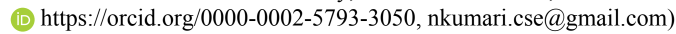
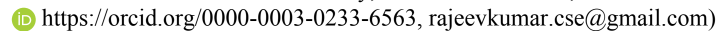
Journal of Universal Computer Science, vol. 31, no. 4 (2025), 312-337
submitted: 9/5/2023, accepted: 9/11/2024, appeared: 28/3/2025 *CC BY 4.0*

A Java Compiler Plugin for Type-Safe Inferences in Generics

Neha Kumari
(School of Computer and Systems Sciences
Jawaharlal Nehru University, New Delhi 110 067, India
 https://orcid.org/0000-0002-5793-3050, nkumari.cse@gmail.com)

Rajeev Kumar
(School of Computer and Systems Sciences
Jawaharlal Nehru University, New Delhi 110 067, India
 https://orcid.org/0000-0003-0233-6563, rajeevkumar.cse@gmail.com)

Abstract: The two most significant yet complex elements of Java generics are wildcards and type argument inference. Both processes rely on the compiler. Even though type argument inference and wildcard execution are implicit processes, a programmer should be aware of them to make the most of the features. A compiler error message tells much about the code and the process mechanism. If the error message is unambiguous and sound, it is easy for the programmer to debug the code. However, in the context of wildcard-type argument inference, the current Javac compiler emits cryptic and imprecise error messages. A programmer may get confused about the inference outcome and failure, so it will be difficult to resolve the errors easily. In this paper, we propose a few additions to the current Wildcard-based type inference algorithm to get detailed and valuable error messages. We implement a Java compiler plugin tool based on the proposed algorithm. The plugin can be easily executed through the Java command line. It gives a comprehensive error message that aids programmers in resolving errors more effectively.

Keywords: Java Generics, Plugin, Type Argument Inference, Type Safety, Wildcards.
Categories: D.1.m, D.2.4, D.2.5, D.3.3
DOI: 10.3897/jucs.106159

1 Introduction

Type safety is crucial for any typed language to ensure the code does not go wrong. In Java, Generics has been added to ensure such type safety by eliminating explicit cast. Java generics is based on the type erasure principle. The effect of type erasure on generics is its invariant behavior. A List<Integer> can not be a subtype of List<Number>. The subtyping limitation has been resolved by including wildcards as the generic parameter. Wildcards enable the co and contra-variant nature of generic parameters. For example, List<Integer> can be a subtype of List<? extends Number>. However, wildcards do not represent a concrete type but represent a bound. The Javac compiler performs a capture conversion process to substitute a valid concrete type instead of wildcard bounds. The other crucial work that the Javac compiler performs to ease Java generics is type argument inference. Type argument inference helps to avoid boilerplate code. Type argument inference is mainly discussed among researchers for its unsoundness [Tate et al. 2011, Amin and Tate 2016, Bierhoff 2022]. The major reasons for the unsoundness

in the type argument inference algorithm highlighted in literature are the subtyping rules and the expansive inheritance path [Kennedy and Pierce 2007]. Even the Java Language Specification (JLS) clearly states that it does not guarantee these properties. Also, a few studies show that there is negligible impact of these vulnerabilities on real Java projects [Greenman et al. 2014, Bierhoff 2022]. However, type inference in Java generics provides a choice for flexible programming. The type inference algorithm should be viewed as a heuristic designed to perform well in practice[1]. The focus is on ensuring that the programmer comprehends the nuanced type inference mechanism, allowing them to utilize it effectively and reap the advantages of this mechanical process. A good understanding of the type inference process reduces more mishaps than fixing the existing unsound type inference algorithm. Here, understanding the inference process does not imply the compiler's internals but the valid usage of type inferences. For example, assigning a null value to contravariant is not encouraged [Amin and Tate 2016]. The existing Javac compiler fails to indicate errors due to null assignment in such cases, prompting wrong error messages. A detailed compile-time error message that follows the type error diagnosis manifesto may contribute significantly to understanding the type argument inference in Java generics [Yang et al. 2000].

In Java, the capture conversion of wildcards during type inference and the target type assertion are not easily understandable. The Java Language Specification (JLS) describes the resolution of wildcard-based type inference more subtly. Moreover, the cryptic and incomplete error messages associated with the type inference process cause lots of confusion among programmers. The common error messages we get are "incompatible type", "inferred type does not conform to the upper bound(s)," and "`CAP#N` cannot be converted." Error messages show how the Javac compiler resolves inference constraints to infer the type. The vagueness of error messages represents the incompleteness of the type inference algorithm. The sample code and compiler error message in Listing 1 shows one such situation.

```
 1 public class Temp<T extends Number> {
 2  <T extends Number>
 3     List<T> method1( List<? super T> t1, T t2)   {...}
 4     List<T> method2( List<? super T> t1, T t2)   {...}
 5  <T extends Number>
 6     T method3(List<? super T> t1, T t2)           {...}
 7
 8 public static void main(String ar[]) {
 9    Temp<Number> obj=new Temp<>(90.0);
10    List<Number> nlist=new ArrayList<>();
11     nlist .add(78);
12    List<Double> dlist1=obj.method1(nlist,15.9); //ok
13    List<Double> dlist2= obj.method2(nlist,15.9); //error
14    Double temp=obj.method3(nlist,67.8); //ok
15    }
16    }
17 ============Compile Time Error Message=============
18  Temp.java:80: error: incompatible types: List<Number> cannot be
     converted to List<Double> ::List<Double> dlist2= obj.method2(nlist,15.9);
```

Listing 1: Method invocation with wildcards

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.2.2

Listing 1 has three method invocation expressions. Two invocation expressions are valid, and one is invalid Java code. However, all three invocation expressions seem similar. We analyze the compile time error message to understand why the second expression is invalid. The error message states an incompatible type mismatch between List<Number> and List<Double>. If a naive programmer looks back at the code at Line 13, he can straightly say that the incompatibility is between the type argument nlist and the target type variable dlist2. However, this is not all true. Line numbers 12 and 13 have similar type arguments with similar method invocation expressions. The only difference between both expressions is that the type parameter of method1() is locally constrained, and the type parameter of method2() is dependent on the class type parameter. However, the type parameter constraint is similar in all cases: <T extends Number>. The type parameter of class instance obj is instantiated as the type Number, and it expects the invocation type of method2() to be the same. The inferred invocation type that depends upon the target type dlist2 is List<Double>. Therefore, the error message should indicate the incompatibility between the instance type obj and the target type dlist2. The Javac compiler error message is based on the available constraints during the resolution phase of type inference. The Javac compiler first forms a bound set containing constraint formulas for the method arguments (List<Number> <: List<α> and α <: Double). Next, the bound set adds new constraints in the form of target type (List<α> = List<Double>). During the assertion of the target type constraint and the method argument constraint, the compiler gets incompatible bounds for the inference variable α, resulting in a compile-time error. Here, the compiler lacks other details, such as the scope of the type variable to be inferred and its implicit constraint. A naive programmer tries to resolve the error by seeing the compiler error message and the invocation expressions; he may get confused. Although the first and third invocation expressions have incompatible target and argument types similar to the second, these are valid expressions. Therefore, the error message should be more descriptive.

Wildcard-based type argument inference is mainly discussed among the researchers for the unsoundness [Tate et al. 2011, Amin and Tate 2016, Bierhoff 2022]. The major reasons for the unsoundness in the type argument inference algorithm highlighted in literature are the subtyping rules and the expansive inheritance path [Kennedy and Pierce 2007]. For example, assigning null values to a contravariant type may lead to an infinite inheritance relation. However, the JLS states that it does not guarantee subtyping relations in case of contravariant. Based on the literature and JLS statement, it can be said that the usage abnormalities of wildcard-based type inference cause unsoundness in the type system. Several pieces of literature suggested restricted usage of wildcards too [Greenman et al. 2014, Bierhoff 2022]. However, the limited use of wildcards also does not guarantee soundness [Amin and Tate 2016]. Understanding wildcard-based type systems is complex. There are several cases where the cause of type unsoundness due to wildcard-based type inference goes undetected [Amin and Tate 2016]. In such a situation, the compiler-generated error messages have a pivot role that helps to understand and resolve the exact error easily. The current Javac compiler emits cryptic and imprecise error messages in the case of wildcard-based type inference. A detailed compile-time error message may contribute significantly to understanding the type argument inference in Java generics.

