Improving Type Error Messages for Generic Java

Nabil el Boustani

27th April 2008

Abstract

Generics, alternatively called parametric polymorphism, are a powerful programming concept that makes higher and cleaner abstractions possible, which help developing reusable code. Java supports parametric polymorphism since its 1.5 release, where it was perceived as one of the great and most advanced features added to the 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 Java, quite complex.

To help novice programmers get accustomed to the new typing rules, we describe a framework in this thesis that can be used by the type checking process to generate error messages that explain why a type error occurs and contain additional hints, which describe how to correct a type error or prevent new errors in future compilations. This framework only supports the type checking of generic method invocations, where the compiler must infer the type arguments of a method. First we show what constitutes this framework, and then discuss a number of heuristics to further improve the error messages. We also provide many examples of error messages generated by our framework and compare them with the error messages generated by the Sun java compiler and Eclipse java compiler.

Contents

Chapter 1

Introduction

A quick browse through the literature shows that most compilers for functional languages such as ML and Haskell, sometimes generate error messages that are not easily understood by novice programmers. One of the reasons why these error messages are regarded as cryptic by novices is that compilers do not or cannot always locate the real source of a type error [14].

Since the introduction of generics to Java, we believe that beginner programmers will experience the same thing. In this thesis we aim to develop an extension for the type checking algorithm already present in the Java compilers to provide better feedback to novice programmers. The thesis considers only the problem of generating constructive error messages for generic method invocations. The extension presented here is used by the type checking algorithm only when generic method invocation fails to type check. In this case, the type checking algorithm will request the extension to examine the failed invocation to provide an error message. This extension analyzes the method invocation to locate all possible sources of the type errors, and may subsequently apply a number of type heuristics that will attempt to describe corrections for the type conflicts. The type heuristics describe only how to change method invocations and not method declarations. Suggesting to change the later requires a global analysis of a program instead of a pratial analysis as we do.

1.1 Motivating examples

Consider the code in Listing 1.1. The method invocation is rejected by the compiler, which claims that there is no method declared with the signature *foo(Map<Number, Integer>)*. This can be very confusing to a programmer, because he/she is trying to invoke the generic method declared with the signature *foo*(*Map*<*T,T*>). The programmer probably thinks that since *Integer* is a subtype of *Number*, it might be legal to call *foo* with *Map*<*Number, Integer*>. This is, however, not correct, because a type variable can only be instantiated with a single type. The error messages generated when compiling the code with

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

Listing 1.1: Type equality conflict

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

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

Figure 1.1: javac and ejc error messages for the invocation in Listing 1.1

the Sun java compiler (javac) and the Eclipse java compiler (ejc) are given in Figure 1.1, respectively. Both compilers state in their error message that the method *foo* cannot be applied to the argument `m1`. It would be, however, instructive to the programmer if the error message would explain why the method invocation fails. In this case, the invocation fails because the type system expects the generic type *Map* in the actual parameter to be instantiated with exactly the same type, i.e. either both with *Number,* or both with *Integer*.

Now take for example the code given in Listing 1.2. The method call is rejected by javac and ejc complaining that the method with the signature *<T>bar(Map<T,* ? *extends T>)* cannot be applied to *Map<Integer*, *Number>* (see Figure 1.2). These error messages do not clarify why the method is not applicable to the actual parameter. Therefore, they do not provide the programmer with the slightest idea on how to correct his/her mistake. For this example, it would be more appropriate to inform the user that the type system instantiates the type variable T to *Integer*, which then causes a conflict with *Number* in the actual parameter, because *Number* is not a subtype of *Integer*.

Consider the last example given in Listing 1.3. The compiler javac generates the same error message for this method call as it did for the invocation in Listing 1.1. Aside from the fact that the error message is not very helpful, the compiler should have generated a different error message just like ejc, because

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

Listing 1.2: Subtyping conflict

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

```
1. ERROR in Test.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>>)
```

Figure 1.2: javac and ejc error messages for the invocation in Listing 1.2

the reason why this invocation failed is completely different from why the invocation in Listing 1.1 is rejected. The method call 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 method invocation is incorrect.

1.2 Structure of this thesis

In Chapter 2 we provide some background information about generics in Java, and introduce some basic concepts and definitions needed to understand this thesis. Next, in Chapter 3 we review the current type checking of method invocations in Java. Chapter 4 describes how the type checking process is extended to produce better error messages, and provide two examples to illustrate how

```
static <T extends Number> void baz(List<T> a){}
...
List<Comparable<Integer>> l = ...;
baz(l); // Error
```

Listing 1.3: Bound conflict

```
Test.java:16: <T>baz(java.util.List<T>) in Test cannot be applied to
(java.util.List<java.lang.Comparable<java.lang.Integer> >)
        baz(l);
```

```
ERROR in Test.java (at line 16)
        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 para-
meter <T extends Number>
```

Figure 1.3: javac and ejc error messages for the invocation in Listing 1.3

the extension works in general. Chapter 5 provides a list of different kinds of type errors and shows how the error messages generated for these type errors by our framework compare to the error messages generated by the only two Java compilers that support generics at the moment: The Sun java compiler and the Eclipse java compiler. In Chapter 6 we present a number of heuristics that we have developed to further improve the quality of an error message by providing repair hints. We also provide examples for each type heuristic to illustrate how the additional hints improve an error message. Chapter 7 will conclude the thesis.

Chapter 2

Type Systems and Java Generics

In this chapter we provide some background information about generics (parametrized polymorphism) in Java and introduce some of the basic concepts and terminology used in the rest of this thesis. Readers who have some programming experience with Java generics may skip to the next chapter.

2.1 Type systems

A type system in its broadest sense is defined in Types and Programming Languages [8] as:

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

This definition consists of two parts. The first part identifies a type system as a means of reasoning about the behavior of a program. The second part describes a type system as a classification system that arranges and groups program values and expressions into categories based on their meaning. These categories are called types. The meaning of the term type in computer science very much resembles the meaning of sets in mathematics. In mathematics the symbol $\mathbb{N}$ denotes a set of natural numbers, also known as counting or ordering numbers. Thus, when defining a variable α as a member of $\mathbb{N}$ ($\alpha \in \mathbb{N}$), α can only take one of the values contained in $\mathbb{N}$. The same principal also holds in Java's type system. When declaring a variable of type unsigned *int*, then this variable can only take a value that is defined in $\mathbb{N}$. Note, however, that due to hardware limitations the type *int* is a finite set, thus *int* is actually only a finite subset of the infinite set $\mathbb{N}$. Furthermore, types in computer science also restrict the operations that can be performed on a value, e.g. the strings "Hello" and

"world!" can be concatenated together, but it makes no sense to multiply them together.

Type systems vary considerably from one programming language to another, but they are all implemented for the same purposes:

- Safety: detect meaningless operations such as: "hello world" + 2, which can lead to run-time errors.
- Efficiency: information about types of expressions can help the compiler choose better program representations and more efficient machine instructions.
- Abstraction: allow programmers to think about programs at higher level instead of just as a sequence of machine instructions or bits.

Type systems can be classified in several ways. One reasonable way of categorizing type systems is to look at how they reason about programs. There are basically two kinds of type systems:

- Type checkers: require program code, typically the identifiers, to be annotated with type information which they can use to verify that every expression in the program can be assigned a correct and appropriate type.
- Type inferencers: require a minimal amount or no type information at all to be present in the program code, because they can deduce the type of expressions and identifiers based on how they are used.

2.2 Java Generics

Generic or parametrized typing was an advanced feature that has been long anticipated by the Java community. It was introduced in Java version 5 (JDK 1.5) with the primary purpose of creating type-safe collections. Before the release of JDK 1.5, collections in Java were heterogeneous, meaning that collections can contain elements of different types. This was possible because every reference type in Java is a subtype of *Object* and collections were made to contain *Object* references. So in a situation where a homogeneous collection is required, programmers had to rely on casts to make sure that an element returned from a collection is indeed of the right type.

```
List l = ...;
l.add(new Object());
l.add("string");
l.add(new Integer(1));
String myStr = (String)l.get(1);
myStr        = (String)l.get(2);
```

Listing 2.1: Heterogeneous collection

The second assignment in Listing 2.1 will succeed at compile time, but will fail at run-time, because the third element in the list is not a string. Having to keep track of types of elements added to and read from a collection is a tedious and error-prone job that can be better performed by the compiler. Java generics added that extra safety to Java, by detecting errors such as the one above at compile time instead of run-time[2]. To be able to detect errors at compile-

```
interface List<E>{
    public boolean add(E o);
    ...
}
```

Listing 2.2: Declaration of a homogeneous collection

time, the existing *List* interface has to be modified to include a type parameter. The type parameter functions as placeholder for the list's element real type, see Listing 2.2. The type checker ensures that a list contains only elements of the same type or elements with a type that is a subtype of the list's element type, e.g. a list of numbers can contain elements of type number and elements of type double or integer. Instantiating the *List* with a type argument *String* makes casting unnecessary when reading from the list, because any attempt to add anything to the list but a *String* will be immediately detected by the compiler as an error, as illustrated in Listing 2.3.

```
List<String> l = ...;
l.add(new Object()); // compile-time error
l.add("string");
l.add(new Integer(1)); // compile-time error
String myStr = l.get(1); // no cast required
```

Listing 2.3: Using a homogeneous collection

Java not only allows you to define parametrized types, but also parametrized or generic methods. The utility class *Collections* in java.util package contains several examples of generic methods. One of these methods is *<T> void fill(List<? super T>, T)*, which replaces all the elements in a list with the second parameter in the method declaration. Generic methods can be called in two ways:

- the user supplies the type arguments that type variables need to be instantiated with, e.g. *Collections.<Integer>fill(someIntegerList, 23),* or
- the type arguments are omitted, e.g. *Collections.fill(someIntegerList, 23).* In this case, the compiler is smart enough to instantiate the type variables with types that will make invocation succeed.

2.3 Subtyping and Containment

Since types are much like sets, they can also be partially ordered. The ordering operator for types is a subtyping operator denoted by '<:'. A type C is considered to be a subtype of D ($C <: D$), if D can be substituted by C everywhere D is used or expected. In terms of sets, every element of C is a element of D. The subtyping operator is reflexive and transitive, meaning that $C <: C$ and $(C <: D \wedge D <: E) \Rightarrow C <: E$. To give a concrete example consider the classes *Reader, BufferedReader* and *LineNumberReader* available in the package *java.io,* which can be used to read character streams. Since the class *LineNumberReader* extends *BufferedReader* and this later class extends the class *Reader,* then *LineNumberReader* also extends *Reader, i.e. LineNumberReader* $<:$ *Reader.*

Subtyping among normal types, i.e. non-generic types, is simple and straightforward, but subtyping among generic or parametrized types requires an additional kind of ordering known as containment, denoted by '$\leq$:'. A parametrized type $C<S_1, \ldots, S_n>$ is considered to be a subtype of $D< T_1, \ldots, T_n >$ if $C <: D$ and $1 \leq i \leq n$, T_i contains S_i ($S_i \leq: T_i$). Containment itself is also partially defined in terms of subtyping as shown below:

- ? *extends* T $\leq:$? *extends* S if $T <: S$
- ? *super* T $\leq:$? *super* S if $S <: T$
- $T \leq: T$
- $T \leq:$? *extends* T
- $T \leq:$? *super* T

2.4 Type variance

From the previous section it is clear that types can be partially ordered, but sometimes it is also important to express how types can vary from each other in a type system. The notions covariance, contravariance and invariance were introduced to represent this change.

2.4.1 Covariance

Covariance indicates that applying a type operator to two types preserves the ordering on the result types. In Java, for example, *Integer* $<:$ *Number* and *Integer*[] $<:$ *Number*[]. Thus arrays in Java are covariant, because their ordering is the same as the ordering of their component type, in this case Integer and *Number*. Another example of covariance in Java is the return type of a method. If a method in class returns some type C, then it can be overridden by another method in a subclass to return a subtype of C.

2.4.2 Contravariance

Contravariance is the exact opposite of covariance. Applying a contravariant operator to two types reverses the ordering of the result types. Suppose that a function f takes an argument of type C and returns a value of type D (written as $C \to D$), then substituting f with another function g of type $C' \to D'$ requires that $C' \to D' <: C \to D$. This is a very logical condition, but what does that mean for the types C, D, C' and D'? Assuming that $(\to)$ is a covariant type operator, one can conclude that $C' <: C$ and $D' <: D$. For the types D and D' this is correct, because D' can replace D anywhere. However, the condition $C' <: C$ breaks the substitution. Assuming that f takes a *Number* as a parameter, then g must take a subtype of *Number*, such as *Integer* or *Double*, as its parameter. But in this case g can not replace f completely, since a call to g with a *Number* as a parameter would be illegal. Thus, instead of $C' <: C$, we must demand the opposite $C <: C'$. Now it is guaranteed that any call to f is also a legal call to g. Thus, the operator $(\to)$ is contravariant in the parameter types. Because this is somewhat counter-intuitive and because of some other issues, some type systems avoid contravariance and prefer to restrict the types of parameters of the substituting function to be exactly the same as the parameter types of the substituted function.

2.4.3 Invariance

Invariance is the middle ground between covariance and contravariance. When a type operator is both covariant and contravariant, then it is said to be invariant. It was mentioned earlier that arrays in Java are covariant, but that is not entirely correct. An array of bytes (*byte*[]) is not a subtype of array of ints (*int*[]). *bytes*[] and *int*[] are in fact completely different from each other, despite the fact that *byte* is subtype of *int*. Arrays in Java are invariant if the base type is a primitive type (not a reference type).

Aside from arrays of primitive types, concrete parametrized types in Java are also invariant. Consider for example the interface *Set*<*E*> in the package *java.util*, where E is a type variable that can be instantiated with any reference type. Although *Integer* <: *Number*, it is not the case that *Set*<*Integer*> <: *Set*<*Number*>. Note, however, that it is allowed to add integers to a set of type *Set*<*Number*>, but only numbers can be read from the set.

2.5 Wildcards

Although in most cases, programmers use homogeneous collections or data structures in general, it is sometime necessary to be able to write code that does not depend on the data structure's type parameter. Prior to the introduction of generics, one could write the method in Listing 2.4 to print all the objects in a collection.

```
public void printAll(Collection co){
```

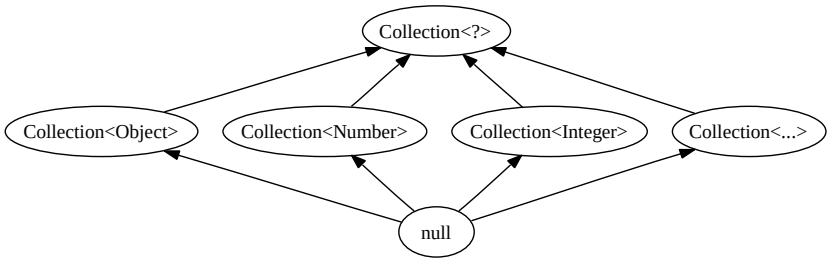

Figure 2.1: Generic subtyping relation

```
    for(int i = 0; i<co.size(); ++i){
        System.out.println(co.get(i));
    }
}
```

Listing 2.4: Print a heterogeneous collection

Although the code above will function just fine with generics code, it is considered bad style to use parametrized types in their raw form. Besides, support for the raw types may become deprecated in the future. Therefore, adapting the above code to create a polymorphic method may yield the following method signature: *void printAll(Collection<Object>)*. However, a method with this signature will accept only collections that contain objects with type *Object*. This seems a bit odd at first, but it actually makes sense if you know that *Collection<Object>* is not a supertype of all parametrized *Collection* types. In fact the subtype relationship between the types defined by the interface *Collection<T>* is like a lattice that is infinite in width with *null* as its bottom and *Collection<?>* as its top, see Figure 2.1. The symbol (?) stands for the *unknown* type, that can be replaced by any other type. Thus, the correct signature of the above method, must be: *void printAll(Collection<?>)*.

The invariance of concrete parametrized types is in some occasions very restrictive and even problematic. Consider, for example, defining a class field whose type is *List* with an arbitrary element type [13, 12]. Declaring a field of type *List<Object>* would be incorrect due to the invariance, but *List<?>* will be correct, because every possible instantiation of *List* is a subtype of *List<?>*. Wildcards add a great deal of flexibility to Java, because types instantiated with a wildcards can be either covariant or contravariant. The subtype relationship between wildcard parametrized types is illustrated in Figure 2.2.

