preceding widening reference conversion; this second formulation may be more confusing, though, because boxing conversions do not support a similar widening primitive conversion before boxing.

- The type argument inference algorithm (15.12.2.7) could produce more general results by invoking capture on A where A is a wildcard-parameterized class type. For example, given class declaration `Foo<X extends A>` (let `A` and `B` be interface names), the invocation `Foo<? extends B>` $<<$ `Foo<? extends T>` produces `T` $:>$ `B`; it could produce the less restrictive `T` $:>$ `A & B`. The type `A` meets the constraints on `T` in the first case, but not the second.

- *lub* (a.k.a. *join*, 15.12.2.7), due to its dependence on erasure, does not satisfactorily handle variables or arrays of parameterized class types. For example, given variable declarations `<X, Y extends X, Z extends X>`, $lub(\texttt{Y}, \texttt{Z}) = $ `Object`, because $|\texttt{X}| = $ `Object`. Clearly, `X` would be a more useful result in this case. (In the case of arrays, $lub(\texttt{List<Integer>[]}, \texttt{List<Double>[]}) = $ `List[]`, based on the JLS. `javac` provides the more useful `List<? extends Number>[]`.)

- Where *lub* (a.k.a. *join*, 15.12.2.7) produces no useful upper bound for a wildcard, it would be convenient to produce a lower bound instead. For example, $lub(\texttt{List<Object>}, \texttt{List<String>}) = $ `List<?>`, but could produce the more useful `List<? super String>`.

- Since the type argument inference algorithm (15.12.2.7) requires in some cases the incorporation of the expected return type, it would be trivial to allow its use in *all* cases. This would allow invocations like `List<Number> l = Arrays.asList(1, 2, 3)`, which is currently incorrect because the algorithm chooses `Integer` as the type of the list, not `Number`. This change could have no ill effects on backwards compatibility, because it would only produce a different return type for the method where the currently-inferred type is incompatible with the calling context—only programs that contain type errors currently would be typed differently.

- It's not clear that the type argument inference algorithm's (15.12.2.7) choice of a parameter's inferred upper bound as the result in some cases is warranted. The upper bound is chosen where the lower bound is `null`. For example, `someMethod(Arrays.asList())` passes a `List<Object>` to `someMethod`. Our suggested choice of the *lower* bound in all cases is better in some situations, is not obviously *worse* in general, and has the benefit of consistency. In the `someMethod` example, it's likely that a `List<null>` would be more useful than a `List<Object>`, assuming the method signature is something like `void someMethod(List<? extends Number> l)`.

A.3 Confusing Content

- The distinction between a type variable bound involving multiple types (4.4) and an intersection type (4.9) is unclear and unmotivated. The assertion is made that "it is not possible to write an intersection type directly as part of a program; no syntax supports this," but the type variable bound *does* provide a syntax to describe the very same notion, even if it is not defined as such.

- The notion of *containment* (4.5.1.1) would be better described as *direct containment*, with *containment* being the reflexive, transitive closure of this relation. It is only used in the definition of *direct supertype*, which is closed to produce the *supertype* relation (4.10), so, strictly speaking, it is correct as it stands.

However, the name is confusingly inconsistent with the definition. As demonstrated here, this concept is not essential in the definition of subtyping anyway (it can instead be expressed concisely in the subtype definition), and could be removed entirely.

- $S <: T$ is defined to mean $T :> S \vee S = \texttt{null}$ (4.10). Since the definition of $:>$ includes `null`, the additional check that S is null is redundant.

- The descriptions of parameterized types (4.5) and subtyping (4.10) are inconsistent in their use of metavariables. T, for example, represents a *type* in the subtyping section, while representing a *type argument* (possibly a wildcard) in the parameterized types section. This makes the presentation very difficult to decipher. It's also not clear whether class types with no parameters, including raw types, are to be considered degenerate parameterized types (the definition seems to imply that they are not, while the subtyping section, to be correct, must assume that they are).

- The presentation of the type argument inference algorithm (15.12.2.7) is difficult to follow and appears more complicated than necessary due to the excessive use of "discussion" blocks explaining the logic behind the algorithm. These discussions might be less distracting if presented separately from the main body of the algorithm.

- The discussion overview of type argument inference (15.12.2.7) confusingly merges the concepts of *lub* and *intersection*. It even suggests that $lub(\texttt{I}, \texttt{J}) =$ `I & J`, which is simply incorrect. The use of the word *intersect* in this context is unfortunate, since it suggests the incorrect use of intersection types and the *glb* function.

