Corrective Hints for Type Incorrect Generic Java Programs

Nabil el Boustani

Jurriaan Hage

Technical Report UU-CS-2009-009
April 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Abstract

Since version 1.5, generics (parametric polymorphism) are part of the Java language. Experience with implementations of the Java Language Specification such as EJC and JAVAC have shown that the type error messages provided by these tools leave more than a little to be desired. Type error messages are often uninformative and sometimes show artifacts of the type checking process in the messages. Evidently, providing good type error messages for a language as large and complex as Java currently is, is not as easy as one would hope.

To alleviate the problem, we describe a number of heuristics that suggest fixes for generic method invocations in Generic Java, and illustrate their effect by means of examples. The heuristics are part of an extension to the original type checking process that has been implemented into the JastAdd Extensible Java Compiler.

Keywords: compilers, type checking, error reporting, error correcting, heuristics, Java generics

```
<T> List<T> foo(Map<T, ? super T> a){}
...
Map<Number, Integer> m = null;
List<Integer> ret = foo(m);
```

javac:

```
cxt_heuristic/Test1.java:6:
<T>foo(Map<T,? super T>) in Test1 cannot be applied
to (Map<Number,Integer>)
```

ejc:

```
The method foo(Map<T, ? super T) in the type Test1
is not applicable for the arguments
(Map<Number,Integer>)
```

Figure 1: Some not so informative type error messages

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 explain why a given generic method invocation fails to type check. For example, in Figure 1, neither compiler explains *why* the method invocation can't possible address the method provided by the programmer, and also does not suggest how the poblem might be fixed. It is up to the programmer to find out that out for himself.

Of course, the only duty of the Java Language Specification (JLS) [6] is to specify which programs are type correct and which are not, a simple yes or no answer. It is the task of the tools that implement the JLS to provide additional information such as the locations that play a role in the type inconsistency, or the reason why a particular invocation fails to type check. It may be that the implementors decided against providing such information for didactic reasons, but some of the examples in Section 3 have led us to believe otherwise.

Still, it seems that the JLS is at least part of the problem. Indeed the current JLS has raised more than a few eyebrows, both in the academic world [13] as well as programming forums around the world. The transition from 1.4 to 1.5 not only added generics to the language, but also features such as enums, variable arity methods, autoboxing and unboxing, that further complicate and lengthen the specification. Indeed, the fact that industry strength compilers such as JAVAC and EJC fail to implement (the generic part of) the JLS consistently is a further indication that something is amiss (see [4] for some examples).

The complications of the JLS are certainly not without reason: the combination of subtyping and generics is a notoriously hard one. Indeed, as we shall reiterate in Section 2, some of the design choices in the JLS were meant specifically to retain expressiveness and some backwards compatibility in a language that has both generics and subtyping. In turn, these choices have made type checking process, and indirectly, type error diagnosis even more complicated.

This, however, is not a paper that aims to address the shortcomings of the JLS. It intends to address the potential shortcomings in the compilers that implement it, and in doing that, to help mitigate some of the complications that the JLS may have caused.

Having studied how to improve type error messages for polymorphic functional languages such as Haskell [7], we quickly found that the differences are at least as large as the similarities. A first important difference is that in the functional programming world there is quite a difference between the declarative specification of the polymorphic lambda-calculus [10] and the implementation of the inferencer [1]. Indeed, to be able to decide whether an expression is type correct or not, no details of the type inference process need to be remembered. In the case of Java, the type system and the type checking process are essentially the same: the JLS specifies the set of type correct

programs as those programs that are designated as such by the type checking algorithm.

A first step to improve type error messages for Java was made by El Boustani and Hage who describe an extension to the Java type checking process to provide more feedback for method invocations that involve generics [4]. One of the essential aspects of their approach is to leave the type checking process as implemented in the compiler exactly as it was, and implement an extension to the process that is only invoked when an error has occurred in a particular method invocation. In this way, there is no danger of changing the set of accepted programs. We share their belief that because of the complexity of the JLS, any change to the original process is a danger by itself.

The work described in this paper extends that of El Boustani and Hage, by providing heuristics that have been implemented on top of their extension. These heuristics typically capture expert knowledge about the mistakes that programmers tend to make when programming with Java generics, or to help reduce the bias that seems to be the rule in implementations of type systems. The concrete effect of using heuristics is that they may attach a suggestion to the type error message on how to repair the type error. For example, with the help of the Context Type Invariance heuristic we describe in Section 5.2, our implementation generates the following type error message for the source code fragment in Figure 1.

```
cxt_heuristic/Test1.java:6
Method <T>foo(Map<T, ? super T>) of type
Test1 is not applicable for the argument of
type (Map<Number, Integer>), because:
  [*] The type Integer in Map<Number, Integer>
  on 5:9(5:21) is not a supertype of the inferred
  type for T: Number. However, replacing Number
  on 5:13 with Integer may solve the type conflict.
```

The heuristic exploits the additional information that the result of the call to `foo` will be stored in a variable of type `List<Integer>`, which is why it suggests to replace `Number` with `Integer` and not the other way around. Note that Eclipse also suggests fixes for the program, but they are restricted to changing the formal parameters to exactly the type of `m`, to change the type of `m` to exactly the formal parameters of `foo`, or to simply introduce a `foo` with exactly that type. None of these fixes actually take the invocation or its context into account.
Our contributions are:

- to show how heuristics can be used to improve type error messages, in particular,
  - to render type error messages less sensitive to operational choices made in the type checking process and its implementation. In particular, to avoid artifacts constructed by the inference process showing up in the type error messages,
  - to capture expert knowledge to decide what is the cause of certain classes of type errors, e.g., mistakes that arise from common misconceptions about the type system, and
  - to provide hints to suggest corrections to these mistakes,
- an implementation in a real Java compiler, the JastAdd Extensible Java Compiler, and
- anecdotal evidence that it works, in the form of over a hundred (stripped-down) examples as part of our implementation.

Besides these contributions, we also have the hope that this paper shall serve to put type error reporting on the research agenda. Indeed, except for the work by El Boustani and Hage, we have not been able to find any publication that deals with type error reporting in (generic) Java. This also explains why we have omitted a Related Work section, and decided to refer to pertinent articles throughout the paper.

What we have not yet done is to perform an empirical study of the benefits of our work. We come back to this in Section 7.

The paper is structured as follows. In Section 2 we reiterate some of the essential elements of the Generics extension in Java, and introduce some basic notations. Section 3 then provides a few examples of type error messages constructed by our extension, together with the messages provided by EJC and JAVAC. In Section 4 we prepare the way by shortly decribing the type inference process and the extension described by El Boustani and Hage [4]. Section 5 forms the bulk of the paper, by describing and illustrating a total of eight heuristics, some with variations. In Section 6 we shortly describe our implementation and where it may be obtained, Section 7 reflects and Section 8 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 at least familiar with (the non-generic part of) Java. For more details, the reader can consult the JLS [6].

Arguably, the main reason for introducing generics 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. When you think about it, this is not so strange: a List<Integer> is not supposed to store, e.g., Doubles, but if we assign it to a variable of type List<Number>, then this becomes possible. However, a value of type HashMap<Integer> may be passed safely to a parameter of type Map<Integer>, 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. In a way List<?> plays the same role in Generic Java that List<Object> played before generics. In many cases, List<?> can simply be replaced by List<T> for a fresh type variable T. However, there is a distinction: in the case of List<T>, the inferencer must be able to find a concrete type for T in every invocation, it does not need to do so for ?.

The wildcard introduces a problem by itself. If we would like to write a reverse function for lists, we can easily do so for non-wildcard types, and only for wildcard 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* [14], in which case the compiler can determine that although it may not know the concrete type at compile-time, it can be sure it will do so at run-time. Wildcard capture conversion only works when wildcards are at top-level; it will not apply to a type like List<List<?>>.

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.

2.1 Notation and terminology

We shortly introduce some notation uses throughout the paper.

Types essentially describe sets of values, e.g., integers and lists of employees. We use $C <: D$ to denote that the values of type C are also values of type D. In Java, this relation contains the transitive closure of the extends relation between classes. For example, LineNumberReader $<:$ BufferedReader and BufferedReader $<:$ Reader and, because of this, also LineNumberReader $<:$ Reader.

When generics enter the picture, the situation becomes somewhat more complicated, formalized by the notion of *containment*. For parameterized/generic types,

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

if and only if C $<:$ D and for all $1 \leq i \leq n$: Si $\leq:$ Ti 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.