Wildcards add a nice abstraction and expressive power to the language, but they also introduce a layer of complexity and problems. Consider a reverse method, which takes a list and returns a copy of this list with the elements in the reverse order. This method could be declared as: *<T>List<T> reverse(List<T>)*. This method will work for any kind of list except for lists instantiated with a wildcard, such as *List<?>*. This is because the exact type

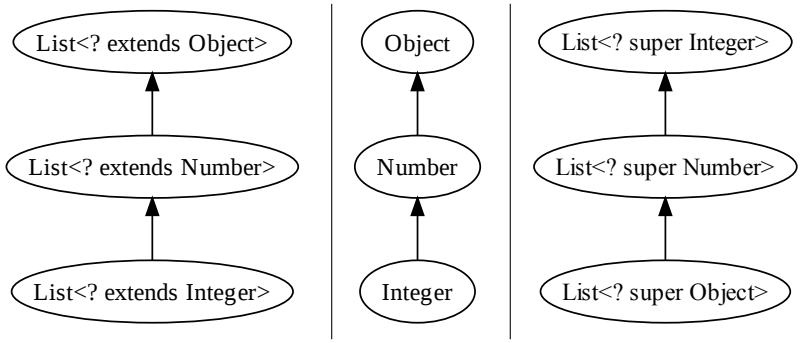

Figure 2.2: Subtyping between wildcard parametrized types

of the list's element cannot be deduced. Weakening the method's signature to: *List<?> reverse(List<?>)* will solve the problem, but we will lose the information that the returned list is of the same type as the list passed to the method. To handle this situation a mechanism called *wildcard capture conversion*[13] was defined that makes it possible in some type safe situations to call generic methods even though no concrete type can be inferred for the type parameters. Wildcard capture is based on the observation that even though the actual type of *List<?>* may not be known, at the moment that the method *reverse* is called, the list will have a certain specific element type. Thus, the run-time type behind the wildcard is captured and used to instantiate the type variable *T* with, hence the name wildcard capture. This, however, works only on top level type variables, e.g. *List<List<?>>* can not be passed to a method expecting *List<List<T>>*, because two elements of the list with the type *List<?>* may have different element types, which can not be captured by *T*.

Since wildcards are not concrete types that are known at compile-time, it is forbidden in Java to use them as type arguments to call generic methods. For example, *Collections.<? extends Number>fill(someIntegerList, 23)* is illegal.

2.6 Bounds

Since type parameters are merely placeholders for real types provided at class instantiation or method invocation, it is not possible to call a method or access a field of the value of a type parameter in the body of the declaring class or method. Consider, for example, a method *compare* with the signature *<T> int compare (T a, T b)* that can be used to compare objects. In the body of *compare* we cannot call the method *compareTo* provided by the interface *Comparable*, because the type that the type variable *T* will be instantiated to during an invocation it is not guaranteed to be a subtype of the *Comparable* interface. To narrow the range of possible types that a type parameter can take, Java provides a way to restrict the instantiation of parametrized types to make polymorphic code somewhat specific. To illustrate this, suppose that we have a class *Matrix*

in Listing 2.5 which allows to perform basic math operations on it such as "add" and "multiply". In this case it is very reasonable to constrain the methods of Matrix to work only on Matrix objects and to restrict the instantiation of *Matrix* to types that are numbers.

```
class Matrix<T extends Number>{
    <S extends Matrix<T>>void add(S other){ ...}
    <S extends Matrix<T>>void multiply(S other){ ... }
}
```

Listing 2.5: Upper bound restriction

The *extends* keyword in the class declaration forces the type parameter passed to the *Matrix* constructor to be at most of type *Number*. Thus, any subtype of *Number* can be used to instantiate the *Matrix* class. Since *Number* is a class, any subclass of *Number* is a good candidate. Types such as *Object* or *String* are not accepted, because they do not extend the class *Number*. On the other hand, if *Number* was an interface than any class implementing the *Number* interface will make the instantiation of *Matrix* correct. This restriction ensures that elements of a matrix can indeed be added together and multiplied by each other. The same effect is achieved in the method declarations.

Besides the upper bound, Java also allows to specify a lower bound, which is useful in situations where the user is interested in the upper part of the inheritance tree instead of the lower part. This can be illustrated by an example from Java Collection Framework. The class *TreeSet<E>* has four different constructors and one of these constructors is *TreeSet(Comparator<? super E>)*. This constructor provides the user with the flexibility to use the comparator defined for a supertype of E in case E itself does not override the implementation of the *Comparator* interface. Another situation where it is useful to have a lower bound is given in Listing 2.6. Instead of restricting the type of the sink to be *Sink<T>*, we can use any sink object that was instantiated with a supertype of *T*.

```
public static <T> T writeAll(Collection<T> coll
                             , Sink<? super T> snk){
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last;
}
```

Listing 2.6: Lower bound restriction

Chapter 3

Current Type Checking

This chapter discusses the portion of the type system of Java which is responsible for type checking method invocations. An overview of method invocation type checking is given in Figure 3.1. The type checking of method invocations is discussed here in a top-down fashion. First method resolution is examined, the initial step in the type checking process. Then constraint decomposition is discussed. This is a pre-processing step that allows method resolution to trigger the generic instantiation algorithm. And at last generic instantiation itself is discussed; it infers the type arguments of generic methods, so that they can be processed by method resolution.

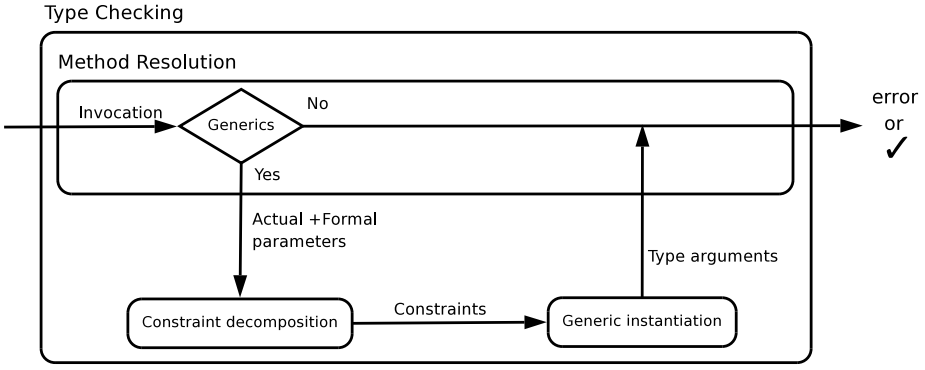

Figure 3.1: An overview of the method invocation type checking

Generics instantiation is referred to in reference [3] as inference. However, in this thesis we prefer to call it generic instantiation because the inference in Java as we will see never fails and may infer nonsensical types. However, type inference in its broadest sense, unlike generics instantiation in Java, might fail in some occasions and never infers absurd types. These two properties of type inference become more apparent when one realizes that type inference problem is just another form of constraints satisfaction problem. Thus, depending on

the constraints, type inference might fail if:

- the constraints are unsatisfiable, or
- the constraints have too many solutions, but no best solution is can be determined.

In this thesis we will also adhere to the same naming convention as in [3]:

- Type expressions are represented by the letters *A, F, U,* and *V*, where *A* is used to denote the type of an actual parameter, and *F* the type of a formal parameter.
- Type variables are represented using the letters *S* and *T*.
- Generic type, i.e. class and interface, declarations are represented by the letters *G* and *H*.

3.1 Method Resolution

All programming languages that support modularity or some sort of overloading need to implement name resolution, that determines what program entity is being referred to by an identifier. This name resolution or name look-up can take place at different levels, such as packages, module, types and methods. Name resolution varies in complexity from one language to another. A function look-up in Haskell is almost as easy as looking up a keyword in a dictionary, if it did not have support for modules and aliases, because function names and type constructors are only allowed to be used once. In Java, however, function look-up or method resolution [3], is much more complicated due to the presence of overloading, overriding, visibility and other factors. Method resolution in Java is responsible for finding the most specific method declaration that matches a method invocation, and it consists of three phases.

1. Determine the name of the method to be invoked and the receiver. The receiver is the class or interface to whom the method definition belongs. A method invocation in Java can take several forms (Appendix C.5), and for these forms the name of the method and the receiver are determined as follows:
    - If the invocation has the form of *MethodName*[1] then:
        - If *MethodName* is a simple *Identifier,* then *Identifier* is the method name. If *Identifier* appears within the scope of a visible method declaration, the receiver is the innermost class or interface that encloses the method declaration.

[1] We do not agree with the term *MethodName* used in the language definition. It gives the impression that we are referring to the name of a method, while in fact *MethodName* can also be a concatenation of a method name and some other identifiers separated with a dot(.), e.g. this.field.meth.

- If *MethodName* is *TypeName.Identifier,* where *TypeName* is the name of a class, then the receiver is the class named *TypeName* and the method name is *Identifier.*
- If *MethodName* is *FieldName.Identifier*, then let T be the declared type of the field or local variable named *FieldName.* If T is a class or interface then the receiver is T, or the upper bound of T if T is type variable.

Example 3.1. The code given below provides an example of each case discussed above. In the first call the receiver is the enclosing class *Foo* and the method is *bar.* In the second call the receiver is the class *Foo*, which is indicated by the identifier Foo before the period (.), and the method is again *bar.* In the third call the receiver would be the type of the parameter *a,* but since the type of *a is* a type variable *T,* the receiver is then the upper bound of T : *Number.* The method in this case is *byteValue.*

```
class Foo{
    static <T extends Number>void bar(T a){
        bar(null);
        Foo.bar(null);
        a.byteValue();
    }
}
```

- If the invocation has the form:
  Primary.NonWildTypeArguments$_{opt}$*Identifier*[2], then let T be the type of the *Primary* expression (see Appendix C.5); the receiver is T if T is a class or interface, or the upper bound of T if T is a type variable.

  Example 3.2. The following code gives a couple examples of this kind of method invocation. The receiver in the first call is the type of the expression *this*, which is always the innermost class that encloses the method invocation, and the method is *foo.* In the second call the receiver is the return type of *Collections.<String>emptySet*() and the method is *size.* The receiver in the last call is the class *Class,* which is the type of *int.class,* and the method is *getCanonicalName.* An example of an invocation where the receiver is the upper bound of type variable is given in the previous example. The method *byteValue* invoked in the last invocation belongs to the upper bound of *T: Number.*

```
this.foo();
Collections.<String>emptySet().size()
int.class.getCanonicalName();
```

[2]It is impossible in Java to parametrize calls with wildcards.

- If the invocation has the form *super.NonWildTypeArguments$_{opt}$Identifier*, then the name of the method is *Identifier* and the receiver is the superclass of the class whose declaration contains the method invocation.

  Example 3.3. The following code is an example of this form of invocation. The receiver in this case is the supertype of *Bar*, which is *Foo*, and the method is *intVal*.

```
class Foo{
   public int intVal(){ return 0; }
}

class Bar extends Foo{
   public void baz(){ super.intVal(); }
}
```

- If the invocation has the form *ClassName.super.NonWildTypeArguments$_{opt}$Identifier*, then the name of the method is *Identifier* and the receiver is the superclass of the class named *ClassName*.

  Example 3.4. The code below is an example of this kind of invocation. The method name in this example is *intVal* and the receiver is the superclass of *Bar Foo,* which was defined in *Example 3.*

```
class Baz extends Foo{
   class Inner{
      public void test(){ Baz.super.intVal(); }
   }
}
```

- If the invocation has the form *TypeName.NonWildTypeArguments Identifier*, then the name of the method is *Identifier* and the receiver is the class named *TypeName*.

  Example 3.5. This kind of invocation has been introduced before in the previous examples. It is used for invoking static methods. An example of this invocation is: $Collections.<String>emptySet()$. The receiver in this case is the class *Collections* and the method is *emptySet*.

2. The second step searches in the receiver obtained in the first step for all accessible and applicable method members. The search is first performed using only the name determined in the first step to identify potential applicable methods. A method is considered potentially applicable if all following conditions hold:

- The name of the method is identical to the name in the invocation.
- The method is accessible from the method invocation site.
- If the method is a variable arity method with arity n, the arity of the method invocation must be greater than or equal to n-1.
- If the method is a fixed arity method with arity n, then the arity of the method invocation must be equal to n.
- If the method invocation includes explicit type parameters, and the method is a generic method, then the number of actual type parameters must be equal to the number of formal type parameters.

After having identified the potential methods, the compiler continues by eliminating any method that can not be correctly invoked given the arguments in the method invocation. This elimination takes place by comparing the actual parameters with the formal parameters of the each potentially applicable method. Due to the support of Java for subtyping, auto-boxing and variable arity methods, determining whether a method is truly applicable is a complicated process. Therefore, it is divided into three phases. These phases are executed in the exact order presented below, and a phase is executed only if the phases that proceed it fail to deliver any applicable methods.

(a) Identify methods with an arity that matches the number of actual parameters in the method invocation and is applicable only by subtyping. Let F_1 to F_n be the types of the formal parameters and A_1 to A_n be the types of the actual parameters, then a method is applicable by subtyping if

for $1 \leq i \leq n$:

- A_i is a subtype of F_i ($A_i <: F_i$), or
- A_i is a raw type that can be converted to a parametrized type C_i, and $C_i <:$ F$_i$

If a method is generic, then all type variables in F_1 to F_n are substituted with the type parameters provided by the method invocation. If these type parameters are absent, then the inferred types are used (see Section 3.3). The generic method is then applicable only if besides the conditions above, all its type parameters are within their bounds. For example, given the following type variable declaration $<T\ extends\ Number>$, T is said to be within its bounds if the type that T is instantiated to is a subtype of *Number*.

(b) Identify methods with an arity that matches the number of actual parameters in the method invocation and is applicable by method invocation conversion, i.e. widening in combination with (un)boxing. Let F_1 to F_n be the types of the formal parameters and A_1 to A_n be the types of the actual parameters, then a method is applicable by

method invocation conversion if for $1 \leq i \leq n$, A_i can be converted by a method conversion to F_i.
If the method is generic then type variables in the formal parameters are first substituted with either the given type parameters in the method invocation, or by the inferred type arguments. The method is then considered to be applicable if all type parameters are within their bounds.

(c) Identify applicable methods with variable arity. Let F_1 to $F_m[]$ be types of the formal parameters and A_1 to A_n be the types of the actual parameters, where $1 \leq m \leq n$. The method is applicable if the following conditions hold:

- For $1 \leq i < m$, A_i can be converted by method conversion to F_i
- If $n \geq m$, then for $m \leq i \leq n$, A_i can be converted by method conversion to the type of F_m

If the method is generic, then type variables in the formal parameters are first substituted with either the given type parameters in the method invocations, or by the inferred type arguments. The generic method is then applicable if the above conditions hold and all its type parameters are within their bounds.

If more than one applicable method was found, then the most specific method is chosen.
For fixed arity methods a method *m1* with the parameters T_1 to T_n is considered more specific than *m2* with the parameters S_1 to S_n, if for $1 \leq i \leq n$ T_i is subtype of S_i.
For variable arity methods a method *m1* with the parameters T_1 to $T_n[]$ is more specific then *m2* with the parameters S_1 to $S_m[]$, where $m \leq n$, if the following conditions are met:

- For all $1 \leq i < m-1$, $T_i <: S_i$
- For all $m \leq i \leq n$, $T_i <: S_n$

If it cannot be determined which method is the most specific, then the compiler deals with the ambiguity as follows:

- If all methods have override-equivalent signatures, then:
  - If all methods are *abstract* except for one, then return this non-abstract method.
  - If all methods are *abstract*, then choose arbitrarily the method with the most specific return type.
- otherwise report an ambiguous method invocation error.

3. The third step checks whether the method chosen in the previous step is appropriate, e.g. an instance method cannot be called from a static

context, an abstract method in class can not be called from subclass (*super.abstractMethod()*), and a method with *void* as return type can be used as top level expression only.

3.2 Constraint Decomposition

The generic instantiation process in Java proceeds by running a constraint decomposition algorithm on the method arguments to collect as much type information as possible that will later help and guide the inference algorithm to compute the best possible types for the type parameters. The purpose of constraint decomposition is to simplify complicated type constraints, by decomposing them into simple atomic constraints. For example, if generic instantiation starts with a single constraint $List\langle Integer\rangle <: List\langle T\rangle$, where T is type variable, then the constraint decomposition algorithm will return the constraint $T = Integer$. There are three kind of atomic constraints in Java; equality constraints of the form of $T = \sigma$, supertype constraints denoted as $\sigma <: T$, and subtype constraint represented by $T <: \sigma$, where σ is a type expression.

The layout sensitive pseudo-code in Listing 3.1, 3.2, and 3.3 describe how the constraints are decomposed. Each procedure takes an actual parameter A, a formal parameter F and a set of constraints TC, which is initially empty, and returns the set TC augmented with new constraints. The procedure in Listing 3.1 checks whether the formal parameter F contains any type variables, and if not it returns the set TC unchanged. The procedure returns also the set TC without any changes if the actual parameter is the expression "null". If the procedure does not stop at the first if-statement, then it checks whether the actual parameter A is a primitive type, such as *int* or *char*. If A is indeed a primitive type, then the procedure replaces A with its corresponding reference type and calls itself. If A is not a primitive type, then the procedure adds the constraint $A <: F$ to the set TC if F is a type variable. If F is an array type instead of a type variable, then the procedure ensures that A is an array type too or it is a type variable with an array type as a bound and then calls itself with the component type of the array types $U[]$ and $V[]$. If F is an instantiation of a generic type G and A itself or a supertype of it is also an instantiation of G, then the procedure performs a case statement on the type arguments of G in which it calls itself or the other procedures in Listing 3.2 and 3.3. If a pair of type arguments (U_i and V_i) of G do not match any of the patterns in Listing 3.1, then the procedure simply ignores these type arguments and moves on to next pair.

