Adding overloading to Java type inference

Andreas Stadelmeier and Martin Plümicke

Baden-Württemberg Cooperative State University Stuttgart
Department of Computer Science
Florianstraße 15, D-72160 Horb
{a.stadelmeier, m.pluemicke}@hb.dhbw-stuttgart.de

Zusammenfassung

In this paper we extend our Java with type inference by adding methods. Functions had been realized as lambda expressions defined in fields until now, which led to the restrictions that no overloading is available. Therefore the main challenge of adding methods is to deal with overloading. We present the change of the data-structures and the algorithm.

1 Introduction

In [Plü11, Plü14] we have proposed to introduce a type inference system for Java. The reason for developing such a system are the often complex and sometimes confusing principal types of Java-expressions with functional interfaces and wildcards. Let us consider the following example defining a lazy map-function for a declared class.

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

class IntVector extends Vector<Integer> {

    void m() {
        Function<Function<Integer, Integer>, IntVector> map1;
        map1 = f -> this.stream().map(f)
                    .collect(IntVector::new, IntVector::add, IntVector::addAll);
        Function<Function<Integer, Integer>,Vector<Integer>> map2;
        map2 = map1; //not correct(!)
      }
  }
```

There are two local variables `map1` and `map2`, which represent a function which takes an (`Integer`→ `Integer`)-function and gives elements of different result types. The first result type is `IntVector` and the second is its supertype `Vector<Integer>`. But the assignment `map2 = map1;` is not correct in Java,

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at http://ceur-ws.org

although in the view of type theory the type of `map1` is a subtype of the type of `map2`. The reason is that the function constructor is normally contravariant in the argument types and covariant in the result types. In Java contra- and covariance have to be defined by wildcards. This means for a declaration of `map2` with the principal type

```
Function<? super Function<Integer,Integer>,? extends Vector<Integer>> map2;
```

the assignment `map2 = map1;` would be correct. Our type inference system allows to leave out the types such that problems like this do not arise any longer. The principal types are inferred automatically. The program looks then like

```
class IntVector extends Vector<Integer> {

    void m() {
        map1;
        map1 = f -> this.stream().map(f)
                    .collect(IntVector::new, IntVector::add, IntVector::addAll);
        Function<Function<Integer, Integer>,Vector<Integer>> map2;
        map2 = map1; //correct
      }
 }
```

Our type inference algorithm works on a subset of the Java 8 programming language. The subset of the Java language supports mostly all features of Java 8 including generics and lambda expressions. Only methods are simulated by lambda expressions of fields. This leads to the restriction that overloading is impossible. Furthermore every type declaration, which is necessary in Java 8, is optional in the given subset. The type inference algorithm is able to infer all possible types for a missing type declaration in the source code.
To publish a workable implementation of the algorithm it ships bundled with an Eclipse plugin [SP14]. The Eclipse plugin is able to parse the Java subset and inject missing types into the source to generate valid Java 8 source code. Afterwards a Java 8 compiler can be used to compile the inferred source. Due to this approach only one type solution is possible, because the standard Java compiler needs an explicit typed Java source code as input. Every type solution is unified to a most general version.
At the moment our algorithm excludes methods and method invocation. We will consider the addition of them in this paper. The main challenge is the consideration of overloading and overriding.
The paper is structured as follows: First we consider the basic structure of the type inference algorithm. Then we consider the extensions using overloading. In the third section the consequences for the whole algorithm are considered. After that we close with a conclusion and an outlook.

2 Basic structure of the algorithm

The algorithm is integrated into a compiler for the described subset of Java 8. The following modifications differ this application from a standard Java compiler. The semantic check of the compiler is replaced by the type inference algorithm. Instead of Java Bytecode the compiler generates a typed output of the parsed source code.
The type inference algorithm itself consists of two functions **TYPE** and **SOLVE** and works on the abstract syntax tree of a parsed input. In **TYPE** first type placeholders (fresh type variables) are added for every unknown type in the syntax tree. Subsequently the **TYPE**-function of the algorithm generates constraints by traversing the given tree. After that in the function **SOLVE** the generated constraints are unified [Plü09]. The actual type inference algorithm generates a single constraint set and therefore only a single unification is needed.

Every statement implies a single set of constraints. A constraint describes a correlation between a type and type placeholders.

2.1 Type Assumptions

Preceding to the **TYPE**-function the algorithm generates a set of type assumptions. This set contains the following informations: A listing of the classes available in the scope of the inferred source as well as the fields and methods of these classes. A class assumption additionally holds information about their subclasses.
A method assumption includes the name of the method, its parameter types and its return type.
This set of type assumptions is used by the **TYPE**-function as well as the **UNIFY**-method.

3 Overloading

In the case of the descriped type inference algorithm *overloading* has a slightly different meaning than the overloading of methods known from the Java programming language. While in normal Java only method names can be overloaded, in Java with type inference method invocations can be overloaded too. In case of a method call the type inference algorithm has to generate constraints out of the given information. In an untyped environment only the name of the method and its parameter count is known. Neither the types of the parameters nor the Java class which defines the called method.
Therefore the algorithm searches the type assumptions for methods with the same name and number of parameters. Every matching method is considered as a possible option. For each of these options the **TYPE**-function generates a constraint set. At the time of the **TYPE**-function it is impossible to determine which of the generated constraints leads to a correct constraints set. Therefore every constraint set generated for one of the possible methods gets added to the final constraints set. These are bundled to an **OR**-constraint set (see Section 3.1).
For the following cases the type inference algorithm generates multiple constraints for a single invocation.

- The used method for the invocation is overloaded with more than one method, which has the same number of parameters.
- There is more than one known class which implements a method with the same signature and parameter count as the invoked one.

3.1 Constraint Set

Out of usability concerns a special data structure is implemented for a constraint set. This data structure consists of two different subgroups.

OR-constraint set: This set represents a set of constraint sets which are connected by **OR** operator.

AND-constraint set: This set represents a set of constraint sets which are connected by **AND** operator. The **AND**-constraint set can also contain single constraints. Only this set would be needed to collect the constraints of the **TYPE**-function without overloading. To support overloading the **AND**-constraint set is also capable of holding **OR**-constraint sets.

Due to the **OR** connected constraints this structure allows the storage of multiple constraint compositions in a compact manner.
But the **UNIFY**-algorithm works on a set of single constraints. So preceding to this step the described constraints set needs a conversion to a set consisting out of single constraints. This is done by a cartesian product operation. Every composition of the given constraints is possibly a right solution.

3.2 Overloading example

Let us consider the following example:

```
class OL {
    m(Integer x) { return x + x; }
    m(Boolean x) { return x || x; }
}

class Main {
    main(x) {
        ol;
        ol = new OL();
        return ol.m(x);
    }
}
```

First the algorithm gives a fresh type variable to every statement or expression, which has an unknown type for now. In the given example the fresh type variables or type placeholders (TPH) are assigned as the following. A type placeholder (TPH) named `ol` is assigned to the untyped Field `ol`. The type placeholder `TPH B` is used as the type variable for the Parameter `x` and the `TPH C` for the return type of the `main` method. The `TPH A` represents the type of the return statement `ol.m(x)`.
Afterwards the **TYPE** algorithm creates a constraint set. The generated constraints for the class `Main` in this example look like the following.

```
(OL <. TPH ol) & // generated by "ol = new OL();"
//"ol.m(x)" generates following OR constraints due to overloading:
[[ (java.lang.Integer <. TPH A), (TPH B <. java.lang.Integer), (TPH ol <. OL) ] |
 [ (java.lang.Boolean <. TPH A), (TPH B <. java.lang.Boolean), (TPH ol <. OL) ]] &
(TPH A <. TPH C) // TPH C is the return type of the method "main"
```

4 Consequences for the whole algorithm

The most time consuming operation of the type inference algorithm is the **UNIFY**-part. For every set of constraints generated by the cartesian product of the **OR**- and **AND**-constraints sets the **UNIFY**-operation is applied. But the size of the cartesian product grows exponentially with every overloaded method that comes into account. Therefore the type inference algorithm becomes very slow when inferring large untyped source code.
A solution to this problem is trying to filter out incorrect constraint sets before the cartesian product gets applied.
Figure 1 shows an exemplary situation. The type assumptions in this example contains multiple assumptions for the *intValue*-method. The classes `Double`, `Integer`, `Float`, `Long` and the class `Example` implement a method named *intValue* with zero parameters. The **TYPE**-function generates constraints for each of these assumptions. This happens for both of the *intValue*-method invocation in the given example. So two **OR** constraint sets are generated which contain five constraint sets each. This equals 25 different constraints after the cartesian product was built.
Only one of the constraint sets can be unified to a valid mapping of type placeholders to Java types. The other constraint sets hold inconsistent constraints. These constraint sets can be eliminated before building the cartesian product, such that only one unification is necessary.
This happens in several steps. At first all **OR** constraint sets are separated from the **AND** constraint sets. Every constraint in the **OR** constraint sets needs to pass a test. In this test the constraint set from the **OR** constraints gets unified together with all **AND** constraints. If the **UNIFY**-method succeeds

```
class Example {
        test(){
                var1;
                var2;
                var1 = this;
                var2 = var1.intValue();
                return var1.intValue();
        }

        int intValue(){
                return 1;
        }
}
```

Abbildung 1: Simple source code which generates a huge amount of constraint sets

on this subset of the constraints, the **OR** constraint remains in its constraints set. Otherwise the tested constraint set gets removed out of its **OR** constraint set. Afterwards the cartesian product can be applied on the remaining constraints.
For the example in Figure 1 this means the following. The **TYPE**-method generates two **OR** constraint sets with five elements each. For 10 different constraint sets the test and therefore the **UNIFY**-method gets applied. Only one constraint for each of the **OR** constraint sets passes this test. So the cartesian product returns only a single constraint set, which has to be unified.
By sorting out inconsistend constraints from the **OR** constraint sets the size of the cartesian product can be reduced.
This method works best when only one of the overloaded methods is applicable. Otherwise multiple constraints remain in the **OR** constraint sets which leads to a exponential increase of the resulting cartesian product with every overloaded method call.

5 Conclusions and outlook

In this paper we have described the addition of methods to our Java subset set with type inference, the main challenge in this context is overloading. We showed the data-structure for storing the contraint sets and gave an optimization possibility to reduce the complexity of the induced cartesian product.
As there is the possibility that the cartesian product still grows enormously, we have the further idea to split the set of constraings before building the cartesian product. Each part of constraints would then consist only of constraints that type variables interdepend, which means that only a small number of cartesian products would be necessary.

Literatur

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü11] Martin Plümicke. Well-typings for Java$_\lambda$. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 91–100, New York, NY, USA, 2011. ACM.

[Plü14] Martin Plümicke. More type inference in Java 8. In *Perspectives of Systems Informatics - 9th International Andrei Ershov Memorial Conference, PSI 2014, St. Petersburg, Russia*, Lecture Notes in Computer Science. Springer, 2014. (to appear).

[SP14] Andreas Stadelmeier and Martin Plümicke. Java type inference as an Eclipse plugin. In *Programmiersprachen und Rechenkonzepte: 31. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte", Bad Honnef, 28. – 30. April 2014*, Technische Berichte der TU Wien, 2014. (to appear).