- The type argument inference algorithm (15.12.2.7) performs boxing, suggesting that recursive invocations of the algorithm could lead to incorrect boxing (for example, `int[]` $<<$ `T[]` could produce `T` $:>$ `Integer`). Such incorrect results are prevented by never invoking the algorithm recursively with a primitive type, but this invariant is never stated, and discovering it requires careful reading (the array rule requires that the element type be non-primitive). A simpler presentation would separate issues related to boxing from the core algorithm.

- The handling of inferred equality constraints in the type argument inference algorithm (15.12.2.7) is far more general than necessary: the algorithm allows arbitrary equivalences to be established between parameters, while the process that *produces* these constraints follows a discipline guaranteeing that, given constraint $S = T$, S is a parameter to be inferred and T is a type that is valid in the calling context. There is no need to equate two different inference parameters and perform a unifying substitution on all other constraints.

- The specification of *lub* (a.k.a. *join*, 15.12.2.7) is far removed from the specification of subtyping (4.10) and relies heavily on raw types, which were intended to be merely a bridge with legacy code. This gives rise to bugs in the JLS, and makes the definition difficult to follow. Implementations are certainly free to define their algorithms in terms of sets of erased types, but a much clearer presentation would follow the definition of subtyping as closely as possible and describe the function recursively.

Appendix B
Summary of `javac` Bugs

In the process of experimenting with the Sun implementation of a Java compiler, `javac`, we have noted the following bugs. In addition, there are a variety of inconsistencies between the compiler and the specification related to the specification bugs above (the compiler may, for example, implement something *correctly* but inconsistently due to problems in the specification).

B.1 Present in Versions 5.0 and 6.0

- Capture is not used to statically check the bounds of of a wildcard-parameterized type:

```
class ABClass<A, B extends A> {}
// This instantiation should be illegal:
ABClass<? extends String, String> ab2;
```

- The inference algorithm produces unsound results and does not check their accuracy in some cases. The following program, for example, is compiled by `javac` without warning, but throws a `ClassCastException` at run time.

```
<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);
}
```

- Infinite types are not produced by *join*, as the following code demonstrates:

```
class Box<T> {}
class SelfBox1 extends Box<SelfBox1> {}
class SelfBox2 extends Box<SelfBox2> {}
// This should be legal:
Box<? extends Box<? extends Box<?>>> b2 =
  true ? new SelfBox1() : new SelfBox2();
```

B.2 Fixed in Version 6.0

- Capture is not implemented correctly, ignoring the wildcard's bound when there is a bound on the parameter:

```
class NumBox<T extends Number> {
  public T get() { return null; }
}
NumBox<? extends Cloneable> b = null;
// This should be legal:
Cloneable c = b.get();
```

- Unboxing conversion cannot handle *subtypes* of the wrapper types, such as bounded variables, at compile-time; an internal error occurs (the specification is not extremely clear on this, but regardless of the interpretation, the compiler should not unexpectedly fail):

```
<T extends Integer> void method(T arg) {
  // This should be legal:
  int x = arg + 3;
}
```

Bibliography

[1] Gilad Bracha, Martin Odersky, David Stoutamire, & Philip Wadler. *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language.* OOPSLA, 1998.

[2] James Gosling, Bill Joy, Guy Steele, & Gilad Bracha. *The Java Language Specification, Third Edition.* 2005.

[3] Atshushi Igarashi, Benjamin Pierce, & Philip Wadler. *Featherweight Java: A Minimal Core Calculus for Java and GJ.* OOPSLA, 1999.

[4] Atsushi Igarashi & Mirko Viroli. *On Variance-Based Subtyping for Parameteric Types.* ECOOP, 2002.

[5] Atsushi Igarashi & Hideshi Nagira. *Union Types for Object-Oriented Programming.* Journal of Object Technology, vol. 6, no. 2, February 2007.

[6] Andrew J. Kennedy & Benjamin C. Pierce. *On Decidability of Nominal Subtyping with Variance.* FOOL/WOOD, 2007.

[7] Kresten Krab Thorup & Mads Torgersen. *Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes.* Lecture Notes in Computer Science, 1999.

[8] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, & Neal Gafter. *Adding Wildcards to the Java Programming Language.* 2004.

[9] Mads Torgersen, Erik Ernst, & Christian Plesner Hansen. *Wild FJ.* FOOL, 2005.

[10] DrJava IDE. `http://drjava.org`.

[11] "Type variables should have lower/super bounds." Java Request for Enhancement. `http://bugs.sun.com/view_bug.do?bug_id=5052956`.

[12] "Please introduce a name for the 'null' type." Java Request for Enhancement. `http://bugs.sun.com/view_bug.do?bug_id=5060259`.

[13] "Multiply-bounded reference type expressions." Java Request for Enhancement. `http://bugs.sun.com/view_bug.do?bug_id=6350706`.