Higher-Order Symb Comput (2011) 24:3–39
DOI 10.1007/s10990-011-9070-3

Improving type error messages for generic Java

Nabil el Boustani · Jurriaan Hage

Published online: 15 June 2011

Abstract Since version 1.5, generics (parametric polymorphism) are part of the Java language. However, the combination of parametric polymorphism and inclusion polymorphism is complicated, particularly so for Generic Java. Indeed, the main Java compilers, Eclipse's EJC and Sun's JAVAC, do not even accept the same set of programs. Moreover, experience with these compilers shows that the error messages provided by them leave more than a little to be desired.

To alleviate the latter problem, we describe how to adapt the type inference process of Java to obtain better error diagnostics for generic method invocations. Although the extension by itself already helps to improve type error messages to some extent, another major advantage of the new type inference process is that it also paves the way for further heuristics can provide additional diagnostic information. The extension has been implemented into the JastAdd Extensible Java Compiler.

Keywords Compilers · Type checking · Error diagnosis · Java generics

1 Introduction

Since the introduction of generics in Java, the programmers who seek to actually use this powerful feature may have discovered that production strength compilers such as Eclipse's EJC and Sun's JAVAC, do not always carefully explain why a given generic method invocation fails to type check.

This paper is an extended version of [1] that was presented at PEPM'09.

N. el Boustani · J. Hage (✉)
Center for Software Technology, Department of Information and Computing Sciences,
Universiteit Utrecht, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail: jur@cs.uu.nl

N. el Boustani
e-mail: alumno@gmail.com

```
import java.util.*;

class Listing1{
    <T> void foo(Map<T,T> a){
        Map<Number, Integer> m1 = null;
        foo(m1);
    }
}
```

```
listings/Listing1.java:6: <T>foo(java.util.Map<T,T>) in Listing1 cannot be
applied to (java.util.Map<java.lang.Number,java.lang.Integer>)
        foo(m1);
        ^
```

JAVAC

```
1. ERROR in listings/Listing1.java (at line 6)
    foo(m1);
    ^^^
The method foo(Map<T,T>) in the type Listing1 is not applicable for the
arguments (Map<Number,Integer>)
```

EJC

```
listings/Listing1.java:6
Method <T>foo(Map<T, T>) of type Listing1 is not applicable for the argument of
type (Map<Number, Integer>), because:
  [*] The type variable T is invariant, but the types:
  - Integer in Map<Number, Integer> on 5:9(5:21)
  - Number in Map<Number, Integer> on 5:9(5:13)
  are not the same type.
```

ours

Fig. 1 A Java code fragment harbouring a type equality conflict, and the type error messages for JAVAC, EJC and our implementation respectively

Consider the code and the corresponding error messages[1] in Fig. 1. Both EJC and JAVAC merely claim that there is no method declared with the signature `foo(Map<Number, Integer>)`. However, they do not explain why the `foo` method that is declared does not match the invocation. Our message, on the other hand, does make such an attempt.

In this paper, we describe how to rearchictect the Java type checking process in order to obtain more informative error messages. We have implemented our solution into the JastAdd Extensible Java Compiler and provide many examples of programs that benefit from our approach. Our modifications also pave the way for the implementation of special heuristics that can go beyond explaining what the problem is, by showing how the problem can be fixed. These heuristics are, however, considered in another paper [2].

Having some experience in improving type error messages for functional languages [9, 13], it turns out that in the current study we faced a number of additional problems. The first problem is that the Java Language Specification (JLS) [7] is large, complicated and full of all kinds of restrictions and limitations that the programmer needs to know about. The fact that not even EJC and JAVAC always agree on the type correctness of a program

[1] All examples in the paper first give the code to be compiled, followed by a number of error messages, accompanied by an indication of which compiler generated the message. The indication **ejc** refers to the Eclipse Java Compiler (0.883_R34x, 3.4.1 release, in mode 1.5, MacOSX), **javac** to Sun's Javac (version 1.5.0_22, MacOSX), and **ours** refers to our own implementation based on the JastAdd Extensible Java Compiler (see Sect. 8 for information on how to obtain this implementation). In our examples we sometimes write (...) to indicate that part of the error message has been deleted. These omitted parts are hints that serve to suggest how to fix the type error and are considered outside the scope of the current paper [2].

suggests that also the implementers of Java have problems coming to grips with it. Furthermore, Smith and Cartwright have shown that type inference in Generic Java is neither sound nor complete [27]. It is worthwhile to note here that type inference is not the same as type checking: type inference, in the case of Java, refers to the process that tries to come up with suitable instantiations for type variables that occur in the program. In the case of Java this process can come up with instantiations that will later be found to be inconsistent. In that sense, type inference can be considered to be unsound, but it does not mean that the specification allows type incorrect programs. A further illustration of the complexity of Generic Java is the sizable FAQ maintained by Angelika Langer [16].

The second problem is that the combination of generics with subtyping, i.e., the fact that you can bind an object of type T to an identifier of type S if T is a subtype of S, yields a type system that is inherently complex. This is further complicated by other demands on the language, such as backwards compatibility and the recent addition of various other features such as variable arity methods and autoboxing and unboxing.

Given the size of the language, we restrict ourselves to type error diagnosis for *generic method invocations*. In our solution we adhere to three general principles: we do not change, but only add to the original type checking process, in that our extension only comes into play after an inconsistency has been detected. Second, in order to provide more elaborate error messages, we retain the constraints generated for an inconsistent invocation in order to better explain what went wrong, and to be able to execute heuristics that work directly on the constraints. The final principle is to avoid acting on those parts of the program that are more likely to contain mistakes. Since we believe that the generic arguments of the generic types are likely to be the source of a mistake when an inconsistency arises, this implies we should ignore these arguments as long as we can.

To better substantiate and position our contribution, we consider the properties proposed by Yang and others [34] that a good type error message should have: *correctness*, *accuracy*, *intuitiveness*, *succinctness* and *source-basedness*. Although these properties tend to be hard to establish formally, we believe the examples in this paper show that compared to the error messages provided by EJC and JAVAC our messages are more intuitive and source-based. We also believe that we strike a better balance in being more informative, although at the price of being less concise. Finally, we believe our implementation is not less correct than that of the compiler into which our research has been implemented (but proving that would be very hard). In general, we think that we have improved the situation with respect to JAVAC and EJC in particular by striking a better balance between verbosity and terseness, and that our messages are less tied the internal type inference process.

The paper is structured as follows. In Sect. 2 we reiterate some of the essential elements of the Generics extension in Java, and introduce some basic notations. Section 3 then provides quite a number of examples of type error messages constructed by our implementation, together with the messages provided by EJC and JAVAC. In Sect. 4 we explain in some detail the type checking process as specified by the JLS, and discuss our extension to that process in Sect. 5. In Sect. 6 we briefly describe our implementation, Sect. 7 describes related work, and Sect. 8 reflects, concludes and gives directions for future work.

2 An overview of Java generics

For completeness we run through the essentials of the generics of Java. We assume the reader is familiar with the non-generic part of Java. For more details, the reader can consult the Java standard [7] and the Java Generics FAQ [16].

Arguably, the main reason for introducing generics into Java was to counter the large number of casts needed to deal with collection classes, e.g., sets and vectors. Before generics, all collection classes were defined so that any `Object` could be stored in them. However, this makes all collections potentially heterogeneous and makes it necessary to explicitly downcast objects obtained from such a collection. This is both cumbersome and potentially unsafe.

With generics, the programmer can specify upper bounds, besides `Object`, for the objects that can be stored in a collection. For example, a `List<Number>` can store `Numbers` and objects of any type that is a subclass of `Number`, such as `Integer`. If an object is retrieved from such a list, then it can be stored as a `Number` without any type cast, and if the need arises it can be further downcast to, say, `Integer`.

An important and maybe subtle point is that although `Integer` is a subclass of `Number`, `List<Integer>` is not a subclass of `List<Number>`. Indeed, this holds for all collection classes: they are all invariant type constructors. All things considered, this is not so strange: a `List<Integer>` is not supposed to store `Numbers`, but if we assign it to a variable of type `List<Number>` we cannot safely guarantee this. However, a value of type `HashMap<Integer,String>` may be passed safely to a parameter of type `Map<Integer,String>`, because `HashMap` extends `Map`.

A consequence of the invariance restriction is that there is no type that denotes a list of any kind of element. We still need such a type, for example to write a length method for lists. This is why the wildcard was introduced: `List<?>` denotes a list for which nothing is known of the element type. We can store a `List<T>` for any type `T` in a variable of such a type. The price to be paid is that we cannot store anything into the list, and we can only read `Objects` from it.

The wildcard introduces a problem by itself. If we would like to write a method that computes the reverse of a list, we can easily do so for non-wild card types, and only for wild-card types if we do not mind losing the knowledge that the input list and the output list have the same element type. In Java this can, in some cases, be solved by *wildcard capture conversion* [30], in which case the compiler can determine that although it may not know the concrete type at compile-time, it is still known to be safe at run-time.

In many cases, we can be more precise in our estimation of the possible types that a certain type variable or a wildcard may have. For that reasons *bounds* were introduced. For example `T extends Number` expresses that the type inferred for `T` should be a subtype of `Number`. Similarly, `T super Number` expresses the inverse relation. The same applies to wildcards. For example, the declaration

```
void foo (Map<? extends Number, ? super Integer> mp)
```

expresses that the key type of an actual parameter should be a subtype of `Number` and that the value type should be a supertype of `Integer`. Note however, that the fact that a wildcard `?` refers to both key and value type does *not* imply that the types are the same, or even related. For example, we can pass a value of type `Map<Double, Integer>`, `Map<Number, Number>` or even `Map<Integer, Object>` to `foo`.

In Generic Java, the *raw type*, e.g., a type constructor such as `List` without its type arguments, is assignment compatible with all instantiations of the generic type. So, if you write `List` as a type somewhere, the inferencer can decide to interpret it as `List<T>` for whatever `T` it finds suitable at that point. This mixing of type constructor and type is used in dealing with legacy code.

The JLS imposes a number of seemingly arbitrary restrictions. To give a few examples: Wildcard capture conversion only works when wildcards are at top-level, and not for a type

like `List<List<?>>`. Wildcards may only occur as arguments to a generic type constructor, i.e, you cannot write `? x = ...` to describe an identifier of unknown type. Primitive types may not occur as parameters of a generic type constructor, e.g., `Vector<int>` is not allowed, but `Vector<Integer>` is. There are also some restrictions on how type variables may be bounded: we may write `T extends Number`, but not `T super Number` [7, §4.4]. This restriction does not apply to wildcards: both `? extends Number` and `? super Number` are allowed [7, §4.5.1].

2.1 Notation and terminology

We shortly introduce some notation used throughout the paper, linking these to the pertinent paragraphs of the JLS [7], to which we refer for more details.

Since we deal exclusively with method invocations in this paper, and focus on generic classes and interfaces, the basic operation is that of method invocation conversion (§5.3). It specifies when actual arguments are compatible with formal parameters. In our case, the main conversion operation is widening reference conversion (§5.1.5), which basically specifies that compatibility is governed by the subtype relation (§4.10): `T` $<:$ `S` if `T` is a subtype of `S`; we also write `S` $:>$ `T` for `T` $<:$ `S`. The relation $:>$ is defined as the reflexive and transitive closure of the direct supertype relation $>_1$, which contains the *extends* relationship between classes and interfaces: if `A` extends `B` then `B` $>_1$ `A`. As §4.10.2 of the JLS makes clear, the subtyping relation among (generic) classes and interfaces is quite complicated. For the purposes of the examples in this paper, we need not to consider these definitions in full detail. Instead, we prefer to give a restricted version of the definition[2]:

$$\texttt{C<S1, ..., Sn>} <: \texttt{D<T1, ..., Tn>}$$

if and only if `C` $<:$ `D` and for all $1 \leq i \leq n$: `Si` $\leq:$ `Ti` (pronounced `Si` *contains* `Ti`) where

- `T` $\leq:$ `T`,
- `T` $\leq:$ `? extends T`,
- `T` $\leq:$ `? super T`,
- `? extends T` $\leq:$ `? extends S`, if `T` $<:$ `S`,
- `? super T` $\leq:$ `? super S`, if `S` $<:$ `T`.