The procedure in Listing 3.2 is very similar to the one in Listing 3.1, but it only adds equality constraints to the set TC, because it only calls itself recursively. The procedure in Listing 3.3 looks complicated, but in fact it is very similar to the procedure in Listing 3.1. The only difference between these procedures is that the formal parameter F should be a subtype of the actual parameter A in the procedure COLLECT_SUB_CONSTRAINTS instead in of the other way around.

The generic instantiation of type variables for a method invocation starts by invoking the procedure in Listing 3.1 on the initial constraints $A <: F$, where A is an actual parameter and F is the formal parameter. This will lead to number of type constraints depending on the form of the types A and F (see Table 3.1 for some examples). For a detailed textual description of this process the reader is advised to read section 15.12.2.7 of [3]

Initial constraint	Final constraints
$List<Integer> <: List<T>$	$\{T = Integer\}$
$List<? \ extends \ Integer> <: List<? \ extends \ T>$	$\{T <: Integer\}$
$List<? \ super \ Integer> <: List<? \ super \ T>$	$\{Integer <: T\}$
$List<Integer> <: List<? \ extends \ T>$	$\{T <: Integer\}$
$List<? \ super \ Integer> <: List<? \ super \ T>$	$\{Integer <: T\}$
$List<? \ extends \ List<? \ super \ Integer>> <: List<? \ extends \ List<? \ super \ T>>$	$\{T <: Integer\}$
$List<? \ super \ List<? \ extends \ Integer>> <: List<? \ super \ List<? \ extends \ T>>$	$\{T <: Integer\}$
$List<? \ super \ List<? \ super \ Integer>> <: List<? \ super \ List<? \ super \ T>>$	$\{Integer <: T\}$
$Map<Integer, \ Number> <: Map<T, \ ? \ extends \ T>$	$\{T = Integer, \ T <: Integer\}$

Table 3.1: Example of constraints collection

```
DEF COLLECT_SUPER_CONSTRAINTS(INPUT: A, F, TC, OUTPUT:TC)
    IF F has no type variables OR
       A is the type of the expression 'null' THEN
        EXIT

    IF A is primitive type THEN
        COLLECT_SUPER_CONSTRAINTS(BOX(A), F, TC)

    ELIF F is type variable THEN
        add A <: F  to TC

    ELIF F = U[] THEN
        IF A = V[] OR A is type variable with a bound V[]
            COLLECT_SUPER_CONSTRAINTS(V, U, TC)

    ELIF F = G<U_1 ,... ,U_n> AND G<V_1 ,... ,V_n> is supertype of A THEN
        FOR i = 1 TO n
            IF U_i AND V_i are concrete types THEN
                COLLECT_EQUAL_CONSTRAINTS(V_i, U_i, TC)

            ELIF U_i = '? extends X' THEN
                IF V_i is a concrete type THEN
                    COLLECT_SUPER_CONSTRAINTS(V_i, X, TC)

                ELIF V_i = '? extends Y'
                    COLLECT_SUPER_CONSTRAINTS(Y, X, TC)

            ELIF U_i = '? super X' THEN
                IF V_i is a concrete type THEN
                    COLLECT_SUB_CONSTRAINTS(V_i, X, TC)

                ELIF V_i = '? super Y'
                    COLLECT_SUB_CONSTRAINTS(Y, X, TC)
```

Listing 3.1: Decompose the type constraints $A<:F$ into a set of atomic constraints.

```
DEF COLLECT_EQUAL_CONSTRAINTS(INPUT: A, F, TC, OUTPUT:TC)
    IF F is type variable THEN
        add A = F to TC

    ELIF F = U[] THEN
        IF A = V[] OR A is type variable with a bound V[] THEN
            COLLECT_EQUAL_CONSTRAINTS(V, U, TC)

    ELIF F = G<U_1 ,...,U_n> AND G<V_1 ,...,V_n> is a supertype of A THEN
        FOR i = 0 TO n
            IF U_i and V_i are concrete types THEN
                COLLECT_EQUAL_CONSTRAINTS(V_i, U_i, TC)

            ELIF U_i = '? extends X' AND V_i = '? extends Y' THEN
                COLLECT_EQUAL_CONSTRAINTS(Y, X, TC)

            ELIF U_i = '? super X' AND V_i = '? super Y' THEN
                COLLECT_EQUAL_CONSTRAINTS(Y, X, TC)
```

Listing 3.2: Decompose the type constraint $A{=}F$ into a set of atomic constraints.

```
DEF COLLECT_SUB_CONSTRAINTS(INPUT: A, F, TC, OUTPUT:TC)
    IF F has no type variables OR
       A is the type of the expression 'null' THEN
        EXIT

    IF F is type variable THEN
        add F <: A to TC

    ELIF F = U[] THEN
        IF A = V[] OR A is type variable with a bound V[] THEN
            COLLECT_SUB_CONSTRAINTS(V, U, TC)

    ELIF F = G<U_i ,... ,U_n> THEN
        IF A is not a generic type THEN
            EXIT

        ELIF A = H<V_1 ,... ,V_n> THEN
           IF H<S_i ,... ,S_n> is supertype of F THEN
               {- substitute S_i with U_i -}
               COLLECT_SUB_CONSTRAINTS(A, H<S_1 ,... ,S_n>[S_i/U_i] , TC)

        ELIF A = G<V_1 ,... ,V_n> THEN
            FOR i = 1 TO n
                IF U_i is a concrete type THEN
                    IF V_i is a concrete type THEN
                        COLLECT_EQUAL_CONSTRAINTS(V_i , U_i , TC)

                    ELIF V_i = '? extends Y' THEN
                        COLLECT_SUB_CONSTRAINTS(Y, U_i , TC)

                    ELIF V_i = '? super Y' THEN
                        COLLECT_SUPER_CONSTRAINTS(Y, U_i , TC)

                ELIF U_i = '? extends X' AND V_i = '? extends Y' THEN
                    COLLECT_SUB_CONSTRAINTS(V_i , X_i , TC)

                ELIF U_i = '? super X' AND V_i = '? super Y' THEN
                    COLLECT_SUPER_CONSTRAINTS(Y_i , X_i , TC)
```

Listing 3.3: Decompose the type constraint $F<:A$ into a set of atomic constraints.

3.3 Generic Instantiation

The generic instantiation process is started by the type-checker each time a generic method is marked as a potential method by the method resolution. Its purpose is to compute the best possible types for the type variables declared in the generic method. The best type in this case are not the principal types as the readers who are familiar with a type inference based on the Hindley-Milner(HM) algorithm [6] might expect, but the most specific types that can make the method invocation correct. The generic instantiation used in Java was introduced in the Generic Java project (GJ) [1, 2] as partial local inference [9, 7], because types are inferred based on the arguments in the method invocation and not on the context. There are some cases, however, where context does play a role in instantiating the type variables. For example, context is used when a generic method has no formal parameters, or when the type constraints do not provide enough information about a type variable.
Method invocations are also processed separately, thus if for one method invocation a type variable T is inferred to be *Integer*, it could be that the same type variable T in another method invocation is inferred to be *String*.

```
<T> T idFunc(T a){ return a;}
...
Integer i = idFunc(idFunc(42));
```

Listing 3.4: Type parameters are inferred on method invocation basis

Consider the code in Listing 3.4. The type parameter T is instantiated two times instead of once. Due to strict nature of Java, the inner method invocation must be first evaluated before it can be passed as an argument to the outer method invocation. This means that the outer invocation does not get an argument of type *T,* but an argument of type *Integer*. Thus, there is no constraint propagation here as one might have expected. Generic instantiation is applied for each invocation, and hence T is instantiated twice.

The pseudocode in Listing 3.5 depicts how the generic instantiation algorithm works. From this pseudo-code, it is clear that the algorithm handles the equality constraints by simply assigning the first encountered type to the type variable in equality constraint without performing any unifications. Reader who have experience with the HM inference system might be surprised by this somewhat unusual approach, because HM system uses unification to make sure that all equality constraints can be satisfied simultaneously. Thus, generic instantiation will not fail even if the equality constraints are contradictory. But rest assured, the fact that generic instantiation always succeeds does not mean that method invocation type check succeeds always. An example of an incorrect method invocation is given in Listing 3.6. Although, the type variable T is instantiated with *String*, the call is incorrect because *HashMap<String, Integer>* is not subtype of *HashMap<T, T>*, where T is substituted with *String*.

```
<T> void foo(HashMap<T, T> a){}
..
```

```
foo(new HashMap<String, Integer>());
```

Listing 3.6: Illegal method invocation

In some occasions type parameters can be underconstrained, e.g. have no constraints at all, which means these type parameters can take any type possible. To deal with this ambiguity Java assigns the type *Object* to any type parameter that was not instantiated/inferred.

Unlike the equality constraints, the generic instantiation algorithm does perform unifications for the supertype constraints. This is denoted in Listing 3.5 by the function *lub*, which stands for *least upper bound*. The function, as its name implies, calculates the least supertype of a set of types, for example $lub(Integer, Double) = Number$. However, computing the least upper bound is not as easy it may seem. For example, $lub(Integer, String)$ does not return the type *Object*, but the much more specific and complicated intersection type denoted by:

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

The interface *Serializable* and the generic interface *Comparable* are part of the intersection type, because both types *Integer* and *String* implement these interfaces. It becomes even more complex when one tries to compute the least upper bound of parametrized types. The reason why *lub* computes these intersection types is because they are more informative, i.e. they allow more operations to be performed on an object than when the type is only known to be *Object*.
To compute the intersection type of parametrized types *lub* uses two auxiliary functions *lci*[3] and *lcta*[4]. The complete definition of *lub* function is as follows:

$$\begin{aligned} lub(U_1 \dots U_k) &= Candidate(W_1) \,\&\, \dots \,\&\, Candidate(W_n), \\ &\quad \text{where } W_i \text{ is an element from } MEC \end{aligned}$$

$$Candidate(W) = \begin{cases} CandidateInvocation(W) & \text{if } W \text{ is generic} \\ W & \text{otherwise} \end{cases}$$

$$CandidateInvocation(G) = lci(Inv(G))$$

$$\begin{aligned} lci(e_1 \dots e_n) &= lci(lci(e_1, e_2), e_3 \dots, e_n) \\ lci(G{<}X_1 \dots X_n{>}, G{<}Y_1 \dots Y_n{>}) &= G{<}lcta(X_1, Y_1), \dots, lcta(X_n, Y_n){>} \end{aligned}$$

[3] short for least containing invocation
[4] short for least containing type argument

$$
\begin{aligned}
lcta(U, V) &= \begin{cases} U & \text{if } U = V \\ ?\ extends\ lub(U, V) & \text{otherwise} \end{cases} \\
lcta(U, ?\ extends\ V) &= ?\ extends\ lub(U, V) \\
lcta(U, ?\ super\ V) &= ?\ super\ glb(U, V) \\
lcta(?\ extends\ U, ?\ extends\ V) &= ?\ extends\ lub(U, V) \\
lcta(?\ extends\ U, ?\ super\ V) &= \begin{cases} U & \text{if } U = V \\ ? & \text{otherwise} \end{cases} \\
lcta(?\ super\ U, ?\ super\ V) &= ?\ super\ glb(U, V) \\
glb(V_1, \ldots, V_n) &= V_1 \,\&\, \ldots \,\&\, V_n \text{ is an intersection of types}
\end{aligned}
$$

The function *glb* is the dual function of *lub*. It takes a set of types and returns the *greatest lower bound* of those types. This is done by intersecting all types. Note that this, however, does not mean that the result of *glb* is always an intersection type. For example:

$$
\begin{aligned}
glb(Number, Serializable, Cloneable) &= Number \,\&\, Cloneable \\
&\text{and} \\
glb(Component, TextComponent, TextField) &= TextField
\end{aligned}
$$

The interface *Serializable* is not included the result, because the class *Number* already implements it. The result of the second *glb* is simply *TextField* because this class is a subclass of both *Component* and *TextComponent*.
Unlike *lub*, *glb* does not always succeed. Consider the case of *glb(Integer, String)*; there is no type in the Java language that can be both a subtype of *Integer* and *String* except for the special *null type*[5]. However, the java compiler prefers not to return this type, but will return a compile-time error instead.

$$
Inv(G) = \{V \mid 1 \le i \le n, V \in ST(U_i) : V = G{<}\ldots{>}, G \in MEC\}
$$
where U_i is one of the parameters of *lub*

$$
\begin{aligned}
MEC &= \{V \mid V \in EC, \forall W \in EC : V \neq W,\ W \not<: V\} \\
EC &= \cap\{EST(U) \mid U \text{ in } U_1 \ldots U_k\} \quad \text{where } U_1 \ldots U_k \text{ are the parameters of } lub \\
EST(U) &= \{V \mid W \in ST(U) : V = |W|\}, \text{ where } |W| \text{ is the erasure type of } W \\
ST(U) &= \{W \mid U <: W\}
\end{aligned}
$$

ST is a function that returns all the possible supertypes of a type. The result of this function is subsequently used by the function *EST* to compute a set of raw supertypes, i.e. erased types that are supertypes of the argument *U*. *EC* is an intersection of sets of raw supertypes of the types $U_1, \ldots, U_n$ that are supposed to be subtypes of a type variable *T*, i.e. $U_1 <: T, \ldots, U_n <: T$. *EC* is used to

[5] type of the expression `null`

compute the minimal set of most specific types, denoted by MEC. Given $EC = \{Integer, Number, String, Object\}$ the minimal set of most specific types is $MEC = \{Integer, String\}$. The function Inv is used by lci to compute all the parametrized supertypes, whose erased type is a member of MEC.

```
DEF INFER(INPUT: TC, OUTPUT: SUBST)
    INFER_REC(TC, FALSE, SUBST)

DEF INFER_REC(INPUT: TC, STOP, OUTPUT: SUBST)
    TC_equal ← all T_i = U in TC
    TC_super ← all U <: T_i in TC
    TC_sub ← all T_i <: U in TC
    {- process equality constraints -}
    FOR i = 1 TO n
        FOR T_i = U IN TC_equal
            IF U is a type varibale T_j THEN
                IF T_i = T_j
                    remove  T_i = U from TC_equal
                ELSE
                    rewrite T_i = U to T_j = U in TC_equal, TC_super and TC_sub
             ELSE
                add T_i = U to SUBST
                replace T_i with U in TC_equal, TC_super and TC_sub
    {- process supertype constraints -}
    FOR i = 1 TO n
        {- U_1 <: T_i, ..., U_k <: T_i -}
        subtypes ← {}
        FOREACH U <: T_i IN TC_super
            add U to subtypes
        add T_i = lub(subtypes) to SUBST
    {- process subtype constraints if not all type variables
       have been inferred
    -}
    remaining_vars ← TVARS(TC)\TVARS(SUBST)
    IF  remaining_vars ≠ ∅ THEN
        IF STOP THEN
            FOR i = 1 TO n
                {- T_i <: U_1, ..., T_i <: U_k -}
                supertypes ← ∅
                FOR T_i <: U IN TC_sub
                    add U to supertypes
                add T_i = glb(supertypes) to SUBST
        ELSE
            IF method occurs in an assigenment THEN
                R  ← return type of the method
                R' ← apply SUBST to R
            ELSE
                R' ← Object
            S  ← type of the assigenment
            newTC ← COLLECT_SUB_CONSTRAINTS(S, R', ∅)
            FOREACH T IN remaining_vars
                FOREACH B IN bounds of T
                    B' ← apply SUBST B
                    newTC ← COLLECT_SUB_CONSTRAINTS(B', T , newTC)
            SUBST ← INFER_REC(TC_sub, TRUE, newTC)
            FOREACH T IN TVARS(TC_sub)\TVARS(SUBST)
                add T = Object to SUBST
```


Listing 3.5: Inference algorithm of Java language

Chapter 4

Improved Type Checking

In this chapter we introduce a new type checking algorithm for generic method invocations, which is tailored towards generating error messages that explain why a type error occurs and contain additional hints, which describe how to correct a type error or prevent new errors in future compilations. We discuss the algorithm in similar manner as in Chapter 3. We will first introduce a new method resolution and explain why it is needed. After that we discuss constraint generation, which is a modified version of the constraint decomposition process discussed earlier. In Section 4.3 we discuss our new algorithm for solving type constraints and will conclude the chapter with a few examples to demonstrate how the algorithm works. Figure 4.2 illustrates how our type checking algorithm differs from the original type checking algorithm in Java (see Figure 3.1).