The main contribution of the paper is the proposed enhancement to the current type inference algorithm. The algorithm focuses on providing a detailed error message in case of wildcard-based type inference. This helps to understand the complex type inference algorithm and saves programmers from performing unsound execution of code. A detailed error message helps novice programmers to resolve the error efficiently [McCall 2016].

2 Foundations and Related Work

The type inference in OOPLs is mainly influenced by the algorithms proposed by Damas-Milner and Palsberg-Schwartzbach [Damas and Milner 1982, Palsberg and Schwartzbach 1991]. The Damas-Milner algorithm focuses on inferring principal types, and the other emphasizes precise types. The main difference is that the principal type relies only on the context as an input to determine the type. In contrast, Palsberg-Schwartzbach's algorithm finds the type based on three kinds of information: local constraint, connecting constraint, and the condition attached to the expression. Both influence Java-type inference.

2.1 Terminology

In this section, we introduce the terminology we use throughout this paper.

Target Type

Every expression in Java has either void or some type deduced at compile time. The same expression can have different types in different contexts. The expression must be compatible with the expected type in that context; the expected type is called the target type[2]. For example, in the chain method call, the later method's target type is the previous method's return type. With Java 8, the target-type assertion became a crucial component for the verification of type-argument inference. Earlier, the argument type was inferred based on method invocation and assignment context only. The verification does not match the final expected type. The following example illustrates the impact of target-type assertions.

```
1  // With JDK 7
2  List<? super Integer> list1= new ArrayList<>("string") //error
3  // With JDK 8
4   List<? super Integer> list2= new ArrayList<>("string") // no error
```

Listing 2: Target-type assertion

In Listing 2, it is clear that the earlier inference does not consider a covariant form of Integer (Object type) as an argument of List. Therefore, it emits an unnecessary error message, whereas Java 8 allows the above inference since String is a subtype of Object.

Concrete Type

In this paper, we use the term concrete for parameterized type, which has a valid type parameter during implementation. For example, the parameter of `List<? extends Integer>` generic type does not represent a concrete type but represents a bound that can be substituted with any subtype of Integer. However, the parameter of `List<Integer>` generic type has a concrete type parameter. The concrete parameter, `Integer`, does not require further resolution.

[2] http://cr.openjdk.java.net/ dlsmith/jsr335/jsr335-0.6.2/D.html

Inference Contexts

- Assignment: In this context, the compiler infers the type of argument by directly substituting what is available on the left side of the assignment operator as a named variable.
- Method invocation: In this context, the compiler looks for available method information such as parameter type, return type, and the target type of method invocation expression.

Wildcard Capture

A simple wildcard as a parameter does not specify a well-formed generic type. It either represents a bound `(? extends, ? super)` or no bound `(?)` for the generic type. A wildcard is not considered a type until it converges into a concrete type. The Javac compiler follows the capture process to converge wildcard bound into a concrete type[3]. The Java compiler implicitly performs the conversion. The compiler first replaces the wildcard with a fresh type variable, and next, it validates the constraints associated with the type variable. The substituted type variable can be constrained explicitly or implicitly according to the declaration of wildcards. It has been observed that unsoundness caused by wildcards is primarily due to implicit constraints [Amin and Tate 2016, Bierhoff 2022]. Lastly, the valid concrete type is substituted in place of the type variable.

Inference Phases

- Reduction: forms the constraint formulas using actual and formal type arguments. Constraint formulas are represented as α `-> Iterable<?>,` β `-> Object`. The formulas are later reduced to a bound set that contains all possible bounds for the inference variable (α `<: Iterable<CAP#1>,` β `<: Object`).
- Incorporation: adds new bounds to the bound set by reducing existing bounds in several iterations.
- Resolution: examines the bounds and instantiates the inference variable. The inference errors are generated at this time.

2.2 Related Work

Wildcards are flexible and easy to implement but suffer from usage abnormalities. Several research monograms mention wildcard abnormalities if used in type argument inference context [Smith and Cartwright 2008, Tate et al. 2011, Amin and Tate 2016]. For example, the compiler lacks unification of the same wildcard type variable and implicitly replaces each wildcard type with a separate type variable (namely `CAP#1...CAP#N`). This results in invalid constraint formation during type argument inference [Stadelmeier and Plümicke 2023]. Several solutions or restrictions have been proposed to mitigate the unsoundness caused by such cases. One way to prevent ambiguous type inference would be to permit type-argument inference only when the most precise return type can be inferred [Tate et al. 2011]. Unification of use and declaration-site variance have also been suggested to overcome wildcard complexities such as capture conversion [Altidor et al. 2012, Tate

[3] https://docs.oracle.com/javase/specs/jls/se18/html/jls-5.1.10.html

2013]. An interval type-influenced model has been proposed to generalize and simplify wildcards for type argument inference and other associated tasks [AbdelGawad 2018]. Amin and Tate informally suggest null exclusion during implicit wildcards [Amin and Tate 2016]. [Bierhoff 2022] suggested witness type protection to avoid unsoundness due to null assignment to the contravariant.

A study on compiler bugs finds that parametric polymorphism (generics) and type argument inference are some of the most bug-triggering language features among JVM compilers. The study highlighted the importance of correct and meaningful error messages in debugging such complex bugs [Chaliasos et al. 2021]. Programmers often encounter cryptic compiler error messages that are difficult to understand and thus difficult to resolve [Traver 2010]. Diagnosing compiler error messages helps to understand the abstraction layer among complex programming constructs[4]. Many of the complex programming features are common in contemporary languages. The novice programmer does not successfully grasp these features, resulting in the wrong implementation. Furthermore, the compiler-generated cryptic messages make it more challenging to fix errors [Luoma et al. 2007]. One common feature is generic programming. For example, C++ incorporates generic templates. The template libraries have a certain set of usage requirements that the C++ compiler misses; therefore, it reports dubious errors as mentioned in Listing 3.

```
 1 #include <bits/stdc++.h>
 2 using namespace std;
 3 int main() {
 4    list <int> alist {12,45,8,6};
 5   sort ( alist .begin() ,  alist .end()); }
 6
 7  ===========G++ Compiler Error
    Message================
 8  In  file  included  from /usr/include/c++/11/algorithm:62,
 9                 from
    /usr/include/x86_64−linux−gnu/c++/11/bits/stdc++.h:65,
10                 from main.cpp:11:
11 /usr/include/c++/11/bits/stl_algo.h:1955:50:  error :  no match for
    'operator'− (operand types are 'std :: ' _List_iterator  and
    'std :: ' _List_iterator )
```

Listing 3: Erroneous template code [Becker et al. 2019]

In the C++ family of compilers, the G++ compiler emits an error with a huge message. However, it does not correctly report a simple statement that the code failure is due to requirement incompleteness; `list` does not support the subtraction operation performed by `sort()`. STLFilt was a tool specifically designed to diagnose STL-related errors and warning messages. This tool eliminates unnecessary long messages, as shown in Listing 3, and includes only the template part. However, it does not rewrite a new descriptive message, and the tool is no longer maintained [Hansen et al. 2007]. The enhancement is intended to add in the latest version of C++20 in the form of "concepts" [Gregor et al. 2006]. Concepts are an extension of the templates to enhance generics in C++. It adds enhanced constraint checking for template instantiation and aims to provide a more

[4] https://www.salon.com/1999/09/16/knuth/

readable error message[5]. A similar tool, clang[6] is claimed to give expressive diagnostics for C++ using regular expression rules.