Intersection types are written $\texttt{T}_1$ & ... & $\texttt{T}_n$, $n > 0$, where the $\texttt{T}_i$ are type expressions (Sect. 4.9). At most one of these types may specify a class type, since Java does not support multiple inheritance.

The *least upper bound* (*lub*) of a collection of classes and interfaces $\mathcal{S}$ is the most specific type that is a supertype of each $S \in \mathcal{S}$. For example, *lub*({`Integer`, `Double`}) = `Number`. Often, the *lub* is not simply a class name, but also specifies a number of interfaces. For example, *lub*({`Integer`, `String`}) equals

```
Object & Serializable &
     Comparable<? extends Object & Serializable &
     Comparable<?>>.
```

Dually, the greatest lower bound (*glb*) of $\mathcal{S}$ is the most general type in the hierarchy that extends each $S \in \mathcal{S}$. Note that the *glb* is sometimes undefined, e.g., *glb*({`String`, `Number`}).

[2]For example, we restrict `C` and `D` to have the same number of arguments, there are no intersections, and no capture conversion is performed.

3 Examples

To illustrate what can be gained from the developments in this paper, we compare the messages generated by our implementation with those generated by the standard compilers for Java, EJC (0.883_R34x, 3.4.1 release, in mode 1.5, MacOSX) and JAVAC (version 1.5.0_22, MacOSX). We have taken the liberty to insert some newlines in order to make the output fit the width of a page. The examples we show tend to be contrived in the sense that they show a particular phenomenon with as little code as possible. We shall however start with a somewhat more realistic and intuitive longer example. The number of examples is relatively small, but Appendix describes where to obtain programs to generate many more examples of the messages we can provide, as part of the test set for our implementation.

Consider the code in Fig. 2 which describes a class for doing static operations on `HashMaps`. Four methods are provided: `copy` adds the pairs in `from` to those in `to`, and `project` restricts the domain of the map to the list of elements in the second argument. The method `printRange` can be used to show that the operations work as expected. The final operation is `selfcompose` which composes a map with itself: the result contains a pair (x, y) if there is some u such that (x, u) and (u, y) are pairs in the argument `HashMap`. A critical requirement is that in the argument to `selfcompose`, the map may not have an arbitrary range and domain, but that these should coincide. We believe the code to be sufficiently standard to need no further explanation. If the type-incorrect invocation `hm1 = MapOps.selfcompose(hm1);` is deleted, then the output of the program is `[twenty]`.

If the classes in Fig. 2 are compiled, then we obtain the type error messages given in Fig. 3. We note that JAVAC and EJC are not very informative, and only say that no matching method could be found. Our message explains the inconsistency between the defined `selfcompose` and the invocation: the method expects a map from a type to the same type, but this is not the case for `hm1` which maps numbers to strings.

It is quite easy to change `MapOps.java` to show the behaviour of our system for other kinds of errors. Consider that the line marked as incorrectly typed is omitted. We shall now describe three modifications of the signature of `project`. In each case, EJC and JAVAC continue to generate messages similar to those in Fig. 3, which we omit. The first modification is to change `extends` into `super` in the signature of `project`. If we compile the `MapOps.java` after this first modification, then the error message reads:

```
listings/MapOps.java:49
Method <T, S>project(HashMap<T, S>, ArrayList<? super T>)
of type MapOps is not applicable for the arguments
of type (HashMap<Number, String>, ArrayList<Integer>), because:
  [*] The type Integer in ArrayList<Integer> on 46:8(46:18)
      is not a supertype of the inferred type for T: Number.
```

If we write `? extends S` instead of `? extends T`, then we obtain the message

```
listings/MapOps.java:49
Method <T, S>project(HashMap<T, S>, ArrayList<? extends S>)
of type MapOps is not applicable for the arguments
of type (HashMap<Number, String>, ArrayList<Integer>), because:
  [*] The type Integer in ArrayList<Integer> on 46:8(46:18)
      is not a subtype of the inferred type for S: String.
```

Finally, if we replace `? extends T` with `T`, then the message becomes:

```
import java.util.*;

class MapOps {
    MapOps() {
    }

    static <T,S> void copy(HashMap<T,S> from, HashMap<T,S> to) {
      for (T key : from.keySet()) {
         to.put(key, from.get(key));
      }
    }

    static <T,S> HashMap<T,S> project(HashMap<T,S> cmap,
                                      ArrayList<? extends T> dom) {
        HashMap<T,S> retmap = new HashMap<T,S>();
        copy(cmap,retmap);
        Set<T> keyset = retmap.keySet();
        keyset.retainAll(dom);
        return retmap;
    }

    static <T,S> void printRange(HashMap<T,S> cmap) {
        System.out.println(cmap.values().toString());
    }

    static <T> HashMap<T,T> selfcompose(HashMap<T,T> map) {
        HashMap<T,T> twm = new HashMap<T,T>();
        Set<T> domain = map.keySet();

        for (T key : domain) {
          T img = map.get(key);
          if (domain.contains(img)) {
            twm.put(key, map.get(img));
          }
        }
        return twm;
    }
}

class Main {
    public static void main(String[] parameters)
    {
       HashMap<Number, String> hm1 = new HashMap<Number,String>();
       hm1.put(10,"ten");
       hm1.put(20,"twenty");
       hm1.put(21.5,"twenty-one-and-a-half");
       ArrayList<Integer> l1 = new ArrayList<Integer>();
       l1.add(20);
       l1.add(30);
       hm1 = MapOps.project(hm1,l1);
       hm1 = MapOps.selfcompose(hm1); // Type-incorrect
       MapOps.printRange(hm1);
    }

}
```

Fig. 2 A class defining operations on `HashMaps` that is used type incorrectly

```
listings/MapOps.java:50: <T>selfcompose(java.util.HashMap<T,T>) in MapOps cannot
be applied to (java.util.HashMap<java.lang.Number,java.lang.String>)
       hm1 = MapOps.selfcompose(hm1);
                    ^
```

JAVAC

```
1. ERROR in listings/MapOps.java (at line 50)
    hm1 = MapOps.selfcompose(hm1);
                 ^^^^^^^^^^^
The method selfcompose(HashMap<T,T>) in the type MapOps is not applicable for
the arguments (HashMap<Number,String>)
```

EJC

```
listings/MapOps.java:50
Method <T>selfcompose(HashMap<T, T>) of type MapOps is not applicable for the
argument of type (HashMap<Number, String>), because:
  [*] The type variable T is invariant, but the types:
  - String in HashMap<Number, String> on 42:8(42:24)
  - Number in HashMap<Number, String> on 42:8(42:16)
  are not the same type.
```

ours

Fig. 3 The three error messages for the inconsistency caused by the wrong invocation to `selfCompose`

```
listings/MapOps.java:49
Method <T, S>project(HashMap<T, S>, ArrayList<T>)
of type MapOps is not applicable for the arguments
of type (HashMap<Number, String>, ArrayList<Integer>), because:
  [*] The type variable T is invariant, but the types:
  - Integer in ArrayList<Integer> on 46:8(46:18)
  - Number in HashMap<Number, String> on 42:8(42:16)
  are not the same type.
```

We now continue with the smaller, more contrived examples in order to show the kind of situations we can handle. We already saw an example of a type equality conflict, but we shall list a few more to illustrate some further differences between our compiler and EJC and JAVAC. The first example can be found in Fig. 4. Note that our message explains that there are in fact two independent errors (indicated by `[*]`), and that it also explains why the invocation does not match the given definition. Again, neither EJC nor JAVAC explains what the problem is, only that the invocation does not match.

In Fig. 5, the conflicts are not independent, because all inconsistencies involve the same type variable, and our message reflects this. We have omitted the messages of EJC and JAVAC since they follow the same pattern as in Fig. 4.

As a final type equality conflict consider the fragment in Fig. 6. In this case, both calls are inconsistent, and although the return type of the inner `foo` does not depend on the types of its inputs, all three compilers only mention the inconsistency arising from the inner invocation: the inconsistency will only show up after fixing the innermost one.

We now continue with an example of a subtyping conflict. Consider the code fragment and associated messages in Fig. 7. Such a conflict arises when an equality constraint determines the type of a particular type variable; in this case `T` becomes equal to `Integer`, which then leads to an inconsistency in the second parameter, because `Number` is not a subtype of `Integer`. Neither EJC nor JAVAC explain why the invocation does not match the declared type of `bar`; our message does provide such an explanation.

It is quite easy, but not very interesting, to come up with a similar mistake to that in Fig. 7, but which involves `super` rather than `extends`. A more interesting source of mistakes,

```
import java.util.*;

class Listingindep{
    <T,S> void foo(Map<T,T> a, Map<S,S> b){
        Map<Number,String> m1 = null;
        Map<Integer,String> m2 = null;
        foo(m1,m2);
    }
}
```

```
listings/Listingindep.java:7: <T,S>foo(java.util.Map<T,T>,java.util.Map<S,S>)
in Listingindep cannot be applied to (java.util.Map<java.lang.Number,
java.lang.String>,java.util.Map<java.lang.Integer,java.lang.String>)
        foo(m1,m2);
        ^
```

JAVAC

```
1. ERROR in listings/Listingindep.java (at line 7)
    foo(m1,m2);
    ^^^
The method foo(Map<T,T>, Map<S,S>) in the type Listingindep is not applicable
for the arguments (Map<Number,String>, Map<Integer,String>)
```

EJC

```
listings/Listingindep.java:7
Method <T, S>foo(Map<T, T>, Map<S, S>) of type Listingindep is not applicable
for the arguments of type (Map<Number, String>, Map<Integer, String>), because:
  [*] The type variable T is invariant, but the types:
  - String in Map<Number, String> on 5:9(5:20)
  - Number in Map<Number, String> on 5:9(5:13)
  are not the same type.
  [*] The type variable S is invariant, but the types:
  - String in Map<Integer, String> on 6:9(6:21)
  - Integer in Map<Integer, String> on 6:9(6:13)
  are not the same type.
```

ours

Fig. 4 A code fragment with two independent equality type conflicts (one in each argument), and the type error messages for JAVAC, EJC and our implementation respectively

```
import java.util.*;

class Listingref{
    <T> void foo(Map<T,Map<T,T>> a){
        Map<Number, Map<Integer,String>> m1 = null;
        foo(m1);
    }
}
```

```
listings/Listingref.java:6
Method <T>foo(Map<T, Map<T, T>>) of type Listingref is not applicable for the
argument of type (Map<Number, Map<Integer, String>>), because:
  [*] The type variable T is invariant, but the types:
  - Integer in Map<Number, Map<Integer, String>> on 5:9(5:25)
  - String in Map<Number, Map<Integer, String>> on 5:9(5:33)
  - Number in Map<Number, Map<Integer, String>> on 5:9(5:13)
  are not the same type.
```

ours

Fig. 5 A code fragment with multiple conflicts arising from the variable `m1`, and the error message we provide

```
import java.util.*;

class Listingnested{
    <T> Map<Number,Integer> foo(Map<T,T> a){
        Map<Number,String> m1 = null;
        foo(foo(m1));
    }
}
```

```
listings/Listingnested.java:6: <T>foo(java.util.Map<T,T>) in Listingnested
cannot be applied to (java.util.Map<java.lang.Number,java.lang.String>)
        foo(foo(m1));
            ^
```
JAVAC

```
1. ERROR in listings/Listingnested.java (at line 6)
    foo(foo(m1));
        ^^^
The method foo(Map<T,T>) in the type Listingnested is not applicable for
the arguments (Map<Number,String>)
```
EJC

```
listings/Listingnested.java:6
Method <T>foo(Map<T, T>) of type Listingnested is not applicable for the
argument of type (Map<Number, String>), because:
  [*] The type variable T is invariant, but the types:
  - String in Map<Number, String> on 5:9(5:20)
  - Number in Map<Number, String> on 5:9(5:13)
  are not the same type.
```
ours