For any pair of classes and interfaces, say C and D, the intersection type of the two is denoted by C & D. Since Java does not have multiple inheritance, at most one of the two contains a class name. Since & is an associative, commutative operator, we freely omit parentheses in large type expressions, e.g., Object & Serializable & Comparable<?>.

Related to these notations, 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 $S \in \mathcal{S}$. Note that typically, the *lub* includes not just a class name, but also a number of interfaces. For example, $lub(\{\texttt{Integer}, \texttt{Number}\})$ equals

$$\texttt{Object \& Serializable \& Comparable<? extends Object \& Serializable \& Comparable<?>>},$$

because both Integer and Number support these interfaces.

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 in Java the *glb* for some collections of classes and interfaces is undefined, e.g., for $glb(\{\texttt{String}, \texttt{Number}\})$.

Finally, the *arity* of a method is the number of arguments it takes. Note that in Java it is possible to define methods that have a variable arity.

```
<T> void foo(Map<? super T, ? super T> a,
             Map<? extends T, ? extends T> b){}
...
Map<Number, Integer> ma = null;
Map<Number, String> mb = null;
foo(ma, mb);
```

javac:

```
sub_repair_heuristic/Test3.java:15:
<T>foo(Map<? super T,? super T>,Map<? extends T,
                                    ? extends T>)
in Test3 cannot be applied to (Map<Number,Integer>,
Map<Number,String>)
```

ejc:

```
The method foo(Map<? super T,? super T>,
Map<? extends T,? extends T>) in the type Test3
is not applicable for the arguments
(Map<Number,Integer>, Map<Number,String>)
```

ours:

```
sub_repair_heuristic/Test3.java:15
Method <T>foo(Map<? super T, ? super T>,
Map<? extends T, ? extends T>) of type Test3 is not
applicable for the arguments of type (Map<Number,
Integer>, Map<Number, String>), because:
  [*] The type Number in Map<Number, Integer>
  on 13:9(13:13) is not a supertype of the inferred
  type for T: Serializable.
  [*] The type Integer in Map<Number, Integer>
  on 13:9(13:21) is not a supertype of the inferred
  type for T: Serializable.
  However, replacing
    - String on 14:21
    - Integer on 13:21
  with Number may solve the type conflict.
```

Figure 2: Some not so informative type error messages, and our own

3 Examples

Before we go on to describe how the type checking process specified in the JLS can be extended, and how that allows us to implement various heuristics for improving Java type error messages, we first give some example type error messages generated by the standard compilers for Java, EJC (version 3.4.1) and JAVAC (version 1.5.0_16), and our own implementation as part of the JASTADD EJC 6. The examples are intended to show the benefits of using the modified type checking process, and in particular our heuristics. In the remainder of the paper we shall give many more examples, but then only the type error messages our system provides.

For reasons of space, we silently omit all the parts of the code that are not of interest to explaining the type error messages and have taken the liberty to rewrite some output to make it fit the width of a column. For example, JAVAC invariably gives the complete name for a class, e.g., `java.lang.Number`; in the messages we provide, we have abbreviated this to `Number`.

The number of examples is relatively small, but Section 6 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.

In the introduction we gave an example to show that type error messages can be quite uninformative for EJC and JAVAC. In Figure 2 we give another example. In fact, the messages given by both JAVAC and EJC are typical: they show the signature of the method against which they compared the invocation, give the types of the arguments in the invocation, and tell the programmer they do not match.

The next few examples, both taken from [4], serve to show that indeed EJC and JAVAC allow aspects of the type inference process be exposed through the type error messages. In the first example in Figure 3, both compilers show that during the type checking process, they introduce captures of wildcards, and that they keep these apart by assigning some number to them. As we show in our type error message, there is actually no need to do that. Note that in this case, the JAVAC compiler even states that no symbol `bar` could be found, implying it did not even try to compare the invocation to the provided method `bar`.

Finally, consider the example of Figure 4. The constraints generated for both method calls are the same according to the JLS: `T` <: `Number` and `T` <: `String`, but the type error diagnosis for these very similar programs by EJC is quite different. This is due to how EJC resolves subtype constraints. The messages provided by our implementation are identical, and also suggest how to fix the problem.

4 The Type Checking Process

To avoid any misunderstanding, we first explain our terminology for describing the process of type checking. In this paper, the term *type checking process* refers to the complete process of determining the type correctness of a specific program fragment. In our particular case these program fragments are always *method invocations*. We first describe the type checking process as described by the JLS. This also allows us to illustrate some of the basic ingredients, e.g., constraints, *lub* and *glb*, in their natural habitat. At the end of this section, we spend some time on explaining how the type checking process in our implementation differs from that of the JLS.

The basic type checking process is depicted in Figure 5. It starts off by performing *method resolution*, which determines, for a given invocation, a set of methods that the programmer may be invoking. Method resolution is a rather complex operation, and to give the reader some idea what is happening, we sketch the process here (more details can be found in [6, 3]).

Java supports overloading, so a call can potentially address a multitude of methods, and method resolution tries to narrow that multitude down to a single method. It weeds out methods by first comparing the (fully qualified) name of the identifier, the number of arguments (taking into account variable arity methods if the need arises), accessibility and the number of generic arguments. Thus, a list of candidate methods is obtained.

These methods are then considered one by one, and the types of the actual arguments are

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

javac:

```
Test1.java:20: cannot find symbol
symbol : method bar(Map<capture#954 of ? extends
 Number, capture#0 of ? extends Number>)
location: class Test1
  foo(m);
```

ejc:

```
1. ERROR in Test1.java (at line 20)
  foo(m);
The method bar(Map<T,T>) in the type Test1 is not
 applicable for the arguments
 (Map<capture#1-of ? extends Number,
      capture#2-of ? extends Number>)
```

ours:

```
Test1.java:6
Method <T>bar(Map<T, T>) of type Test1 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.
```

Figure 3: A code fragment with a wilcard equality error followed by the three corresponding type error messages.

```
<T extends Number>
  void foo(Map<? super T, ? super T> a)
...
Map<String, Number> m1 = ...;
foo(m1);
Map<Number, String> m2 = ...;
foo(m2);
```

ejc:

```
1. ERROR in Listing5.java (at line 10)
  foo(m);
Bound mismatch: The generic method foo(
 Map<? super T, ? super T>) of type Listing5 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>

2. ERROR in Listing5.java (at line 12)
  foo(m);
The method foo(Map<? super T,? super T>) in the
 type Listing5 is not applicable for the arguments
 (Map<Number,String>)
```

ours:

```
Listing5-1.java:9
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
  8:9(8:13) and Number in Map<String, Number> on
  8:9(8:21) do not share a common subtype. Most
  likely String needs to be replaced with a
  subtype of the following type: Number.


Listing5-2.java:9
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
  8:9(8:13) and String in Map<Number, String> on
  8:9(8:21) do not share a common subtype. Most
  likely String needs to be replaced with a
  subtype of the following type: Number.
```

Figure 4: Two similar method calls as diagnosed by EJC and our own system.

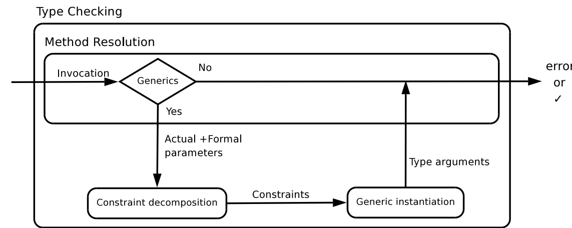

Figure 5: The type checking process

matched with the types of the formal parameters; this part of method resolution also takes into account (un)boxing and variable arity methods.

If there is no matching method, i.e., all potential methods have been eliminated, then the process stops and an error message is generated. If more than one method remains, method resolution tries to determine whether there is a single most specific method in the set of potential methods. At this stage, various heuristics are applied by the method resolution subprocess. For example, it prefers to return a nonabstract method instead of an abstract one.

