Improving Type Error Messages for Generic Java

Nabil el Boustani

Jurriaan Hage

Technical Report UU-CS-2008-038
October 2008

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. However, adding parametric polymorphism to a language that is built on inclusion polymorphism can be confusing to a novice programmer, because the typing rules are suddenly different and, in the case of Generic Java, quite complex. Indeed, the main Java compilers, Eclipse's EJC compiler 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. The extension has been implemented into the Jastad extensible Java compiler.

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.

Consider the code and the corresponding error messages in Figure 1. Both EJC and JAVAC only 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 with the invocation. Our message, on the other hand, does make such an attempt.

```
<T> void foo(Map<T,T> a){
  Map<Number, Integer> m1 = ...;
  foo(m1);
```

javac:

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

ejc:

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

ours:

```
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.
```

Figure 1: A code fragment with a type equality conflict and the three error messages that result.

Coming from a background of improving type error messages for polymorphic functional languages such as Haskell, we were interested to see whether the same ideas could be applied here. (In the functional programming world Java's notion of genericity is called parametric polymorphism; we shall continue to use the term generics.)

As it turns out, some ideas could be reused, but there large differences as well. The type checking process of Java is substantially different from the Hindley-Milner type system for the polymorphic lambda-calculus on which strongly typed functional languages are based [9]. One big difference is that in the functional programming world there is quite a difference between the elegant, declarative specification of the polymorphic lambda-calculus and the implementation of the inferencer [1]. In the case of Java, the type system and the type checking process are the same: the Java Language Specification [4] (henceforth referred to as JLS) specifies the set of type correct programs as those programs that are designated as such by the type checking algorithm. Since its publication, there has been more than one indication that the current JLS is not ideal: Smith and Cartwright show that type inference in Generic Java is neither sound nor complete [10]. It is important to realize that type inference is not the same as type checking: type inference, in the case of Java, refers to the facility that tries to come up with suitable instantiations for type variables that occur in the program. There is also a type checking phase that will verify that these choices are correct, and ultimately decides whether the program is type correct.

As Smith and Cartwright point out, incompleteness can usually be dealt with by adding type information to the program. An even more telling sign of the complexity of the JLS, might be that during our investigation we found that the two most important compilers, Sun's JAVAC and Eclipse's EJC, do not always agree on the type correctness of Generic Java programs.

Some of the complications in the JLS, however, seem unavoidable. Java already contains 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. 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.

In this paper we describe an extension to the type checking process as specified by the JLS. Its goal is to provide better feedback for *method invocations* that involve generics, a particularly important and complex part of the language. The essence of the approach, one that worked for us in the context of functional languages, is to hold on to more information about the program, and to hold on to it longer. The original operational description in the JLS retains information only long enough to make the yes or no decision. For example, it throws all information about the possible types to which a type variable may be instantiated, after it has randomly chosen one of these.

A second principle of our 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 that involves generics. In this way, we are sure not to change the set of accepted programs. In view of the complexity of the original type checking process, we believe that any change to the original process is a danger by itself.

The work described here is part of the master thesis project of the first author [3]. The second part of the thesis deals with heuristics that have been implemented on top of the development in this paper, in order to further improve diagnostics by also providing hints for correcting the mistake. For reasons of space we have decided to report on this subject in a separate paper.

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 quite a number of examples of type error messages constructed by our extension, together with the messages provided by EJC and JAVAC. In Section 4 we explain in some detail the type checking process as specified by the JLS, and discuss our extension to that process in Section 5. In Section 6 we shortly describe our implementation, and Section 7 reflects, concludes and gives directions for future work. Regarding the missing section on Relate Work: we simply could not find any work on improving type error messages for Generic Java. Therefore we have restricted ourselves to mentioning related work when the need for it comes up.

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 [4].

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 `Number`s 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 `Number`s, but if we assign it to a variable of type `List<Number>` we cannot safely guarantee this. 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 `Object`s from it. In a way `List<?>` plays the same role in Generic Java that `List<Object>` played before generics.

The wild card introduces a problem by itself. If we would like to write a reverse function for lists, 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* [11], 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 therefore also `LineNumberReader` $<:$ `Reader`.

When generics enter the picture, the situation becomes somewhat more complicated, formalized by the notion of *containment*. For parameterized/generic types, `C<S1,...,Sn>` $<:$ `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, typically at most one of the two is a class name. Since `&` is an associative, commutative operator, we freely omit parentheses in large type expressions, e.g., `Object & Serializable & Comparable<?>`.