Fig. 6 Although our error message provides more explanation, for all three compilers the type inconsistency in the inner invocation of `foo` eclipses the inconsistency in the outer invocation

```
import java.util.*;

class Listing2{
    static <T> void bar(Map<T, ? extends T> a){
        Map<Integer, Number> m2 = null;
        bar(m2);
    }
}
```

```
listings/Listing2.java:6: <T>bar(java.util.Map<T,? extends T>) in Listing2
cannot be applied to (java.util.Map<java.lang.Integer,java.lang.Number>)
        bar(m2);
        ^
```
JAVAC

```
1. ERROR in listings/Listing2.java (at line 6)
    bar(m2);
    ^^^
The method bar(Map<T,? extends T>) in the type Listing2 is not applicable for
the arguments (Map<Integer,Number>)
```
EJC

```
listings/Listing2.java:6
Method <T>bar(Map<T, ? extends T>) of type Listing2 is not applicable for the
argument of type (Map<Integer, Number>), because:
  [*] The type Number in Map<Integer, Number> on 5:9(5:22) is not a subtype of
      the inferred type for T: Integer.
```
ours

Fig. 7 A code fragment with a subtyping conflict and the three corresponding type error messages

```
import java.util.*;

class Listing6{
    <T extends Number> void foo(Map<? super T, ? super T> a){
        Map<Number, String> m = null;
        foo(m);
    }
}
```

```
listings/Listing6.java:7: <T>foo(java.util.Map<? super T,? super T>) in Listing6
cannot be applied to (java.util.Map<java.lang.Number,java.lang.String>)
        foo(m);
        ^
```

JAVAC

```
1. ERROR in listings/Listing6.java (at line 6)
    foo(m);
    ^^^
The method foo(Map<? super T,? super T>) in the type Listing6 is not applicable
for the arguments (Map<Number,String>)
```

EJC

```
listings/Listing6.java:7
Method <T extends Number>foo(Map<? super T, ? super T>) of type Listing6 is not
applicable for the argument of type (Map<Number, String>), because:
  [*] The types Number in Map<Number, String> on 5:9(5:13) and String in
      Map<Number, String> on 5:9(5:21) do not share a common subtype.
```

ours

Fig. 8 A code fragment without assignment context, so that in absence of equality and subtype constraints the supertype constraints determine the type

that is typical for supertype constraints, is due to the inability to find a single type that extends two different types. In the case of the code fragment in Fig. 8, the return type of `foo` is `void` so the assignment context cannot be used to find a proper instance for `T`. Following the JLS, the type `T` is then computed by taking the largest type that extends both `Number` and `String`. However, such a type does not exist, as our error message explains. As usual, both EJC and JAVAC simply complain that the invocation does not match the method. In Sect. 3.1 we show that these compilers sometimes exhibit strange behaviour for this kind of example.

A similar kind of situation occurs in Fig. 9, but here the cause of the problem is a subtype conflict: because of the second argument type for `Map` and the invariance of collection class type constructors, `T` will be instantiated to `Number`. However, this choice clashes with the fact that `Number` is not a subtype of `Integer`.

We continue with some examples that involve a bound conflict, starting with Fig. 10. Surprisingly, JAVAC gives exactly the same error messages as in Fig. 1, although the reason why this particular invocation fails is completely different: it is illegal, because the type that the type variable `T` should be instantiated with must be a subtype of `Number`. Since the type `Comparable<Integer>` of the actual parameter is not a subtype of `Number`, the invocation is incorrect.

A second example of a bound error can be found in Fig. 11. In this case the error message of EJC is quite reasonable, although our message does not resort to mentioning an intersection type, which is an artifact constructed by the type inference phase. Surprisingly, some versions of JAVAC crash for this particular program due to an infinite number of calls to the least upper bound function, while some other version of JAVAC generate an internal error.

```
import java.util.*;

class Listing8{
   <T> void foo(Map<? super T, T> a) {
      Map<Integer, Number> m = null;
      foo(m);
   }
}
```

```
listings/Listing8.java:6: <T>foo(java.util.Map<? super T,T>) in Listing8 cannot
be applied to (java.util.Map<java.lang.Integer,java.lang.Number>)
      foo(m);
      ^
```
JAVAC

```
1. ERROR in listings/Listing8.java (at line 6)
    foo(m);
    ^^^
The method foo(Map<? super T,T>) in the type Listing8 is not applicable for the
arguments (Map<Integer,Number>)
```
EJC

```
listings/Listing8.java:6
Method <T>foo(Map<? super T, T>) of type Listing8 is not applicable for the
argument of type (Map<Integer, Number>), because:
  [*] The type Integer in Map<Integer, Number> on 5:7(5:11) is not a supertype
      of the inferred type for T: Number.
```
ours

Fig. 9 A conflict due to the fact that `Integer` is not a supertype of `Number`, a constraint that follows from the second argument to `Map`

```
import java.util.*;

class Listing3{
    <T extends Number> void baz(List<T> a){
        List<Comparable<Integer>> x = null;
        baz(x);
    }
}
```

```
listings/Listing3.java:6: <T>baz(java.util.List<T>) in Listing3 cannot be
applied to (java.util.List<java.lang.Comparable<java.lang.Integer>>)
        baz(x);
        ^
```
JAVAC

```
1. ERROR in listings/Listing3.java (at line 6)
    baz(x);
    ^^^
Bound mismatch: The generic method baz(List<T>) of type Listing3 is not
applicable for the arguments (List<Comparable<Integer>>). The inferred type
Comparable<Integer> is not a valid substitute for the bounded parameter
<T extends Number>
```
EJC

```
listings/Listing3.java:6
Method <T extends Number>baz(List<T>) of type Listing3 is not applicable for the
argument of type (List<Comparable<Integer>>), because:
  [*] The type Comparable<Integer> in List<Comparable<Integer>> on 5:9(5:9) is
      not a subtype of T's upper bound Number in 'T extends Number'.
```
ours

Fig. 10 A code fragment with a bound conflict followed by the three corresponding type error messages

```
class JavacError1 {
  <T extends Number> void foo(T a, T b){
    foo(1, false);
  }
}
```

```
listings/JavacError1.java:3: internal error; cannot instantiate <T>foo(T,T) at
JavacError1 to (int,boolean)
    foo(1, false);
       ^
```

JAVAC

```
1. ERROR in listings/JavacError1.java (at line 3)
    foo(1, false);
    ^^^
Bound mismatch: The generic method foo(T, T) of type JavacError1 is not
applicable for the arguments (Integer, Boolean). The inferred type
Object&Comparable<?>&Serializable is not a valid substitute for the bounded
parameter <T extends Number>
```

EJC

```
listings/JavacError1.java:3
Method <T extends Number>foo(T, T) of type JavacError1 is not applicable for the
arguments of type (int, boolean), because:
  [*] The type boolean of the expression 'false' on 3:12 is not a subtype of T's
      upper bound Number in 'T extends Number'.
```

ours

Fig. 11 The value `false` cannot be considered an element of any type that extends `Number`; everything is okay for the first argument `1`

An example of a type error involving wildcards can be found in Fig. 12. The code fragment shows what might be a typical mistake on the part of a novice programmer: assuming that the type `? extends Number` equals `? extends Number` which, if provable, would make the invocation correct. However, these types are not equivalent. Our message follows the tenets of the manifesto of Yang [33] in that an error message should never reveal anything internal to compiler. However, the error messages provided by EJC and JAVAC explicitly refer to a captured wildcard.

3.1 Strange behaviour

In some cases we have observed that the compilers may behave strangely, or simply not according to the JLS.

Sometimes the type inferencer computes the largest subtype of a pair of types, and recall from the preliminaries that such a type may not always exist. This is the case for the program given in Fig. 13, in which `T` should be instantiated to the largest subtype of `Number` and `String`, which is not a valid type. Surprisingly, JAVAC accepts the program, due to the fact that it ignores the bound constraint when inferring a type for `T`.

A second example of a program that is erroneously accepted by JAVAC can be found in Fig. 14. In fact, none of the method invocations in this program should be allowed according to the JLS. Both EJC and our own compiler generate the type error messages as provided.

So why are these invocations type incorrect? The type of a generic type variable `T` for which no constraints of the form `T` = ... or ... <: `T` are generated, is determined by the type to which the result of the call to `bar` is assigned (if present). This, in the case of `bar(m1)` results in `T` being instantiated to `Integer` due to the type given for `l`. For the second call, `T`

```
import java.util.*;

class Listing4{
    <T> void bar(Map<T, T> a){
        Map<? extends Number, ? extends Number> m = null;
        bar(m);
    }
}
```

```
listings/Listing4.java:6: <T>bar(java.util.Map<T,T>) in Listing4 cannot be
applied to (java.util.Map<capture of ? extends java.lang.Number,
capture of ? extends java.lang.Number>)
        bar(m);
        ^
```

JAVAC

```
1. ERROR in listings/Listing4.java (at line 6)
    bar(m);
    ^^^
The method bar(Map<T,T>) in the type Listing4 is not applicable for the arguments
(Map<capture#1-of ? extends Number,capture#2-of ? extends Number>)
```

EJC

```
listings/Listing4.java:6
Method <T>bar(Map<T, T>) of type Listing4 is not applicable for the argument of
type (Map<? extends Number, ? extends Number>), because:
  [*] The type variable T is invariant, but the type '? extends Number' is not.
```

ours

Fig. 12 Two wildcards (even with the same bounds) are never provably the same, which is a likely reason for the inconsistency. Both JAVAC and EJC make explicit reference to captured variables, while our message does not

```
import java.util.*;

class JavacError2 {
  <T extends Number> void foo(List<? super T> a){
    List<String> x = null;
    foo(x);
  }
}
```

```
1. ERROR in listings/JavacError2.java (at line 6)
    foo(x);
    ^^^
Bound mismatch: The generic method foo(List<? super T>) of type JavacError2 is
not applicable for the arguments (List<String>). The inferred type String is not
a valid substitute for the bounded parameter <T extends Number>
```

EJC

```
listings/JavacError2.java:6
Method <T extends Number>foo(List<? super T>) of type JavacError2 is not
applicable for the argument of type (List<String>), because:
  [*] The types String in List<String> on 5:5(5:10) and Number do not share a
      common subtype. (...)
```

ours

Fig. 13 The argument type implies that `T` is `String`, which is clearly not a subtype of `Number`. Still, the program is accepted by JAVAC

```
import java.util.*;

class JavacError3{
    <T extends Number> List<T> bar(Map<? super T, ? super T> a){
        Map<List<String>, List<String>> m1 = null;
        Map<Double, Number> m2 = null;
        List<Integer> l = bar(m1);
        List<Integer> s = bar(m2);
        s = bar(m1);
        return null;
    }
}
```

```
1. ERROR in listings/JavacError3.java (at line 7)
    List<Integer> l = bar(m1);
                      ^^^
The method bar(Map<? super T,? super T>) in the type JavacError3 is not
applicable for the arguments (Map<List<String>,List<String>>)
----------
2. ERROR in listings/JavacError3.java (at line 8)
    List<Integer> s = bar(m2);
                      ^^^
The method bar(Map<? super T,? super T>) in the type JavacError3 is not
applicable for the arguments (Map<Double,Number>)
----------
3. ERROR in listings/JavacError3.java (at line 9)
    s = bar(m1);
        ^^^
The method bar(Map<? super T,? super T>) in the type JavacError3 is not
applicable for the arguments (Map<List<String>,List<String>>)
```