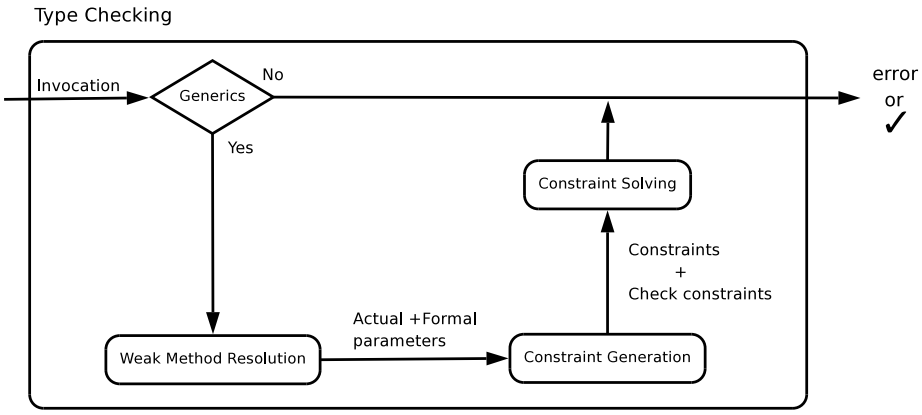

Figure 4.1: An overview of generic method invocation type checking as presented in this chapter

4.1 Weak Method Resolution

The method resolution process presented in Chapter 3 relies on generic instantiation to instantiate all the type parameters of generic methods, so that their instantiated signature can be used to find the most specific method declaration for a given method invocation. Since generic instantiation never fails, it is always possible to obtain an instantiated signature of generic method. The absence of failure in generic instantiation, as it was mentioned earlier, is due to the defaulting mechanism that makes sure that all type variable are instantiated, and the way equality constraints are handled. The method invocation in Listing 4.1

```
import java.util.*;
...
<T> void foo(Map<T, T> a){..}
...
Map<String, Number> map = ...
foo(map);
```

Listing 4.1: Inference failure

will generate the following constraint set: $\{T = String, T = Number\}$. The types *String* and *Number* are clearly not the same, therefore it is impossible to assign a type that satisfies both constraints to the type variable T. A Haskell compiler will halt and report a type conflict to the user. But generic instantiation will, according to Listing 3.5, instantiate the type variable T with the first encountered type *String* and will not report any type conflict. In fact, a type error is only reported when the type checker tries to verify whether the formal parameters of a method match the actual parameters in the method invocation.

The choice of ignoring constraints violations by the generic instantiation algorithm is correct, because any apparent type conflict during the generic instantiation process is guaranteed to be caught later by the type checker. However, there are cases where the method resolution fails to return any method at all. In this situation the only error message reported by the compiler, is that the user is calling a method that does not exist. With the introduction of generics to the Java language, this kind of error message is not very helpful (see the error messages reported by the Eclipse compiler and the Sun compiler for the method invocation in Listing 4.2 in Table 4.1).

```
<T extends Number> void foo(T a){}
<T extends Error> void foo(T a){}
...
foo("Hello");
```

Listing 4.2: No method declaration matches the call

1. ERROR in Test.java (at line 7) foo(""); ^^^ Bound mismatch: The generic method foo(T) of type Test is not applicable for the arguments (String). The inferred type String is not a valid substitute for the bounded parameter <T extends Error>
Test.java:7: cannot find symbol symbol : method foo(java.lang.String) location: class Test foo(""); ^

Table 4.1: Error messages reported by the Eclipse compiler and the Sun compiler respectively

Consider the method declaration in Listing 4.2, where none of the method declarations matches the method invocation, because of a bound mismatch: the type of the argument "`Hello`" is *String*, which is subtype of neither *Number* nor *Error*. This kind of error could be caused by a beginner programmer who does not know much about bounded type variables, or could be a result of removing or modifying a generic method that used to allow arguments of type *String*.

```
<T extends Number, S> void foo(List<S> a, T b){}
<T extends Number, S> void foo(List<T> a, S b){}
...
List<String> list = ...
foo(list, "42");
```

Listing 4.3: Ambiguous method invocation

Another interesting example is given in Listing 4.3. The error in this case arises because both type parameters T and S are instantiated to *String*. Changing the second argument in the method call to a subtype of *Number* will correct the error, but so will changing the first argument to any subtype of $List<?\ extends\ Number>$. Thus, it will be more constructive to make the programmer aware of this fact by producing two different error messages or one error message that involves both method declarations, rather than just one error message for only one of the methods.

Thus for the sake of better error reporting, the method resolution has to be weakened to allow all methods whose name and arity matches the name and number of arguments of the invocation respectively. This would allow us to treat the methods in Listing 4.2 and 4.3 as candidates. Weakening the method resolution does not mean that inaccessible methods such as overridden methods, instance methods in a static context, or simply methods declared with a restricted visibility, are considered as candidates for error reporting. However,

considering inaccessible methods, is a possibility that one could explore, but we prefer not to in this thesis. To weaken the method resolution, the second step that searches a class or an interface for applicable member methods, must ignore the type parameters and determine the most specific method based on raw types only. This is done by erasing the generic parts of types and ignoring bound information. Thus, the signature of the method declarations in Listing 4.2 will be converted to $void\,foo(Object)$ and the signature of the method declarations in Listing 4.3 will be converted to $void\,foo(List, Object)$. Listings 4.5, 4.6, and 4.7 depict how the method resolution is modified compared to the original method resolution introduced in Chapter 3.

The procedure METHOD_RESOLUTION takes a method invocation as an argument and returns a set of most specific method declarations. The procedure starts first by searching for potentially applicable methods, which are methods with the same name as in the invocation and an arity equal to the number of arguments in the invocation if the method has fixed arity, or an arity less than the number of arguments in the invocation if the method has a variable arity. This search is performed by the procedure POTENTIAL_APPLICABLE in Listing 4.6. After constructing a set of potentially applicable methods, METHOD_RESOLUTION tries to narrow this set to a (smaller) set of most specific methods. Computing a set of most specific methods is performed by using the procedures defined in Listing 4.7, which work exactly as described in Section 3.1, except that they ignore any kind of generic information. After having constructed a set of most specific methods, METHOD_RESOLUTION will remove any method declaration that is not appropriate (e.g. not accessible from the call site) from the set and return the remaining methods.

```
<T, S extends Number> void foo(Collection<T> a, List<S> b){}
<T, S extends Number> void foo(List<T> a, ArrayList<S> b){}
...
foo(new LinkedList<Integer>(), new ArrayList<Integer>()); // OK
foo(new LinkedList<Double>(), new ArrayList<Integer>()); // WRONG
```

Listing 4.4: Both declarations match the call, but the second one is more specific

Type checking method invocations with this weak method resolution is slightly more efficient than the original version, because it reduces the number of methods for which the generic instantiation/inference process must be initiated. This efficiency is achieved only when the method parameters are parametrized types, such as in Listing 4.4. During type checking the generic instantiation process must be initiated for both method declarations in Listing 4.4, because they are both potentially applicable. For both method invocations the second method declaration will be eventually chosen as the most specific method, because its parameters are subtypes of the parameters of the first method declaration. The weakened method resolution presented here, will choose the same method to be the most specific one without using generic instantiation to in-

```
DEF METHOD_RESOLUTION(INPUT: inv)
    potential_methods ← ∅
    FOREACH meth IN member methods of the receiver
        IF POTENTIAL_APPLICABLE(meth, inv) THEN
            add meth to potential_methods

    specific_methods ← ∅
    FOREACH meth IN potential_methods
        IF meth is fixed arity AND APPLICABLE_BY_SUBTYPING(meth, inv) THEN
            add meth to specific_methods
    most_specific_methods = MOST_SPECIFIC_METHOD_FIX(specific_methods)

    IF #most_specific_methods == 0 THEN
        specific_methods ← ∅
        FOREACH meth IN potential_methods
            IF meth is fixed arity AND APPLICABLE_BY_METHOD_CONV(meth, inv) THEN
                add meth to specific_methods
        most_specific_methods = MOST_SPECIFIC_METHOD_FIX(specific_methods)

    IF #most_specific_methods == 0 THEN
        specific_methods ← ∅
        FOREACH meth IN potential_methods
            IF meth is variable arity AND APPLICABLE_BY_METHOD_CONV(meth) THEN
                add meth to specific_methods
        most_specific_methods = MOST_SPECIFIC_METHOD_VAR(specific_methods)

    FOREACH meth IN most_specific_methods
        IF meth is not appropriate THEN
            remove meth from most_specific_methods;
    RETURN most_specific_methods
```

Listing 4.5: Weakened method resolution

```
DEF POTENTIAL_APPLICABLE(INPUT: meth, inv)
    IF meth.name ≠ inv.name THEN
        RETURN FALSE
    IF meth is not accessible from inv's location THEN
        RETURN FALSE
    IF meth.arity ≠ #inv.arguments AND meth is fixed arity THEN
        RETURN FALSE
    IF meth.arity < (#inv.arguments − 1) AND meth is variable arity THEN
        RETURN FALSE
    RETURN TRUE

DEF APPLICABLE_BY_SUBTYPING(INPUT: meth, inv)
    FOR i = 1 TO n
        fparam = WIDE_ERASURE(meth.arguments[i])
        aparam = inv.arguments[i]
        IF aparam is not subtype of fparam THEN
            RETURN FALSE
    RETURN TRUE

DEF APPLICABLE_BY_METHOD_CONV(INPUT: meth, inv)
    FOR i = 1 TO n
        fparam = WIDE_ERASURE(meth.arguments[i])
        aparam = inv.arguments[i]
            IF aparam cannot be method converted to fparam THEN
                RETURN FALSE
    RETURN TRUE

DEF WIDE_ERASURE(INPUT: tp)
    IF tp is a type variable THEN
        RETURN Object
    ELIF tp is generic THEN
        {− if tp = G<...> then return G −}
        RETURN raw form of tp
    ELSE
        RETURN tp
```

Listing 4.6: Implementation of method applicability test

```
DEF MOST_SPECIFIC_METHOD_FIX(INPUT: meth_set, OUTPUT: meth_set)
    FOREACH m1 IN meth_set
        FOREACH m2 IN meth_set
            IF MORE_SPECIFIC_FIX(m1, m2) THEN
                remove m2 from meth_set

DEF MORE_SPECIFIC_FIX(INPUT: m1, m2)
    FOR i = 1 TO n
        arg1 = WIDE_ERASURE(m1.arguments[i])
        arg2 = WIDE_ERASURE(m2.arguments[i])
        IF arg1 not subtype of arg2 THEN
            RETURN FALSE
    RETURN TRUE

DEF MOST_SPECIFIC_METHOD_VAR(INPUT: meth_set, OUTPUT: meth_set)
    FOREACH m1 IN meth_set
        FOREACH m2 IN meth_set
            IF MORE_SPECIFIC_VAR(m1, m2) THEN
                remove m2 from meth_set

DEF MORE_SPECIFIC_VAR(INPUT: m1, m2)
    IF #m1.arguments > #m2.arguments THEN
        RETURN MORE_SPECIFIC_VAR(m2, m1)

    k = min(#m1.arguments, #m2.arguments) - 1
    n = max(#m1.arguments, #m2.arguments)

    FOR i = 1 TO k
        arg1 = WIDE_ERASURE(m1.arguments[i])
        arg2 = WIDE_ERASURE(m2.arguments[i])
        IF arg1 not subtype of arg2 THEN
            RETURN FALSE

    FOR i = k TO n
        arg1 = WIDE_ERASURE(m1.arguments[i])
        arg2 = WIDE_ERASURE(m2.arguments[k+1])
        IF arg1 not subtype of arg2 THEN
            RETURN FALSE

    RETURN TRUE
```

Listing 4.7: Determine the most specific method(s)

stantiate any type variables. It should be noted, however, that this weakened version of method resolution is not intended to replace the original one, but it is merely used to discover a larger set of methods the programmer might be trying to call. Having a large set of methods provides more clues and insight of what the user is attempting to achieve, which can be used to improve the error messages.

4.2 Constraints Generation

Recall that the generic instantiation algorithm presented in Chapter 3 does not check for any constraint violations. If a constraint violation does occur, then the algorithm does not report it, because it is unaware of any type conflict. In fact, type errors during the generic instantiation process are just delayed until all type parameters are instantiated and the type checker starts doing its job. The Java language is a statically type-checked language, i.e. the compiler verifies that the types of program expressions satisfy their type declarations. Therefore, we expect that a type error is generated if something does not type check. All type conflicts, such as $T = Integer \wedge T = String$, that might go undetected during the generic instantiation process will always lead to type-checking errors. However, even when there are no inconsistencies in the type constraints, it is still not guaranteed that the type-checking will succeed. In Listing 4.8, the

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

Listing 4.8: Inference succeeds, but type checking fails.

type parameter T is instantiated to *Number*, because the only type constraint available for T is $\{T = Number\}$. But unfortunately, $List<?\ extends\ Number>$ is not a subtype of $List<?\ super\ T>$, where T is substituted by *Number*. In this situation the type checker reports that the method *foo* cannot be applied to the arguments of type $List <?\ extends\ Number >$and $List < Number >$, 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. Keeping the type constraints around after generic instantiation has been performed, will help the type checker produce better error messages. Another solution, which is pursued here, is to provide the constraints solver, that will be introduced in Section 4.3, 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. Therefore the constraints decomposition algorithm in Listings 3.1 and 3.3 is extended to generate additional constraints, which cannot be decomposed into atomic constraints. The new constraints generation algorithm will, for example, generate the following constraints:

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

for the method invocation in Listing 4.8. The constraints above are not joined together, but are kept separate because they serve different purposes. The first set of constraints — left operand of $\Cup$— is used to infer a type for T, while the second set is used to ensure that an inferred type will not cause any

type-checking errors. The second set of constraints also allows the constraint solver to take additional measures in case an inferred type does not satisfy one of the constraints in this set, such as providing the programmer with a possible correction. Moreover the role that the extra constraints play in the inference process, is explained in the next sections. Listing 4.9 and 4.10 illustrate the constraint generation algorithm. The procedures in these listings are very similar to those defined in Section 3.2. The difference between these procedures and those in Section 3.2 is the generation of an additional set of non-atomic constraints, which is indicated in the pseudo-code by the comment "{- add the actual and formal parameter to ETC -}". The part of the algorithm that normalizes equality constraints is equivalent to the one in Listing 3.2.

Java types and type declarations in JastAdd Extensible Java Compiler[11] (JastAdd ECJ) are not directly connected to identifiers and expressions in the AST. The type of an expression can easily be looked up in the type environment, but it is not possible to query an expression for its type, which is necessary in some situations, e.g. printing an expression and its location in the source code, finding the generic type that a certain type in the type constraints originated from, or counting the number of times a certain type is used in a method call. The constraint generation algorithm is also extended to gather additional information about types that is needed, but cannot be directly provided by JastAdd ECJ. This additional information can be used to improve error messages or can be used by the heuristics which will be introduced later on in Chapter 6.

```
DEF COLLECT_SUPER_CONSTRAINTS(INPUT: A, F, AP, FP, KIND, TC, ETC,
                              OUTPUT:TC, ETC)
    IF F has no type variables OR
       A is the type of the expression 'null' THEN
        EXIT
    IF A is primitive type THEN
        COLLECT_SUPER_CONSTRAINTS(BOX(A), F, TC)
    ELIF F is type variable THEN
        add A <: F  to TC
    ELIF F = U[] THEN
        IF A = V[] OR A is type variable with a bound V[]
            COLLECT_SUPER_CONSTRAINTS(V, U, TC)
    ELIF F = G<U1 ,... ,Un> THEN
        IF G<V1 ,... ,Vn> is supertype of A THEN
            FOR i = 1 TO n
                IF Ui AND Vi are concrete types THEN
                    COLLECT_EQUAL_CONSTRAINTS(Vi, Ui, TC)
                ELIF Ui = '? extends X'
                    IF Vi is a concrete type THEN
                        COLLECT_SUPER_CONSTRAINTS(Vi, X, TC)
                    ELIF Vi = '? extends Y'
                        COLLECT_SUPER_CONSTRAINTS(Y, X, TC)
                    ELSE
                        {- add the actual and formal parameter to ETC -}
                        IF KIND is super type check THEN
                            add AP <: FP to ETC
                        IF KIND is sub type check THEN
                            add FP <: AP to ETC
                ELIF Ui = '? super X'
                    IF Vi is a concrete type THEN
                        COLLECT_SUB_CONSTRAINTS(Vi, X, TC)
                    ELIF Vi = '? super Y'
                        COLLECT_SUB_CONSTRAINTS(Y, X, TC)
                    ELSE
                        {- add the actual and formal parameter to ETC -}
                        IF KIND is super type check THEN
                            add AP <: FP to ETC
                        IF KIND is sub type check THEN
                            add FP <: AP to ETC
```