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.

This is the arena in which type inference and type checking take place. In Section 4 we explain in more detail how they work. But first some example type error messages to whet your appetite.

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 (versions 0.771, 3.3.0 (Linux) and 0.780_R33x, 3.3.1 (Windows)) and JAVAC (version 1.6.0_03).

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.

Because we have already illustrated a type equality conflict in the introduction, we immediately continue with an example of a subtyping conflict. Consider the code fragment and associated messages in Figure 2. 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 Figure 2, but which involves `super` and not `extend`. A more interesting source of mistakes, and typical for supertype constraints, are due to the inability to find a single type that extends two different types. In the case of the code fragment in Figure 3, the return type of `foo` is `void` so it cannot be used to instantiate `T`. The type `T` should then be 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 Section 3.1 we show that these compilers sometimes exhibit strange behaviour for this kind of example.

We continue with some examples that involve a bound conflict, starting with Figure 4. Surprisingly, JAVAC gives exactly the same error messages as in Figure 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 Figure 5. In this particular case the error message of EJC is quite reasonable, although our message prefers not to resort to mentioning an intersection type, which is an artifact constructed by the type inference phase. Surprisingly maybe, JAVAC crashes for this particular program due to an infinite number of calls to the least upper bound function!

An example of a type error involving wildcards can be found in Figure 6. The code fragment shows what might be a typical mistake on the part of a novice programmer: that the type `? extends Number` equals `? extends Number` which, if provable, would make the invocation correct. However, this is not the case.

Our message follows the tenets of the manifesto of Yang et al [12] in that an error message should never reveal anything internal to compiler. However, the error messages provided by EJC

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

javac:

```
Listing2.java:11: <T>bar(Map<T,? extends List<T>>)
  in Listing2 cannot be applied to
  (Map<Number, List<Integer>>)
  bar(m2);
```

ejc:

```
1. ERROR in Listing2.java (at line 11)
  bar(m2);
The method bar(Map<T,? extends List<T>>) in the
 type Test is not applicable for the arguments
 (Map<Number,List<Integer>>)
```

ours:

```
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.
```

Figure 2: A code fragment with a subtyping conflict and the three corresponding type error messages.

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

ours:

```
Test6.java:7
Method
<T extends Number>foo(Map<? super T, ? super T>)
of type Test6 is not applicable to 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.
```

Figure 3: A code fragment followed by our type error message.

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

javac:

```
Listing3.java:6: <T>baz(List<T>) in Test cannot be
  applied to (List<Comparable<Integer>>)
  baz(l);
```

ejc:

```
1. ERROR in Listing3.java (at line 6)
  baz(l);
Bound mismatch: The generic method baz(List<T>) of
 type Test 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>
```

ours:

```
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`.
```

Figure 4: A code fragment with a bound conflict followed by the three corresponding type error messages.

```
< T extend Number> void foo(T a, T b){}
...
foo(1, false);
```

ejc:

```
1. ERROR in Test1.java (at line 12)
  foo(1, false);
Bound mismatch: The generic method foo(T, T) of
 type Test1 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>
```

ours:

```
Test1.java:12
Method <T extends Number>foo(T, T) of type Test1
is not applicable to the arguments of type
(int, boolean), because:
  [*] The type boolean of the expression ‘false’
      on 12:16 is not a subtype of T’s upper bound
      Number in ‘T extends Number‘.
```

Figure 5: Another code fragment with a bound error.

and JAVAC, which explicitly refer to a captured wildcard. The message of JAVAC states that it cannot find a particular symbol, which is a bit stronger even than saying that the invocation does not match any particular method signature.

3.1 Strange behaviour

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

In an earlier example, we saw that the type inferencer computes the largest subtype of a pair of types, and that such a type may not always exist. This is also the case for the program give in Figure 7, in which `T` should be instantiated to the largest subtype of `Integer` 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`. In fact, we found quite a few more programs similar, but different from this one that JAVAC accepts, but that should be rejected (see Listing 5.8 of [3]).

The EJC compiler also sometimes exhibits strange behaviour. Consider the code fragment in Figure 8. The constraints generated for both method calls are the same according to the JLS $\{\texttt{T} <: \texttt{Number}, \texttt{T} <: \texttt{String}\}$ but the type error diagnosis for these very similar programs by EJC is very different. This is due to how EJC resolves subtype constraints.