Developing a sound generic-type argument inference algorithm is a challenging task. Moreover, the developer and debugger often face difficulties correctly interpreting inference-based error messages. A conventional way to improve type error messages related to inference is to change how constraints are resolved [Boustani and Hage 2009]. The extension of the previous work includes providing a hint with a type error message to fix the type errors [Boustani and Hage 2010]. [Pavlinovic 2019] proposes an enhancement in liquid type inference that combines classical typing with static analysis to get precise details of type inference that automatically generates meaningful type error messages. A robust program is one that is able to detect and recover from various faults [Fetzer and Felber 2007, Medeiros et al. 2014]. StaDyn is a good example of a hybrid typing language for the .NET platform (similar to the Java platform) that utilizes compile-time type information and dynamically typed code to improve compile-time error detection and runtime performance. The StaDyn compiler collects dynamically typed information, which is used to detect type errors at compilation time [Ortin et al. 2022].

Java 8 enables type-level annotations through `TYPE_USE` and `TYPE_PARAMETER` elements. Several type-based annotation plugins are being used and developed to enhance generic type checking [Papi et al. 2008, Brotherston et al. 2017]. The annotations help the programmer to point out exact program elements. If the annotations help the programmer to point out exact program elements, the plugin will only work for that specific class parameter. In this paper, the proposed plugin uses Java annotation for the same purpose.

3 Dubious Type Argument Inference

Initially, the inference algorithm had limited scope and was more prone to unsound inference, resulting in misleading error messages. The algorithm is primarily improved in JDK 8 and has enhanced type safety. It follows the target type assertion to validate type inference. However, the algorithm behaves uncertainly in some cases of wildcard parameterization. Amin and Tate [Amin and Tate 2016] have shown several unsound code examples that JDK 8 allows. The non-deterministic type checking of wildcards behaves differently for similar constrained code examples, as shown in Listing 6. In such cases, the Javac compiler sometimes emits errors and sometimes does not [Bierhoff 2022]. Also, these errors lack significant error messages; therefore, it is hard to understand the actual reason for code unsoundness. According to the JLS, resolving a simple inference variable depends on a few types of inference contexts. However, the type inference in the presence of a wildcard requires more processes to be resolved. For example, find the least upper bound (lub) if the inference variable has one or more lower bounds. The process of finding lub itself is subtle and criticized for its ill-defined definitions on many occasions[7]. Listing 4 highlights such a dubious statement of the JLS.

```
1  public static <T> List<? super T> method1(List<? super T> t,T t1,
   Class<T> cls){   return t; }
2  int in=90;
3  List<? super Integer> list1= method1(new ArrayList<>(),in,
   Integer.class); //case 1
```

[5] https://www.iso.org/standard/64031.html
[6] https://clang.llvm.org/cxx_status.html
[7] https://bugs.java.com/bugdatabase/view_bug?bug_id=8314380

```
4  List<? super Integer> list2= method1(new ArrayList<>(),in,
   Number.class); //case 2
5  List<? super Integer> list3= method1(new ArrayList<>(),in,
   Object.class); //case 3
```

Listing 4: Multi-type inference

In Listing 4, the target type is an example of a contravariant type with a lower bound: `Integer`. This lower bounded parameterized type is valid for all its supertypes, which vary according to other inference bounds. The `method1()` invocation expression (case 1) generates four constraint formulas based on subtyping and type equality rule: `ArrayList<`α`> -> ArrayList<CAP#1>, int -> `α`, Class<`α`> -> Integer, List<`α`> -> List<CAP#2>`. The formulas further reduce to bounds containing three method arguments and one target type respectively: α `:> Object,` α `:> Integer,` α `= Integer,` α `:> Integer`. During the resolution phase, all bounds are resolved together to instantiate the inference variable. In this example, the bound set contains two bounds of wildcard capture form and one lower bounded inference variable. The inference instantiation requires resolving the lub of these three bounds. As per the available bounds, the rule applied to get lub is `lcta(U, ? super V) = ? super glb(U, V)`. However, the mentioned lub rule about getting supertype of greatest lower bound (glb)[8] is unclear in case of incompatible types resulting a dubious error messages[9]. In this case, `<? super glb(Object, Integer)>` yields `Integer`. The three cases of Listing 4 have a common lub as `Integer`. The only difference among these cases is the type equality constraint: α `= Integer,` α `= Number, and` α `= Object` respectively. Therefore, the inference variable `<T>` can be inferred as an `Integer`, `Number`, or `Object`. If the third argument of method invocation expression that forms equality constraint is removed and the second argument, which is an integer primitive type, is replaced with a String value, then there should be an incompatible glb (`<? super glb(Object, Integer, String)>`). However, there is no error, and the inferred type is assumed to be an Object type as follows:

```
1  List<? super Integer> list4= method1(new ArrayList<>(),"str"); //no
   error
2  List<? super Integer> list1= method1(new ArrayList<>(),"str",
   Integer.class); //error
3 =========Compile-Time Error Message=================
4 error: method method2 in class Example cannot be applied to given types;
5 List<? super Integer> plist1 =
   method2(newArrayList<>),"str",Integer.class);
6
7   required:              List<? super T#1>,T#1,Class<T#1>
8   found:               ArrayList<Object>,String,Class<Integer>
9   reason:                  inference variable T#1 has incompatible bounds
10  equality constraints:     Integer
11  lower bounds:             String
```

Listing 5: Multi-type inference error

[8] https://docs.oracle.com/javase/specs/jls/se17/html/jls-5.html#jls-5.1.10
[9] https://bugs.java.com/bugdatabase/view_bug?bug_id=8071963

The error is generated when the third argument is added, which forms an equality constraint. However, the error message in Listing 5 highlights the incompatibility reason due to the equality constraint and lower bound. Still, it is hard to assume a primitive type as a lower bound for a programmer until he is aware of the dubious lub process. In this section, we highlighted three flaws of wildcard execution during the type inference process. The following are the three flaws.

- Null assignment to contravariant bound.
- Capture unification.
- Capture incompleteness in nested classes.

The type argument inference algorithm additionally deals with wildcard bounds at the incorporation phase. In this phase, new bound sets are created for the wildcard bounds since these bounds contain further constraints to be generated as new bounds for inference. The compiler reduces the wildcard bound into new bounds until it reaches a fixed point.

3.1 Null Assignment

The JDK 7 prohibits null literal as an actual argument for the type parameter inference[10]. However, the later version of Java does not clarify the usage of null literals. Especially the use of null with contravariants has been observed as unsound in many works of literature [Kennedy and Pierce 2007, Smith and Cartwright 2008, Amin and Tate 2016]. According to JLS, if there is a subtyping constraint of the form `S <: T` and `T` is null, then the constraint reduces to a false bound, but this statement is not followed by the Javac compiler[11] in the case of implicit constraint. The following code example we have listed is from [Amin and Tate 2016] and also reproduced in [Bierhoff 2022]. The earlier literature claimed the unsoundness in the following code example was due to the presence of an implicit null constraint. The latter claimed that the unsoundness was rooted in its subtyping rule. However, if we follow the compiler error message (before JDK 17, this code was accepted by Javac compilers), we get the interpretation that the inference fails due to incompatible bounds and not due to null assignment to the wildcard type. The error message is an example of a dubious statement: it states that incompatible bounds are due to a mismatch of lower bounds of type variable `J`. However, the language specification has no concept of lower bounds for type variables. The type variable can have an upper bound and only if it is explicitly defined[12].

```
1 public class Check {
2 static class MultiPara<I, J extends I> {}
3 static class SingleP<I> {
4 <J extends I>
5   I m1(MultiPara<I,J> arg, J j) {
6     return j;
7     }    }
8   static <X,Y> Y m2(X x) {
9       MultiPara<Y,? super X> constrain = null;
```

