Type–Inference in Java

Martin Plümicke, Timo Holzherr and Arne Lüdtke

University of Cooperative Education Stuttgart/Horb
Department of Information Technology
Florianstraße 15
D–72160 Horb
m.pluemicke@ba-horb.de
timo@holzherr.de
arne@g-free.de

Abstract. With the introduction of Java 5.0 [1] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<? extends Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.
Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not. Furthermore there are methods whose principle types would be intersection types. But intersection types are not implemented in Java 5.0. This means that Java 5.0 methods often don't have the principle type which is contradictive to the OOP-principle of writing re-usable code.
This has caused us to develop a Java 5.0 type inference system, which assists the programmer by calculating types automatically. This type inference system allows us to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principle types. In this contribution we present the ideas including type inference for types with wildcards.

1 Introduction

In this paper we present a type inference algorithm for a core language of Java 5.0. In [2] we presented a type inference algorithm for a restricted Java 5.0 type system, which didn't include wildcards. Now we give the whole algorithm which also includes types with wildcards.
Type inference means that we can implement Java 5.0 programs without type annotations for method parameters, return types and local variables. Those are automatically calculated by the type inference system.
In the example in fig. 1 the class `Matrix` extends `Vector<Vector<Integer>>`. `Matrix` has the method `mul`, which implements the multiplication of matrices. The parameters, the return type, and the local variables are explictly typed (underlined in fig. 1). If we consider the type annotations more accurately, we will recognize that there is more than one possibility to give correct type annotations. The return type, the type annotation of `m`, and the type annotation of `ret` are

```
class Matrix extends Vector<Vector<Integer>> {
   Matrix mul(Matrix m){
     Matrix ret = new Matrix();
     Integer i = 0;
     while(i <size()) {
        Vector<Integer> v1 = this.elementAt(i);
        Vector<Integer> v2 = new Vector<Integer>();
        Integer j = 0;
        while(j < v1.size()) {
           Integer erg = 0;
           Integer k = 0;
           while(k < v1.size()) {
              erg = erg + v1.elementAt(k)
                        * m.elementAt(k).elementAt(j);
              k++; }
           v2.addElement(new Integer(erg));
           j++; }
        ret.addElement(v2);
        i++; }
     return ret; }
}
```

Fig. 1. The class `Matrix`

not unambiguous. For example the type `Vector<Vector<Integer>>` would also be a correct type annotation. In this case the type of the method `mul`, has apart from the type `Matrix` $\rightarrow$ `Matrix` also the type `Vector<Vector<Integer>>` $\rightarrow$ `Vector<Vector<Integer>>`, and all mixtures of those two types. Furthermore

`Vector<? extends Vector<? extends Integer>>`
$\rightarrow$ `Vector<? extends Vector<? extends Integer>>`

would also be a correct type. The conjunction of the different types of a method is called intersection type of the method. In Java 5.0 no annotations of intersection types are allowed.
The idea of Java 5.0 type inference is to omit these type annotations. The system automatically calculates a principle intersection type of the methods.
The paper is organized as follows. In the second section we formally introduce the Java 5.0 types. In the third section we give the type unification algorithm, which is the base of the type inference algorithm. The type inference algorithms and a large example of its application follow in the fourth section. We close this paper by a conclusion and an outlook.

2 Type–inference algorithm

The type–inference algorithm determines all absent types of methods, its parameters, and its local variables.

2.1 Overview

There are two approaches of type inference in object-oriented languages. The first approach is based on the Hindley–Milner type system [3]. The main aim of all type inference algorithms following the Hindley–Milner approach is to infer principle types. In contrast the second approach founded by Palsberg and Schwartzbach infers types as precise as possible [4]. This is done for code optimization in compilers.

Our type inference approach for Java 5.0 programs has the aim, to infer principle types. Therefore our algorithm is oriented at the Hindley–Milner approach.

Similar as the Hindley–Milner approach our approach bases on an unification algorithm. We extend the ordinary unification problem (cp. [5]): For two type terms θ_1, θ_2 a substitution is demanded, such that $\sigma(\theta_1) \leq^* \sigma(\theta_2)$. This unification problem is not unitary, but finitary, which means that there are a finite number of general unifiers.

2.2 The algorithm

The basic idea of the algorithm is that each expression, each statement and each block is typed by simple types and that each method is typed by function types. These types are determined step by step during the run of the algorithm.