This is a typical, but not so pleasant phenomenon: the implementation of the type checking process leaks through in the type error messages. Since programmers typically have no knowledge how this is implemented, they are at a disadvantage when trying to interpret the messages. We saw two examples of this earlier: the captured wildcards and the intersection types computed by the inference process.

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

```
  <T> void foo(Map<T, T> a){
    Map<? extends Number, ? extends Number> m = null;
    foo(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:

```
Listing4.java:6
Method <T>foo(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 6: A code fragment with a wilcard equality error followed by the three corresponding type error messages.

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

Figure 7: A code fragment erroneously accepted by JAVAC.

```
<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>)
```

Figure 8: Two similar method calls as diagnosed by EJC.

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 Figure 9.

It starts off by performing *method resolution*, which determines, for a given invocation, a set of methods that the programmer may be invoking. We shall 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 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 quite easy, so we give only the decomposition for the set of constraints found for our example, and omit the details of Listings 3.1 to 3.3 in [3]

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

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

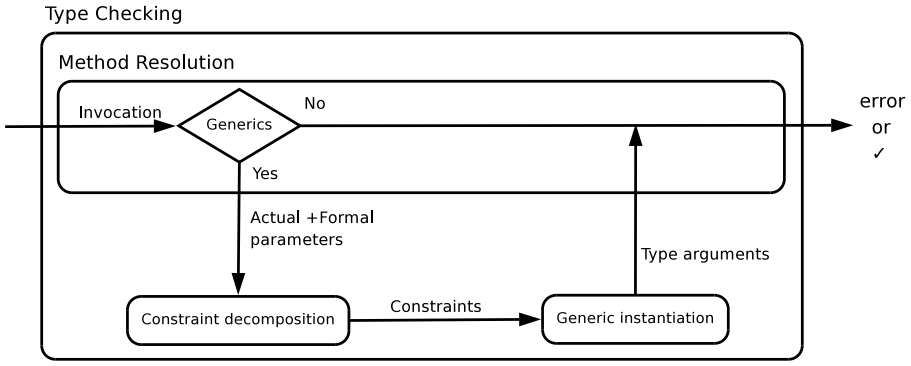

Figure 9: The type checking process

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 *least upper bound (lub)* of the two, `Number`. However, things are not always so simple: for $\{\texttt{Integer} <: \texttt{T}, \texttt{String} <: \texttt{T}\}$, 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 [12], believe a type checking process should have: it should only refer to types or expressions that 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 counterpart 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 that a type variable is involved in are of this kind and the invocation occurs in an assignment context, then the JLS specifies that the context should be used to determine to what type a type variable should be instantiated (if possible). 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 $\texttt{S} = \texttt{Integer}$. Together, the decomposed constraints are

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

The inference process then instantiates `S` to `Integer` on the basis of the third constraint above; `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.

For reasons of space, we cannot give the full details of these processes. Full details for dealing with equivalence constraints, subtype constraints and supertype constraints can be found in Listing 3.5 of [3].

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 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 — 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 Method resolution

The process of method resolution is a complex one, due to such features as overloading, overriding and visibility. It 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 methods of method invocation. Examples respectively are `a.byteValue()`, `this.Foo()`, `super.intVal()`, `Baz.super.intVal()` and `Collections<String>.emptySet()`. In the case of `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 method 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 $1 \leq i \leq n$:

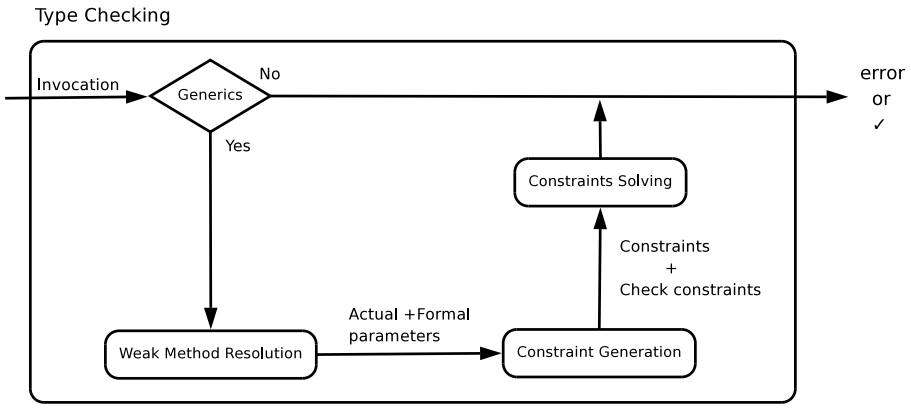

Figure 10: The modified type checking process

- $A_i <: F_i$, 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 and 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, and, 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.