Listing 4.9: Extending generation of super type constraints to include initial constraints

```
DEF COLLECT_SUB_CONSTRAINTS(INPUT: A, F, AP, FP, KIND, TC, ETC,
                            OUTPUT:TC, ETC)
    IF F has no type variables OR
       A is the type of the expression 'null' THEN
        EXIT
    IF F is type variable THEN
        add F <: A to TC
    ELSE IF F = U[] THEN
        IF A = V[] OR A is type variable with a bound V[] THEN
            COLLECT_SUB_CONSTRAINTS(V, U, TC)
    ELIF F = G<U_i ,... ,U_n> THEN
        IF A is not a generic type THEN
            EXIT
        ELIF A = H<V_1 ,.. ,V_n> THEN
           IF H<S_i ,... ,S_n> is supertype of F THEN
               {- substitute S_i with U_i -}
               COLLECT_SUB_CONSTRAINTS(A, H<S_1 ,... ,S_n>[S_i/U_i], TC)
        ELIF A = G<V_1 ,... ,V_n> THEN
            FOR i = 1 TO n
                IF U_i is a concrete type THEN
                    IF V_i is a concrete type THEN
                        COLLECT_EQUAL_CONSTRAINTS(V_i, U_i, TC)
                    ELIF V_i = '? extends Y' THEN
                        COLLECT_SUB_CONSTRAINTS(Y, U_i, TC)
                    ELIF V_i = '? super Y' THEN
                        COLLECT_SUPER_CONSTRAINTS(Y, U_i, TC)
                ELIF U_i = '? extends X' THEN
                    IF V_i = '? extends Y' THEN
                        COLLECT_SUB_CONSTRAINTS(V_i, X_i, TC)
                    ELSE
                        {- add the actual and formal parameter to ETC -}
                        IF KIND is super type check THEN
                            add AP <: FP to ETC
                        IF KIND is sub type check THEN
                            add FP <: AP to ETC
                ELIF U_i = '? super X' THEN
                    IF V_i = '? super Y' THEN
                        COLLECT_SUPER_CONSTRAINTS(Y_i, X_i, TC)
                    ELSE
                        {- add the actual and formal parameter to ETC -}
                        IF KIND is super type check THEN
                            add AP <: FP to ETC
                        IF KIND is sub type check THEN
                            add FP <: AP to ETC
```

Listing 4.10: Extending generation of subtype constraints to include initial constraints

4.3 Constraint Solving

4.3.1 Checking Type Parameter Bounds

Type parameters in a generic method can have bounds just like type parameters in a class declaration. These bounds can be seen as limitations or restrictions that narrow the range of types that certain type parameters can take. But these bounds can also be viewed as type constraints that provide valuable information about the kind of type that will be inferred for a type variable. The bound constraints, however, do not play an active role in inferring type arguments, except if a type variable does not have any equality or supertype constraints. Since we want to improve the error messages, we would like to exploit any kind of information available. Consider the code in Listing 4.11. The method

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

Listing 4.11: The parameter m is not a subtype of the formal parameter b

invocation will not type check, because the second actual parameter is not a subtype of the second formal parameter and the inferred type for the type parameter T is not a subtype of its bound *Number*. To analyze why this error occurs, one needs to view the generated constraints and deduce which type was inferred for the type variable T. The generated constraints for the method invocations are:

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

According to the algorithm given in Listing 3.5, the type that is inferred for T is the result of *lub(String, Integer)*, which is the type *Object*[1]. By substituting the type variable T in all formal parameters with the type *Object*, one can conclude that the type-check error arises because the type '*? super Object*' does not contain *Number,* and *Object* is not subtype of *Number*. This error could have also been noticed during the inference process if the third constraint ($T <: Number$) was checked. Thus, it is possible to determine why the method invocation does not type check during both the inference and type-checking. However, type-checking unlike inference, cannot determine what caused the error. Judging only by the collected constraints above, one could blame:

- the type *Number,* because it is not a supertype of $lub(String, Integer)$, or
- the type *String*, because it is not a subtype of *Number*.

[1] In fact the inferred type is a more complicated type, but for simplicity we approximate it with Object.

Which type could be blamed depends on the order in which the type constraints are solved. Since generic instantiation, see Chapter 3, ignores the subtype constraints ($T <: U$) in the presence of supertype constraints, one could assume that *Number* should be blamed for the type conflict. However, this assumption would be totally wrong. Examining the bounds of T, reveals that actually *String* is the source of the conflict. Since *String* is not a subtype of *Number,* taking the least upper bound of any subtype of *Number* and *String* will always lead to a type that is not a subtype of *Number*.

To benefit from the information encoded in the bounds of type parameters, we propose to check all the types in the equality and supertype constraints against the bounds of the corresponding type variables before solving the generated constraints. To minimize the number of produced error messages, we restrict the checking of types against the bounds of type variables to the types that contribute to inferring type parameters only. For example if the generated constraints for a type variable T are

$$\{T = String,\ Integer <: T,\ T <: Number\},$$

then we only have to verified whether *String* is a subtype of the bounds of *T,* because only the equality constraint determines the inferred type for T in this case. Thus, if *Double* is a bound of T, then an error message is issued informing the programmer that *String* does not satisfy the bounds of T.

```
<T extends Comparable<T>> void foo(T a,...){}
...
<T,S extends Comparable<T>> void bar(S a, T b,...){}
```

Listing 4.12: Bounds involving type variables

While involving the bound constraints in the inference process can improve the generated error messages, it is not always possible because type parameter bounds might contain type variables. Consider the method declarations in Listing 4.12. In the first declaration, the bound of T expresses only that the inferred type for T must implement the interface *Comparable* instantiated with the inferred type. Since we are interested in checking the bounds before we begin inferencing the type parameters, the bound of T does not carry any information that we can exploit. In the second declaration, the bound of S does not carry any information just like in the first declaration. However, this can change if we control the order in which type parameters are inferred. If we manage to infer T before S, then we can substitute T in the bound of S. This would allow us then to check the types in the constraints on S against the instantiated bound.

4.3.2 Ordering Type Variables

In the Haskell and ML community it is known that the famous algorithm $\mathcal{W}$ and folklore algorithm $\mathcal{M}$ do not always report the same program location as where the type conflict occurred[14, 5]. The reason behind this difference is the order in which they infer the sub-expressions, which influences the order

of unification. The order of unification subsequently speeds up or delays the detection of type conflicts. The same thing could be said about our inference algorithm and generic instantiation discussed in the previous chapter. Thus, the order in which constraints are processed has an impact on the reported error messages. Since type parameters are instantiated in Java separately, the order in which they are instantiated has no influence on produced error messages.

```
<T, S extends Number> void foo(T a, S b){
    foo(1, a); // OK or NOT
}
```

Listing 4.13: Independent type variables

To explain what we mean by inferring each type parameter independently, consider the code in Listing 4.13. One may think that the method invocation will type check just fine, because:

- the type parameter T is inferred/instantiated to be *Integer*,
- S is inferred to be T and since T is *Integer*, S also becomes *Integer*
- *Integer* is a subtype of Number; the upper bound of S.

This is unfortunately not correct. The type parameter T is indeed inferred to be *Integer*, but the type inferred for S is T, not *Integer*. Since T is an unbounded type parameter, it can virtually be any type. Therefore, T is chosen to be *Object*, and hence we cannot conclude that T is a subtype of *Number*. Therefore, the method invocation does not type check.

In our inference algorithm type parameters are still inferred separately, but because we involve bound constraints in the inference process, the order of inferring the type parameters does have a substantial impact on the generated error messages.

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

Listing 4.14: Order of type parameters matters

Consider the code in Listing 4.14, which gives rise to the following constraints $\{S = Integer, S = String, Integer <: T\}$. Our constraint solver, unlike generic instantiation, will not infer S to be *Integer* or *String*. The inferred type of S is the special type $\perp$, which denotes that the type constraints can not all be satisfied simultaneously. In this case, we will issue an error message explaining to the user that S is invariant, but that the types *Integer* and *String* are not equivalent. This a good error message compared to what other compilers generate, because it explains the source of the type error. Actually, we can even improve this error message by proposing a possible fix for the type conflict. To

do that, however, we need to change the order in which the type parameters are inferred. If we infer T first and than S, then we can benefit from bound constraints of S. Inferring T first, means that S must be a subtype of *Integer*: the inferred type for T. Looking at type constraints on S we establish that *String* is the type that makes the constraints inconsistent. Therefore, we add a hint in our previous error message explaining to the programmer that replacing *String* with *Integer* may fix the type conflict.

The order in which type variables need to be inferred to provide good error messages is determined by what we call bound dependency. We say that a type variable T_1 depends on another type variable T_2, if T_2 occurs in one of the bounds of T_1. For example, the type parameter S in Listing 4.14 depends on T. Type variables that have a large number of type variables depending on them should be inferred first. For example, given the following type parameter declaration:

$$< T,\ S,\ R\ extends\ Map<S,\ T>,\ U\ extends\ Map<R,\ S>>$$

The type parameters need to be inferred in the following order: S, T, R and U. This ordering is computed as follows:

```
DEF UpdatePrior(tvar)
    tvar.prior++
    FOREACH bound IN tvar.bounds
        FOREACH var IN bound
            IF var ≠ tvar THEN
                UpdatePrior(var)
```

Listing 4.15: Computing the priority of type parameters

The procedure UpdatePrior takes a type parameter as argument and increases its priority and the priority of each type parameter in its bounds. Applying this procedure to the parameters above results in the following ordering priorities:$\{T := 3,\ S := 4\ , R := 2,\ U := 1\}$.

4.3.3 Constraint Solver

We present here a constraint solving algorithm that returns a substitution that satisfies all the constraints of each type parameter. The algorithm, much like generic instantiation in Section 3.3, processes one type parameter at a time and solves the constraints in a fixed order. The algorithm solves the equality constraints for a type parameter first by ensuring that all the types in equality constraints are the same type. Given the following equality constraint set $\{T = \tau_1, \ldots, T = \tau_n\}$ of a type parameter T, the algorithm verifies that $\exists j, 1 \leq j \leq n, \forall i, 1 \leq i \leq n : \tau_i = \tau_j$ and then infers T to be τ_j. If the type parameter T has the following supertype and subtype constraints:

$$\{\alpha_1 <: T, \ldots, \alpha_m <: T, T <: \beta_1, \ldots, T <: \beta_k\}$$

then the algorithm ensures furthermore that $\forall p, q, 1 \leq p \leq m, 1 \leq q \leq k : \alpha_p <: \tau_j \wedge \tau_j <: \beta_q$. If the algorithm fails to find τ_j, or not all subtype and supertype constraints can be satisfied by τ_j, then the algorithm returns no substitution for T and generates an error message.
If a type parameter does not have equality constraints, then supertype constraints have to be solved to infer the type parameter. These constraints are solved by computing the *lub* of all types in the supertype constraints. After computing the *lub*, the algorithm verifies that all subtype constraints of the type parameter that is being inferred, are satisfied. Given the constraints:

$$\{\alpha_1 <: T, \ldots, \alpha_m <: T, T <: \beta_1, \ldots, T <: \beta_k\}$$

The algorithm infers T to be $lub(\alpha_1, \ldots, \alpha_n) = \rho$ and ensures that $\forall q, 1 \leq q \leq k : \rho <: \beta_q$. If the subtype constraints cannot be satisfied, then the algorithm returns no substitution for T and produces an error message.
At last, if a type parameter T does not have either equality or supertype constraints, then the algorithm infers the type parameter based on the combination of constraints from the context and the subtype constraints. If the constraints from the context contain either equality or supertype constraints, then the type parameter is inferred as mentioned above, otherwise it is inferred to be the *glb* of all types in subtype constraints. If *glb* does not exist then the algorithm will not return a substitution for T and generates an error message.
If a type parameter does not have any type constraints at all, then it is simply inferred to be *Object*.

```
<T, S> List<S> foo(Map<T, ? super T> a, List<? super S> b){
    return ...;
}
...
Map<Number, Integer> m = ...;
List<Double> ld        = ...;
List<Float>  lf        = foo(m, ld);
```

Listing 4.16: Type conflicts in constraints on T and S

Consider, for example, the code in Listing 4.16. The generated constraints for the invocation of the method *foo* are $\{T = Number, T <: Integer, S <: Double\}$. The constraint solver solves the constraints for T by inferring T to be *Number*. After that, the constraint solver substitutes T in the constraint $T <: Integer$ with *Number* to verify that inferred type is correct. Since *Number* is not a subtype of *Integer*, the constraint solver will not return a substitution for T, but generates an error message expressing that the inferred type *Number* is not a subtype of *Integer*. Since there are no more constraint involving T, the constraint solver moves on to the next type variable S. The type variable S has a single subtype constraint, therefore additional constraints from the context are required. Since the invocation appears on the right hand side of an assignment,

the constraints $\{S = Float\}$ are generated from the initial constraint $List<S>$ $<: List<Float>$. Combining all the constraints on S yields the constraints $\{S = Float, S <: Double\}$. The constraint solver infers S to be *Float* from these constraints, but upon substituting S with *Float* the constraint solver discovers that *Float* is not a subtype of *Double*. The constraint solver will return no substitution for the type variable S, but generates an error message expressing that *Float* is not a subtype of *Double*.

4.3.4 Extended Constraint Solver

In the previous section we showed how constraints are solved to infer type parameters of generic methods. In this section we extend the constraint solver to check the bounds of type parameters and verify that the second set of constraints obtained by the constraint generation is satisfied. Figure 4.2 shows how bound constraints and the constraints obtained from constraint generation are used to infer type parameters.

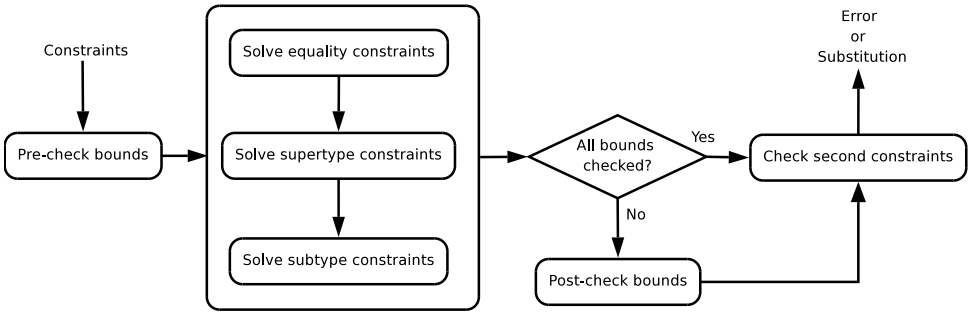

Figure 4.2: Extended constraint solver

The solver first checks that all types in the atomic constraints of a type parameter T satisfy the bounds of T. If these types do not satisfy the bounds, then an error message is generated for each bound conflict. The solver performs this step only if all bounds of T have been successfully instantiated. The constraint solver then proceeds to infer a type parameter using only atomic constraints. Solving atomic constraints, as explained earlier, can result in either returning a substitution or a set of error messages. After having instantiated a type parameter T, the solver ensures that the inferred type satisfies the bounds of T if that was not done before. As a last step, the solver confirms that the inferred types satisfy all the non-atomic constraints obtained during constraint generation.

4.4 Examples

Consider the utility class presented in Listing 4.17 where the method *foo* has been overloaded multiple times. Applying our weak method resolution from

Section 4.1 in Listing 4.5 to the first method invocation disqualifies the first method declaration as a potential candidate, because it does not have enough parameters. The other methods do have the right number of parameters, therefore, are marked as potential candidates and their signatures are converted to the signatures in Table 4.2.

Signature	Method on line
foo(Map, Collection)	4
foo(Map, List)	7
foo(HashMap, List)	10
foo(HashMap, LinkedList)	13
foo(HashMap, Set)	16

Table 4.2: Raw signatures of potential candidate

Since there are no primitive types in the first method invocation and declarations, and the method declarations have a fixed arity, the applicable methods can be determined using subtyping only, i.e. no type promotion is applied to the actual parameters. The second parameter *LinkedList* in the invocation is not a subtype of the generic interface type *Set*, therefore, the last method is not considered to be applicable.

Next, our weak method resolution from Section 4.1 needs to reduce the set of applicable of methods $\{foo_4, foo_7, foo_{10}, foo_{13}\}$[2] to a set of most specific methods. Comparing the parameters of foo_4 with foo_7 yields foo_7 to be more specific, because *List* is subtype of *Collection* but not vice versa. The method foo_7 is then compared with foo_{10}, which yields the last method to be more specific, because *HashMap* is a subtype of *Map*. At last foo_{10} is compared with foo_{13}, which yields foo_{13} to be more specific. Since there are no more methods to compare foo_{13} with, a singleton set $\{foo_{13}\}$ of most specific applicable methods is returned by the weak method resolution from Section 4.1. The reason a set is returned instead of a single method, is that we want to know all the methods that the programmer might be trying to call.

After have identified a set of most applicable methods, we proceed to infer the type variables of the methods in this set one method at a time. But before we can infer anything we need to generate the the constraints first. Applying the procedure in Listing 4.9 to actual and formal parameters returns the following constraint sets:

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

Normally the next steps would be to order the type variables and check the bounds, but since there is only one type variable with no bounds the algorithm skips these steps altogether.

Now we proceed to solve the type constraints starting with the equality constraints. Since there is only one equality constraint, we directly infer T to

[2]Subscript is the line number of the method signature in Listing 4.17

```
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);
```

Listing 4.17: Example of an utility class

be *Integer* and check that supertype and subtype constraints are satisfied after replacing T with *Integer* as illustrated below.

$$\begin{aligned} & \{T = Integer,\ \ Integer <: T,\ T <: Number\} \\ \Rightarrow\ & \{Integer <: T,\ T <: Number\}[T/\underline{Integer}] \\ \Rightarrow\ & \{Integer <: \underline{Integer},\ \underline{Integer} <: Number\} \\ \Rightarrow\ & \checkmark \end{aligned}$$

The last step in the algorithm is to check whether non-atomic constraints are satisfied, but in this case the set of non-atomic constraints is empty. Thus, the first method call type checks and no error message is generated. Normally our algorithm is used only when type checking in Java fails, but this example serves to show that the algorithm can also handle valid calls.