[10] https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.2.7
[11] https://bugs.java.com/bugdatabase/view_bug?bug_id=8314382
[12] https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.4

```
10        SingleP<Y> obj = new SingleP<Y>();
11        return obj.m1(constrain, x);               //error
12     }
13 public static void main(String[] args) {
14 String str = Check.<Integer,String>m2(0);   }}
15
16 =========Compile-Time Error Message==============
17 Check.java:12: error: method m1 in class SingleP<I> cannot be applied to
     given types;
18   return:          SingleP.m1(constrain, x);
19   required:        MultiPara<Y,J>,J
20   found:           MultiPara<Y,CAP#1>,X
21   reason:          inference variable J has incompatible bounds
22   lower bounds:      Y
23   lower bounds:      X
24   where Y,X,J, and I are type variables:
25   Y extends Object declared in method <X,Y>coerce(X)
26   X extends Object declared in method <X,Y>coerce(X)
27   J extends Y declared in method <J>m1(MultiPara<I,J>,J)
28   I extend the Object declared in class SingleP
29   Where CAP#1 is a fresh type-variable:
30     CAP#1 extends Y super: X from capture of ? super T
31   1 error
```

Listing 6: Unsound Java program [Amin and Tate 2016]

Moreover, the JLS states that if there is a constraint formula of the form `S <: T` and T is the null type, the constraint reduces to false. Since the constraint reduces to false, there should be no resolution, and the type inference should be terminated with an error message indicating improper use of null. As shown in Line 8, null is assigned to a contravariant type, and this contravariant is passed as an argument to method `m1()`. The initial constraint formulation for this method argument concerning method type parameter is `MultiPara<Y, CAP#1> -> MultiPara<`α`,` β`>` that further reduces to `Y ->` α`, CAP#1 ->` β and β `<:` α. This shows how null is skipped during constraint formation, and hence, an error message is missed to highlight null unsoundness. In Java, the null type is considered a subtype of all reference types[13]. Listing 6 uses a null literal that bypasses the constraint to validate the implicit type bound of wildcard argument in `MultiPara<Y,? super X>` instance variable.

3.2 Unconfirmed Lower Bound

The sample code on Listing 7 has been taken from the Java bug database[14]. The error message shows incompatible bounds due to a type mismatch between different scoped variables. As mentioned in the bug description, it is claimed that the code should compile successfully. Therefore, when calling the `m2()` method, the Javac should try to unify `<X extends I<CAP#1>` and `<Y extends I<T#1>` and derive it as `T#1 = CAP#1`. However, such inference behavior is unspecified in JLS leading to dubious error messages as follows:

[13] https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.10.2
[14] https://bugs.java.com/bugdatabase/view_bug?bug_id=8251891

```
 1 public class Bug {
 2    interface I<T> {}
 3    interface J<T> {}
 4 static <X extends I<?>> void m1(J<X> vs) {
 5       m2(vs);      }
 6  static <Y extends I<T>, T> void m2(J<Y> vs) { . . . }   }
 7
 8 ============Compile−Time Error Message===========
 9 Bug.java:5: error : method m2 in class Bug cannot be applied to given
     types;
10       m2(vs);
11  required :            J<Y>
12  found:                J<X>
13  reason:               inference  variable  Y has incompatible  bounds
14  equality  constraints : X
15  lower bounds:        I<T>
16  where Y,T,X are type−variables:
17   Y extends I<T> declared in method <Y,T>m2(J<Y>)
18   T extends Object declared in method <Y,T>m2(J<Y>)
19   X extends I<?> declared in method <X>m1(J<X>)
20 1 error
```

Listing 7: Upper bound containing wildcard

The actual type argument used to call method `m2()` is an upper bound with an unbounded wildcard, and the type parameter is an upper bound with a type variable `T`. According to the JLS, the initial constraint formula for such arguments is `J<X> <= J<Y>, X = Y, X <: I<CAP#1>, Y <: I<T>`. The mismatch bounds between `X` and `Y` reduce to a false bound, resulting in no further resolution and compile time error as shown in Listing 7. However, resolving `CAP#1` and `T` results in a common type `Object`.

3.3 Inner Class

According to the JLS, a wildcard cannot be used as a type argument for a supertype (class) or instance creation expression. However, this is not validated in the case of nested generic classes; the Javac compiler allows the use of wildcards in an enclosing class instance expression.

```
 1  public class CaptureInSuperClass {
 2    static class Box<A> {
 3     A a;
 4     Box(A a) { this.a = a; }
 5    class Inner {
 6     void setA(A newA) {
 7       a = newA;      }
 8     A getA() {
 9        return a; }  }  }
10    static Object prev;
11  public static void main(String[] args) {
```

```
12     Box<String> stringBox = new Box<>("1");
13     for (Box<?> box : List.of(stringBox, new Box<>(1))) {
14       box.new Inner() {{    // Erroneous anonymous class declaration
     expression
15           if (prev == null) prev = this;
16           else this.getClass().cast(prev).setA(this.getA());
17       }};       }
18     String s = stringBox.a;   // ClassCastException
19     //GenericSignatureFormatError
20     ((Box<?>)stringBox).new Inner(){}.getClass().getGenericSuperclass();
21   }}
22
23 ===========Compile—Time Error Message===========
24 error: error while generating class <anonymous CaptureInSuperClass$1>
25  ( illegal signature attribute for type CAP#1)
26  where CAP#1 is a fresh type variable:
27     CAP#1 extends Object from capture of ?
28 1 error
```

Listing 8: Inference with inner class

The code on Listing 8 shows the failure of type validation for the anonymous inner class type variable, which is a constraint with the outer class type variable[15]. JDK 17 (onwards) emits an error at compile time, but it's not descriptive. The compiler error message shows one error. However, there are two errors. The first error is at Line 18, where the code has incompatible type access; therefore, it should be `ClassCastException`. The second error is at Line 20, where the anonymous class declaration expression contains no valid constructor argument. According to the JLS, the superclass or the supertype of an anonymous class declaration should not have such type variable (`CAP#1`) that was not declared as a type parameter[16]. Earlier in JDK 10, there was no such rule. Therefore, the Javac compiler accepts the unbounded wildcard as a constructor argument for an anonymous class that later results in heap pollution. However, the current Javac compiler detects the type violation shown in Listing 8. It emits an error but the error message does not specify that the illegal signature format is due to the presence of an unbounded wildcard in a superclass type argument.

4 Methodology

The type-inference algorithm is primarily enhanced in JDK 8. It added a target type assertion to assert type compatibility. Figure 1 clearly shows the difference in the type inference process from the earlier version. The previous algorithms included method-type arguments to form constraints and perform intersection among bound sets for type instantiation. The enhanced algorithm has more constraints and contexts for type resolution. The recursive process between the reduction and incorporation phases includes a maximum element in the bound set, which helps instantiate the accurate type. There is a need to improve the type validation during the inference process to resolve the issues mentioned in Section 3.

[15] https://bugs.java.com/bugdatabase/view_bug?bug_id=8203335
[16] https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html#jls-15.9.3

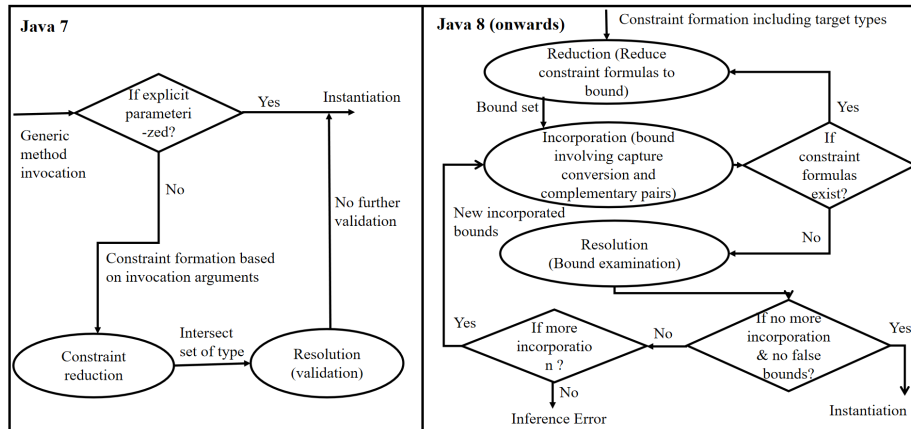


Figure 1: Comparative illustration of Java 7 vs. Java 8 (onwards)

Before Java 8, type validation occurs in the later phase of the algorithm. Therefore, the Javac compiler includes paltry information in error messages. Java 8 improved type checking and performed compatibility assertions at the reduction and incorporation phases. The compiler adds false bounds during reduction if there is any inconsistency. However, the compiler only emits an inference error message at the resolution phase. The detailed constraint information is not available at the resolution phase. A few pieces of information in general terminologies are available. For example, upper bounds, lower bounds, equality constraints, etc. If we analyze the inference error messages in Section 3, we can find it contains only these terminologies. Considering the cases discussed in Section 3, we propose enhancements to the current type inference algorithm.