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 complex, stepwise process. In our extension of the type checking process, which we describe in the following section, we relax the restraints somewhat to try and figure out which method the programmer might have been trying to call.

5 The Modified Type Checking Process

The overall architecture of our modified type checking process can be seen in Figure 10. The structure of the process is not much different, only the phases themselves will 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.

```
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);
```

Figure 11: A utility class.

5.1 Weakened method resolution

First off, 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 side by side to see which method the programmer most likely intended to call.

A major decision we have to make is how exactly we weaken method resolution. Since we are interested in improving type error messages for generic invocations, we choose to drop generic information from the invocation and the candidate methods, and perform our comparison between the results. To be more precise, 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 full specification of the process is given in pseudo code in [3] (Listings 4.5-4.7), but are too large to include. We describe the process informally by an example. Consider the code in Figure 11 where we define a many-times overloaded method `foo`. We designate `foo` on line x by $\texttt{foo}_x$.

We consider the first 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

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

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

```
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 12: A code fragment with two candidate methods.

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

Figure 13: Inference succeeds, but checking fails.

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 of 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 `List` $<:$ `Collection`. Similarly, $\texttt{foo}_7$ and $\texttt{foo}_{10}$ are removed in favour of $\texttt{foo}_{13}$. For the second and third invocation in Figure 11 the same set of candicates is obtained.

In all cases, we ended up with a singleton set, but the our resulotion method does not demand this, contrary to the original resolution method. For the code fragment of Figure 12, for example, both declarations of `bar` pass method resolution.

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 Figure 13. 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 extended version, we provide the constraints 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 constraints decomposition algorithm is extended to generate additional constraints, which are left alone during decomposition. For the example in Figure 13, the new constraints 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

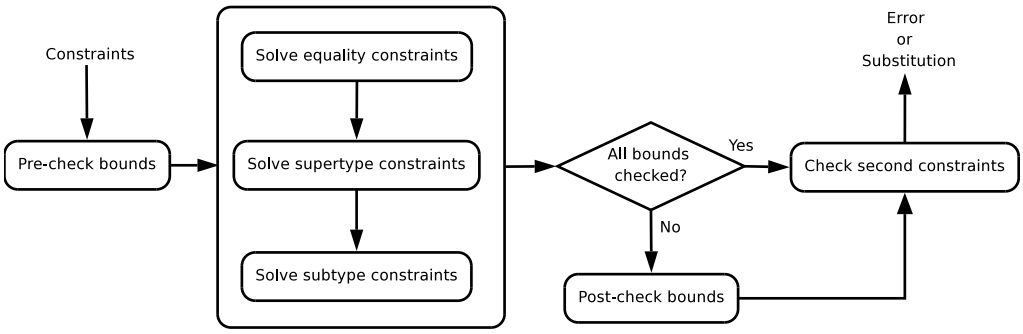

Figure 14: The constraint solving phase

assignments came from. The constraint decomposition phase, only applied to the left operand of $\Cup$, is exactly the same as for the original process.

Coming back to the example of Figure 11, we generate constraints for the invocation of `foo` at line 20 to type check. The only remaining candidate method is $\texttt{foo}_{13}$, in which case we obtain

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

For the second invocation we obtain

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

and for the third

$$\{\texttt{T} = \texttt{Number}, \texttt{Double} <: \texttt{T}\} \Cup$$
$$\{\texttt{LinkedList<? extends Number>} <: \texttt{LinkedList<? super T>}\}$$

5.3 The constraint solving phase

In Figure 14 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 a list of type error messages, in case of failure. 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.

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`[1], the lub of `String` and `Number`. Then it proceeds to instantiate possible other type variables, and afterwards 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 can't 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 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 inconstency will result, and a type error message can be generated immediately.

A 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

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

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

Figure 15: Order of inference matters.

of the type system solve constraints in different orders and that that influences the error message the implementations provide [8, 6].

In our inference algorithm, type variables are considered separately, but because we involve the bound 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 Figure 15.

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`: $\texttt{S} <: \texttt{Integer}$. During the pre-check we can then establish that the constraint $\texttt{S} = \texttt{Integer}$ is consistent with the bound and $\texttt{S} = \texttt{String}$ is not.

We have chosen the following method to order the type variables involved: a type variable `U` *depends on* a type variable `V`, if `V` occurs in a bound for `U`. In our example, `S` depends on `T`. The idea is then to first infer type variables on which many other type variables depend. 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. If we count dependencies transitively for the following type parameter declaration

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

then we obtain 3 for `T`, 4 for `S`, 2 for `R` and 1 for `U`. Hence, type inference should start by inferring `S`.

5.4 The inferencer

The large rounded rectangle in Figure 14 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 in the way described in the previous section.

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 Figure 16 gives the pseudocode to describe the inference process.

We conclude this section by revisiting the running example of Figure 11.

For the first invocation we found

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

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 $\texttt{Double} <: \texttt{Number}$, the constraints are satisfied. However, the non-atomic constraint is not satisfied, so the method invocation fails to type check.