Given a method that has survived method resolution, the concrete parameter types of the invocation are paired with the formal parameter types of the method declaration to form a set of constraints, as follows:

```
<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 for this invocation to type check.

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 intuitively easy, so we give only the decomposition for the set of constraints found for our example, and omit further details (see Listings 3.1 to 3.3 of [3])

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

Note that at this point, the type checking process has lost knowledge of the exact constraints generated by the program, and has to make do with the decomposed constraints. For verifying type correctness, this is not a problem; for generating suitable type error messages, it is better to also keep the old constraints around, because they are more closely tied to the source program.

The type checking process then proceeds by *inferring* 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 decide that the instantiation is incorrect. The JLS states that if a conflict exists, then it will indeed show up 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 *lub* of the two, `Number`.

As the reader may recall from Section 2.1, the *lub* of the types `Integer` and `String` is quite a bit more complicated, involving interfaces and wildcards. We have observed 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 [16], believe a type checking process should have: it should only refer to types or expressions that are part of the original source program.

In the absence of equality constraints and subtype constraints, the type inferencer will consider the context of the invocation, in particular, whether a type variable can be instantiated based on information from the assignment context. 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; `T` was already instantiated to `Integer`. If, at this point, a type variable is still not assigned a concrete type, then subtype constraints, i.e., of the form `T` $<:$ `Number`, will be considered. For declared type variable that are not constrained in any way, the JLS specifies that they should 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 each invocation is considered *in isolation*. This even holds for nested invocations `foo(bar(x),y)`, where 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 may be used to determine the proper instantiations for `bar`. This lack of propagation in Java has its advantages — types are instantiated based on local information only and not through a long and complicated sequence of unifications —, but may also surprise the programmer, particularly in the case of the ambigious method invocation.

4.1 The Modified Type Checking Process

The overall architecture of the modified type checking process can be seen in Figure 6. Although, the modified process has been described in more detail already in [4], we include a summary of the process to both give the reader a better idea of what is going on, and to be able to state more precisely when and how heuristics are applied in the system.

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. In this way, it cannot interfere with the decision of type correctness, but only provide more detailed feedback whenever the original type checking process failed.

The essential differences for the various phases in the type checking process are:

- The method resolution phase allows more than one method to be returned as its result.
- When matching the invocation with a potential method during method resolution, any generic type information will be ignored.

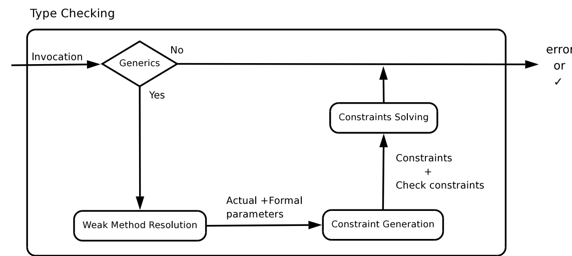

Figure 6: The modified type checking process

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

Figure 7: A code fragment with two candidate methods.

- Although constraint decomposition will still take place, the system keeps track of the original constraints as well.
- type inference is combined with checking that bounds on the inferred types may still be satisfiable.

We shall go through the modified type checking process in some more detail, and sketch by means of examples what happens in the various phases.

In the modified version 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`. In this way, the code fragment of Figure 7, for example, allows both declarations of `bar` to pass method resolution.

Constraint generation is the modified 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 Figure 8. 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.

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

Figure 8: Inference succeeds, but checking fails.

In the modified version, the constraint solver is provided with more constraints, which will ensure that a type is inferred only if all type constraints are satisfied and no type checking error will occur. For that reason, the original constraints decomposition algorithm is extended to generate additional constraints, which are left alone during decomposition. For the example in Figure 8, the new constraint generation algorithm will collect the following constraints:

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

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

This is a general principle when improving type error messages: one of the operands, the simplified one on the left, serves to easily decide type correctness, while the right operand provides the type error construction process with additional information where the inconsistent type variables assignments came from. The constraint decomposition phase, only applied to the left operand of $\Cup$, is exactly the same as for the original process.

The constraints obtained in this way are then fed to the constraint solver which can use both the left and right argument to $\Cup$ to come up with a suitable error message, and heuristics may be triggered to consider the set of constraints to come up with suggestions for fixing the problem. More details will be given later in Section 6. The details of the constraint solver are not important for this paper; the interested reader is referred to [4].

5 The heuristics

Before we go on to describe the heuristics and their effect on the error message in detail, we first give some more general information about the heuristics and how they are used.

Except for a single heuristic that generates warnings (see Section 5.5), all the other heuristics that we present are program correcting heuristics: they explain to the programmer how to modify the source code to resolve the type error. Most heuristics operate directly on the generated type constraints for a method invocation, and all of the heuristics are triggered by type conflicts.

But what exactly does it mean for a heuristic to be "program correcting"? Type conflicts arise from different parts in the program generating conflicting constraints. Often, it boils down to certain parts of the program pointing to one thing, e.g., `T` should be instantiated to `Number`, and other parts pointing to a different conclusion, e.g., `T` should be instantiated to `Integer`. If, for whatever reason, we have more confidence that the latter is the correct reading, and not the former, the type error may explain to the programmer how the program should be modified in order to make sure that the constraints that imply that `T` should be `Number` will not be generated anymore. This ensures that, with all other things equal, the inconsistency will not arise during the next compilation. It is the task of each heuristic to decide, for a particular type error, whether there is an observed, substantial difference in confidence, and what the suggested modification should be.

In our architecture, several heuristics can be triggered by the same type conflict. To avoid multiple repair suggestions, which can easily confuse the programmer, we have implemented a prioritization of the heuristics. If multiple heuristics have a suggestion how to fix the problem only the fix suggested by the heuristic with highest priority is included. The priorities are assigned to the heuristics by the error manager, the component that has the responsibility to collect all the type conflicts discovered by the constraint solver (see Section 6). At the moment the priorities of heuristics are static; making priorities more dynamic is future work.

When we consider the heuristics below we shall give plenty of examples of the effect of using them on pieces of generic Java. Note however, that the type error messages have been edited to fit the two-column format of the paper, and that this may hamper readability.

5.1 The majority heuristic for equality constraints

This heuristic goes back to the work of Johnson and Walz [15], and encapsulates a simple but important idea: if two parts of the program are in type conflict, blame that part for which you

```
<T> void foo(Map<T, T> a, T b){}
...
Map <Integer, Number> m = ...; ...;
Number n = ...;
foo(m, n);
```

```
Test1.java:18 Method <T>foo(Map<T, T>, T) of type
Test1 is not applicable to the arguments of type
(Map<Integer, Number>, Number), because:
  [*] Type variable T is invariant, but the types
   - Number in Map<Integer, Number> on 6:9(6:22)
   - Integer in Map<Integer, Number> on 6:9(6:13)
  are not the same.
However, replacing Integer on 6:13 with Number may
solve the type conflict.
```

Figure 9: Equality type conflict, followed by our error message

```
<T> void foo(List <T> a, List <T> b, List <T> c){}
...
List <Number> src = ...;
List <Integer> big = ...;
List <Integer> small = ...;
foo(src, small, big);
foo(src, src, big);
```

Figure 10: Equality conflict with two solutions

have the least evidence that it is correct. Blame, in our case, means that a repair suggestion will indicate how the parts in which we have least confidence ought to be changed to get rid of the inconsistency. In this section, we start with a simple way of weighing evidence, and progressively make refinements.

The majority heuristic for equivalence constraints is used to correct type equality conflicts by choosing a type based on the equality constraints that satisfies as many type constraints as possible. Consider the code in Figure 9. The method invocation leads to the constraints:

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

The type variable `T` gives rise to a type equality conflict because `Integer` is not equal to `Number`. This conflict can be resolved in two ways; we can choose `T` to be `Integer` or we can choose `T` to be `Number`. Since the former gives rise to a new conflict with the third constraint $\mathtt{Number} \not<: \mathtt{T}$, the error message generated for the method invocation (Figure 9) is extended with a repair hint that describes how to make sure `T` will be instantiated to `Number`.

Sometimes an equality conflict can be resolved in multiple ways. The code fragment in Figure 10 shows two instances of such a situation. The invocations have exactly the same constraints $\{\mathtt{T} = \mathtt{Number}, \mathtt{T} = \mathtt{Integer}\}$, thus we can decide to replace Integer with Number or the other way around. In such situation, we can either extend the error messages generated with an arbitrary chosen possible repair, or not to extend the error messages at all, because none of the solutions is better than the other. To solve this problem of choice, we can define a function that can assess the quality of solutions, and help us to find the best solution. For this heuristic we choose to use the minimal number of edits or modifications required to implement the suggested repair(s). The notion of an edit here, is the act of adding or removing a type or a type constructor. A parametrized

```
<T> void foo(Map<T, T> a, List <T> b,
             List <T> c, List <T> d) {}
Map <Number, Number> m = l...]
List <Integer> l = ...;
List <Integer> l2 = ...;
foo(m, l, l, l2);
```