EJC

```
listings/JavacError3.java:7
Method <T extends Number>bar(Map<? super T, ? super T>) of type JavacError3
is not applicable for the argument of type (Map<List<String>, List<String>>),
because:
  [*] The type List<String> in Map<List<String>, List<String>> on 5:9(5:9) is
      not a supertype of the inferred type for T: Integer.

listings/JavacError3.java:8
Method <T extends Number>bar(Map<? super T, ? super T>) of type JavacError3
is not applicable for the argument of type (Map<Double, Number>), because:
  [*] The type Double in Map<Double, Number> on 6:9(6:13) is not a supertype of
      the inferred type for T: Integer.

listings/JavacError3.java:9
Method <T extends Number>bar(Map<? super T, ? super T>) of type JavacError3
is not applicable for the argument of type (Map<List<String>, List<String>>),
because:
  [*] The type List<String> in Map<List<String>, List<String>> on 5:9(5:9) is
      not a supertype of the inferred type for T: Integer.
```

ours

Fig. 14 Another code fragment erroneously accepted by JAVAC

is instantiated to `Integer`. However, in both cases, and the third one as well, the constraint derived from the invocation itself `T <: List<String>` then cannot be satisfied.

The EJC compiler also sometimes exhibits strange behaviour. Consider the code fragments in Figs. 15 and 16. The only difference between the two is in the order of the type arguments of the value passed to the method. The constraints generated for both method

```
import java.util.Map;

class Listing51{
    <T extends Number> void foo(Map<? super T, ? super T> a){
        Map<String, Number> m = null;
        foo(m);
    }
}
```

```
listings/Listing5-1.java:6: <T>foo(java.util.Map<? super T,? super T>) in
Listing51 cannot be applied to
(java.util.Map<java.lang.String,java.lang.Number>)
        foo(m);
        ^
```

JAVAC

```
1. ERROR in listings/Listing5-1.java (at line 6)
    foo(m);
    ^^^
Bound mismatch: The generic method foo(Map<? super T,? super T>) of type
Listing51 is not applicable for the arguments (Map<String,Number>). The inferred
type String is not a valid substitute for the bounded parameter
<T extends Number>
```

EJC

```
listings/Listing5-1.java:6
Method <T extends Number>foo(Map<? super T, ? super T>) of type Listing51 is not
applicable for the argument of type (Map<String, Number>), because:
  [*] The types String in Map<String, Number> on 5:9(5:13) and Number in
      Map<String, Number> on 5:9(5:21) do not share a common subtype. (...)
```

ours

Fig. 15 An inconsistency that for EJC reveals some of the underlying type checking machinery: the diagnosis given by EJC for the erroneous invocation should be the same as that for Fig. 16, but this is not the case

calls are the same according to the JLS:

$$\{\mathtt{T} <: \mathtt{Number}, \mathtt{T} <: \mathtt{String}\}$$

One would expect, therefore, the error messages for the two programs to be identical, as they are for our implementation. However, the type error diagnosis for these programs by EJC is quite different. This is due to how EJC resolves subtype constraints. Note that although the error messages generated by JAVAC are the same, neither is very informative.

Obtaining different type error messages for very similar, in some sense isomorphic, programs is an often observed but not so pleasant phenomenon. It is typically due to the fact that the implementation of the type checking process leaks through to the type error messages. Since programmers typically do not have any knowledge of the implementation, they are at a disadvantage when trying to interpret the messages. We saw two examples of this earlier: the captured wildcards (Fig. 12) and the intersection types (Fig. 11) computed by the inference process.

4 The type checking process

To be able to describe our modifications to the Java type checking process, we first discuss the original process. The modifications are then described in the next section.

```
import java.util.Map;

class Listing52{
    <T extends Number> void foo(Map<? super T, ? super T> a){
        Map<Number, String> m = null;
        foo(m);
    }
}
```

```
listings/Listing5-2.java:6: <T>foo(java.util.Map<? super T,? super T>) in
Listing52 cannot be applied to (java.util.Map<java.lang.Number,
java.lang.String>)
        foo(m);
        ^
```

JAVAC

```
1. ERROR in listings/Listing5-2.java (at line 6)
    foo(m);
    ^^^
The method foo(Map<? super T,? super T>) in the type Listing52 is not applicable
for the arguments (Map<Number,String>)
```

EJC

```
listings/Listing5-2.java:6
Method <T extends Number>foo(Map<? super T, ? super T>) of type Listing52 is not
applicable for the argument of type (Map<Number, String>), because:
  [*] The types Number in Map<Number, String> on 5:9(5:13) and String in
      Map<Number, String> on 5:9(5:21) do not share a common subtype. (...)
```

ours

Fig. 16 The inconsistency is very similar to that in Fig. 15, but nonetheless diagnosed differently by EJC

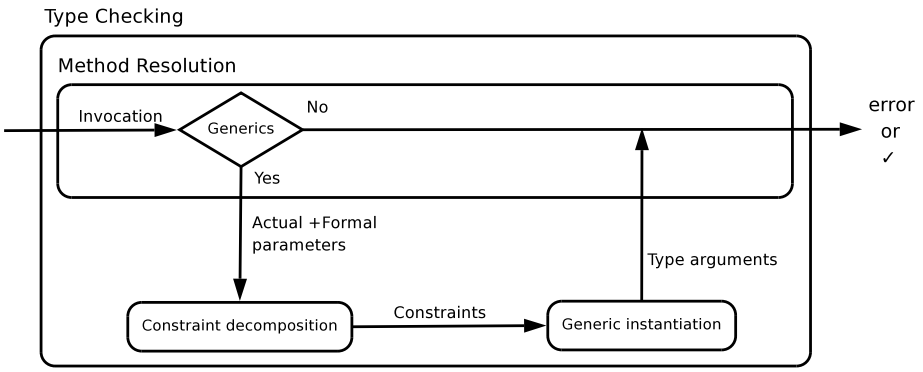

Fig. 17 The type checking process

To avoid any misunderstanding, we first explain our terminology. In this paper, *type checking process* refers to the complete process of determining the type correctness of a particular program fragment. In our particular case these program fragments are always *method invocations*.

The type checking process is depicted in Fig. 17. It starts off by performing *method resolution*, which determines, for a given invocation, a set of methods that the programmer may be invoking. We describe the complex process of method resolution in some detail below.

The set of methods obtained by method resolution may contain a number of generic methods. Pairing the concrete parameter types to the formal parameter types of a generic method:

```
<T, S> List<S> foo (Map<T, T> a,
                    List <? super S> b);
...
Map<Integer, Number> m = ...;
List<String> l = ...;
List<Integer> ret = foo(m, l);
```

results in a set of constraints

$$\{\texttt{Map<Integer, Number>} <: \texttt{Map<T, T>},\ \texttt{List<String>} <: \texttt{List<? super S>}\}.$$

that should hold if this invocation is to typecheck.

The set of constraints is subsequently decomposed into atomic constraints between type variables on the one hand and types on the other. Although there are quite a few cases to be covered, this part of the process is conceptually quite easy and is described in detail in §15.12.2.7 of [7]. Here, we give only the result of decomposition for our example:

$$\{\texttt{T} = \texttt{Integer}, \texttt{T} = \texttt{Number}, \texttt{String} :> \texttt{S}\}.$$

The type checking process then proceeds with *generic instantiation* to infer the types of the generic variables, essentially a process of finding a concrete type for each type variable. Although its name might imply otherwise, the inference process has a surprising property: if multiple, conflicting instantiations for a type variable are possible, then the inference process simply selects one, leaving it up to the later type checking phase to find that the instantiation is incorrect. The JLS states that if a conflict exists, then it will indeed not show up until in the type checking phase at the end of the type checking process. In the above example, a possible outcome of the inference phase is:

$$\texttt{T} = \texttt{Integer}.$$

In the presence of multiple supertype constraints, say

$$\{\texttt{Integer} <: \texttt{T}, \texttt{Double} <: \texttt{T}\},$$

this results in the instantiation of `T` to the *least upper bound* (*lub*) of the two, `Number`. However, things are not always so simple: for {`Integer` <: `T`, `String` <: `T`}, the lub is

```
Object & Serializable & Comparable<? extends Object &
                Serializable & Comparable<?>>,
```

because both `Integer` and `String` implement these interfaces. Note that in many compilers, not only are these types computed by the inference process, they are sometimes also used in the type error message displayed to the programmer. This contradicts one of the crucial properties that we, and others [33], believe a type checking process should have: it should only refer to types or expressions that are part of the original source program. Without any information on how the lub was computed, it can be very difficult for programmers to reconstruct what has happened and why.

The dual of the least upper bound is the *greatest lower bound* (*glb*), which is used to capture the most general type that extends both argument types (which may include classes *and* interfaces). However, if all constraints in which a type variable is involved are of this kind and the invocation occurs in an assignment context, then the JLS specifies that the type of the variable to which the result is assigned should, if possible, be used to determine to what type a type variable should be instantiated. This happens to be the case in our example for `S`. Therefore, the constraint `List<S> <: List<Integer>` is added to the constraint set. It decomposes into `S = Integer`. Together, the decomposed constraints are

$$\{\texttt{T} = \texttt{Integer}, \texttt{T} = \texttt{Number}, \texttt{S} = \texttt{Integer}, \texttt{String :> S}\}.$$

The inference process then instantiates `S` to `Integer` on the basis of the third constraint above; note that `T` was already instantiated to `Integer`.

One may wonder what happens if a declared type variable is not constrained in any way. The JLS specifies that the variable should then be instantiated to `Object`; this helps the JLS deal with legacy code.

In the final step, it is determined whether the remaining constraints, which are by now all equivalence and subtype relations between concrete types, are consistent. This part of the type checking process we call the type checking *phase*.

It is important to realize that in the type checking phase each invocation is considered *in isolation*. This even holds for nested invocations `foo(bar(x),y)`, where first the `bar` invocation will be considered in isolation from its context. It can therefore well be that the `bar` invocation type checks, but that types chosen by the inference phase turn out to be inconsistent with the enclosing call to `foo`. Or, it may be that the call to `bar` is not valid due to a case of ambiguous method invocation, but that on the basis of the type of `foo`, this ambiguity could have been resolved. By contrast, in the polymorphic lambda-calculus type information from the encapsulating call would be used to determine the proper instantiations for `bar`. This lack of propagation in Java has its advantages, as types are instantiated based on local information only and not through a long and complicated sequence of unifications. But this may also surprise the programmer, particularly in the case of an ambiguous method invocation.

4.1 Method resolution

The goal of method resolution, for a given invocation, is to come up with a single method in the program that the programmer intends to call. This decision has to be made statically, and cannot depend on run-time type information.

The process of method resolution is rather complex, due to such features as overloading (multiple methods having the same name), overriding (methods can be overridden by subclasses), variable arity methods, and visibility. Visibility depends not only on lexical scope, but also on, e.g., the access modifiers of the method. This includes the fact that you may not call a private method from certain contexts, but also that you cannot access (non-static) instance fields from a static method.