The second method invocation is processed exactly in the same way as the first invocation. The set of most specific applicable methods is again $\{foo_{13}\}$, but the set of generated type constraints is different:

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

The type variable T is inferred to be *Number* due to the equality constraint, and the rest of the constraints are verified after having substituted T with *Number*.

$$\begin{aligned} & \{T = Double,\ Number <: T,\ T <: Integer\} \\ \Rightarrow\ & \{Number <: T,\ T <: Integer\}[T/\underline{Double}] \\ \Rightarrow\ & \{Number \not<: \underline{Double},\ \underline{Double} \not<: Integer\} \\ \Rightarrow\ & \bot \end{aligned}$$

Clearly this method call is not valid because of the supertype and subtype conflicts shown above. The algorithm in this case generates an error message reporting about these conflicts.

For the third invocation, foo_{13} is also the only specific method and the generated constraints are:

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

The algorithm proceeds again by inferring T to be *Number* and verifies that the rest of the atomic constraints are satisfied. This time there are no conflicts as shown below.

$$\begin{aligned} & \{T = Number,\ Double <: T\} \\ \Rightarrow\ & \{Double <: T\}[T/\underline{Number}] \\ \Rightarrow\ & \{Double <: \underline{Number}\} \\ \Rightarrow\ & \checkmark \end{aligned}$$

Having successfully inferred the type of T, the algorithm verifies whether the inferred type is a valid substitution for T by checking the non-atomic constraints.

$$\begin{aligned} & \{\mathit{LinkedList}\ <?\ \mathit{extends}\ \mathit{Number}> <: \mathit{LinkedList}\ <?\ \mathit{super}\ T>\}[T/\underline{\mathit{Number}}] \\ \Rightarrow\ & \{\mathit{LinkedList}\ <?\ \mathit{extends}\ \mathit{Number}> \not<: \mathit{LinkedList}\ <?\ \mathit{super}\ \underline{\mathit{Number}}>\} \\ \Rightarrow\ & \bot \end{aligned}$$

The type *Number* as shown above is not a valid a substitution for T, therefore, the algorithm generates an error message describing the type conflict above.

For one last example consider the code in Listing 4.18. In this example all *bar* methods are considered potential candidates because their name and number of parameters match the name and number of parameter in the method invocations, respectively. Since our weak method resolution ignores all bound information, the signature of all methods *bar* are converted to *bar*(*Object*, *Object*).

```
1 class BarUtil{
2     static <T extends Number>void bar(T a, T b){}
3     static <T extends Integer>void bar(T a, T b){}
4 }
5 ...
6 BarUtil.bar('0', 3.14);
```

Listing 4.18: Type variables as parameters

Observing that the type of parameters in the invocation are primitive types, we can deduce that the weak method resolution will not find any applicable candidates using only subtyping. Therefore, the weak method resolution will try to find methods that are applicable using method invocation conversion. This leads to returning the following set as applicable method candidates: $\{bar_2, bar_3\}$.

Starting with the method bar_2, we generate the following constraints:

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

Primitive types are first promoted to their equivalent reference types, before they are added as constraints. Since we are dealing with a single type variable, the algorithm skips the ordering of type variables, and proceeds to bound checking. Bound checking is performed by taking all the types in supertype constraints of T and ensuring that they are subtypes of all the bounds of T. Thus in this case, we have one conflict $\mathit{Character} \not<: \mathit{Number}$. The algorithm generates an error message for this bound conflict, but continues to infer T to be *lub*(*Character*, *Double*). The result of *lub* is not returned as a substitution, because the we already know that the result type will be incorrect, but it is used to uncover all the type conflicts. For example, if T had one additional subtype constraint $T <: \mathit{Integer}$, then this constraint will emphasize the fact that the types used to infer the type variable T are not correct. Providing an error message that contains all type conflicts that contribute to making an invocation incorrect, will prevent the programmer from providing a fix for the bound

error only and compile the program again without ensuring that the other type conflicts are resolved.

After having inferred T to be $lub(Character, Double)$, the algorithm stops because there are no more constraints that have to be resolved, and returns the error message generated earlier about the bound conflict.

The algorithm infers the last method bar_3 using the exact constraints given above. This time the algorithm reports two bound conflicts $Charachter \not<: Integer$ and $Double \not<: Integer$. Just like before, the algorithm halts after computing $lub(Character, Double)$ and generates an error message describing both bound conflicts.

To summarize this last example, the algorithm generates an error message for each method declaration explaining why it is not applicable to arguments in the invocation.

Chapter 5

Error Messages

In this chapter we present a number of type conflicts and show the error messages our extension of the type-checking process generates for these conflicts. We also compare the error messages generated by our extension with the error messages produced by the two best Java compilers available at the moment; the Sun java compiler (javac) and the Eclipse java compiler (ejc). First we start with some examples of equality conflicts, and then will move on to supertype and subtype conflicts, and we end the chapter with few examples of bound conflicts.

5.1 Equality Errors

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

Listing 5.1: Simple equality error

Listing 5.1 provides a simple example of a type error caused by an inconsistency in the equality constraints. The error message generated for this example by our type-checking extension is the following:

```
Test1.java:19
Method <T>foo(Map<T, T>) of type Test1 is not applicable to the argument
of type (Map<Integer, String>), because:
  [*] The type variable T is invariant, but the types:
  - String in Map<Integer, String> on 18:9(18:22)
  - Integer in Map<Integer, String> on 18:9(18:13)
  are not the same type.
```

```
<T> void swap(Map<T, T> a){
    ...
    T key = ...;
    a.put(a.remove(T), key);
}
...
Map<Integer, Double> m1 = ...;
Map<? extends Number, ? extends Number> m2 = m1; // line 8
swap(m2);
```

Listing 5.3: Type variance

ejc and javac generate error messages that are somewhat similar. ejc informs the programmer that the method *foo(Map<T, T>)* is not applicable to the argument *Map<Integer, String>*, while the javac complains about not being able to find any method with the signature: *foo(Map<Integer, String>)* .

A good example of a mistake that novice programmers may make when working with wildcards is given in Listing 5.2.

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

Listing 5.2: Wildcard equality error

The call is illegal because type parameters in Java are invariant, while upper and lower bound wildcards are covariant and contravariant, respectively. To further illustrate what we mean by this, consider the code in Listing 5.3. The assignment on line 8 is legal because both types *Integer* and *Double* are subtypes of *Number*. If we would allow the call to the method *swap,* which swaps the keys and values in a map, with the argument *m2,* then we will be replacing integers with doubles and vice-versa.

The error message we generate for the illegal call in Listing 5.2 is:

```
Test1.java:20
Method <T>bar(Map<T, T>) of Type Test1 is not applicable to the argument
of type (Map<? extends Number, ? extends Number>), because:
  [*] The type variable T is invariant, but the type '? extends Number' is not.
```

The error messages reported by ejc and javac are:

```
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>)
```

```
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);
```

The first thing that one may notice when comparing the error messages of javac and ejc, is that they are very similar and they both include internal representation of wildcard types that are subjected to capture conversion. In our type system we try to follow the guide lines of the manifesto specified by Yang [5] to measure the quality of error messages. Therefore, we try to avoid any kind of information that is not directly available in the source code from being reported in an error message.

When comparing our error message with that of javac and ejc, it is not easy to judge whether our brief error message is better than that of javac and ejc. The understanding of our error message depends on whether the user is familiar with the term type invariance.

5.2 Supertype Errors

This kind of type errors occur when a type variable is inferred using equality constraints, but the inferred type does not satisfy the supertype constraints. Consider the code in Listing 5.4, where the type of T is instantiated to *Integer* from the equality constraint set $\{T = Integer\}$. Aside from the equality constraints, T also has a set of supertype constraints $\{Number <: T\}$.

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

Listing 5.4: Supertype error

Instantiating T with *Integer* means that we can substitute T with *Integer* in $Number <: T$. However, *Number* is not a subtype of *Integer*. This reasoning is captured in our error message given below.

Test1.java:6
Method <T>foo(Map<? extends T, T>) of Type Test1 is not applicable to the argument of type (Map<Number, Integer>), because:
[*] The type Number in Map<Number, Integer> on 5:9(5:13) is not a subtype of the inferred type for T: Integer.

javac and ecj both reject the method call above and generate the same error message given below; none of them explains why the invocation is invalid.

Test1.java:6:

<T>foo(Map<? extends T,T>) in Test1 cannot be applied to (Map<Number,Integer>)

foo(m);

5.3 Subtype Errors

A subtype error, much like a supertype error, occurs when the inferred type from equality or super constraints does not satisfy the subtype constraints. Consider the example given in Listing 5.5. The type parameter T is instantiated with *Number* due to the equality constraint $\{T = Number\}$. The type variable T also has a subtype constraint $\{T <: Integer\}$, which cannot be satisfied, because *Number* is not a subtype of *Integer*.

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

Listing 5.5: Subtype error

In the error message generated by our constraint solver, we explain that the method call is incorrect because of this inconsistency. The error message is given below:

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

javac and ejc produce a similar error message, but they do not explain why the method call is incorrect.

Subtype errors could also occur as a result of a failure to compute *glb*. An example of this case is in Listing 5.6. Since the method has *void* as its return type, we cannot instantiate the type variable T based on the context. Therefore, T must be instantiated based on the subtype constraints $\{T <: Number, T <: String\}$ with $glb(Number, String)$. But since *Number* and *String* have no common subtype, we generate the following error message:

```
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.
```

ejc and javac generate the same error message as they produced for the illegal call in Listing 5.5.

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

Listing 5.6: *Number* and *String* have no common subtype

Type errors that occur due to a failure to compute *glb* can be very confusing, because ejc and javac do not yet provide complete support for generics. Take for example the code in Listing 5.7. JLS specifies that the type variable T should be instantiated using the subtype constraint $T <: String$ and the bound constraint $T <: Number$. Thus, T must be $glb(String, Number)$. However, we have already established in the previous example that *Number* and *String* do not have a common subtype. Therefore, the call to the method *foo* is illegal, yet javac does not reject it. The reason this call is accepted by javac is that the bound constraint is ignored when instantiating T.

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

Listing 5.7: Simple and confusing

A few more examples of how odd subtype constraints are resolved by javac are given in Listing 5.8. All the method invocations need to be rejected, because the type that T is instantiated with in each invocation is in conflict with the

types in the parameters of that invocation. The constraints generated for the first call are $\{T <: List <String>\}$. Recall from Listing 3.5 that type variables that have only subtype constraints are inferred based on the context if the return type of the method where these type variables are declared in is not *void* and the result of the invocation occurs in an assignment context. The type variable T will, therefore, be instantiated based on the constraints generated from the assignment. These constraints are $\{T = Integer\}$. Thus, T is instantiated to *Integer*. Because *Integer* is not a subtype of *List<String>*, the call is illegal. Our constraint solver rejects this call as it should with the error message below, while javac accepts it.

```
Test.java:17
Method <T extends Number>bar(Map<? super T, ? super T>) of type Test is
not applicable to the argument of type (Map<List<String>, List<String> >),
because:
  [*] The type List<String> in Map<List<String>, List<String> > on
15:9(15:9) is not a supertype of the inferred type for T: Integer.
```

The second invocation is also invalid, for the same reason as the first call. The type variable T is instantiated from the context as a *Float*. But since *Float* is not subtype of *Double*, our constraint solver rejects the call and reports the following error message. javac on the other hand accepts the call without any complaints.

```
Test.java:18 Method
<T extends Number>bar(Map<? super T, ? super T>) of type Test is not
applicable to the argument of type (Map<Double, Number>), because:
  [*] The type Double in Map<Double, Number> on 16:9(16:13) is not a
supertype of the inferred type for T: Float.
```

The third call resembles the first call except that this time the type variable T is instantiated as *Float* instead of *Integer*. *Float* is not a subtype of *List<String>*. Therefore, our constraint solver rejects this call too.

Some may argue that it is safe to allow the invocations in Listing 5.8 because no heap pollution can take place. Since no write operations can be performed on a generic type that is instantiated with a lower bound wildcard without using casts, this might be true. However, allowing these calls contradicts with the type checking rules of method invocation. A method invocation in Java is legal if the actual parameters are subtypes of the formal parameters. In the first call, for example, T is instantiated to *Integer*, which means that the formal parameter becomes *Map<? super Integer, ? super Integer>*. The instantiated formal parameter is definitely not a supertype of the actual parameter, which has the type *Map <List<String>, List<String>>*.

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

Listing 5.8: Context

ejc also has its share of strange behaviour when dealing with subtype constraints. The code example in Listing 5.9 causes an interesting change in the error messages generated by the compiler. The constraints generated for

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

Listing 5.9: Subtype conflicts

both method calls are according to the JLS the same. These constraints are $\{T <: Number, T <: String\}$, or $\{T <: Number, T <: String, T <: Number\}$ if we would allow duplicates. Thus, it is obvious why the method invocations are rejected. Nevertheless, the error messages generated for the invocations are very different. The error messages produced for the first and second invocation are given below, respectively.

1. ERROR in Test8.java (at line 10)
   foo(m);
Bound mismatch: The generic method foo(Map<? super T,? super T>) of type Test8 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>

1. ERROR in Test8.java (at line 11)
   foo(m);
The method foo(Map<? super T,? super T>) in the type Test8 is not applicable for the arguments (Map<Number,String>)

This unexpected change in the reported error messages, is due to the way the ejc resolves subtype constraints. ejc instantiates the type variable T to be the first type it encounters in the subtype constraints, which explains why it claims in the first error message that T is instantiated to *String*. Our constraint solver, however, generates the same error message for both invocations, because the source of the type conflict is the same.

5.4 Bound Errors

Consider the code in Listing 5.10. The method call fails, because the type of its argument *int* is not a subtype of the bound *Cloneable*. This exact explanation is given below in the error message produced by our type checking extension.

```
Test1.java:6
Method <T extends Number & Cloneable>foo(T) of type Test1 is not applicable to the argument of type (int), because:
  [*] The type int of the expression '1' on 6:13 is not a subtype of T's upper bound Cloneable in 'T extends Number & Cloneable'.
```

```
<T extends Number & Cloneable> void foo(T a){}
...
foo(1);
```

Listing 5.10: Simple bound error

If we compile the same code above with javac and ejc, we get the following error messages:

```
Test1.java:6:
<T>foo(T) in Test1 cannot be applied to (int)
   foo(1);
```

```
ERROR in Test1.java (at line 6)
   foo(1);
Bound mismatch: The generic method foo(T) of type Test1 is not applicable for the arguments (Integer). The inferred type Integer is not a valid substitute for the bounded parameter <T extends Number & Cloneable>
```

We clearly see that ejc provides a good error message compared to javac. However, ejc does not say which bound exactly is conflicting with the type that T was instantiated to. It also includes the promoted type of the argument *Integer* in the message. In our messages we try to stay as close as possible to the

types given in the source code to prevent confusion.

Extending the method *foo* given above with one more parameter and relaxing the bounds of its type parameter T yields the code in Listing 5.11.

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

Listing 5.11: Bound error

Compiling this code with ejc gives the error message below, while javac fails to report an error message due to an internal error[1] in the compiler.

```
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 Ob-
ject&Comparable<?>&Serializable is not a valid substitute for the bounded
parameter <T extends Number>
```

ejc informs the programmer that the intersection type that T was instantiated with is not a subtype of its bound *Number*. This seems acceptable, but we believe that the type error can be explained without resorting to the inferred intersection type. The source of the error is the second parameter in the method invocation; the type *boolean* is not a subtype of *Number*, hence taking the *lub* of *int* and *boolean* will not be a subtype of *Number* either. The error message produced with our type system is provided below.

```
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'.
```

[1] This is caused by the infinite number of calls to the function *lub(Integer, Boolean)* from the function *lcta* when trying to compute the type that the interface *Comparable* needs to be instantiated with.

Chapter 6

Heuristics

In this chapter we present a number of heuristics that we have developed to improve the generated error messages beyond what was shown in Chapter 5. Except for a single heuristic that generates warnings, all the other heuristics that we present are program correcting heuristics that explain to the user how to modify the source code to resolve the type errors. Eight of the nine correcting heuristics that we implemented operate on the generated type constraints for a method invocation, and all of the heuristics are triggered by type conflicts. Since several heuristics can be triggered by the same type conflict, we have prioritized the heuristics so that only the heuristic with highest priority can extend the content of an error message. In reality, all the heuristics have the freedom to extend an error message, but only the extension of the heuristic with the highest priority will be visible in the final error message presented to the programmer. The priorities are assigned to the heuristics by an error manager, who has the responsibility to collect all the type conflicts discovered by the constraints solver. At the moment the priorities of heuristics are static, and a heuristic cannot change its priority. This is something that we might have to experiment with in the future.

In the first section, we present a heuristic for correcting type equality conflicts, and after that we discuss a heuristic that can be used in any kind of type conflict. In the third section we present a number of heuristics that are all targeted at correcting type conflicts caused by wildcards. In the last two sections we present a heuristic that can correct subtype conflicts and a heuristic that tries to compensate for the fact that the constraint solver sometimes might fail to report all type conflicts that contribute to rejecting a method call.