```
Test6.java:9
Method <T>foo(Map<T,T>, List<T>, List<T>, List<T>)
of type Test6 is not applicable to the arguments
of type (Map<Number, Number>, List<Integer>,
List<Integer>, List<Integer>), because:
  [*] Type variable T is invariant, but the types:
    - Integer in List<Integer> on 7:9(7:14)
    - Number in Map<Number, Number> on 5:9(5:13)
  are not the same type. However, replacing Number
  on 5:13 with Integer may solve the type conflict.
```

Figure 11: Equality conflict with two equivalent solutions

type is naturally described as a tree, which we subsequently flatten into a list of types and type constructors, e.g., `Map<Integer, Integer>` is mapped to the list [`Map`, `Integer`, `Integer`] and `List<Integer>` maps to [`List`, `Integer`]. The similarity between the two is then the number of minimum number of additions and deletions of types and type constructors to map the former into the latter. In the example, `Map` and `Integer` must be deleted, and `List` must be added: a total of three edits.

Applying this recipe to the first invocation in Figure 10, we can propose to change the type `Number` with `Integer`, because the programmer needs only to change the type of the parameter `src`. Proposing to change `Integer` to `Number` would mean that the user has to perform more edits, because both types of the parameters small and big have to be modified. For the second invocation we do not propose a repair because both solutions require the same number of modifications.

The minimal number of edits is a reasonable assessment function, but it is not the only criterion that can be used determine the best solution. Consider for example the code in Figure 11, where the number of edit operations required to change the type Integer to Number or vice-versa is exactly the same. In this case the heuristic can not determine the best solution based on the number of edits. However, the heuristic also tends to favor localized changes. Often it is easier for the user to change the types in one place instead of multiple places or files. Therefore, we generate the error message in Figure 11.

Localized changes, however, are not always desired. For example, the user may have compiled the code in the middle of an unfinished refactoring. To change the default behaviour of the system, the user can use the command line option `-distinct` to prevent the system from suggesting a repair for the invocation in Figure 11.

Until now, the presented examples only featured a single type variable. We now continue with some example in which multiple type variables play a role. Consider the code in Figure12, where the heuristic must take the type variable `S` into consideration when trying to solve the equality conflict of `T`.

The problem of finding a solution for the equality conflict of `T` is a constraint satisfaction problem (CSP). The variables in the CSP are the type variables $\{\texttt{T}, \texttt{S}\}$, the domains of the variables are $\{\texttt{Integer}, \texttt{String}\}$ for `T` and $\{\texttt{String}\}$ for `S`, and the constraints are $\{\texttt{S} <: \texttt{T}\}$. Solving the CSP by trying all possible types for `T` and `S` quickly shows that `T` should be `String`.

To speed up the process of finding a solution of a type variable group in general we make use of two search heuristics: the degree heuristic and forward checking [12]. The degree heuristic tries

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

Figure 12: Conflict involving two type variables

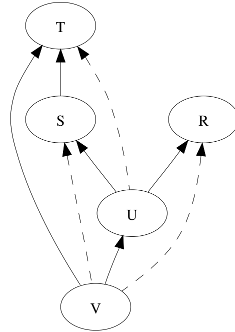

Figure 13: Type variable dependency graph

to minimize the number of branches when looking for solutions by selecting the most constrained variable and assign a type to it. For example, consider the following type variable declaration:

$$< \texttt{T}, \texttt{S extends T}, \texttt{R}, \texttt{U extends Map<R, S>},$$

$$\texttt{V extends Map<U, T>} >$$

From these constraints, the dependency graph given in Figure 13 can be constructed as follows: each of the type variables becomes a vertex in the graph, and there is a solid edge from S to T if T occurs in a bound for S, in other words the type for S depends on the concrete choice for T. The dashed edges in the graph denote the transitive edges. The vertex in the graph with the most incoming edges is the most constrained variable, in the sense that the highest number of type variables depends on it.

Forward checking is used to ensure that when instantiating a type variable `X`, the type variables directly depending on `X` continue to have at least one type in their domain that is within their bounds. For example, when instantiating the type variable `T` to some concrete type σ, the domains of the type variables `S` and `V` are checked to verify that `S` can be instantiated so that `S` $<:$ σ and `V` can be instantiated so that `V` $<:$ `Map<X,`σ`>` for some X in the domain of `U`.

Figure 14 provides an example of what kind of errors this heuristic can repair. Note that every type variable has an equality conflict. Since `S` depends on `T`, and `R` depends on `S` and indirectly on `T`, the heuristic processes all type variables as a single type variable group.

Grouping type variables based on their bound dependencies when solving equality conflicts is better than solving the conflicts for each type variable separately, because we can prevent proposing repairs that cause conflicts with the constraints on the bounds. However, sometimes grouping type variables only on the basis of their bound dependencies does not suffice. Consider the code in Figure 15, where the type variables in the first method declaration form one type group, and in the second declaration they form two separate groups.

Note that the suggested repair for the type variable `T` in the first invocation will solve the equality conflict, but it will also create a new conflict. The conflict arises when checking the non-atomic constraints, because `Map<Integer, ? extends Double>` $\not<:$ `Map<S, ? extends T>`, where `S` and `T` are substituted by `Integer` and `Number`, respectively. In the second invocation the type variable groups $\{$`T`, `S`$\}$ and $\{$`R`$\}$ are solved separately, which causes the heuristic to propose to

```
<T, S extends T, R extends S> void foo(
  Map<T, T> a, Map<S, S> b, Map<R, R> c) {}
...
Map <Integer, Double> m1 = ...;
Map <Integer, Byte> m2 = ...;
Map <Integer, Byte> m3 = ...;
foo(m1, m2, m3);
```

```
Test8.java:12
Method <T, S extends T, R extends S> foo(Map<T, T>,
Map<S, S>, Map<R, R>) of type Test8 is not
applicable to the arguments of type(Map<Integer,
Double>, Map<Integer, Byte>, Map<Integer, Byte>),
because:
  [*] Type variable T is invariant, but the types:
  - Double in Map<Integer, Double> on 9:9(9:22)
  - Integer in Map<Integer, Double> on 9:9(9:13)
  are not the same type. However, replacing Double
  on 9:22 with Integer may solve the type conflict.
  [*] Type variable S is invariant, but the types:
  - Byte in Map<Integer, Byte> on 11:9(11:22)
  - Integer in Map<Integer, Byte> on 11:9(11:13)
  are not the same type. However, replacing Byte
  on 11:22 with Integer may solve the type conflict.
  [*] Type variable R is invariant, but the types:
  - Byte in Map<Integer, Byte> on 11:9(11:22)
  - Integer in Map<Integer, Byte> on 11:9(11:13)
  are not the same type. However, replacing Byte
  on 11:22 with Integer may solve the type conflict.
```

Figure 14: Single type variable group

```
<T, S extends T> void bar(Map<S, ? extends T> a,
                          Map<T, T> b) { }
...
Map<Integer, ? super Double> sm = ...;
Map<String, Number> tm = ...;
bar(sm, tm);
...
<T, S extends T, R> void baz(Map<S, ? extends R> a,
                    Map<T, T> b, Map<R, R> c) { }
...
Map<Integer, ? super Double> sm = ...;
Map<String, Number> tm = ...;
Map<Object, Number> rm = ...;
baz(sm, tm, rm);
```

```
Test9.java:23
Method <T, S extends T>bar(Map<S, ? extends T>,
Map<T, T>) of type Test9 is not applicable to the
arguments of type(Map<Integer, ? super Double>,
Map<String, Number>), because:
  [*] Type variable T is invariant, but the types:
  - Number in Map<String, Number> on 22:9(22:21)
  - String in Map<String, Number> on 22:9(22:13)
  are not the same type. However, replacing String
  on 22:13 with Number may solve the type conflict.
```

```
Test9.java:38
Method <T, S extends T, R>baz(Map<S, ? extends R>,
Map<T, T>, Map<R, R>) of type Test9 is not
applicable to the arguments of type(Map<Integer,
? super Double>, Map<String, Number>, Map<Object,
Number>), because:
  [*] Type variable T is invariant, but the types:
  - Number in Map<String, Number> on 36:9(36:21)
  - String in Map<String, Number> on 36:9(36:13)
  are not the same type. However, replacing String
  on 36:13 with Number may solve the type conflict.
  [*] Type variable R is invariant, but the types:
  - Number in Map<Object, Number> on 37:9(37:21)
  - Object in Map<Object, Number> on 37:9(37:13)
  are not the same type.
```

Figure 15: Conflicts in non-atomic constraints