Method resolution consists of three main steps. For a given method invocation,

i. Determine the name of the method to be invoked, say `mthd`, and the class or interface that receives the invocation. Java has five different forms of method invocation. Examples respectively are `a.byteValue()`, `this.Foo()`, `super.intVal()`, `Baz.super.intVal()` and `Collections<X>.emptySet()`. In the case of the

invocation `a.byteValue()`, `byteValue()` is the name of the method, and the receiver is the innermost class or interface that encloses the method declaration (if indeed, `byteValue` is visible from the invocation site). Note that for this case alone there are two additional variants: `a.` may be omitted, and a type name may be used instead of `a`. For more details see the JLS.

ii. Consider every method of the receiver in turn to find all possible accessible and applicable method members. A method `potential` in the receiver is a candidate if and only if

- the names `potential` and `mthd` are the same,
- `potential` is accessible from the invocation site,
- if `potential` is a variable arity method of arity, say n, then the number of arguments passed to `mthd` must be greater than or equal to $n - 1$,
- if `potential` is a fixed arity method of arity n, then the number of arguments passed to `mthd` must be equal to n,
- if the method invocation includes explicit type parameters, and `potential` is a generic method, then the number of actual type parameters must equal the number of formal type parameters.

Then the compiler tries to weed out potential methods by comparing actual to formal parameters. Due to the presence of subtyping, auto-boxing, and variable arity methods, this is quite a complicated process, consisting of three alternative decision procedures. A decision procedure is only applied if all the preceding decision procedures eliminated *all* candidates. Below, we assume the method invocation is `mthd(`$A_1, \ldots, A_n$`)`.

(a) Identify methods `mthd(`$F_1, \ldots, F_n$`)`, and in which only "weakening by subtyping" is allowed to match actual argument types to formal argument types. In other words, for all $1 \leq i \leq n$:

- $A_i <: F_i$, or
- A_i is a raw type that can be parametrized into a type C_i so that $C_i <: F_i$.

If the method is generic, then all type variables in the F_i are bound to a concrete type provided by the method invocation. If such type information is unavailable then type inference, as described earlier, is used to find concrete types. The potential method is then only applicable if all instantiated type variables are within their stated bounds.

(b) Similar to the previous case, but now in combination with (un)boxing.

(c) Similar to the previous case, but now allowing also variable arity methods. Details can be found in the JLS.

If all the sets of candidates delivered by the three previous cases are empty, then no matching method exists, and an error message is produced. Otherwise, we take the first non-empty one, say S, and proceed to try to eliminate candidates until only one is left. For example, we remove from S those methods for which a more specific signature in S exists, e.g., if both `mthd(List<T>)` and `mthd (List<Integer>)` are in S, then the former is deleted. A similar but more complicated rule can be formulated for variable arity methods.

If for any pair of methods it cannot be decided which is the most specific, the compiler has a few rules to deal with this, largely by preferring non-abstract over abstract methods. In the absence of the former, an arbitrary abstract method with the most specific return type is chosen.

If that still does not work, then an ambiguous method invocation error message is generated.

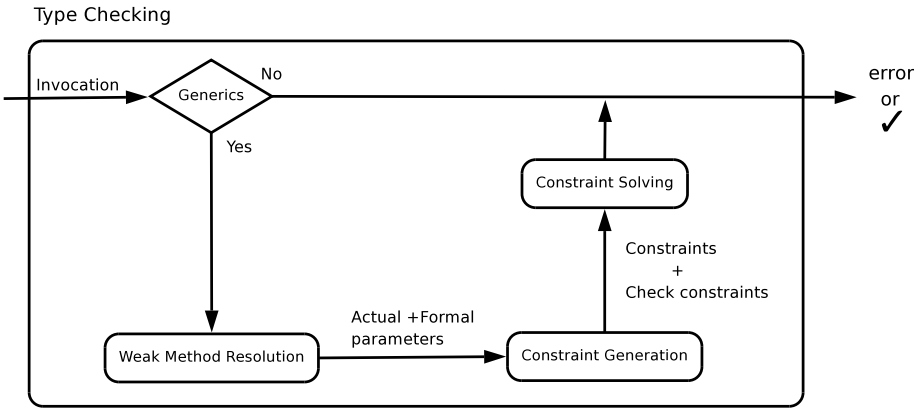

Fig. 18 The modified type checking process

iii. In the third and final step, the method chosen in the previous step is screened for appropriateness. For example, an instance method cannot be invoked from a static context.

As the reader can see, method resolution is indeed a complicated, stepwise process. In our extension of the type checking process, which we describe in the following section, we relax the constraints somewhat to try and figure out which method the programmer might have been trying to call, and to use that information in our type error message.

5 The modified type checking process

Figure 18 shows the overall architecture of our modified type checking process. The structure of the process is not much different, only the contents of the phases themselves change. It is important to realize that this modified process is only invoked after the original process has found a particular invocation to be type incorrect.

5.1 Weakened method resolution

First, we define a weaker form of method resolution that allows more candidate methods to be targeted by the method invocation. We are not interested in identifying a single method that is being called, but want to consider multiple candidate methods and provide a diagnosis for each method that is reasonably close to what the programmer intended to call.

A major decision we have to make is how exactly we weaken method resolution. Our choice here is to erase *generic* information from the invocation and the candidate method. In other words, when we are looking for a suitable method in the second step (ii) of method resolution we base our comparison between the signatures on the raw types, instead of the generic types. Conversion to raw types involves replacing type variables (and possible bounds) with `Object` and changing generic types like `List<Number>` to the raw type `List`.

The original process of method resolution aims to come up with a single method to be called (or it may fail to find one at all). Because we drop information that might be used during the original method resolution phase, the set of candidates might now be larger. This is not a problem. Remember that we already know that in the end none of these methods

will be acceptable. But since we have more candidates to pass on to the part of the process that is still to come, that part of the process may then be able to tell us more precisely *why* each of the candidates fails to qualify. In our particular case, for example, we have defined and implemented a number of heuristics [2] that work directly on the constraints derived from a particular method invocation and method definition, but these heuristics will only be applied to method definitions that pass (weakened) method resolution.

The (heuristic) assumption we make is that when it comes to the types of arguments, the generic parts of Java are more likely to contain mistakes than the raw types. This assumption derives from the fact that the generic part of Java is pretty complicated and therefore more easily misunderstood. Moreover, it is a fairly recent addition to the language, and a refinement of the raw types.

Another way to view our assumption, is that we prefer not to commit too early to certain choices because that immediately biases the type checking process, in that it becomes less likely that we shall ever blame that which we commit upon first. This indicates that, as a rule, it pays to commit first to facts in which your confidence is the highest. Since our assumption is that we should have less confidence in the generic parts of the types, it makes sense to ignore them in the early phases of the process.

Another instance of this idea can be found in work on removing the bias from Algorithm W [3]: in a standard implementation, constraints are solved (i.e., unifications are performed) while traversing the abstract syntax tree of the program (from left to right), thereby building a type substitution on the fly. So if we have multiple, inconsistent constraints for a particular type variable, the leftmost constraint is seen first and will determine the type we find for the type variable, even in cases where we find that all other constraints for that type variable say differently. In [9] it was shown how the bias can be removed by a constraint solving approach in which sets of constraints are considered simultaneously, and heuristics that work on these sets of constraints, either select the constraint(s) that are most likely to be the cause of the inconsistency, or eliminate constraints for which the opposite is the case.

The specification of weakened method resolution in the form of pseudocode can be found in Figs. 19, 20 and 21. The function `methodResolution` takes a method invocation as an argument and returns a set of most specific method declarations. The procedure starts by searching for potentially applicable methods, by considering all methods that are members of the receiver of the invocation. These are methods with the same name as the method that is invoked, and have an arity equal the number of arguments in the invocation (if the method is of fixed arity), or an arity less than the number of arguments in the invocation (if the method is of variable arity). This part is performed by the procedure `potentiallyApplicable` in Fig. 19. As explained before, three attempts will be made to come to a non-empty set of methods: first we are only allowed to match up to subtyping, then we allow method conversion (`methodConvertible`, that considers candidate methods modulo boxing, unboxing and various forms of widening, §5.3 of [7]), and finally we also allow methods of variable arity to be considered.[3]

In each case, methods will be eliminated if they are less specific than at least one other in the list. This check is implemented for the fixed arity and variable arity case in `mostSpecificMethodFixed` and `mostSpecificMethodVariable`, as given in Fig. 20. The methods behave exactly as in the original type checking process, except that they ignore any kind of generic information. The specification of the auxiliary function

[3]The construct `foreach (x, y) in (xs,ys)` iterates through the lists `xs` and `ys` in a way that if `x` is the *i*th element of `xs`, then `y` is the *i*th element of `ys`.

```
[Method] methodResolution(invocation)
  potentialMethods = [m in invocation.receiver
                      | potentiallyApplicable(m,invocation)]
  specificMethods = [m in potentialMethods
                     | m of fixed arity and
                       applicableBySubtyping(m,invocation)]
  mostSpecificMethods = mostSpecificMethodFixed(specificMethods)
  if empty mostSpecificMethods then
    methodConvertedMethods = [m in potentialMethods
                              | m of fixed arity and
                                applicableByMethodConversion(m,invocation)]
    mostSpecificMethods = mostSpecificMethodFixed(methodConvertedMethods)
    if empty mostSpecificMethods then
      methodVarArity = [m in potentialMethods
                        | m of variable arity and
                          applicableByMethodConversion(m,invocation)]
      mostSpecificMethods = mostSpecificMethodVariable(methodConvertedMethods)
  return [m in mostSpecificMethods | m appropriate]

boolean potentiallyApplicable(method, invocation)
  return
     method.name == invocation.name and
     method accessible from invocation.location and
     (method.arity == length(invocation.arguments) or
      (method of variable arity and
       length(invocation.arguments) >= method.arity-1))

boolean applicableBySubtyping(method, invocation)
  foreach (parameter,argument) in (method.arguments,invocation.arguments) do
    erasedParameter = wideErasure(parameter)
    if (argument <: erasedParameter) then
      continue
    else
      return false
  return true

boolean applicableByMethodConversion(method, invocation)
  (* Works both for variable and fixed arity methods *)
  k = method.arity-1
  for i = 1 to k
    erasedParameter = wideErasure(method.arguments[i])
    argument  = invocation.arguments[i]
    if methodConvertible(argument, erasedParameter) then
      continue
    else
      return false
  (* The "variable" part *)
  erasedParameter = wideErasure(method.arguments[k+1]
  for i = k+1 to n
    argument   = invocation.arguments[i]
    if methodConvertible(argument, erasedParameter) then
      continue
    else
      return false
  return true
```

Fig. 19 Pseudocode for weakened method resolution (part 1)

```
Type wideErasure(tp)
  if tp is a type variable then
     return Object
  else
     if tp is generic then
        return raw form of tp
     else
        return tp

[Method] mostSpecificMethodFixed(methods)
  foreach m1 in methods
    foreach m2 in methods
      if moreSpecificFixed(m1,m2) then
        methods.delete(m2)

[Method] mostSpecificMethodVariable(methods)
  foreach m1 in methods
    foreach m2 in methods
      if moreSpecificVariable(m1,m2) then
        methods.delete(m2)
```

Fig. 20 Pseudocode for weakened method resolution (part 2)

`moreSpecificVariable` in Fig. 21 is rather lengthy, because we have to deal with the fact that the number of parameters can be unequal in two ways, but that the subtype relation is independent of the respective lengths. Note that we could have abstracted away from the particular type comparison operator in order to save code, but for reasons of clarity and correspondence with the JLS (§15.12.2.5) we preferred to write it out.