4.1 The Proposed Algorithm

The proposed algorithm focuses on wildcard-dependent inference. Here, the wildcard-dependent inference implies the presence of a wildcard type as an inference component (type parameter or actual type argument). We aim to provide detailed, accurate error messages related to wildcard-dependent inference. For this, we propose adding the following three crucial components to the inference process, which are not given much attention in the current type-inference validation process:

- Scope of the inference variable as a constraint.
- Containment rule for type validation during resolution.
- No constraint formation for the null type argument.

The JDK 8 inference algorithm added the target type assertion for the validation and initialization of an inference variable. As mentioned in Section 2.1, the target type implies the expected type in that expression context. The current inference algorithm uses assignment and method invocation contexts to infer. The target type in the context of method invocation is the invocation type of method. JLS states that there are multiple rounds of reduction and resolution processes to find the type of inference variable. This

includes the generation of additional constraints for resolving wildcards. The target type compatibility assertion occurs only at the last round of the resolution and determines the real type or emits a failure message. However, there can be different target types at different rounds of inference resolution. For example, in the method invocation expression as shown in Listing 1 at Line 13, the initial target type for the `method2()` parameter is the class instance parameter type. Listing 9 is a part of Listing 1. The class parameter type in this expression is `Number`, and the final target type for the inference is `Double`.

```
1 //Temp is a class with parameter type Number
2     Temp<Number> obj=new Temp<>(90.0);
3 //Target type for following expression is Double.
4     List<Double> dlist2= obj.method2(nlist,15.9); //error
```

Listing 9: Target-type

`Double` is a subtype of `Number`. Therefore, both are compatible types, and the constraint is reduced to a valid bound. The next round of the resolution bound set contains the reduced type parameter bound but misses instance type parameter information (scope type). Therefore, the error message only mentions the incompatibility between the method invocation type and the method argument. We propose including type information for the respective inference variable scope as a separate constraint. The scope of an inference variable is significant in forming a valid constraint formula that helps to resolve an accurate type. If the scope is added as a constraint, it helps provide more precise inference information at compile-time. Section 6 justifies the scope inclusion with a detailed example.

Another issue with wildcard inference resolution is capture variable unification, as highlighted in Section 3.2. Unification is finding a substitution such that two terms become equal. A proper unification rule for type inference delivers a proper justification in cases of rule violation, and hence, the compiler may emit an elaborate error message. The containment rule for wildcards[17] is a suitable constraint for unification. According to the rule, the type variable `T1` is said to contain `T2 (T2 <= T1)` if the set of types denoted by `T2` is provably a subset of the set of types denoted by `T1`. The type variable `S` indicates the type parameter of the generic method, and `T` indicates the type argument.

```
1  ? extends T <= ? extends S if T <: S
2
3  ? extends T <= ?
4
5  ? super T <= ? super S if S <: T
6
7  ? super T <= ?
8
9  ? super T <= ? extends Object
10
11 T <= T
12
13 T <= ? extends T
14
15 T <= ? super T
```

[17] https://docs.oracle.com/javase/specs/jls/se17/html/jls-4.5.1.html

```
16
17  T <= ? (addition)
```

Listing 10: Type argument containment rules

We propose an addition to Listing 10 at Line 17 for unbounded wildcard containment. An unbounded wildcard represents an unknown type. However, Java treats it as an `Object` type if no explicit type is assigned. The upper bound of any type parameter `T` is also `Object`. This implies `T` is a subset of `<?>` and the relation `T <= ?` is a valid addition. Listing 6 shows how containment is violated. The constraint on the second parameter of the `MultiPara<>` class and the argument passed during the declaration of the class inside the method `m1()` don't match and violate the containment rule. Using the containment rule for the type argument unification may help to detect the mismatch early. This especially benefits in the case of implicitly constrained wildcards, as it does not require an additional incorporation process for failed containments. The proposed algorithm uses the listed containment rule to validate the constraints of different scoped type variables too. Figure 2 represents the proposed additions to the existing inference process. The

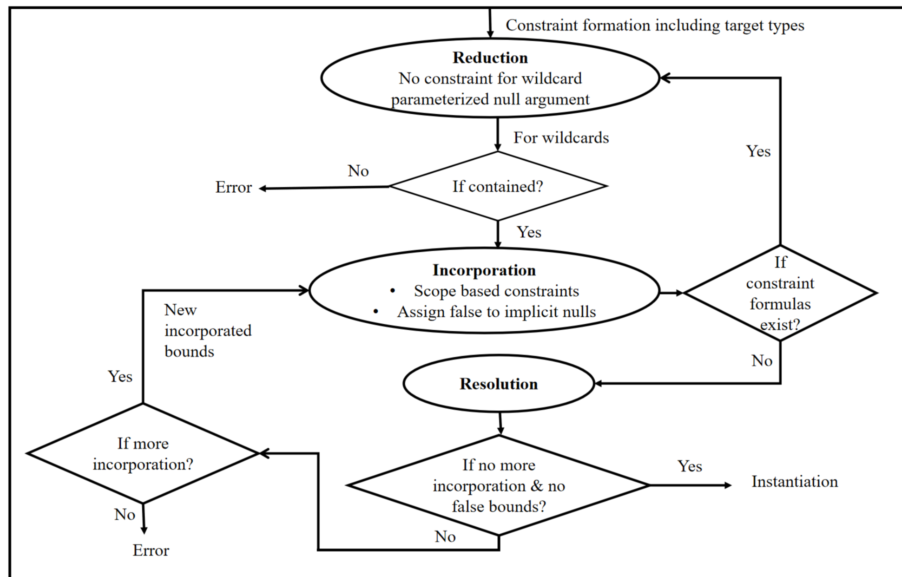


Figure 2: Schematic of the proposed algorithm

figure depicts three phases of the inference process, as mentioned in Section 2.1. At the reduction phase, we propose complete elimination for constraint formation if the actual type argument (explicit) to the wildcard parameter is null. Further, the constraint formation of wildcard arguments has to face a unification check before incorporation. Only the valid wildcard arguments will form constraint formulas at the incorporation phase. We propose the incorporation of scope-based constraints for each wildcard. This approach helps the compiler emit error messages with a precise cause of the error.

Algorithm 1 The Proposed Algorithm

```
 1: When any of the type parameter (P) or the actual type argument (A) is wildcard-type,
 2: function Reduction
 3:     while no expression or type do                          ▷ Constraint generation
 4:         if A is the type of null then
 5:             no constraint is implied on P.
 6:         end if                                              ▷ Constraint validation
 7:         if P contain A (A <= P ) then               ▷ Satisfies the containment rule
 8:             Incorporation()
 9:         else
10:             Error: type mismatch for inference.
11:             break
12:         end if
13:     end while
14: end function
15: function Incorporation
16:     while no bounds to incorporate do
17:         if bounds contain wildcards then
18:             perform capture conversion and add capture variable scope constraint
    bound                               ▷ Scope constraint: scope of the actual argument
19:             if (Cap#N is lower-bounded or unbounded) && (Cap#N == null) then
20:                 α -> false
21:                 break
22:             else
23:                 if (Cap#N is upper-bounded) && (Cap#N == null) then
24:                     no constraint is implied on Cap#N
25:                 end if
26:             end if
27:             if more constraint formulas ? then
28:                 Reduction()
29:             else
30:                 Resolution()
31:             end if
32:         end if
33:     end while
34: end function
35: function Resolution
36:     while no more bounds for instantiation do
37:         if all bounds of the bound set are TRUE and fully incorporated then
38:             initialize inference variable
39:         else
40:             if new incorporated bounds ? then
41:                 Incorporation()
42:             else
43:                 emits an inference failure error message
44:                 break
45:             end if
46:         end if
47:     end while
48: end function
```

The current algorithm has a different approach to inconsistency. In the current JDK algorithm, inconsistency during the reduction and incorporation phases is being validated recursively until the final bound set is prepared. The existing algorithm emits an error at the resolution phase, whereas the proposed algorithm emits an immediate error at the reduction phase for initial constraint violations. However, the proposed algorithm follows the existing approach of recursive bound-set formation for other cases. The Algorithm 1 contains a snippet of the existing inference process where the proposed additions are included. Our algorithm mainly proposes enhancements in constraint generation and validation processes such that the compiler emits elementary and precise information about inference failure.