```
<T> void foo(Map<T, T> a,
            Map<? super T, ? extends T> b) {}
...
Map<Number, Integer> m = ...;
Map<? super Integer, ? super Integer> m2 = ...;
foo(m, m2);
```

Figure 16: Conflicts in all constraints

repair the type conflict of `T` by replacing `String` with `Number`. For the type variable `R`, no repair was suggested because both types `Object` and `Number` are equivalent solutions. Note, however, that the proposed repair in this case will also lead to a type conflict when checking the non-atomic constraints. To help the heuristic find a solution that will satisfy all the constraints, the user can use the command-line option `-strict`. This flag will force the heuristic to combine the solutions of all type variable groups whose members are involved in non-atomic constraints, e.g. $\{\texttt{T},\texttt{S}\}$ and $\{\texttt{R}\}$ will become $\{\texttt{T},\texttt{S},\texttt{R}\}$, and then filter out all the solutions that do not satisfy all the non-atomic constraints involving type variables from the joined type variable groups. The generated error messages when using the `-strict` flag are the same as the ones give nearlier, except that the line

```
However, replacing String on 22:13 with Number may
solve the type conflict.
```

is omitted from the first message (`Test9.java:23`), and the line

```
However, replacing Number on 37:21 with Object may
solve the type conflict.
```

is added to the second (`Test9.java:38`).

Observe that the system retracted the repair it suggested before, because now it knows that the repair is not a complete solution. In the second message we see that system proposed an additional repair, because now it has the necessary criteria to judge whether Number should be replaced by Object or the other way around.

One may wonder why we do not always run this heuristic in its strict mode since it will provide safer solutions. The reason is that non-atomic constraints themselves can be a source of conflicts. To illustrate, consider the code in Figure 16. Running the heuristic in strict mode will not propose a repair of the invocation, because the type variable `T` is overconstrained with the (unsatisfiable) non-atomic constraints.

If the heuristic is run in non-strict mode instead, then a repair hint suggesting to replace `Number` in the type of the parameter `m` with `Integer` will be presented to the programmer. If the programmer applies the suggested repair and compiles for the second time, then he will be presented with another error. However, the system will present a repair for this second error too, which is suggested by the opposite wildcards heuristic discussed in Section 5.3.2.

5.2 Context type invariance

In this section we present a heuristic that relies on information from the context to correct type conflicts. The heuristic exploits the fact that if the return value of a method has a parameterized type, e.g., `List<T>`, and we assign the result for a particular invocation to a variable of type `List<X>` for some concrete `X`, then collection class invariance provides additional evidence that `T` should be `X`.

The heuristic can only be applied in a situation where all the following conditions are met:

- The method has a parameterized return type, that contains type variables.
- The method invocation appears in an assignment context.

```
<T> List<T> foo(Map<T, ? super T> a) {}
...
Map<Number, Integer> m = ...;
List<Integer> ret = foo(m);
```

```
Test1.java:6
Method <T>foo(Map<T, ? super T>) of type Test1 is
not applicable to the argument of type
(Map<Number, Integer>), because:
  [*] The type Integer in Map<Number, Integer> on
  5:9(5:21) is not a supertype of the inferred type
  for T: Number. However, replacing Number on 5:13
  with Integer may solve the type conflict.
```

Figure 17: Return type invariance

```
<T, S extends Map<Number,T>>
  List <T> baz(Map<T, ? super T> a, S b) {}
...
Map<Object, Number> mt = ...;
Map<Number, String> ms = ...;
List<Integer> ret = baz(mt,ms);
```

Figure 18: Bound conflict

- If `S` is the type of the lvalue and `R` the return type of the method, then `R` <: `S` should give rise to an equality constraint.

Consider the code and corresponding type error message in Figure 17. Here the type variable `T` has a subtype conflict, because `T` is instantiated to `Number`, but `Number` is not a subtype of `Integer`. Because the result of the invocation is assigned to a local variable, we can conclude that whatever type `T` is instantiated to, `List<T>` must be a subtype of `List<Integer>`. Due to type invariance we can immediately conclude that `T` can only be instantiated to `Integer`. Thus, the heuristic substitutes the type inferred for `T` with `Integer` and verifies that all constraints are satisfied.

This heuristic runs in strict mode by default, meaning that it considers all the constraints when verifying whether the type deduced from the context can solve the type conflicts. Thus, bound and non-atomic constraints are also checked. Therefore, the heuristic does not propose any repairs for the invocations in Figure 18 and 19. In the former, the heuristic solves the subtype conflict (`Object` $\not<$: `Number`) of `T` by inferring from the context that `T` must be `Integer`. However, substituting `T` with `Integer` in the bound of `S` causes a bound conflict, because `Map<Number, String>`

```
<T extends Number> List <T>
  bar(Map<T, T> a, List<? extends T> b) {}
...
Map<Boolean, String> m = ...;
List<? super Number> l = ...;
List<Integer> i = bar(m, l);
```

Figure 19: Non-constraint conflict

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

```
Test8.java:7
Method <T extends Number, S>foo(Map<T, S>,
List<? extends S>) of type Test8 is not
applicable to the arguments of type(Map<String,
String>, List<? super Number>), because:
  [*] The type String in Map<String, String> on
  5:9(5:13) is not a subtype of T's upper bound
  Number in 'T extends Number'. However,
  replacing String on 5:13 with Integer may
  solve the type conflict.
  [*] The type List<? extends S>, where S was
  inferred to be String is not a supertype of
  List<? super Number> on 6:9.
```

Figure 20: Indirect conflict

is not a subtype of `Map<Number, Integer>`. In Figure 19 the heuristic solves the equality conflict of `T` by inferring from the context that `T` must be `Integer`. Substituting `T` with `Integer`, however, will not satisfy the non-atomic constraint: `List <? super Number>` $<:$ `List <? extends T>`.

Although the heuristic does check the bound and non-atomic constraints before suggesting a repair, it will still propose a repair for the call in Figure 20. The reason we suggest a repair in this case is because we want to help the programmer as much as possible. By default, the heuristic proposes repairs for type variables that are not directly responsible for bound conflicts or a conflict in the non-atomic constraints, like in Figure 20. These repairs, however, are not always in line with the intentions of the programmer. Therefore, to disable printing any repairs which will not guarantee that the invocation will type check, the user can provide the command-line option `-verystrict` to the heuristic.

Note that this heuristic does not only resolve type conflicts of the type variables that occur in the return type of a method, but also bound conflicts and non-atomic conflicts of other type variables. Conflicts of other type variables, however, are only resolved under the condition that these conflicts involve at least one type variable from the return type. For example, consider the code in Figure 21. The type variables `T` and `R` have no conflicts at all, but `S` has a bound conflict that involves both. Therefore, the heuristic suggests to change the types that `T` and `R` were instantiated to.

5.3 Wildcards

In this section we present a number of heuristics that specialize in correcting type conflicts caused by wildcards.

5.3.1 Bounded lower bound wildcard

This heuristic targets only bound conflicts caused when a bounded type variable was instantiated with a lower bound wildcard. Consider, for example, the code in Figure 22. The type variable `T` is involved in a single equality type constraint: $\{$`T` $=$ capture(`? super Integer`)$\}$. Therefore,

```
<T, R, S extends Map<R, T>>
    Map<T, R> baz(T a, R b, S c) {}
...
Number t = ...;
Number r = ...;
Map<Double, Integer> ms = ...;
Map<Integer, Double> ret = baz(t, r, ms);
```

```
Test9.java:15
Method <T, R, S extends Map<R, T>>baz(T, R, S)
of type Test9 is not applicable to the arguments
of type(Number, Number, Map<Double, Integer>),
because:
  [*] The type Map<Double, Integer> on 14:9 is
  not a subtype of S's upper bound Map<Number,
  Number> in `S extends Map<R, T>`. However,
  replacing the types:
  - Number on 13:9 with Integer
  - Number on 13:9 with Double
  may solve the type conflict.
```

Figure 21: Indirect bound conflict

```
<T extends Number> void foo(List<T> a) {}
...
List<? super Integer> l = ...;
foo(l);
```