6.1 Maximal Equality

The Maximal Equality heuristic 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 Listing 6.1. The method invocation

leads to the constraints:

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

The type variable T gives rise to a type equality conflict because $Integer \not\equiv Number$. 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 first possible choice will give rise to a new conflict with the third constraint ($Number \not<: T$), the error message generated for the method invocation is extended with a repair hint that can lead to T being instantiated to *Number*, as given below.

```
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:
  [*] The 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 type. However, replacing Integer on 6:13 with Number may solve the type conflict.
```

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

Listing 6.1: Equality type conflict

In some occasions it might be the case that an equality conflict has multiple solutions. By a solution we mean how the types in the invocation should be modified in order for the invocation to be correct. Listing 6.2 provides two examples of an equality conflict that has two solutions.

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

Listing 6.2: Equality conflict with two solutions

The invocations have exactly the same constraints $\{T = Number, T = Integer\}$, 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

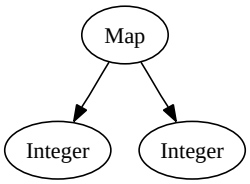

Figure 6.1: Tree representation of *Map<Integer, Integer>*

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 in the source code 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. We consider a parametrized type to be a tree, and a non-parametrized type to be a tree with a single node. Non-parametrized types are treated as single atomic entities. Thus, removing an *int* is as costly as removing a *String*. The algorithm that computes the costs of converting a type to another type works by flattening the tree structure of types into lists and then counting the number of differences between these lists. For example, converting *int* to *String* takes 2 edits, because we have to remove *int* and add *String*. Converting *List<Integer>* to *Map<Integer, Integer>* would take 3 edits. Flattening these types results in the lists [*List, Integer*] and [*Map, Integer, Integer*](see figure 6.1 for the tree representation of this type), which we call the source and target list respectively. The source list does not contain the type *Map* from the target list, thus this type must be added (1 edit). *Integer* occurs only once in the target list, thus we need to add an extra *Integer*(1 edit). At last, the target does not contain the type *List*, so this this type must be removed (1 edit).

Thus, for the first invocation in Listing 6.2 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 criteria that can be used determine the best solution. Consider for example the code in Listing 6.3, 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 following error message for the invocation in Listing 6.3:

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

Listing 6.3: Equality conflict with two equivalent solutions

```
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:
  [*] The 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.
```

Localized changes, however, are not always desired. For example, the user may have compiled the code in the middle of an unfinished re-factoring. 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 Listing 6.3.

Until now, the presented examples for this section have been rather simple and straightforward. Now we will show some more complicated examples. Consider the code in Listing 6.4, where the heuristic must take the type variable S into consideration when trying to solve the equality conflict of T.

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

Listing 6.4: Conflict involving two type variables

The problem of finding a solution for the equality conflict of T is a constraint satisfaction problem (CSP). The variables in our CSP are the type variables $\{T, S\}$, the domains of the variables are $\{Integer, String\}$ for T and $\{String\}$ for S, and the constraints are $\{S <: T\}$. 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* [10]. The degree heuristic tries 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:

$< T,\ S\ extends\ T,\ R,\ U\ extends\ Map <R,\ S> \&\ T,\ V\ extends\ Map <U,\ T>>$

Using the variable dependency discussed in Section 4.3.2 we obtain the dependency graph given in Figure 6.2. The arrows denote how type variables depend on each other. The solid arrows are direct dependencies, and the dashed arrows are indirect dependencies. The node in the graph with the most incoming edges is the most constrained variable. Note how assigning a type to the type variable T limits which types can be assigned to the type variables S, U and V. The type variable T and R are independent of each other, because instantiating one type variable does not affect how the other can be instantiated. If the type variable R had the same number of incoming edges in the graph as T, then the heuristic would have composed a set of partial solutions by simply taking the cartesian product of the domains of T and R.

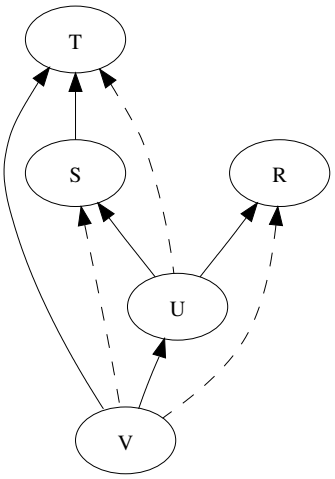

Figure 6.2: Type variable dependency graph

Forward checking is used to ensure that when instantiating a type variable, the type variables directly depending on the instantiated this type variable continue to have at least one type in their domain that is within their bounds. For example, when instantiating the type variable T to σ, the domains of the type variables S and V are checked to verify that S can be instantiated so that $S <: \sigma$ and V can be instantiated so that $V <: Map <?,\ \sigma>$.
Listing 6.5 provides an example of what kind of errors this heuristic can repair. Note that every type variable has an equality conflict. Since all type variables are

```
<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);
```

Listing 6.5: Single type variable group

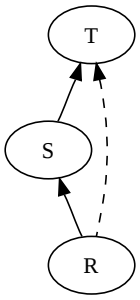

Figure 6.3: Dependency graph of type variables in Listing 6.5

members of the same dependency graph (see Figure 6.3), the heuristic processes all type variables as a single type variable group. The error message generated by our system using the maximal equality heuristic is:

> 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:
> [*] The 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.
> [*] The 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.
> [*] The 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.

Grouping type variables based on their bound dependencies when solving equality conflicts is definitely better than solving the conflicts for each type variable separately, because we can prevent proposing repairs that would otherwise cause conflicts with the bound constraints. However, sometimes grouping type variables just based on their bound dependencies is not enough. Consider the code in Listing 6.5, where the type variables in the first method declaration form one type group, while in the second they form two separate groups.

```
<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);
```

Listing 6.6: Conflict in non-atomic constraints

The error messages generated by our system for the method invocations are:

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:
[*] The 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:
[*] The 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.
[*] The 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.

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 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. $\{T, S\}$ and $\{R\}$ will become $\{T, S, R\}$, 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 given below:

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:
[*] The 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.

Test9.java:38

```
<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);
```

Listing 6.7: Conflict in all constraints

> 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:
> [*] The 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.
> [*] The 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. However, replacing Number on 37:21 with Object may solve the type conflict.

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.

Someone may argue why we do not always run this heuristic in its strict mode since it will only provide safer solutions. The reason we do not do this is because non-atomic constraints themselves can be a source of conflicts. To illustrate this, consider the code in Listing 6.7. Running this heuristic by default in strict mode will not propose a repair of the invocation in Listing 6.7, because the type variable T is overconstrained with the (unsatisfiable) non-atomic constraints. In this case the programmer has to figure out by himself what is wrong with the invocation. But if the heuristic is run in its normal mode, then a repair hint suggesting to replace *Number* in the type of the parameter `m` with *Integer* will be presented to the programmer. In this case, if the programmer applies the suggested repair and compiles for the second time, then he will be presented with another error. However, the constraint solver will present a repair for this second error too, which is suggested by the opposite wildcards heuristic discussed in Section 6.3.2.

6.2 Context Type Invariance

In this section we present a powerful heuristic that relies on information from the context to correct type conflicts. The heuristic is based a very simple logic: type variables in Java are invariant. This heuristic, however, can only be applied in a situation where all the following conditions are met:

- The method has parametrized return type, that contains type variables.
- The method invocation appears in an assignment context.
- Let S be the type the left-value and R be the return type of the method, then $R <: S$ should give rise to an equality constraint.

Consider the code in Listing 6.8, where the type variable T has a subtype conflict, because T is instantiated to *Number*, but *Number* is not a subtype of *Integer*.

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

Listing 6.8: Return type invariance

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 further 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. The error message generated after having applied the context type invariance heuristic is given below.

```
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.
```

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 Listings 6.9 and 6.10.

In Listing 6.9 the heuristic solves the subtype conflict ($Object \not<: Number$) of T by inferring from the context that T must be *Integer*. However, substituting T

```
<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);
```

Listing 6.9: Bound conflict

```
<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);
```

Listing 6.10: Non-constraint conflict

with *Integer* in the bound of S causes a bound conflict, i.e. *Map<Number, String>* is not a subtype of *Map<Number, Integer>*.

In Listing 6.10 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:

$$List<?\ super\ Number> \not<: List<?\ extends\ T>$$

Although, the heuristic does check the bound and non-atomic constraints before suggesting a repair, it will, however, propose a repair for the call in Listing 6.11 as given below.

> 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.

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 always proposes repairs for type variables that are not directly responsible for bound conflicts or a conflict in the non-atomic constraints, like in Listing 6.11. Suggested repairs,

```
<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);
```

Listing 6.11: Indirect conflict

however, are not always in line with the beliefs and intention 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 Listing 6.12. The type variables T and R have no conflicts at all, but S has a bound conflict that involves both T and R, therefore, the heuristic suggests to change the types that T and R were instantiated to, as given in the error message below.

```
<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);
```

Listing 6.12: Indirect bound conflict

```
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.
```

6.3 Wildcards

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

6.3.1 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 Listing 6.13. The type variable has only one equality type

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

Listing 6.13: Instantiating bounded type variable with a lower bound wildcard

$\{T = capture(?\, super\, Integer)\}$. Therefore, the constraint solver instantiates T to ω_1, which is obtained by performing capture conversion on "? *super Integer*"[1]. 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. Therefore, we know that the programmer might have intended to use an upper bound wildcard instead of the lower bound. Thus, the heuristic changes the type *List<? super Integer>* to *List<? extends Integer>* and verifies whether all constraint can be satisfied simultaneously. The change of the lower bound to an upper wildcard in Listing 6.13 will cause the constraints to change to $\{T = capture(?\, extends\, Integer)\}$. The type variable will be instantiated to a capture of "? *extends Integer*" ω_2. Since ω_2 is subtype of *Integer,* ω_2 will also be a subtype of *Number*. The heuristic informs the programmer that bound change in the wildcard may solve the conflict as given below.

> 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

Consider the code in Listing 6.14. The heuristic will not suggest a repair for this invocation. Converting *List<? super Integer>* to *List<? extends Integer>*

[1] see Section 2.5 about capture conversion

would yield the constraints:

$$\{T = capture(?\ extends\ Integer)\}$$
$$\Cup$$
$$\{List<?\ extends\ Integer> <: List<?\ super\ T>\}$$

Because T has only one type equality, 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[2] "? *super Integer*".

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

Listing 6.14: Replacing '*super*' with '*extends*' doe not solve the conflict

6.3.2 Opposite Wildcards

This heuristic can be considered to be a variation of the previous heuristic, because it solves the type conflicts caused by incompatible wildcards. Consider, for example, the code in Listing 6.15. The invocation does not type check, be-

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

Listing 6.15: Wildcard conflict

cause *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` as shown below.

> 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:

[2] see Section 2.3 about type containment

[*] 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>

The heuristic can also repair conflicts where the lower bound wildcard is used in the formal parameters and the upper bound in the actual parameter (see Listing 6.16 for an example). The invocation of the method *bar* does not type check

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

Listing 6.16: Another wildcard conflict

for the same reason as in Listing 6.15: 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 as illustrated below.

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>

The heuristic can also handle nested wildcards, such as in Listing 6.17, because it checks iteratively whether the bounds in the wildcards are compatible. The error messages produced for the method invocations are presented below.

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>>

```
<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);
```

Listing 6.17: Nested wildcards

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> >

6.3.3 Wildcard Reduction

In the previous sections we showed how our heuristics can revert the bounds of wildcards in the actual parameters until they match the wildcards in the formal parameters to correct type conflicts. In this section we present two heuristics that correct conflicts by reducing wildcard instantiated types to concrete types.

6.3.3.1 Super Object

Consider, for example, 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 Listing 6.18, where a type conflict is caused by the use of the wildcards "? *super Object*". The captures of wildcards are never equivalent

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

Listing 6.18: Equality conflict caused by wildcards

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

Listing 6.19: Type error caused by '? *super Object*'

even if the wildcards are syntactically equivalent. Replacing the wildcards with just *Object* resolves this type error; therefore the heuristic hints at this repair in the error message below.

> 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.

Besides equality conflicts, this heuristic can also revolve type conflicts in the non-atomic constrains (see Listing 6.19 for an example). The invocation does not type check, because T is instantiated to a capture of "? *super Object*" 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. This repair is suggested in the error message presented below.

> 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.

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

Listing 6.20: Type conflict caused by wildcards

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

Listing 6.21: Supertype conflict

> However, replacing '? super Object' on 5:29 with Object may solve the type conflict.

6.3.3.2 Extends Final

This heuristic is very similar to the previous one, because 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 Listing 6.20. 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 below, that the substitution of the wildcards will correct the type conflict.

> 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:
> [*] The 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.

Wildcards cannot only cause type equality conflicts, but also supertype conflicts. Consider, for example, the code in Listing 6.21. The constraints gener-

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

Listing 6.22: Subtype conflict

ated for this invocation are: $\{T = \omega, String <: T\}$, where $\omega = capture(? \ extends \ String)$. 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. The heuristic includes a hint to this repair as shown in the error message below.

> 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.

Upper bound wildcards are used sometimes to disallow write operations on parametrized types. For example, given a list l_1 of type *List<Integer>*, we can prevent adding elements to this list by assigning it to a list l_2 of type List<? extends Integer>. Now, if we allow the access to l_1 only through l_2, then we cannot write to l_2 because the type system will not allow it. This is not something we encourage, but if the programmer wants to use this practice, then he/she can disable the use of the extends final heuristic by providing the command-line option "**-df**" to the type system.

6.4 Maximal Subtyping

This heuristic corrects subtype errors in a way similar to how the maximal equality heuristic, presented in Section 6.1, corrects 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 Listing 6.22 are $\{Integer <: T, Number <: T, T <: Double, T <: Number\}$. If we ignore the order in which constraints are processed, then we could say that T will either be *Number*[3] or *Double*[4]. If we let T be *Number*, then

[3] Result of *lub({Integer, Number})*
[4] Result of *glb({Double, Number})*

```
<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);
```

Listing 6.23: Non-computable *glb*

we have the following conflict: $Number \not<: Double$. But if we let T be *Double*, then we obtain the following conflicts: $Integer \not<: Double$ and $Number \not<: Double$. Thus, instantiating T with *Number* will satisfy more constraints than instantiating with *Double* will. Hence the error message below.

> 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.

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 Listing 6.23. In this case, the heuristic computes the best possible *glb,* which is $FocusEvent = glb(\{AWTEvent, FocusEvent\})$. Comparing the number of conflicts that would arise if we instantiate T to *FocusEvent* or *AWTEvent*[5] yields the following results:

$$T = FocusEvent \Rightarrow \begin{cases} ComponentEvent \not<: FocusEvent \\ AWTEvent \not<: FocusEvent \\ FocusEvent \not<: String \end{cases}$$

$$T = AWTEvent \Rightarrow \begin{cases} AWTEvent \not<: \quad FocusEvent \\ AWTEvent \not<: \quad String \end{cases}$$

[5] Result of *lub({FocusEvent, ComponentEvent, AWTEvent})*

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

```
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.
```

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.

6.5 Equality Warnings

We have mentioned in one of the previous chapters that the order in which type constraints are processed has an impact on the generated error messages. 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 Listing 6.24. The constraint solver stops at 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

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

Listing 6.24: Hidden error

that warns the programmer that the types in the equality are not only in conflict with each other, but also with the supertype and/or subtype constraints. For the invocation in Listing 6.24, the heuristic generates the warning shown below.

> 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:
> [*] The 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)

The warning in this case is an essential part of the error message, because it will prevent the user from providing a repair for 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. For example, given the following constraints:

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

The heuristic will report that only *String* will be in conflict with *Integer* and *Number*, even though $Comparable<Number>$ will also cause a conflict if the user decides to replace *Integer* with *Number*. The reason why we do not report the type $Comparable<Number>$, 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 presented at the beginning of this chapter had found a repair for the equality conflicts.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we have shown that an existing Java type system can be extended to produce better feedback when generic method invocations fail to type check. This extension does not require any drastic changes in the original type checking process. It is invoked only when a type error occurs in a generic method call. We have shown that even though generic instantiation of type variables can be performed without any type conflicts, it is still not guaranteed that a method invocation will type check. Therefore, as a counter measurement, we require that the constraints passed to the constraint solver in our extension should also include non-atomic constraints, which will be used to verify that type variables are correctly instantiated. Due to the way generic instantiation is interwoven in the method resolution, we also require a weaker version of the method resolution that will always return a non-empty set of methods, even if generic instantiation fails (It is, however, allowed to return a empty set if the user is calling a non-existing method). It is important that the weaker method resolution returns a set of methods instead of a single method, because we would like to know all the methods that the user might be trying to call in order to produce better error messages. But, to prevent redundant and unnecessary error messages, we restrict the number of returned methods to a set of most specific methods only, which are determined without using any generic type information.

7.2 Future Work

There are different directions in which future research could continue. One can conduct an empirical research to study into what extent our framework helps programmers to learn generics, and how the the framework can be further improved so that programmers can take full advantage of it. One could also implement additional heuristics or improve the current ones. In this thesis, it was assumed that all actual parameters are available in source form. This can

lead to repair heuristics suggesting to change a type located in a class file. This, however, is not always possible if the source code is not available. Consider, for example, proposing to the programmer to change the type of the field `out` in the class *System*, which is part of the package *java.lang*. To prevent proposing such repair hints, heuristics must be aware of the origins of all the types in the constraint set.