Constraint Generation

The constraint formation in the current inference algorithm is based on compatibility and subtyping assertions. The actual arguments and target types are directly used to form constraints.

Initially, the upper and lower bound wildcards have the constraint relation with inference variables as α :>CAP#1 and α <: CAP#1 respectively. It further reduces, and new precise formulas are formed at the incorporation phase. In this phase, we propose adding scope constraints for the implicit wildcards that passed the containment validation. The embedded scope of the actual argument can be represented as List<α> <: List<CAP#1>, (CAP#1) -> C<T>. The notation implies that the capture variable CAP#1 is within the scope of the class C with a parameter T. The unbounded wildcard may form subtyping or type equality constraints. If the unbounded wildcard parameter has a null literal as a type argument, then there is no constraint formation. We further elaborate on how null is resolved for constraint formation in the case of wildcard bounds.

Resolving Null

Some literary work has indicated a pattern of null uses that impacts Java's generic unsoundness and, thus dubious error messages. For example, lower bounded null [Smith and Cartwright 2008], implicit nulls [Amin and Tate 2016], and null as $\perp$ [Bierhoff 2022]. The JLS treats null as a default lower bound for wildcards. The constraint formation for null arguments is allowed, except in the case where null is a supertype. However, it fails to specify how the null bounds were resolved. We didn't find any type-inference error message that mentions a violation due to null presence; instead, the actual reason is null. We propose to resolve two different patterns of null presence. The first scenario is an explicit or direct null. If the wildcard has an equality relation with null, such as List<?> = null, we propose ignoring constraint formation for the explicit null argument because there is no proper specification for the lub (null). The null type does not provide any useful information about the relation between two types during type inference resolution; rather, it may result in a dubious inference type. For example, infer(Object, Integer) clearly infers an Integer as lub(Object, Integer) and infer(String, Integer) as an incompatible type error. However, infer(String, null) results in String, and in some situations, it results in Null, too. The second scenario is an implicit null assignment to the bounded wildcards. At the incorporation phase, the implicit constraint wildcards are reduced to their final bound (actual type). Therefore, we propose a reduction of captured bounds to False if the lower bounded wildcard and the unbounded wildcard are reduced to a null. Assigning null to the upper-bound wildcard cannot violate type soundness, as it always follows the subtyping principle.

The null assignment to the upper-bound wildcard can participate in inference resolution, but it does not contribute any kind of type information for the inference. Therefore, it is better to skip constraining the null argument and the upper-bounded wildcard parameter. During resolution, the compiler finds false for the null assignment in the bound set and emits a null value error message.

Constraint Validation

The current algorithm recursively reduces constraint formulas to bounds. If the reduction or incorporation process finds any inconsistency, it marks the inconsistency as false. At the end of the resolution process, when there is no more iteration left, the false bounds result in an inference error. This validation process skips a few elementary details in the context of wildcard-type inference, resulting in dubious and abstract error messages. For example, implicit constraints and the scope of an actual wild-type argument. Therefore, we propose constraint validation at the reduction phase. The constraint validation at this phase is specifically for wildcard-type argument containments. If the compatibility of the type parameter and the actual type argument is not satisfied, the compiler immediately emits an inconsistency error, and the inference process terminates.

5 Implementation

We developed an annotation-based Java compiler plugin by implementing the Plugin interface. This plugin is limited to a compile-time extension since it does not contain any modules for the JVM extension. The annotation will help to get the exact type elements on which the plugin processes and provide possible inferred type information for that annotated element. The annotation is the medium to connect the user interface with the plugin. The following sub-sections contain the complete process of plugin implementation.

5.1 Plugin Life-Cycle

The Java compiler calls the plugin through the init(JavacTask task, String... args) method. The first parameter is an instance of the JavacTask class, which provides access to the Javac compiler's functionality. The second parameter is an array of strings to hold arguments for plugins, if any. The four crucial stages of plugin implementation and their respective interfaces are as follows:

1. **Parse**: Building an abstract syntax tree (AST) using TaskEvent.Kind.PARSE.
2. **Enter**: Using TaskEvent.Kind.ENTER to import classes and methods of source code.
3. **Analyze**: Analyzing abstract syntax trees (ASTs) for errors. In this phase, we can do customized type checks, which can be added by modifying the AST.
4. **Generate**: Generating class files using GENERATE Enum.

As highlighted, the TaskEvent class is essential in completing the plugin cycle. The TaskEvent class is a part of the Compiler Tree API, and it provides details about the work that the JDK Javac Compiler has done.

5.2 Compiler Tree API

The compiler API has been available with JDK 6 and is primarily used to develop compiler plugins. It provides interfaces representing source code as ASTs and utilities to work on these ASTs. As mentioned above, the `init()` method has a parameter, an instance of JavacTask that has direct access to the Javac compiler. Utilizing the instance of JavacTask, we can call the TaskListener to monitor the activity of the Javac compiler. It invokes the start-and-finish method according to the occurrence of the event. When the parsed event is finished, we can get the AST through the `TaskEvent.getCompilationUnit()` method. The AST details can be analyzed through the TreeVisitor interface. Only a Tree element, for which the `accept()` method is called, dispatches events to the given visitor. When a visitor is passed to a tree's accept method, the visitXYZ() method most applicable to that tree is invoked. In our plugin, we override three visitXYZ() methods, namely `visitNewClass(), visitParameterizedType(), and visitWildcard().` By visiting these methods, we get information about the possible bounds of wildcards, and based on that, with few constraints, the plugin fetches inferred type information.

5.3 Implementation Setup

The Java Plugin API was added in Java 8. Therefore, the implementation requires JDK 8 or a newer version. The Plugin interface is available in the Java util package. We need to import the plugin interface, and then we can create a Java class that implements the Plugin API. The Java class should be discoverable through the ServiceLoader framework. We must create a "META-INF/service/" directory on the project classpath. This directory contains a file with the fully qualified name of the abstract service class. In our case, it is "META-INF/service/com.source.GPluginClass". After that, it is possible to add the plugin's *.class file(s) and service discovery descriptor to the classpath and to call Javac with the -Xplugin= option. The following are the steps followed to develop our Java compiler plugin using the Plugin interface:

- The InferredT annotation interface is created where the element types are TYPE_USE and TYPE_PARAMETER.
- The Java class named "GPluginClass" implements the Plugin interface and overrides the `getName()` and `init()` methods. The `getName()` method is used to get the plugin's name, and the init() method is used to initialize the plugin for the mentioned compilation task.
- The plugin API fetches details from compiled code, which are ASTs, with the help of a Tree API. In our case, we override the visitNewClassTree(), visitParameterizedType(), and visitWildTree() methods.
- Based on the information gathered by parsing these trees and applying the constraints, the plugin pops up the inferred type information at compile time.

The pseudocode of the plugin implementation is in Appendix A.

6 Results

The proposed algorithm fixes the dubious type inference error messages in the following cases of wildcard type inferences. Listing 11 contains a code snippet from Listing 1. This shows that when we use the `@InferredT` annotation along with the method definition, the plugin returns an improved-type error message.