```
Test1.java:6
Method <T extends Number>foo(List<T>) of type
Test1 is not applicable to the argument of type
(List<? super Integer>), because:
  [*] The type '? super Integer' in
  List<? super Integer> on 5:9(5:14) is not a
  subtype of T's upper bound Number in
  `T extends Number`. However
  '? extends Integer' is a subtype of Number
```

Figure 22: Instantiating bounded type variable with a lower bound wildcard

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

Figure 23: Replacing `super` with `extends` does not solve the conflict

```
<T> void foo(Map<? extends T, T> a) {}
...
Map<? super Integer, Number> m2 = ...;
foo(m2);
```

```
Test3.java:16
Method <T>foo(Map<? extends T, T>) of type
Test3 is not applicable to the argument of
type(Map<? super Integer, Number>), because:
  [*] The type Map<? extends T, T>, where T
  was inferred to be Number is not a supertype
  of Map<? super Integer, Number> on 15:9.
  However Map<? extends Integer, Number> is a
  subtype of Map<? extends T, T>
```

Figure 24: Wildcard conflict

the constraint solver instantiates `T` to ω_1, which is obtained by performing capture conversion on `? super Integer`. Since it is not guaranteed that ω_1 will be a subtype of `Number`, the bound of `T`, a bound conflict arises. We know that the bound conflict will arise no matter what the lower bound is in the wildcard. So maybe the programmer intended to use an upper bound (`extends`) instead of a lower bound (`super`). Therefore, the heuristic changes the type `List<? super Integer>` to `List<? extends Integer>` and verifies whether all constraints can be satisfied simultaneously. Changing the lower bound to an upper bound in Figure 22 will cause the constraints to change to $\{\texttt{T} = \text{capture}(\texttt{? extends Integer})\}$. The type variable will be instantiated to a capture of `? extends Integer`, say ω_2. Since ω_2 is a subtype of `Integer`, ω_2 will also be a subtype of `Number`. The heuristic therefore informs the programmer that a bound change in the wildcard may solve the conflict.

Consider the code in Figure 23. The heuristic will not suggest a repair for this invocation. Converting `List<? super Integer>` to `List<? extends Integer>` gives the constraints:

$$\{\texttt{T} = \text{capture}(\texttt{? extends Integer})\} \Cup$$

$$\{\texttt{List<? extends Integer>} <: \texttt{List<? super T>}\}$$

Because `T` is involved in just one type equality constraint, it will be instantiated to ω_3, where ω_3 is the capture of `? extends Integer`. However, substituting `T` with ω_3 in `List<? super T>` will not resolve the non-atomic constraints, because `? super` ω_3 does not contain `? super Integer`.

5.3.2 Opposite wildcards

This heuristic is a variation of the previous heuristic, because it addresses type conflicts caused by incompatible wildcards.

```
<T> void bar(Map<? super List<T>, T> a) {}
...
Map<? extends List<Integer>, Integer> m = ...;
bar(m);
```

```
Test4.java:7
Method <T>bar(Map<? super List<T>, T>) of type
Test4 is not applicable to the argument of type
(Map<? extends List<Integer>, Integer>), because:
  [*] The type Map<? super List<T>, T>, where T
  was inferred to be Integer is not a supertype
  of Map<? extends List<Integer>, Integer> on 6:9.
  However Map<? super List<Integer>, Integer> is
  a subtype of Map<? super List<T>, T>
```

Figure 25: Another wildcard conflict

Consider the code in Figure 24. The invocation does not type check, because `Map<? super Integer, Number>` is not a subtype of `Map<? extends T, T>`, where `T` is instantiated to `Number`. However, changing the keyword `super` to `extends` in the declaration of the local variable `m2` will make the invocation type check. Therefore, the heuristic proposes to change the type of `m2`.

The heuristic can also repair conflicts where the role of lower and upper bound is reversed. Figure 25 provides an example: the invocation of the method `bar` does not type check for the same reason as in Figure 24: the first type parameter in formal parameter `? super List<T>` does not contain the type `? extends List<Integer>`, where `T` is instantiated to `Integer`. The situation changes, however, when we change the bound of the wildcard in the actual parameter. The heuristic discovers this and reports it to the user.

The heuristic can also handle nested wildcards, such as in Figure 26, because it checks iteratively whether the bounds in the wildcards are compatible.

Another approach to correcting programs is to detect situations in which the program actually does not need wildcards, and their presence leads to type incorrectness.

5.3.3 Super Object

Consider the type `List<? super Object>`. We can add an element of any type to this list, because every possible type in Java is a subtype of `Object`, and hence it is also a subtype of `? super Object`. Furthermore, this list can only refer to a list instantiated with a type that can contain `? super Object`. The only type in Java that can do this, is Object itself. Since we can only read elements of type `Object` from this list, we can say that this list behaves almost exactly the same as a list of type `List<Object>`. Therefore, it will be a good idea to check whether type conflicts caused by the wildcard `? super Object` can be resolved by using `Object` instead. Consider the code in Figure 27, where a type conflict is caused by the use of the wildcards `? super Object`. Two different captures of a wildcard are never equivalent even if the wildcards and their bounds are identical. Replacing the wildcards and their bounds with just `Object` resolves this type error; therefore the heuristic hints at this repair in the error message.

Besides equality conflicts, this heuristic can also resolve type conflicts in the non-atomic constraints (see Figure 28 for an example). The invocation does not type check, because `T` is instantiated to a capture of `? super Object`, say ω, but `? extends` ω does not contain the wildcard `? super Object` from the actual parameter. Replacing the second wildcard in the type of the variable `m` with `Object`, will cause `T` to be instantiated to `Object`. Since `Object` is the supertype of all types, the invocation will type check. Therefore the repair is suggested in the error message.

```
<T> void foo1(List<? extends List<? extends T>> a)
...
List<? super List<? super Integer>> l = ...;
foo1(l);
...
<T> void foo6(List<? extends List<? super T>> a)
...
List<? super List<? extends Integer>> l = ...;
foo6(l);
```

```
Test5_1.java:7
Method <T>foo1(List<? extends List<? extends T>>)
of type Test5_1 is not applicable to the argument
of type(List<? super List<? super Integer>>),
because:
  [*] The type List<? extends List<? extends T>>,
  where T was inferred to be Object is not a
  supertype of List<? super
  List<? super Integer>> on 6:9. However,
  List<? extends List<? extends Integer>> is
  a subtype of List<? extends List<? extends T>>
```

```
Test5_2.java:37
Method <T>foo6(List<? extends List<? super T>>)
of type Test5_2 is not applicable to the argument
of type(List<? super List<? extends Integer>>),
because:
  [*] The type List<? extends List<? super T>>,
  where T was inferred to be Object is not a
  supertype of List<? super
  List<? extends Integer>> on 36:9. However,
  List<? extends List<? super Integer>> is
  a subtype of List<? extends List<? super T>>
```

Figure 26: Nested wildcards

```
<T> void foo(Map<T, T> a) {}
...
Map<? super Object, ? super Object> m = ...;
foo(m);
```

```
Test1.java:6
Method <T>foo(Map<T, T>) of type Test1 is not
applicable to the argument of type
(Map<? super Object, ? super Object>), because:
  [*] The type variable T is invariant, but the
  type '? super Object' is not. However, replacing
  - '? super Object' on 5:13
  - '? super Object' on 5:28
  with Object may solve the type conflict.
```

Figure 27: Equality conflict caused by wildcards

```
<T> void foo(Map<? extends T, T> a) {}
...
Map<? super Object, ? super Object> m = ...;
foo(m);
```

```
Test2.java:6
Method <T>foo(Map<? extends T, T>) of type
Test2 is not applicable to the argument of type
(Map<? super Object, ? super Object>), because:
  [*] The type Map<? extends T, T>, where T was
  inferred to be '? super Object' is not a
  supertype of Map<? super Object, ? super Object>
  on 5:9. However, replacing '? super Object' on
  5:29 with Object may solve the type conflict.
```

Figure 28: Type error caused by `? super Object`

```
<T> void foo(Map<T, T> a) { }
...
Map<? extends String, ? extends String> m = ...;
foo(m);
```

```
Test1.java:9
Method <T>foo(Map<T, T>) of type Test1 is not
applicable to the argument of type(Map<? extends
String, ? extends String>), because:
  [*] Type variable T is invariant, but the type
  '? extends String' is not. However, replacing
  - '? extends String' on 8:13
  - '? extends String' on 8:31
  with String may solve the type conflict.
```

Figure 29: Type conflict caused by wildcards

```
<T> void foo(Map<? extends T, T> a) { }
...
Map<? extends String, ? extends String> m1 = ...;
foo(m1);
```