After having constructed a set of most specific methods, `methodResolution` will remove any method declaration that is not appropriate from the set (e.g. not accessible from the call site), and returns the set of remaining methods.

We illustrate the process by means of an example. Consider the code in Fig. 22 where we define a many-times overloaded method `foo` (we designate `foo` on line x by $\texttt{foo}_x$). Consider first the invocation on line 20. The method $\texttt{foo}_2$ does not qualify as a candidate because it has the wrong number of parameters. All the other declarations have the right number of parameters, so they are marked as candidates in step (i). Their signatures are converted into their raw form, and we obtain

$$\texttt{foo}_4(\texttt{Map},\texttt{Collection}),\quad \texttt{foo}_7(\texttt{Map},\texttt{List}),\ \texttt{foo}_{10}(\texttt{HashMap},\texttt{List}),\quad \texttt{foo}_{13}(\texttt{HashMap},\texttt{LinkedList}),\quad \texttt{foo}_{16}(\texttt{HashMap},\texttt{Set}).$$

In the absence of primitive types and variable arity methods, the applicable methods can be determined using subtyping only, i.e., we only need to look at case (a) of step (ii) in the method resolution phase. The second parameter `LinkedList` in the invocation is not a subtype of the generic interface type `Set`. Therefore, $\texttt{foo}_{16}$ is disqualified as a candidate. Next, our weak method resolution reduces the set of applicable methods

$$\{\texttt{foo}_4, \texttt{foo}_7, \texttt{foo}_{10}, \texttt{foo}_{13}\}$$

to a set of most specific methods. Comparing $\texttt{foo}_4$ with $\texttt{foo}_7$ results in the removal of $\texttt{foo}_4$, because $\texttt{List} <: \texttt{Collection}$. Similarly, $\texttt{foo}_7$ and $\texttt{foo}_{10}$ are removed in favour of $\texttt{foo}_{13}$. For the second and third invocation in Fig. 22 the same set of candidates is obtained.

```
boolean moreSpecificFixed(m1,m2)
  (* In case both methods are of fixed arity *)
  foreach (paramm1,paramm2) in (m1.arguments, m2.arguments) do
    paramm1 = wideErasure(paramm1)
    paramm2 = wideErasure(paramm2)
    if (paramm1 <: paramm2) then
      continue
    else
      return false
  return true

boolean moreSpecificVariable(m1,m2)
  (* In case both methods are of variable arity *)
  if m1.arity >= m2.arity then
     k = m2.arity - 1
     m = m1.arity
     for i = 1 to k
       paramm1 = wideErasure(m1.arguments[i])
       paramm2 = wideErasure(m2.arguments[i])
       if (paramm1 <: paramm2) then
         continue
       else
         return false
     paramm2 = wideErasure(m2.arguments[k+1])

     for i = k+1 to n
       paramm1 = wideErasure(m1.arguments[i])
       if (paramm1 <: paramm2) then
         continue
       else
         return false
     return true
  else
     k = m1.arity - 1
     m = m2.arity
     for i = 1 to k
       paramm1 = wideErasure(m1.arguments[i])
       paramm2 = wideErasure(m2.arguments[i])
       if (paramm1 <: paramm2) then
         continue
       else
         return false
     paramm1 = wideErasure(m1.arguments[k+1])

     for i = k+1 to n
       paramm2 = wideErasure(m2.arguments[i])
       if (paramm1 <: paramm2) then
         continue
       else
         return false
     return true
```

Fig. 21 Pseudocode for weakened method resolution (part 3)

In all cases, we ended up with a singleton set, but our resolution method does not demand this, contrary to the original resolution method. For example, for the code fragment of Fig. 28, which we shall discuss in more detail later, both declarations of `bar` pass method resolution.

```
1  class FooLib{
2     <T> void foo(Map<T, ? extends T> a){}
3
4     <T> void foo(Map<T, ? extends T> a,
5                  Collection<? super T> b){}
6
7     <T> void foo(Map<T, ? extends T> a,
8                  List<? super T> b){}
9
10    <T> void foo(HashMap<T, ? extends T> a,
11                 List<? super T> b){}
12
13    <T> void foo(HashMap<T, ? extends T> a,
14                 LinkedList<? super T> b){}
15
16    <T> void foo(HashMap<T, ? extends T> a,
17                 Set<? super T> b){}
18 }
19 ...
20 UtilLib.foo(new HashMap<Integer, Integer>(),
21              new LinkedList<Number>());
22 UtilLib.foo(new HashMap<Double, Number>(),
23              new LinkedList<Integer>());
24 LinkedList<? extends Number>  wl = ...;
25 UtilLib.foo(new HashMap<Number, Double>(), wl);
```

Fig. 22 A utility class for illustrating weakened method resolution

```
<T> void foo(List<T> a, List<? super T> b){
...
List<Number> l1 = ...;
List<? extends Number> l2 = ...;
foo(l1, l2);
```

Fig. 23 Inference succeeds to find instantiations for all variables, but then checking fails

5.2 Constraint generation

Constraint generation is our version of the phase of constraint decomposition. Recall that the original type inference phase does not check for inconsistencies. Inconsistencies are discovered later during the type checking phase. This choice leads to type error messages that cannot explain very well what the problem is, because information has been lost between the type inferencing and type checking phase.

Consider the code fragment in Fig. 23. Here, the type parameter `T` is instantiated to `Number`, because `l1` is passed as the first argument. But unfortunately, `List<? extends Number>` is not a subtype of `List<? super Number>`. In this situation, an implementation based on the JLS will typically say that `foo` cannot be applied to the variables `l1` and `l2`, but it cannot for example explain to the programmer why the error occurred or how to fix it: it does not have enough knowledge to do so.

In our extension, we provide the constraint solver, introduced in detail below, with more constraints that will ensure that a type is inferred only if all type constraints are satisfied and no type-checking error will occur. To that purpose, the original constraint decomposition algorithm is extended to keep track of constraints that were left alone during decomposition,

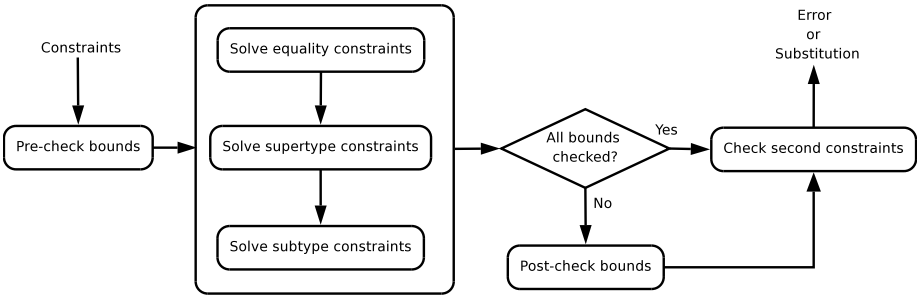

Fig. 24 The constraint solving phase

because they did not match any of the cases of the constraint decomposition procedures. In fact, the new procedures can be obtained from the old ones (based on §15.12.2.7 of [7]), simply by adding cases that store the non-atomic constraints for which no case is defined, into a special set of constraints. In the following we therefore keep track of two sets of constraints X and Y by writing $X \Cup Y$. Here, the left operand X contains the decomposed constraints, and Y contains those constraints that could not be decomposed.

For the example in Fig. 23, the new constraint generation algorithm will collect the following constraints:

$$\{\mathtt{T} = \mathtt{Number}\} \Cup$$

$$\{\mathtt{List<?\ extends\ Number>} <: \mathtt{List<?\ super\ T>}\}$$

Coming back to the example of Fig. 22, we generate constraints for the invocation of `foo` at line 20 to type check. The only remaining candidate method is $\mathtt{foo}_{13}$, in which case we obtain

$$\{\mathtt{T} = \mathtt{Integer}, \mathtt{Integer} <: \mathtt{T}, \mathtt{T} <: \mathtt{Number}\} \Cup \emptyset$$

For the second invocation we obtain

$$\{\mathtt{T} = \mathtt{Double}, \mathtt{Number} <: \mathtt{T}, \mathtt{T} <: \mathtt{Integer}\} \Cup \emptyset$$

and for the third

$$\{\mathtt{T} = \mathtt{Number}, \mathtt{Double} <: \mathtt{T}\} \Cup$$

$$\{\mathtt{LinkedList<?\ extends\ Number>} <: \mathtt{LinkedList<?\ super\ T>}\}$$

5.3 The constraint solving phase

In Fig. 24 we summarize the constraint solving phase. In the pre-check for bounds, we check that all types in the atomic constraints of a type parameter `T` satisfy the bounds of `T`. If not, a type error message is generated for each failed check. Then it infers the instantiations for every type variable, based on the decomposed constraints (the left operand of $\Cup$). This either results in a substitution or, in case of failure, a list of type error messages. If there are still bounds for `T` left unchecked, e.g., they involve also other type variables, then these bounds checks are performed next. Finally, we verify that the non-atomic constraints (the right operand of $\Cup$) are satisfied.

```
<T, S extends T> void foo(Map<S, S> a, T a){
...
Map<Integer, String> m = ...;
foo(m, 1);
```

Fig. 25 The order of inference for type variable matters

The reason for doing the pre-check is best illustrated by an example. Consider the following set of constraints

$$\{\texttt{String} <: \texttt{T}, \texttt{Integer} <: \texttt{T}, \texttt{T} <: \texttt{Number}\}.$$

The original algorithm instantiates `T` to `Object`, the lub of `String` and `Integer`.[4] Later, during type checking it finds that `Object` is not a subtype of `Number`. Since the type checker does not have information available about how `T` got its type, it cannot really say what went wrong. If on the other hand, the bound had been checked immediately (or alternatively, information about the inference of `T` had been retained), we would have found that `String` $<:$ `T` is not consistent with `T` $<:$ `Number`; and choosing any type that is a supertype of `String` is not going to help. In other words, for constraints of the given form, it can be determined at an early stage that an inconsistency will result, and a type error message can be generated immediately.

The second modification we made is to tune the order in which type inference instantiates the type variables. It is well known that for the polymorphic lambda-calculus the different implementations of the type system solve constraints in different orders and that the order influences the error message the implementations provide [12, 18]. In our inference algorithm, type variables are considered separately, but because we involve the bounds constraints at an early stage, the inferred type for a particular type variable may impact that of another.

To illustrate, consider the code fragment in Fig. 25. If we first infer the type of `S`, then we cannot exploit the information that the bound `S extends T` might give us. On the other hand, if we first infer `T` (to be `Integer`, obviously), then we obtain an additional pre-check bound for `S`: `S` $<:$ `Integer`. During the pre-check we can then establish that the constraint `S` $=$ `Integer` is consistent with the bound and `S` $=$ `String` is not.

It should not be a surprise, therefore, that we have chosen the *degree heuristic* for determining the order of inference [26]. This heuristic selects first the type variables that determines the largest number of other variables. Consider the following generic parameter:

```
<T, S extends T, R, U extends Map<R, S> &
  T, V extends Map<U, T>>
```

From this generic parameter we can construct the graph in Fig. 26 as follows: the type variables form the vertices of the graph, and we have a solid edge from `S` to `T` if `T` occurs in a bound for `S`. The dashed edges in the graph are edges that additionally result from taking the transitive closure of the direct dependencies. The degree heuristic now specifies that we should first instantiate the type variable that has the highest number of incoming edges, whether solid or dashed. The reasoning is that an instance for such a type variable provides the most information, i.e., the largest number of type variables can potentially profit from the additional information. Hence, in the example type inference should start by inferring `T`.

[4] Actually, the type is somewhat more complicated, but never mind that now.