6.1 Case 1: Scope constraint for improved error message

```
1 @InferredT
2 List<T> method2( List<? super T> t1, T t2){...}
3 Temp<Number> obj=new Temp<>(90.0);
4 List<Double> dlist2= obj.method2(nlist,15.9); //error
5 =============== Plugin Message ================
6 Temp.java:80: error : incompatible types: List<Number> cannot be
    converted to List<Double> ::List<Double> dlist2= obj.method2(nlist,15.9);
7
8 reason:     target type dlist2 doesn't confirm CAP\#1 scope type
    parameter of method2(CAP\#1, T).
9 where CAP\#1 extends Number from Temp<T extends Number> and T ->
    Number
```

Listing 11: Relevant error message

The previous error message shown in Listing 1 does not mention the implicit constraint over the type parameter of `t1`. The method type parameter `t1` is lower bound with `T` which is the class (Temp) parameter type. In the main method, the Temp class parameter is instantiated with the `Number` type, as shown in Line 3. This implies the parameter `t1` must initialize the same type (Number). However, the target type of method invocation expression is `Double`. The existing inference algorithm performs target-type assertion along with method arguments as the last step of constraint resolution. That results in a dubious error message, as shown in Listing 1. The proposed algorithm suggests adding more constraints to the target type assertion to get an exact reason for type incompatibility. Adding scope-based constraints, according to the proposed algorithm, provides a comprehensive compiler error message. In Listing 11, the scope of `CAP#1` is the class `Temp<T>` which is initialized with `Number` type. The target type assertion finds a mismatch with the scope constraint type, and hence the compiler points to the exact reason for type incompatibility.

6.2 Case 2: Null rejection

```
 1 class NullTest{
 2  interface I<X> {}
 3    <T> T make(I<?> ts) { return null; }
 4    <E extends CharSequence> E take(I<? super E> arg) { return null; }
 5  void test(I<I<? super String>> arg) {
 6      take(make(arg)).intern();     }}
 7============Compile-Time Error Message===========
 8  NullTest.java:6: error: cannot find symbol
 9          take(make(arg)).intern();
10   symbol:   method intern()
11    location: interface CharSequence
12  1 error
13============Plugin Message===================
14 NullTest.java:6: error: invalid null assignment
```

```
15 reason:  null cannot be super of E::  take(CAP\#3) ->
16  take(null)
17 where CAP\#3 extends CharSequence
```

Listing 12: Rejecting null assignment to contravariant

According to the existing Java inference algorithm, the initial constraints of Listing 12 are as follows: `I<CAP#1> -> I<CAP#2>, I<CAP#3> :> null, E <: CharSequence`. The captured type variable `CAP#3` is implicitly constrained with null, but it is skipped during further constraint reduction, and the inference algorithm resolves as `CharSequence` instead of a null type error. However, the proposed algorithm finds the implicit null constraint in an intermediate phase (incorporation) of constraint reduction and emits a null-type error.

6.3 Case 3: Containment validation

```
1 @InferredT
2 <J extends I>
3       I m1(MultiPara<I,J> arg, J j) {
4         return j;}}
5 =============== Plugin Message ===============
6 Check.java:5:  error:  method m1(MultiPara<I,J> arg, J j) does not contain
    m1(constrain, x)
7 reason:  the argument for type parameter J does not  satisfy  the
    containment rule::  ?  super X <= ? extends Object
8  where I,J,X,Y are type-variables:  X, Y are scoped within method
    <Y,X>m2(T) and J, I is scoped within class MultiPara<I, J extends I>.
```

Listing 13: Error due to containment rejection

The code snippet is from Listing 6. As mentioned in previous works [Amin and Tate 2016, Bierhoff 2022], the Javac compiler did not mention the valid reason for incompatible bounds, and the error message does not provide sufficient information to be debugged. Our plugin first verifies the code based on the containment rule and finds there is a containment violation. According to the listed containment rule 10, the upper-bounded type argument contains a lower bound only when the upper bound is an object type. The code in Listing 6 does not satisfy the containment rule, and its compilation emits an error diagnostic. Here, we didn't get a null error since the process terminates early at the reduction phase and does not process wildcard incorporation.

7 Conclusion

Wildcard-based type inference in Java is a complex and implicit process that developers need to understand for accurate implementation and debugging. However, the Javac compiler often lacks consistent and informative error messages for wildcard type inference in most situations. In this study, we explored various wildcard inference codes along with associated error messages from the literature and the Java bug database. We identified and highlighted the shortcomings of both the Javac compiler and the Java Language

Specification (JLS) concerning misleading and insufficient error messages with these Java codes.

In this work, we proposed additions to the existing inference algorithm, specifically for wildcard-type inference, focusing on constraint generation and validation. The proposed additions introduce more bounds during the resolution process, providing additional inference information and resulting in more meaningful error messages.

Our Javac plugin, based on the proposed algorithm, resolves the identified issues, generating more significant error messages for inference failures. However, at this stage, the plugin works only in a few limited cases of wildcard-type inference. Future research will explore additional aspects of type argument inference to further advance the accuracy and applicability of this approach.

A Appendix: Pseudo Code

The pseudocode of the plugin process initiation and the four stages of implementation as mentioned in Section 5 are as follows:

Algorithm 2 Plugin process initialization

```
 1: Implement Plugin interface and define its two methods: getName() and init().
 2: function getName
 3:     return name                                  ▷ get the name of the plugin
 4: end function
 5: function init(JavacTask task, String args) ▷ calling TaskListener() to invoke start
    and finish processes
 6:     task.addTaskListener(new TaskListener()
 7:     function start(TaskEvent e)
 8:     end function
 9:     function finished(TaskEvent e)
10:         if e.getKind() ≠ TaskEvent.Kind.PARSE then
11:             return
12:         end if
13:         e.getCompilationUnit()
14:     end function
15:     )
16: end function
```

Algorithm 3 Stages 1 & 2: Parsing ASTs and Source code import

1: **function** getCompilationUnit.accept(new TreeScanner<$Void$, $Void$>()) ▷ accept() to implement visitor pattern for specific Tree element
2: visiting element-specific visitor pattern
3: **function** visitNewClass(ClassTree $node$, Void $aVoid$)
4: return super.visitNewClass(node, aVoid)
5: **end function**
6: **function** visitParameterizedType(ParameterizedType $node$, Void $pVoid$)
7: return super.visitParameterizedType(node, pVoid)
8: **end function**
9: **function** visitWildcard(WildcardTree $node$, Void $wVoid$)
10: return super.visitWildcard(node, wVoid)
11: **end function**
12: **end function**

Algorithm 4 Stage 4: Final binary generation

1: Gathers elementary information about the parameter and embeds it into final error message
2: **function** generateError(TreeMaker $factory$, Names $symbolsTable$, Tree $parameter$)
3: **if** parameter.getKind()==ParameterizedTypeTree **then**
4: parameter.getType() and parameter.getTypeArguments() ▷ get parameter and argument type
5: **else**
6: **if** parameter.getKind()==WildcardTree **then**
7: parameter.getbound() ▷ get bounds of wildcardtype
8: **end if**
9: **end if**
10: symbolsTable.fromString(parameter) ▷ get the scope of enclosing parameter
11: print error message
12: **end function**

Algorithm 5 Stage 3: AST modification

```
 1: New AST elements can be created through a TreeMaker instance
 2: function CheckType(MethodTree node, Context context)        ▷ context assists in
    extending the compiler by extending its components ▷ Context instance is obtained
    at init(), ((BasicJavacTask) task).getContext()
 3:     TreeMaker factory = TreeMaker.instance(context)
 4:     List<? extends Tree> parametersToInstrument = method.getParameters();
 5:     if parametersToInstrument instanceof WildcardTree then
 6:         checkNull(parametersToInstrument)
 7:     end if
 8: end function
 9: function checkNull(ParameterizedTypeTree parameter)
10:     if parameter.getTypeArguments() == null then
11:         break
12:     elsecheckContainment(parameter)
13:     end if
14: end function
15: function checkContainment(WildcardTree parameter)
16:     if Match(parameter) then          ▷ match() method matches the parameter with
    containment argument
17:         generateConstraint(parameter, symbolsTable)
18:     else
19:         generateError(factory, symbolsTable, parameter)  ▷ generate error message
20:         break
21:     end if
22: end function
23: function generateConstraint(WildcardTree parameter, Names symbolsTable)
24:     captured = captureConversion(parameter, parameter.getTypeArguments())
25:     if captured = null then
26:         generateError(factory, symbolsTable, parameter)
27:         break
28:     else
29:         constraintFormation(parameter, symbolsTable)        ▷ add scope constraint
30:     end if
31: end function
```

References

[AbdelGawad 2018] AbdelGawad, M. A.: “Towards taming Java wildcards and extending Java with interval types,”; arXiv preprint arXiv:1805.10931 (2018).