```
Test3.java:10
Method <T>foo(Map<? extends T, T>) of type Test3
is not applicable to the argument of type
(Map<? extends String, ? extends String>), because
  [*] The type String in Map<? extends String,
  ? extends String> on 8:9(8:13) is not a subtype
  of the inferred type for T: ' ? extends String'.
  However, replacing ' ? extends String' on 8:31
  with String may solve the type conflict.
```

Figure 30: Supertype conflict

5.3.4 Extends Final

This heuristic is the dual of the previous one. It attempts to solve type conflicts caused by upper bound wildcards, where the bound is a final type, e.g. `String` and `Integer` in the `java.lang package`. Consider the code in Figure 29. The method invocation fails, because `T` can not be instantiated to two captures of `? extends String`. Since the type `String` is final, i.e. can not have subtypes, the variable `m` can only be instantiated with `Map<String, String>`. Therefore, the heuristic substitutes the wildcards with `String` and verifies whether the invocation will type check. It is clear from the error message, that the substitution of the wildcards will correct the type conflict.

Wildcards cannot only cause type equality conflicts, but also supertype conflicts. Consider, for example, the code in Figure 30. The constraints generated for this invocation are: $\{\texttt{T} = \omega, \texttt{String} <: \texttt{T}\}$, where $\omega = \text{capture}(\texttt{? extends String})$. It is clear from the constraints that `T` will be instantiated to ω, but ω is not a supertype of `String`; ω itself is a subtype of `String`. But if we change the second wildcard in the declaration of `m1` to `String`, then the invocation will type check, because `T` will be instantiated to `String`, and `String` is of course a subtype of itself.

Upper bound wildcards are used sometimes to disallow write operations on values of parame-

```
<T> void foo(T a, T b,
             Map<? super T, ? super T> c) {}
...
Map<Number, Double> m = ...;
Number n = ...;
foo(1, n, m);
```

```
Test1.java:13
Method <T>foo(T, T, Map<? super T, ? super T>) of
type Test1 is not applicable to the arguments of
type(int, Number, Map<Number, Double>), because:
  [*] The type Double in Map<Number, Double> on
  11:9(11:21) is not a supertype of the inferred
  type for T: Number. However, replacing Double on
  11:21 with Number may solve the type conflict.
```

Figure 31: Subtype conflict

terized type. For example, given a list `l1` of type `List<Integer>`, we can prevent adding elements to this list by assigning it to a list `l2` of type `List<? extends Integer>`. Now, if we allow the access to `l1` only through `l2`, then we cannot write to `l2` because the type system will not allow it. To prevent repair suggestions, the extends final heuristic can be turned off with the command-line option `-df`.

5.4 The majority heuristic for subtype conflicts

This heuristic corrects subtype errors in a way similar to how the majority heuristic of Section 5.1 corrects equality constraint conflicts. The heuristic attempts to correct subtype conflicts by finding a type, or set of types, that satisfies as many constraints as possible. The constraints generated for the invocation in Figure 31 are

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

If we ignore the order in which constraints are processed, then we could say that `T` will either be $lub(\{\texttt{Integer}, \texttt{Number}\}) = \texttt{Number}$ or $glb(\{\texttt{Double}, \texttt{Number}\}) = \texttt{Double}$. If we choose `T` to be `Number`, then we have the following conflict: $\texttt{Number} \not<: \texttt{Double}$. But if we set `T` to `Double`, then we have two conflicts: $\texttt{Integer} \not<: \texttt{Double}$ and $\texttt{Number} \not<: \texttt{Double}$. Thus, instantiating `T` with `Number` will satisfy more constraints than instantiating `T` with `Double` will. Hence the error message.

In the previous example, the heuristic solved the conflict by comparing the number of conflicts caused by the result of *lub* and *glb*. But, sometimes *glb* cannot be computed such as in Figure 32. In this case, the heuristic computes the best possible *glb*, which is $\texttt{FocusEvent} = glb(\{\texttt{AWTEvent}, \texttt{FocusEvent}\})$. Comparing the number of conflicts that would arise if we instantiate `T` to `FocusEvent` or to `AWTEvent`, the *lub* of `FocusEvent`, `AWTEvent`, and `ComponentEvent` yields the following results:

- $\texttt{T} = \texttt{FocusEvent}$ implies $\texttt{ComponentEvent} \not<: \texttt{FocusEvent}$, $\texttt{AWTEvent} \not<: \texttt{FocusEvent}$ and $\texttt{FocusEvent} \not<: \texttt{String}$
- $\texttt{T} = \texttt{AWTEvent}$ implies $\texttt{AWTEvent} \not<: \texttt{FocusEvent}$ and $\texttt{AWTEvent} \not<: \texttt{String}$

Instantiating `T` with `AWTEvent` will obviously satisfy more constraints, thus the heuristic adds a repair hint to the error message, which also specifies all the reasons why the invocation does not type check.

```
<T> void foo(T a, T b, T c,
             Map<? super T, ? super T> d,
             List <? super T> e) { }
...
FocusEvent p1 = ...;
ComponentEvent p2 = ...;
AWTEvent p3 = ...;
Map<AWTEvent, FocusEvent> m = ...;
List <String> l = ...;
foo(p2, p3, p1, m, l);
```

```
Test2.java:34
Method <T>foo(T, T, T, Map<? super T, ? super T>,
List<? super T>) of  type Test2 is not applicable
to the arguments of type(ComponentEvent,
AWTEvent, FocusEvent, Map<AWTEvent, FocusEvent>,
List<String>), because:
  [*] The types AWTEvent in Map<AWTEvent,
   FocusEvent> on 30:9(30:13) and String in
   List<String> on 31:9(31:14) do not share a
   common subtype.
  [*] The types FocusEvent in Map<AWTEvent,
   FocusEvent> on 30:9(30:23) and String in
   List<String> on 31:9(31:14) do not share a
   common subtype.
  [*] The type FocusEvent in Map<AWTEvent,
   FocusEvent> on 30:9(30:23) is not a supertype
   of the inferred type for T: AWTEvent.
  [*] The type String in List<String> on
  31:9(31:14) is not a supertype of the inferred
  type for T: AWTEvent. However, replacing
  - String on 31:14
  - FocusEvent on 30:23
  with AWTEvent may solve the type conflict.
```

Figure 32: Non-computable glb

```
<T> void foo(List<? extends T> a, Map<T, T> b) { }
...
List<String> l = ...;
Map<Integer, Double> m = ...;
foo(l, m);
```