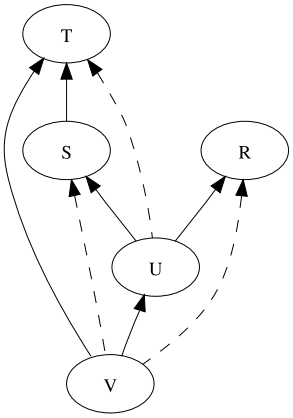

Fig. 26 A type variable dependency graph, with *solid edges* denoting a direct dependency, and *dashed edges* for indirect ones

```
if empty E then
  if empty P then
    if empty B then
      set T = Object
    else
      set C = constraints from the context
      if T inferred on the basis of union(B,C)
        set T = inferred type
      else
        generate error message
  else
    set A = { alpha | alpha <: T in P }
    set T = lub(A)
    if B are satisfied then
      okay
    else
      generate error message
else
  if all constraints in E of the form T = X then
    set T = X
    if union(P, B) are satisfied then
      okay
    else
      generate error message
  else
    generate error message
```

Fig. 27 A pseudocode algorithm for type inference

5.4 The inferencer

The large rounded rectangle in Fig. 24 is the core of the constraint solving phase, where inference takes place. The algorithm processes the type variables one at the time, in the order obtained using the dependency graph, as described in Sect. 5.3.

Suppose we now deal with type variable `T`, and `E`, `P` and `B` contain the type equality, supertype and subtype constraints involving `T`, respectively. Then Fig. 27 gives the pseudocode to describe the inference process.

We conclude this section by revisiting the running example of Fig. 22.

For the first invocation we find

$$\{\texttt{T} = \texttt{Integer}, \texttt{Integer} <: \texttt{T}, \texttt{T} <: \texttt{Number}\} \uplus \emptyset.$$

In this case, there is a single equality constraint for `T`, so we infer `T` to be `Integer` and proceed to verify the remaining constraints: the supertype and subtype constraints turn out to be satisfied as well. Because the set of non-atomic constraints is empty, and therefore trivially satisfied, the invocation on line 20 invokes the method on line 13 correctly and unambiguously.

For the second invocation we obtained

$$\{\texttt{T} = \texttt{Double}, \texttt{Number} <: \texttt{T}, \texttt{T} <: \texttt{Integer}\} \uplus \emptyset.$$

The type variable `T` is inferred to be `Double`, but in this case both subtype and supertype constraints fail to be satisfied. Hence an error message is generated.

Finally, for the third

$$\{\texttt{T} = \texttt{Number}, \texttt{Double} <: \texttt{T}\} \uplus$$

$$\{\texttt{LinkedList<? extends Number>} <: \texttt{LinkedList<? super T>}\}$$

we infer `T` to be `Number`, and since `Double` $<:$ `Number`, the constraints are satisfied. However, the non-atomic constraint is not satisfied, so the method invocation fails to type check.

For a somewhat different example, consider the code fragment in Fig. 28. Because in both cases the names match, and the number of formal parameters matches the number of actual arguments in the call, both methods are considered candidates and will be considered further. Since weak method resolution ignores all generic information, both method signatures are implicitly converted to `void bar(Object a, Object b)`. Because the arguments to the call happen to be of primitive type, weak method resolution will attempt to match the call to a method by means of method invocation conversion (§5.3 of [7]). This results in both methods to be considered applicable. Then the type checking process will consider each method in turn to determine which of these, if any, matches the call.

Let us consider the first definition of `bar`. Matching the types of the invocation with that of the first definition of `bar` results in the following constraints, in which the primitive types are automatically promoted to their corresponding reference type:

$$\{\texttt{Character} <: \texttt{T}, \texttt{Double} <: \texttt{T}\} \uplus \emptyset.$$

Because only one type variable is involved, the ordering phase for type variables can be ignored, and we proceed to perform bounds checking. Bounds checking ensures that for all types `S` with `S` $<:$ `T`, that `S` is a subtype of all the types `U` that bound `T` from above. Choices for `S` in this particular case are `Double` and `Character`, the only possibility for `U` is `Number`. The combination `Double` and `Number` is fine, but since `Character` $\not<:$ `Number`, the method does not match. Notwithstanding, the process does set `T` equal to the lub of `Double` and `Character`. This type is not returned as the type for `T`, but, instead, used to further uncover potential type conflicts. For example, if there would be another constraint on `T`, say `T` $<:$ `Integer`, then this may help establish more firmly that the types used in computing the lub to obtain `T` are not the right ones. When we provide an error message to fix the problem, we can then avoid to suggest fixing it in a way that in the next compile a clash with the constraint `T` $<:$ `Integer` can occur.

```
class BarUtil{
   static <T extends Number>void bar(T a, T b){}
   static <T extends Integer>void bar(T a, T b){}
   void dummy() {
     BarUtil.bar('0', 3.14);
   }
}
```

```
listings/Listing7.java:5: internal error; cannot instantiate <T>bar(T,T) at
BarUtil to (char,double)
     BarUtil.bar('0', 3.14);
                 ^
```

JAVAC

```
1. WARNING in listings/Listing7.java (at line 3)
    static <T extends Integer>void bar(T a, T b){}
                       ^^^^^^^
The type parameter T should not be bounded by the final type Integer. Final
types cannot be further extended
----------
2. ERROR in listings/Listing7.java (at line 5)
    BarUtil.bar('0', 3.14);
            ^^^
Bound mismatch: The generic method bar(T, T) of type BarUtil is not applicable
for the arguments (Character, Double).
The inferred type Object&Serializable& Comparable<?> is not a valid substitute
for the bounded parameter <T extends Number>
```

EJC

```
listings/Listing7.java:5
Method <T extends Number>bar(T, T) of type BarUtil is not applicable for the
arguments of type (char, double), because:
  [*] The type char of the expression ''0'' on 5:18 is not a subtype of T's
      upper bound Number in 'T extends Number'.

listings/Listing7.java:5
Method <T extends Integer>bar(T, T) of type BarUtil is not applicable for the
arguments of type (char, double), because:
  [*] The type char of the expression ''0'' on 5:18 is not a subtype of T's
      upper bound Integer in 'T extends Integer'.
  [*] The type double of the expression '3.14' on 5:23 is not a subtype of T's
      upper bound Integer in 'T extends Integer'.
```

ours

Fig. 28 A code fragment with two candidate methods, both of which fail to qualify. JAVAC and EJC provide only one message, but our system provides a diagnosis for both failures

For the second definition of `bar`, we similarly obtain an inconsistent set of constraints. In this case `U` ranges over the set {`Integer`}, and neither the constraint `Double <: T` nor the constraint `Character <: T` can be satisfied. For both method definitions an error message will be provided by our implementation, as seen in Fig. 28. Both EJC and JAVAC give only a single type error message for the invocation. It is not made clear why the method call matches neither, and, in the case of JAVAC it is even not clear to which of the two method definitions the method invocation has been compared.

As an aside, EJC also warns that it makes little sense to have `T` extend a final class. This is quite similar to one of the heuristics we describe in [2]. However, before we would give such a hint, we make sure that replacing `T` by its upper bound actually solves the problem. Since that is not the case here, our implementation does not provide such a hint.

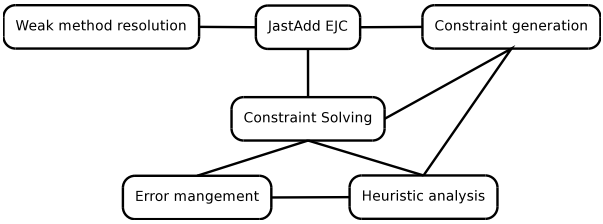

Fig. 29 Architecture of our extension to the JastAdd EJC

6 Implementation

We have implemented our work as an extension to the JastAdd Extensible Java Compiler (JastAdd EJC) [6], which in turn was built on top of JastAdd [11]. The latter is an attribute grammar compiler that allows specifying compiler semantics in an aspect-oriented way by means of declarative attributes and semantic rules using ordinary Java code. Some more information on this implementation can be found in Appendix.

For the convenience of the weak method resolution, the ordering of type variables and the computation of greatest lower bound have been implemented using JastAdd. We have contributed the module that we have developed for computing the greatest lower bound to the maintainers of JastAdd EJC; it has been added to the repository.

The architecture of the resulting compiler is shown in Fig. 29: the type checker sends a method invocation which fails to typecheck to the weak method resolution which returns a set of methods. The type checker then generates type constraints for the method invocation and each method declaration using the constraint generation algorithm described in Sect. 5. The type checker passes these constraints along to our constraint solver together with the return type of the method declaration under consideration and the type of the value if the invocation appears in an assignment context. The constraint solver will then solve the constraints and return an error message to the type checker if the constraints are unsatisfiable. The error messages returned by the constraint solver are maintained and collected by a separate error manager. This is mainly to facilitate the implementation of a number of heuristics that we use to suggest fixes for the type error [2].

Although we have only discussed method invocations, our extension already gives some limited support for constructor invocations.

7 Related work

Thus far, work on improving type error diagnosis has concentrated on strongly typed, higher-order, polymorphic functional languages, such as ML and Haskell. Indeed, one of the reasons we set out to do the current study was to see how far our experience in improving type error diagnosis for Haskell would help us in a different setting.

The PhD thesis of Bastiaan Heeren [12] compiles a number of papers with the second author and includes an extensive bibliography on improving type error messages for functional languages. We mention a few of the more important papers. We omit [34], which was already discussed in the introduction.

Heeren categorises papers based on their approach. The simplest approach to improve type error messages is to change the order in which unifications take place, i.e., constraints are solved, because that will change the unification that will be held responsible for the inconsistency [10, 17, 18, 20].

The second category consists of explanation systems, which in one way or another keep track of how a conclusion about a particular type was derived in order to provide that explanation to the programmer [5, 9, 32, 33, 35]. We make in particular note of Yang et al. [35], because at its basis lies algorithm $\mathcal{H}$, which was inspired by how human subjects (in an experimental setting) perform type checking. In particular, they discovered that human subjects focus on the concrete types derived from the use of literals, and make heavy use of the two-dimensional inspection of code. Contrary to how most algorithms work, humans avoid type variables when explaining the type of an expression.

A potential problem of explanation systems is that explanations can quickly become very verbose, because type information can potentially propagate to all corners of the program. This is less of a problem in the context of Generic Java, because each method invocation is considered in isolation; reasoning is thereby much more localised.

Repair systems try to isolate a particular cause of a problem, suggesting that this part of the program should be changed to remedy the mistake. Particular examples are [2, 9, 21, 31]. The danger of these systems is that they are heuristic: there are always situations where they will suggest the wrong fix. Thus, some researchers prefer to report a set of program locations, usually with some guarantee that no location outside this set can be responsible for the problem. Such a set forms a program slice, and such approaches are said to perform type error slicing [4, 8, 25]. It is still unknown, however, whether such a system can be useful in practice as there is not enough information yet on how large slices can become, and how well the system works in the presence of multiple independent and/or somewhat related mistakes. We believe that the combination of type error slicing with type error repair to be very promising, but we have not yet seen any substantial work in this direction.

A final category consists of systems that allow the programmer to interactively investigate the types of expressions in the program. Such a system can usually give more precise feedback, because from the interaction it may obtain more information from the programmer about his/her intentions. The most well-known attempts in this direction are Typeview [14] and Chameleon [28, 29].

Characteristic of all these attempts is that the authors redesign the type inference process to come up with better error messages. A totally different approach can be found in [19], which uses a Caml compiler as a black box, and presents programmers with complete program fixes for parts of their program. This is done by enumerating variations on the faulty program and submitting these to the compiler to decide on type correctness.