[Agesen 1996] Agesen, O.: “Concrete type inference: Delivering object-oriented applications”; Ph.D. Thesis, Stanford University (1996).

[Altidor et al. 2012] Altidor, J., Reichenbach, C., Smaragdakis, Y.: “Java wildcards meet definition-site variance”; Proc. 26th ECOOP, Springer (2012), 509-534.

[Amin and Tate 2016] Amin, N., Tate,R.: “Java and Scala’s type systems are unsound: The existential crisis of null pointers”; Proc. 31st OOPSLA, ACM (2016), 838-848.

[Becker et al. 2019] Becker, B.A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D.J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P.M. and Pearce, J.L.: “Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research”; Proc. Working Group Reports Innovation & Technology in Computer Science Education (ITiCSE-WGR) (2019), ACM, 177-210.

[Bierhoff 2022] Bierhoff, K.: “Wildcards need witness protection”; Proc. ACM Program. Lang. 6 OOPSLA2, Article 138 (2022), 373-394.

[Boustani and Hage 2009] El Boustani, N., Hage, J.: “Improving type error messages for generic Java”; Proc. PEPM, ACM (2009), 131-140.

[Boustani and Hage 2010] El Boustani, N. and Hage, J.: “Corrective hints for type incorrect generic Java programs”; Proc. ACM SIGPLAN Workshop Partial Evaluation Program Manipulation (2010), ACM, 5-14.

[Brotherston et al. 2017] Brotherston, D., Dietl, W., Lhoták, O.: “Granullar: Gradual nullable types for Java”; Proc. 26th Int. Conf. Compiler Construction, ACM (2017), 87-97.

[Cassola et al. 2022] Cassola, M., Talagorria, A., Pardo, A., Viera, M.: “A gradual type system for Elixir”; Journal of Computer Languages 68, 101077, (2022).

[Chaliasos et al. 2021] Chaliasos, S., Sotiropoulos, T., Drosos, G.P., Mitropoulos, C., Mitropoulos, D. and Spinellis, D.: “Well-typed programs can go wrong: A study of typing-related bugs in JVM compilers”; Proc. OOPSLA (2021), ACM, 1-30.

[Damas and Milner 1982] Damas, L., Milner, R.: “Principal type-schemes for functional programs, in”; Proc. 9th ACM SIGPLAN-SIGACT Symp. POPL (1982), 207-212.

[Fetzer and Felber 2007] Fetzer, C., Felber, P.: “Improving Program Correctness with Atomic Exception Handling”; J.UCS - Journal of Universal Computer Science 13 (8) (2007), 1047-1072.

[Greenman et al. 2014] Greenman, B., Muehlboeck, F., Tate, R.: “Getting f-bounded polymorphism into shape”; ACM SIGPLAN Notices 49 (6) (2014) 89-99.

[Gregor et al. 2006] Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G. and Lumsdaine, A.: “Concepts: Linguistic support for generic programming in C++”; Proc. 21st OOPSLA (2006), ACM, 291-310.

[Hansen et al. 2007] Hansen, A.H., Riis, C. and Hovard, J.: “https://www.bdsoft.com/tools/stlfilt.html”, Software Tools: STLFilt (2007).

[Horstmann 2014] Horstmann, C. S.: “Java SE 8 for the Really Impatient, Pearson Education”; (2014).

[Kennedy and Pierce 2007] Kennedy, A.J. and Pierce, B.C.: “On decidability of nominal subtyping with variance”; Proc. FOOL/WOOD (2007), ACM, 1-25.

[Kumar and Agrawal 2007] Kumar, R., Agrawal, V.: “Multiple dispatch in reflective runtime environment”; Computer Languages, Systems Structures 33 (2) (2007) 60-78.

[Kumari and Kumar 2019] Kumari, N., Kumar, R.: “Evolution of Generic Programming in OOPLs”; ACM SIGSOFT Software Engineering Notes 44 (1) (2019), 35-43.

[Leivant 1983] Leivant, D.: “Polymorphic type inference”; Proc. of the 10th ACM SIGACT-SIGPLAN Symposium on POPL, ACM (1983), 88-98.

[Luoma et al. 2007] Luoma, H., Lahtinen, E. and Jarvinen, H. M.: “CLIP, a command line interpreter for a subset of C++”; Proc. 7th Baltic Sea Conf. Computing Education Research (2007), Australian Computer Society, 199-202.

[McCall 2016] McCall, D.: “Novice Programmer Errors - Analysis and Diagnostics”; Ph.D. Thesis, University of Kent (2016).

[Medeiros et al. 2014] Medeiros, F., Ribeiro, M., Gheyi, R., Fonseca, B.: "A Catalogue of Refactorings to Remove Incomplete Annotations"; J.UCS - Journal of Universal Computer Science 20 (5) (2014), 746-771.

[Muehlboeck and Tate 2017] Muehlboeck, F., Tate, R.: "Sound gradual typing is nominally alive and well,"; Proc. 32nd OOPSLA, ACM (2017), 1-30.

[Ortin et al. 2022] Ortin, F., Garcia, M., Perez-Schofield, B. G., Quiroga, J.: "The StaDyn programming language"; SoftwareX 20 (2022).

[Palsberg and Schwartzbach 1991] Palsberg, J., Schwartzbach, M., I.: "Object-oriented type inference"; ACM SIGPLAN Notices 26 (11) (1991), 146-161.

[Papi et al. 2008] Papi, M. M., Ali, M., Correa, T. L., Perkins, J. H., Ernst, M. D.: "Practical pluggable types for Java"; Proc. ISST, ACM (2008), 201-212.

[Pavlinovic 2019] Pavlinovic, Z.: "Leveraging Program Analysis for Type Inference"; Ph.D. Thesis, New York University (2019).

[Plevyak and Chien 1994] Plevyak, A., Chien A.: "Precise concrete type inference for object-oriented languages"; Proc. 9th OOPSLA, ACM (1994), 324-340.

[Plümicke 2015] Plümicke, M.: "More type inference in Java 8"; Proc. Perspectives of System Informatics, Springer (2015), 248-256.

[Smith and Cartwright 2008] Smith, D., Cartwright, R.: "Java type inference is broken: Can we fix it?"; Proc. 23rd OOPSLA, ACM (2008), 505-524.

[Stadelmeier and Plümicke 2023] Stadelmeier, A. and Plümicke, M.: "Type Inference for Java: Unification of Type Constraints Involving Existential Types"; Proc. 22nd Kolloquium Programmiersprachen und Grundlagen der Programmierung (2023), 159-174.

[Tate et al. 2011] Tate, R., Leung, A., Lerner, S.: "Taming wildcards in Java's type system"; Proc. 32nd PLDI, ACM (2011), 614-627.

[Tate 2013] Tate, R.: "Mixed-site variance"; Proc. 20th FOOL (2013), 614-627.

[Torgersen et al. 2004] Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: "Adding wildcards to the Java programming language,"; Proc. 18th SAC, ACM (2004), 1289-1296.

[Traver 2010] Javier Traver, V.: "On Compiler Error Messages: What They Say and What They Mean"; Advances in Human-Computer Interaction (2010).

[Yang et al. 2000] Yang, Jun and Michaelson, Gregory John and Trinder, Phil and Wells, Joseph Brian: "Improved type error reporting"; Proc. 12th Int. Workshop Implementation Functional Languages (2000).