```
Test1.java:13
Method <T>bar(List<? super T>, Map<T, T>) of type
Test1 is not applicable to the arguments of type
(List<String>, Map<Integer, Double>), because:
  [*] Type variable T is invariant, but the types:
  - Double in Map<Integer, Double> on 12:9(12:22)
  - Integer in Map<Integer, Double> on 12:9(12:13)
  are not the same type.
[Warning]:
The types Double and Integer will cause a type
conflict with String in List<String> on 11:9(11:14)
```

Figure 33: Hidden error

In case the best possible *glb* causes the same number of conflicts as *lub*, the heuristic tries to compute the best possible *lub* that does not necessarily include all the types in the supertype constraints, but does satisfy more subtype constraints than the original *lub*. If after comparing the best possible *glb* and *lub*, it was determined that they both cause the same number of conflicts, then the heuristic resorts to comparing all the types in the constraints separately. If the heuristic still cannot find a type with a minimal number of conflicts, then the heuristic just chooses the type that is used the most, if possible.

5.5 Equality Warnings

Since the type constraints in Java are solved in a fixed order, there are certain errors that can remain undetected, because the constraint solver abruptly halts. Consider, for example, the code in Figure 33. The constraint solver stops while solving the equality constraints and does not check the supertype constraints because no type can be assigned to `T`. Thus, the user receives only a complaint about the type `Integer` and `Double` not being equal. The programmer might be completely unaware of the fact that even if he/she substitutes one of the types in the equality constraints with the other, the invocation will still not type check, because neither Integer nor Double are supertypes of `String`. Thus, in order to compensate for the fact that our type checking algorithm does not report all the reasons why the invocation does not type check, we have developed this heuristic that warns the programmer that the types in the equality are not only in conflict with eachother, but also with the supertype and/or subtype constraints. For the invocation in 33, the heuristic indeed generates an additional warning.

The purpose of the warning is to avoid the situation that the programmer repairs the equality conflict without verifying that *all* constraints will be satisfied. The heuristic, in general, generates a warning with a minimal set of types that will cause conflicts. Consider, for example, the following set of constraints:

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

The heuristic will report that only `String` is in conflict with `Integer` and `Number`, even though `Comparable<Number>` will also cause a conflict if the programmer decides to replace `Integer` with `Number`. The reason why we do not report the type

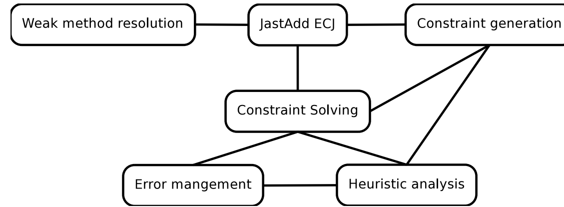

Figure 34: Architecture of our extension to the JastAdd EJC

`Comparable<Number>`, is because we suspect that `String` is the type that does not belong in the type constraints. Note that this heuristic will normally not report a warning if the maximal equality heuristic of Section 5.1 suggested a repair for the equality conflict.

6 Implementation

We have implemented our work as an extension to the JastAdd Extensible Java Compiler (JastAdd ejc) [2], which in turn was built on top of JastAdd [8]. The latter is an attribute grammar compiler that allows to specify compiler semantics in an aspect-oriented way by means of declarative attributes and semantic rules using ordinary Java code.

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 depicted in Figure 34: the type checker sends a method invocation which fails to type check to the weak method resolution subprocess which returns a set of methods. The type checker then generates type constraints for the method invocation and each method declaration returned by weak method resolution, using the constraint generation algorithm described informally in Section 4.

The type checker passes these constraints along to the constraint solver together with the return type of the method declaration and the type of the lvalue 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. If a set of constraints is not satisfiable, then the heuristics described in this paper may apply themselves, and they each may suggest a program fix by communicating their suggestion to the error manager.

Some heuristics are triggered by the presence of constraints of a given form. For example, it makes little sense to execute the majority heuristics for equality constraints in the absence of such constraints. Each heuristic decides for itself, again by communicating with the error manager, whether it makes sense to execute itself for a given incorrect method invocation.

If multiple heuristic extend the type error message with a suggestion, the error manager decides which of these will end up in the returned error message. At most one suggestion will be allowed for each error message, but this can be easily changed. For example, an IDE might offer all the suggestions ordered by priority, and update the priorities dynamically based on indications by the programmer. Currently, the error manager assigns static priorities to each of the heuristics.

Further experimentation is needed to establish the right priorities, if they are to be kept static. It is more likely, however that we shall follow the route taken in Hage and Heeren [7] to have the heuristics give an indication of their “belief” that the suggestion they provide indeed reflects the mistake that was made. For example, in the case of the majority heuristic the ratio between satisfiable and unsatisfiable constraints could be a measure of such a belief.

Although we have not discussed anything but method invocation in the paper, our extension already gives some limited support for constructor invocations.

To use the system you need `Ant` (http://ant.apache.org) and `subversion` (http://subversion.tigris.org/). Once you have these installed on your system, simply execute

```
svn co https://svn.cs.uu.nl:12443/repos/Swa5/project/
```

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.

7 Reflection

To guide the development of (new) heuristics it would be really helpful to know what kind of mistakes programmers actually make. Program logging systems like BlueJ might be able to help us there [9]. In fact, we have asked Matthew Jadud, the author of [9] whether there were any type incorrect programs involving generics collected during his investigation. Unfortunately, his work focused on novice Java programmers, and type errors are at that point not very common, let alone type errors that involve generics.

One promising idea, suggested by Pierre-Evariste Dagand in private communication, is to offer our system as a webbased 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 problematic programs that we can use to validate or improve on our work.

We would like to conclude with a number of design principles that served us really well:

- *Leave the (very complicated) type checking process intact.* It avoids proving the type system is not changed by the modifications, and the separation of concerns will make it both easier to engineer and easier to add to an existing compiler.
- *Ignore, at first, what is likely to be wrong.* A major problem to obtain to good type error messages when following the JLS, is that potential methods may be disqualified at an early stage, due to a mistake in the generic parts of a type. By ignoring the generic type information at that stage, more methods will be considered, and for each of these a more precise type error message may be obtained.
- *Do not easily throw away information.* For example, decomposing type constraints into their constituent parts may make solving easier, but it complicates generating good feedback.

Furthermore, we advise Java programmers to beware for mistakes in the implementation of the current JLS, even in industry-strength compilers. Finally, we hope that the next version of the JLS will take into account that from the specification tools must be built that not only implement the JLS, but can also provide useful feedback for programmers. Although type errors may not be as frequent in Java as they are in Haskell, the addition of new type system features will certainly work to redress the balance.

8 Conclusion and Future Work

We have described how to add heuristics to an extension of the type checking process of Generic Java that keeps track of more information about the constraints on types in the program and keeps the information around longer. The heuristics were developed to counter the bias that may be introduced by the JLS and the particular implementations of the type checking process specified by the JLS, and to capture knowledge about often made mistakes in generic method invocations. The benefit of these heuristics is that they can advise the programmer on what might be wrong with the program, and suggest how the inconsistencies may be resolved. We have illustrated our work

by a number of examples and have made a download available in which our work is implemented as an extension to the JastAdd ejc.

There are three important directions for future work: to empirically validate our work, to devise new heuristics, and to extend our work beyond method (and constructor) invocations. We have already indicated how we might go about the first of these in Section 7.

Regarding the topic of devising new heuristics, the issue consists of two parts: to devise more heuristics for generic method invocations, that assume that a mistake has been made in the generic part, and to devise heuristics for dealing with other issues, e.g., accessibility and visibility of identifiers.

The heuristics we have devised thus far follow the JLS in considering one method invocation at the time. However, if a lack of understanding of this fact is the reason for a mistake, then there is no way our heuristics can come up with a correct diagnosis. For that they need to be extended into considering larger sets of constraints, arising from, e.g., multiple invocations at the time. In other words, to discover a misunderstanding of the way type checking is compartimentalized in the JLS, the heuristics themselves need to go beyond that compartimentalization.

A particularly interesting language construct when we want to extend beyond method invocations, is that of inner classes. In that case, we shall have the additional problem of dealing with the scope of type variables, and all the mistakes programmers can make in these situations. A can of worms waiting to be opened.

We note that our work takes the JLS as a starting point, and we have yet to consider alternative approaches, as part of, e.g., Scala [5] and Timber [11]. 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.

In view of all the possibilities for future work, it would really be useful to have some idea where to extend the work first. For that it would be really helpful to have a collection of type incorrect Java programs. The website mentioned in Section 7 can be of use, but since it may take some time before that starts to deliver, we would like to take this opportunity to advertise that right here: if you have a collection of programs to contribute, please contact the second author via e-mail.

Acknowledgments

We acknowledge the involvement of Martin Bravenboer during the early stages of this project.

References

[1] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[2] T. Ekman and G. Hedin. JastAdd Extensible Java Compiler. http://jastadd.cs.lth.se/web/extjava.

[3] N. el Boustani. Improving type error messages for generic java, 2008. `http://www.cs.uu.nl/wiki/Hage/MasterStudents`.

[4] N. el Boustani and J. Hage. Improving type error messages for Generic Java. In G. Puebla and G. Vidal, editors, *Proceedings of the ACM SIGPLAN 2009 Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '09)*, pages 131–140. ACM Press, 2009.

[5] M. Odersky et al. The Scala homepage, 2008. `http://www.scala-lang.org/`.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The Third Edition*. Addison-Wesley Professional, 2005.

[7] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation of Functional Languages – IFL 2006*, volume 4449, pages 199 – 216, Heidelberg, 2007. Springer Verlag.

[8] G. Hedin and E. Magnusson. The JastAdd system - an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, April 2003. http://www.cs.lth.se/ gorel/publications/2003-JastAdd-SCP-Preprint.pdf.

[9] M. C. Jadud. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1):25 – 40, March 2005.

[10] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[11] J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The Timber homepage, 2008. `http://www.timber-lang.org`.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 5, pages 137–151. Pearson Education, second edition, 2003.

[13] D. Smith and R. Cartwright. 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)*, pages 505–524, New York, NY, USA, 2008. ACM.

[14] M. Torgersen, C. Plesner Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.

[15] J. A. Walz and G. F. Johnson. 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*, pages 44–57, St. Petersburg, FL, January 1986.

[16] J. Yang. Explaining type errors by finding the sources of type conflicts. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.