To further improve the repair hints, it might be interesting to explore how the type constraints of nested calls can be solved in one step. The framework infers currently each method invocation separately. Given, the invocation $g(f(a_1, \ldots, a_m), b_2 \ldots, b_n)$, our framework (and Java type checker) infer the type variables of the method f first, and then instantiates f with the inferred types. The instantiated method f is subsequently used to infer g. The disadvantage of this approach is that a repair heuristic might suggest to change the type of $f(a_1, \ldots, a_m)$. Since the return type of $f(a_1, \ldots, a_m)$ was inferred, it might not be easy for a beginner programmer to figure out how to change the return type. It will be a lot easier if a heuristic could instruct the programmer to change the types of the parameters $a_1, \ldots, a_m$ instead of the return type.

In this thesis we have focused only on the method invocations where the compiler has to infer the omitted type parameters. But, there is also room to improve the error messages produced for method invocations and constructors that provide their own type arguments, such as *Collections.<Integer>.fill(...,...)*. Our constraint solver can be used as a basis to improve the generated feedback for these kind of method and constructor calls. As proof of concept, we have added minimal support for type parametrized method invocations and constructor calls. The provided type arguments in an invocation are first checked to make sure they satisfy the bound constraints. If they do, then we proceed to check whether actual parameters are subtypes of formal parameters, where all type variables in the later parameters are substituted by the user provided type arguments. If not all actual parameters are subtypes of their corresponding formal parameters, then we ignore the provided type arguments and try to instantiate the type variables using our constraint solver. If the constraint solver successfully instantiates all the type variables, then we advise the programmer to change the provided type arguments to the types found by the constraints solver.

Appendix A

Architecture and Implementation

A.1 Architecture

The architecture of our framework can be best described as a set of modules and classes that are connected and centered around the type checker of the JastAdd Extensible Java Compiler (JastAdd EJC). This illustrated in Figure A.1. 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 4.2. After having generated the constraints, the type checker passes these constraints along to our constraint solver with the return type of a method declaration and the type of the lvalue if the invocation appears in a 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. The

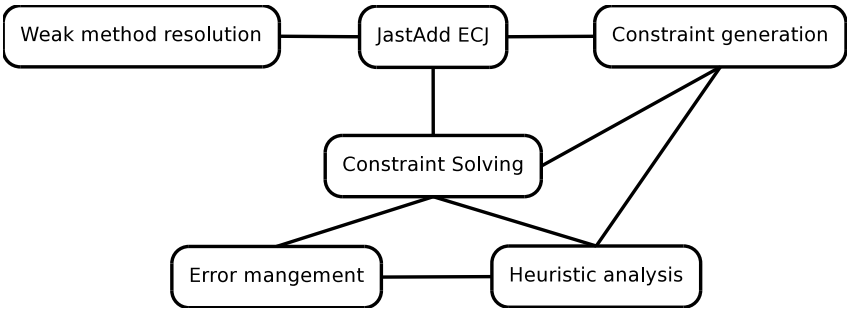

Figure A.1: Architecture

error manager is also used by the heuristics to extend the error messages generated by the constraint solver. The heuristics use the error manager also to check whether they should run or not. For example, the Maximal Equality heuristic runs only if it encounters an equality error message in the error manager.

A.2 Implementation

Our framework is implemented as a combination of JastAdd code and normal Java code. JastAdd[4, 11] 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. The framework is built as an extension to the JastAdd EJC which is built entirely using JastAdd.

For the convenience of the weak method resolution discussed earlier, the ordering of type variables, and the computation of greatest lower bound, which was not present in the version of JastAdd EJC used in this project, are implemented using JastAdd. We have contributed the module that we have developed for computing the greatest lower bound to the creator and maintainer of JastAdd EJC, who has added it to the repository as a part of the compiler. Several other aspects are implemented in JastAdd code such as modifying the type parameters of a parametrized type, printing simple names for types used in an invocation and all other required functionality that was not present in the classes of JastAdd EJC. Since JastAdd EJC is a compiler that is still in development, there were also some bugs that needed fixing. For example, the containment of lower bound wildcards was incorrect, which did not only impact on our project but on the entire type checker of the compiler. This bug was fixed in JastAdd.

All the other aspects of the project that did not require augmenting the classes of the JastAdd EJC are implemented in Java. The constraint solver, for example, is implemented in three classes as illustrated by the class diagram in Figure A.2. The superclass BasicSolver has two important fields named "constraintsMap" and "errMan". The first field constraintsMap is a map that contains all constraints on each type variable of a method as it is returned by the ConstraintMapBuilder, which is responsible for generating constraints from the actual and formal parameters as discussed in Section 4.2. The type ConstraintSet contains all the types of a type variable which are divided into three separate lists for ease of access. The equality constraints list contains all the types that should be equal to the inferred type of a type variable. For example, if a type variable S has the constraints $\{S = Number, S = Integer\}$, then the equality constraints list contains $Number$ and $Integer$. The supertype constraint list contains all the types that should be subtypes of the type that the type variable will be instantiated to. The subtype constraint list contains all the types that should be supertypes of the type that the type variable will be instantiated to. The second field errMan is an error manager which collects all the type error messages generated by the solver. The fields supertypeAfterChecks and subtypeAfterChecks are used to collect the non-atomic constraints.

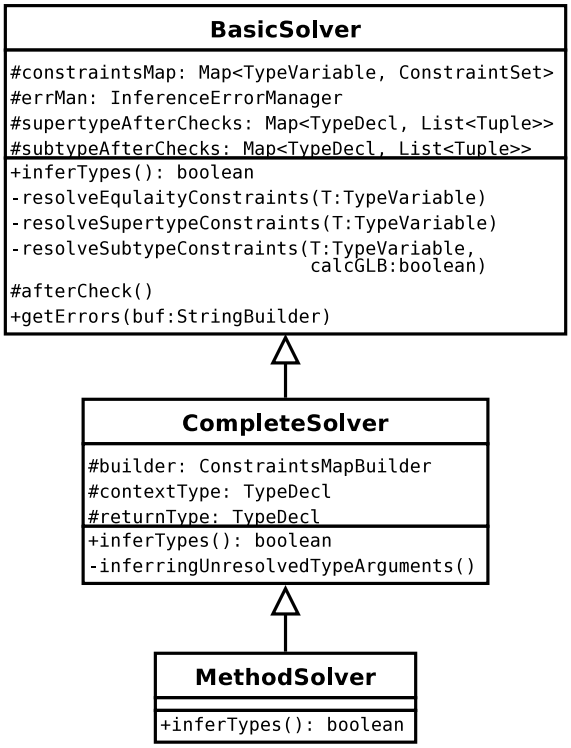

Figure A.2: Constraint solvers classes

To infer the types of each type variable, the method inferTypes can be used. The method iterates through constraintsMap and calls the methods resolveEqulaityConstraints, resolveSupertypeConstraints, and resolveSubtypeConstraints on each type variable. The method resolveEqulaityConstraints checks whether all the types in the equality constraints are the same and instantiates T (the type variable passed to it as an argument) to the type in equality constraints. If the types in the equality constraints are not the same, such as *Integer* and *Number* or '? *super String*' and '? *super String*', then it adds an equality error message to the field errMan. The method resolveSupertypeConstraints checks first whether T was not already instantiated by the method resolveEqulaityConstraints. If T was indeed instantiated, then the method checks that all the types in the supertype constraints are subtypes of the type that T was instantiated to. If T was not instantiated before, then the method resolveSupertypeConstraints will instantiate T by computing the least upper bound of all the types in supertype constraints. The method for computing the least upper bound is provided by the class *TypeDecl*, which is a part of the JastAdd EJC. The last method, resolveSubtypeConstraints, checks whether T was already instantiated by resolveSupertypeConstraints. If T was instantiated, then the method checks that types in the subtype constraints are supertypes of the type that T was instantiated to. If T was not instantiated, then resolveSubtypeConstraints will instantiate it to the greatest lower bound of all the types in the subtype constraint set if the parameter calcGLB is set to true.

After having iterated though all the type variables in constraintsMap, the method inferTypes calls the method afterCheck to verify that the types that the type variables were instantiated to can satisfy the non-atomic constraints.

The CompleteSolver class augments the class BasicSolver by adding the possibility to infer type variables that do not have any constraints. The MethodSolver class extends the CompleteSolver class by allowing to run type heuristics after having processed all the type constraints on all the type variables.

The heuristics are also implemented in plain Java. All the heuristics implement a simple interface which provides a single method named analyze, which is called from the MethodSolver class (see Figure A.3). The method findSourceOfBoundError in AbstractHeuristics is used by the heuristics to search for the type variables that are directly responsible for an bound conflict. Each heuristic provides its own implementation of the analyze method which performs an analysis on the constraints obtained from the constraints solver and reports its finding to the error manager (also obtained from the MethodSolver) by adding an instance of interface MessageExtension to it or adding the extension directly to error message that triggered the heuristic. The error manager, as mention earlier, decides based on the priority of the heuristic which message extension will be shown to the programmer.

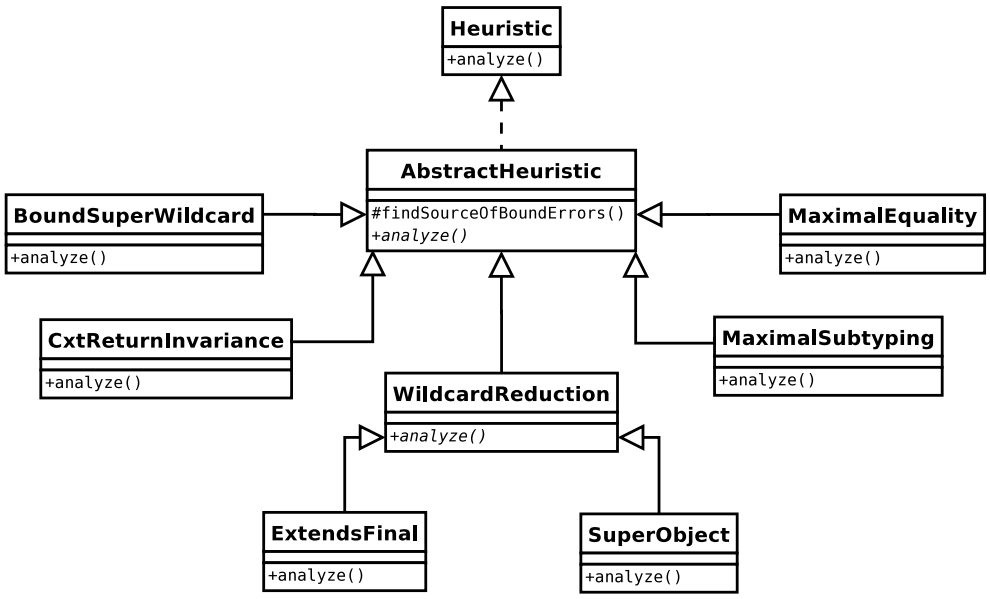

Figure A.3: Heuristics classes

Appendix B

Manual

B.1 Download and install

All the required files to use and test our framework are currently available from the svn repository at:
https://svn.cs.uu.nl:12443/repos/Swa5/project.

To check out a copy, users must install the version control system Subversion available at http://subversion.tigris.org/. The files can be checked out from a console using the following command:

```
svn co https://svn.cs.uu.nl:12443/repos/AFP_Exercise_2006/project
```

After having downloaded the files, users can build the framework by going to the directory Java1.5Backend, which contains an ant build file. Running the command "ant" will perform all the steps necessary to build the framework and will create a bin directory in the same folder where the svn checkout was performed. Users that do not have ant installed on their system can download it from http://ant.apache.org. The bin folder will contain all the compiled files needed to run JastAdd EJC combined with our framework. The repository contains also a large number of test files categorized by the kind of type error, which can be used to learn about the kind of error messages our framework can generate.

B.2 Usage

The JastAdd EJC can be used to compile java files by running the following command in the bin directory created by ant:

```
java JavaCompiler [options] <java files>
```

Running JavaCompiler with the option "-help" will show all the available options the compiler can take, as illustrated below:

```
Usage: java Java5Compiler <options> <source files>
  -verbose                  Output messages about what the compiler is doing
```

```
-classpath <path>          Specify where to find user class files
-sourcepath <path>         Specify where to find input source files
-bootclasspath <path>      Override location of bootstrap class files
-extdirs <dirs>            Override location of installed extensions
-d <directory>             Specify where to place generated class files
-help                      Print a synopsis of standard options
-version                   Print version information

Heuristic options:
-strict                    Run heuristics in strict mode
-verystrict                Run heuristics in very strict mode
-df                        Disable running the final heuristic
-collapse                  Print each bound error on a line
-distinct                  Print minimal repairs even if they are not localized
```

The standard options of JastAdd EJC are printed first and the options that are specific to our framework are printed below the line "Heuristic options:". The option "-strict" is used by the heuristics Maximal Equality, Extends Final, Super Object and Maximal Subtyping discussed in Chapter 6. When this option is set, the heuristics will print a repair for a type variable only if the non-atomic constraints involving this type variable are satisfied by the repair. The option "-verystrict" is used by the heuristic Context Type Invariance to make sure that the entire invocation will type check. Thus, if an invocation, for example, has a conflict in the non-atomic constraints, but the non-atomic constraints do not involve the type variables for which the heuristic suggests to be repaired, then these repairs will be printed, unless the option "-verystrict" is set. The option "-df" is used in the constraint solver to disable running the Extends Final heuristic for the reason that was discussed in Section **??**. The "-collapse" option is used to force the error manager to print each bound error of a type variable on a separate line. Printing each bound error an separate line has the advantage of quickly letting the programmer know how many errors he has to deal with. But, the disadvantage of printing each bound error separately is that an error message can become very large. The final option "-distinct" is used in the Maximal Equality heuristic to print repairs that require a minimal number of edits, even though they might have to be performed in different places in the source code.

Appendix C

Syntax

C.1 Name syntax

PackageName:
    Identifier
    PackageName . Identifier

TypeName:
    Identifier
    PackageOrTypeName . Identifier

MethodName:
    Identifier
    AmbiguousName . Identifier

PackageOrTypeName:
    Identifier
    PackageOrTypeName . Identifier

AmbiguousName:
    Identifier
    AmbiguousName . Identifier

C.2 Type syntax

ReferenceType:
    ClassOrInterfaceType
    TypeVariable ArrayType

ClassOrInterfaceType:
    ClassType

InterfaceType

ClassType:
TypeDeclSpecifier
TypeArguments$_{opt}$

InterfaceType:
TypeDeclSpecifier
TypeArguments$_{opt}$

TypeDeclSpecifier:
TypeName
ClassOrInterfaceType . Identifier

TypeName:
Identifier
TypeName . Identifier

TypeVariable:
Identifier

ArrayType:
Type [ ]

C.3 Type argument syntax

TypeArguments:
< ActualTypeArgumentList >

ActualTypeArgumentList:
ActualTypeArgument
ActualTypeArgumentList , ActualTypeArgument

ActualTypeArgument:
ReferenceType
Wildcard

Wildcard:
? WildcardBoundsOpt

WildcardBounds:
extends ReferenceType
super ReferenceType

NonWildTypeArguments:
< ReferenceTypeList >

ReferenceTypeList:
ReferenceType
ReferenceTypeList , ReferenceType

C.4 Primary expression syntax

Primary:
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal Type . class
void . class
this
ClassName.this
(Expression)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess

C.5 Method invocation synatx

MethodInvocation:
MethodName (ArgumentList$_{opt}$)
Primary . NonWildTypeArguments$_{opt}$ Identifier (ArgumentList$_{opt}$)
super . NonWildTypeArguments$_{opt}$ Identifier (ArgumentList$_{opt}$)
ClassName . super . NonWildTypeArguments$_{opt}$ Identifier (ArgumentList$_{opt}$)
TypeName . NonWildTypeArguments$_{opt}$ Identifier (ArgumentList$_{opt}$)

ArgumentList:
Expression
ArgumentList , Expression

Bibliography

[1] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[2] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ: Extending the Java Programming Language with type parameters. 1998.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, third edition, July 2005.

[4] 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.

[5] Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[6] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[7] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference (colored version) (black and white version). In *POPL 2001*, 2001.

[8] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Massachusetts Institute of Technology Cambridge, Massachusetts 02142, February 1 2002. ISBN:0262162091.

[9] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265, New York, NY, 1998.

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

[11] Görel Hedin Torbjörn Ekman. Jastadd extesible java compiler. http://jastadd.cs.lth.se/web/extjava.

[12] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *Proceedings of FOOL 12*. School of Informatics, University of Edinburgh, 2005.

[13] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.

[14] Jun Yang. Explaining Type Errors by Finding the Source of a Type Conflict. In *SFP '99: Selected papers from the 1st Scottish Functional Programming Workshop (SFP99)*, pages 59–67, Exeter, UK, UK, 2000. Intellect Books.