```
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
```

Figure 16: A pseudocode algorithm for type inference

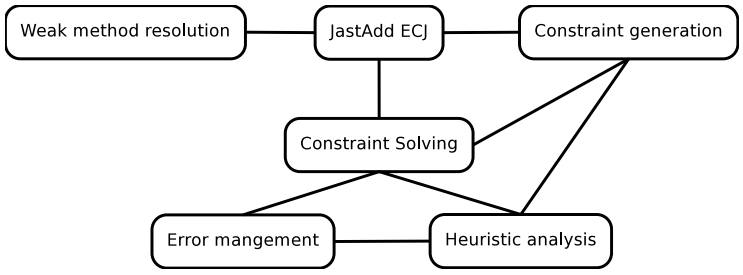

Figure 17: Architecture of our extension to the JastAdd EJC

6 Implementation

We have implemented our work as an extension to the JastAdd Extensible Java Compiler (JastAdd EJC) [2], which in turn was built on top of Jastad [5]. 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 can be found in Figure 17: the type checker sends a method invocation which fails to type check 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 Section 5.

The type checker passes these constraints along to our constraint solver together with the return type of a 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. This is mainly to facilitate a number of heuristics we have implemented to suggest fixes for the type error. Due to reasons of space, we report on the heuristics in a separate paper.

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

6.1 Using the system

To use the system you need `subversion` (http://subversion.tigris.org/) and `Ant` (http://ant.apache.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. Note that many error messages also suggest a problem fix using heuristics that we have not discussed in this paper; feel free to ignore these. The developments in this paper are responsible for the remainder of the error message. Have fun.

7 Conclusion, Reflection 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. The main ideas are to keep track of more information and either to keep that information around longer or to perform certain checks earlier to benefit from information still being around.

Another general idea that we have applied is that if you want to say something sensible about, e.g., the methods a programmer might, erroneously, want to call, you should not dismiss eligible methods soon in the type checking process. As a general rule, you should make the implementation roomier, allowing for mistake and not judge and dismiss too soon.

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 Extensible Java Compiler.

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, but if a lack of understanding on the part of the programmer of this particular fact is the reason for the mistake, we can only find out by going beyond the compartments.

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 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. To have some idea in which direction to look it would be really helpful to know what kind of mistakes programmers make (if the mistake does not involve only generics). Program logging systems like BlueJ might be able to help us there [7].

We would like to conclude with a number of lessons we have learnt during our investigation:

- The combination of subtyping and generics is a hard one.
- Programmers: beware of a lack of type information propagation, especially if you come from a background of the polymorphic lambda-calculus.
- Compiler builders: be thankful for the lack of propagation, because it makes it easier to explain type errors.
- Leave the (very complicated) type checking process intact. It avoids needing to prove that the existing process was not changed in any essential way.
- Beware for mistakes in the implementation of the current JLS, even in industry-strength compilers.
- The amount and complexity of our work suggests that the current JLS is not declarative and intuitive enough to be used by (novice) programmers.

Smith and Cartwright [10] show how the Java JLS might be "fixed". Although soundness and completeness are obviously 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 to the JLS will take those properties into consideration as well.

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. `http://www.cs.uu.nl/wiki/Hage/MasterStudents`.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley Professional, third edition, July 2005.

[5] 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.

[6] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/phdthesis`.

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

[8] O. Lee and K. Yi. A generalization of hybrid let-polymorphic type inference algorithms. In *Proceedings of the First Asian Workshop on Programming Languages and Systems*, pages 79–88, National university of Singapore, Singapore, December 2000.

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

[10] 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 ??–??, 2008.

[11] 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.

[12] 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.