We consider the current paper to belong to the category of explanation systems, although our explanations typically will not be long. Our work also has some elements from [19]: although our work is implemented directly in a Java compiler, we do leave the original type inference process intact.

What sets our work somewhat apart from the literature above is that we need to deal with subtyping and overloading. Because a programmer may be attempting to call any number of methods, our method compares the invocation to multiple method definitions, and for each describes why the call is inconsistent with that method. For the languages that are the subject of the papers above, such a situation never occurs. Moreover, our work is complicated by the fact that instead of the elegant Hindley-Milner type system for the polymorphic lambda-calculus [22], we have to cope with the large, operational specification of Java's type system, which actually coincides with its type checking process.

We are not aware of any other work on improving type error messages for languages besides functional ones, except for el Boustani and Hage [1], which is a shorter version of the current paper, and a follow-up paper that deals with heuristics that can offer suggestions on how to fix the type error [2]. Changes with respect to the former can be summarized as

follows: since we have fewer restrictions on length we have recompiled all the programs with the three tools and have included all the outcomes without changing the content of the messages. We made one exception: our implementation also sometimes suggests how to change the program to get rid of a type error. Since these suggestions are the subject of [2], we have omitted these from our messages, and replaced them with `(...)`. We have also added quite a few examples including a realistic one, and a related work section. Finally, we describe more details of the processes involved.

Somewhat related to our work is that of Jadud who performed an extensive study of parse errors and compiler usage for Java [15].

8 Conclusions, reflections and future work

We have described how the type checking process of Generic Java can be extended to provide more informative type error messages, particularly for method invocations that involve generics. We have illustrated our work by a sizable number of examples and have made a download available in which our work is implemented as an extension to the JastAdd Extensible Java Compiler.

Our work follows three design principles: Principle number one is to leave the (very complicated) type checking process intact, because any modification of the process risks changing the set of typable programs. The second principle is to avoid acting on what you trust the least, e.g., a major hurdle to obtain good type error messages when following the JLS, is that candidate methods may be disqualified at an early stage of the type checking process. In this paper, we chose to ignore the generic parts of types, based on the assumption that this is where programmers make the most mistakes. The third and final principle is to hold on to type information longer than is necessary for following the JLS. In our particular case, we kept some of the original constraints around to make a more informed decision on the causes of the inconsistencies.

There are plenty of directions for future work. The first is to perform a more global analysis to come up with an even better estimate of what might be the mistake. For example, type inference is highly compartmentalized in the JLS: each method invocation is considered more or less in isolation and independent of others. However, if a lack of understanding of this particular fact is the reason for a mistake, we can only find that out by going beyond these compartments. This is an often observed pattern, and one that makes type error diagnosis significantly more complicated than type checking by itself: to find out what kind of faulty reasoning on the part of the programmer is responsible for a type error, the improved type inference process must be able to distinguish between these forms of reasoning, and therefore be able to maintain and manipulate what, according to the original type system, is inconsistent information.

Beyond method invocations, a particularly interesting and complicated language construct is that of inner classes. In that case, we have the additional problem of dealing with the scope/shadowing of type variables, and all the mistakes programmers can make in these situations.

Furthermore, although we have weakened method resolution somewhat so that we may determine the method the programmer might have wanted to invoke, there are plenty of variations left unconsidered: why not also consider methods that are not visible or accessible and suggest to modify the program so that they become visible and accessible? There is, in fact, a huge number of possibilities here, and thus far we have barely scratched the surface.

To have some idea in which direction to look it would be really helpful to know what kind of mistakes programmers make. Program logging systems like BlueJ might be able to

help us there [15]. Having consulted Jadud on the subject, however, we have learned that the programs he has collected do not contain generics. One promising idea, suggested by Pierre-Evariste Dagand in private communication, is to offer our system as a web-based service for Java programmers, and, as part of the service, ask them to rate the output of our system and that of the standard compilers. As a bonus, we obtain a collection of programs that we can use to validate and improve on our work. Recently, we found another valuable source of information which is Angelika Langer's FAQ on Java Generics [16]. Based on this FAQ, we plan to perform a systematic study of the kind of mistakes programmers make, and misunderstandings they might have about Java Generics. As a more labour-intense alternative to the two previous two, the empirically based route followed in [35] could be explored to find out, e.g., what kind of mistakes programmers make, and how they themselves derive the types of expressions in their program. The work presented here can serve to prepare the way for such further studies.

We note that our work takes the JLS as a starting point, and we have yet to consider alternative approaches to combining generics with subtyping, as part of, e.g., Scala [24] and Timber [23]. However, the second author, together with the developers of the Timber language, has recently started to look at the latter language, as an example of a language that adds subtyping to a language based on the polymorphic lambda-calculus.

Finally then, in [27] it is shown how the generics of Java might be "fixed" to obtain a process that is sound and complete. Although soundness and completeness are clearly important issues, we believe that intuitiveness and elegance of the type system is important too, particularly for a language that may well be the first programming language novice programmers encounter. We therefore hope that any fix will take those properties into consideration as well.

Acknowledgements We acknowledge the involvement of Martin Bravenboer during the early stages of this project, and thank the anonymous referees for their suggestions and corrections.

Appendix: Using the system

To compile the system you need `subversion` (http://subversion.tigris.org/) and `Ant` obtainable from http://ant.apache.org. Once you have these installed on your system, checkout the repository

```
staff.jur.javagenerics/project
```

on the subversion server located at

```
https://subversion.cs.uu.nl/
```

(click on `info` to get the exact location for the checkout). The `README` file that you obtain in the process explains how to proceed: run `ant` in the `Java1.5Backend` directory, and afterwards proceed to the `bin` directory where the invocation `java JavaCompiler -help` tells you how the compiler should be invoked. The subdirectory `testing` contains a large number of example programs on which to try out the compiler. Most of these programs also explain in comments which constraints are generated and how these are used to determine type (in)correctness. Note that many error messages also suggest a problem fix

using heuristics; that part of our work is discussed in another paper [2]. The examples in the current paper have all been included in the special subdirectory `testing/listings`. To compile, e.g., `Listing1.java` simply write the following at the command prompt while inside the `testing` subdirectory:

```
java -cp ../bin JavaCompiler -d /tmp listings/Listing1.java
```

References

1. el Boustani, N., Hage, J.: Improving type error messages for Generic Java. In: Puebla, G., Vidal, G. (eds.) Proceedings of the ACM SIGPLAN 2009 Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'09), pp. 131–140. ACM Press, New York (2009)
2. el Boustani, N., Hage, J.: Corrective hints for type incorrect Generic Java programs. In: Gallagher, J., Voigtländer, J. (eds.) Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM'10), pp. 5–14. ACM Press, New York (2010)
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982, pp. 207–212. ACM Press, New York (1982)
4. Dinesh, T.B., Tip, F.: A slicing-based approach for locating type errors. In: 264, Centrum voor Wiskunde en Informatica (CWI), p. 24 (1998). ISSN 1386-369X
5. Duggan, D., Bent, F.: Explaining type inference. Sci. Comput. Program. **27**, 37–83 (1996)
6. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA'07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, pp. 1–18. ACM, New York (2007)
7. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional, Reading (2005)
8. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. In: Proceedings of the 12th European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 2618, pp. 284–301. Springer, Berlin (2003)
9. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) Implementation of Functional Languages—IFL 2006, vol. 4449, pp. 199–216. Springer, Heidelberg (2007)
10. Hage, J., Heeren, B.: Strategies for solving constraints in type and effect systems. Electron. Notes Theor. Comput. Sci. **236**, 163–183 (2009). Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008)
11. Hedin, G., Magnusson, E.: The JastAdd system—an aspect-oriented compiler construction system. Sci. Comput. Program. **47**(1), 37–58 (2003). http://www.cs.lth.se/~gorel/publications/2003-JastAdd-SCP-Preprint.pdf
12. Heeren, B.: Top quality type error messages. PhD thesis, Universiteit Utrecht, The Netherlands (2005). http://www.cs.uu.nl/people/bastiaan/phdthesis
13. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Eighth International Conference on Functional Programming, pp. 3–13. ACM Press, New York (2003)
14. Huch, F., Chitil, O., Simon, A.: Typeview: a tool for understanding type errors. In: Mohnen, M., Koopman, P. (eds.) Proceedings of 12th International Workshop on Implementation of Functional Languages (IFL 00). Aachner Informatik-Berichte, pp. 63–69 (2000)
15. Jadud, M.C.: A first look at novice compilation behaviour using BlueJ. Comput. Sci. Educ. **15**(1), 25–40 (2005)
16. Langer, A.: Java generics FAQs—frequently asked questions (2008). http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html
17. Lee, O., Yi, K.: Proofs about a folklore let-polymorphic type inference algorithm. ACM Trans. Program. Lang. Syst. **20**(4), 707–723 (1998)
18. Lee, O., Yi, K.: A generalization of hybrid let-polymorphic type inference algorithms. In: Proceedings of the First Asian Workshop on Programming Languages and Systems, pp. 79–88. National University of Singapore, Singapore (2000)
19. Lerner, B., Grossman, D., Chambers, C.: Seminal: searching for ml type-error messages. In: ML'06: Proceedings of the 2006 Workshop on ML, pp. 63–73. ACM, New York (2006)
20. McAdam, B.J.: On the Unification of Substitutions in Type Inference. In: Hammond, K., Davie, A.J.T., Clack, C. (eds.) Implementation of Functional Languages (IFL'98), London, UK. LNCS, vol. 1595, pp. 139–154. Springer, Berlin (1998)

21. McAdam, B.J.: How to repair type errors automatically. In: Hammond, K., Curtis, S. (eds.) Trends in Functional Programming, Intellect, Bristol, UK, vol. 3, pp. 87–98 (2002)
22. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**, 348–375 (1978)
23. Nordlander, J., Carlsson, M., Gill, A., Lindgren, P., von Sydow, B.: The Timber homepage (2008). http://www.timber-lang.org
24. Odersky, M.: The Scala homepage (2008). http://www.scala-lang.org/
25. Rahli, V., Wells, J.B., Kamareddine, F.: A constraint system for a SML type error slicer. Tech. Rep. HW-MACS-TR-0079, Herriot Watt University, Edinburgh, Scotland (2010)
26. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn., pp. 137–151. Pearson Education, Upper Saddle River (2003). Chap. 5
27. Smith, D., Cartwright, R.: Java type inference is broken: can we fix it. In: Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08), pp. 505–524. ACM, New York (2008)
28. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Haskell'03: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 72–83. ACM Press, New York (2003)
29. Stuckey, P.J., Sulzmann, M., Wazny, J.: Improving type error diagnosis. In: Haskell'04: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 80–91. ACM Press, New York (2004)
30. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04), pp. 1289–1296. ACM Press, New York (2004)
31. Walz, J.A., Johnson, G.F.: A maximum flow approach to anomaly isolation in unification-based incremental type inference. In: Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, FL, pp. 44–57 (1986)
32. Wand, M.: Finding the source of type errors. In: 13th Annual ACM Symp. on Principles of Prog. Languages, pp. 38–43 (1986)
33. Yang, J.: Explaining type errors by finding the sources of type conflicts. In: Michaelson, G., Trinder, P., Loidl, H.W. (eds.) Trends in Functional Programming. Intellect Books, pp. 58–66 (2000)
34. Yang, J., Michaelson, G., Trinder, P., Wells, J.B.: Improved type error reporting. In: Proceedings of 12th International Workshop on Implementation of Functional Languages. LNCS, vol. 2011, pp. 71–86. Springer, Berlin (2000)
35. Yang, J., Michaelson, G., Trinder, P.: Explaining polymorphic types. Comput. J. **45**(4), 436–452 (2002)