Type assumptions: First, we assume for each expression, for each statement, and for each block a fresh type variable as a type–placeholder. The types of the methods are assumed as function types, which consists also of type–placeholders for each argument and the return type.

Run through the abstract syntax tree: During a run through the abstract syntax tree, the types of each expression, each statement, and each block are determined. This is done step by step. At each position of the abstract syntax tree there are type assumptions of the expressions, the statements, and the blocks, respectively, and there are conditions for these types given by the Java 5.0 type inference system for each Java 5.0 construct of the core language. The type assumptions are unified by type unification as the corresponding rules in the type inference system define.
After that the resulting unifiers are applied to the respective type assumptions.

Multiplying the assumptions: If the result of the type unification contains more than one unifier, for each unifier a new set of type assumptions is generated. The algorithm is continued on both sets of type assumptions.

Erase type assumptions: If the type unification fails, the corresponding set of type assumptions is erased.

New method type parameters: If at the end, there are type–placeholders contained in type assumptions, these type–placeholders are replaced by new introduced method type parameters.

Intersection types: If at the end, there is more than one set of type assumptions for a method, this method has an intersection type, which is then generated.

2.3 Type inference example

We consider again the *matrix* example from the introduction. We apply the algorithm to the corresponding abstract syntax tree of the class `Matrix` (fig. 1), where the underlined type annotations are erased.

Type assumptions: All expressions, statements, and the block are typed by fresh type variables as type–placeholders (denoted by greek letters).

```
class Matrix extends Vector<Vector<Integer>> {
   {$\alpha$} mul({$\beta$} m) {
      {$\gamma$} ret = new Matrix();
      Integer i = 0;
      while(i <size()) {
         {$\iota$} v1 = this.elementAt(i);
         {$\kappa$} v2 = new Vector<Integer>();
         Integer j = 0;
         while(j < v1.size()) {
            {$\chi$} erg = 0;
            Integer k = 0;
            while(k < v1.size()) {
               {$\chi$} erg = {$\chi$} erg + ({$\xi$}({$\iota$} v1).elementAt(k))
                  * ({$\psi$}({$\phi$} ({$\beta$} m).elementAt(k)).elementAt(j)); k++;}
            v2.addElement({$\chi$} erg); j++; }
         ret.addElement({$\mu$} v2); i++; }
      return ret; }}
```

In the following all type–placeholders are determined by unification, gradually. As an example we consider the assignment $\{\gamma\}$ `ret = new Matrix()`. We have to unify $\{\texttt{Matrix} \lessdot \gamma\}$. The result of this unification is:
$\{\{\gamma \mapsto \texttt{Matrix}\}, \{\gamma \mapsto \texttt{Vector<Vector<Integer>>}\},$
$\{\gamma \mapsto \texttt{Vector<}_?\texttt{Vector<Integer>>}\}, \{\gamma \mapsto \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>}\},$
$\{\gamma \mapsto \texttt{Vector<}_?\texttt{Vector<}^?\texttt{Integer>>}\}, \{\gamma \mapsto \texttt{Vector<}^?\texttt{Vector<Integer>>}\}\}$
Now, the set of assumptions is multiplied, scuh that fore each result there is an own set of type assumptions.
The algorithm is continued for all type–placeholders step by step.
As result we get following the intersection type:

$$\texttt{mul}:\&_{\beta,\alpha}(\beta \rightarrow \alpha), \tag{1}$$

where

$\beta \in \{\texttt{Vector<Vector<Integer>>}, \texttt{Vector<Vector<}_?\texttt{Integer>>},$
$\texttt{Vector<}_?\texttt{Vector<Integer>>}, \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>},$
$\texttt{Matrix}\}$

$\alpha \in \{\texttt{Matrix}, \texttt{Vector<Vector<Integer>>}, \texttt{Vector<}^?\texttt{Vector<Integer>>},$
$\texttt{Vector<}_?\texttt{Vector<Integer>>}, \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>},$
$\texttt{Vector<}_?\texttt{Vector<}^?\texttt{Integer>>}\}$

2.4 Principle Type

First, we consider the definition of Damas and Milner [3]: *A type-scheme for a declaration is a* principal type-scheme*, if any other type-scheme for the declaration is a generic instance of it.*
We generalize this informal definition for Java 5.0 programs as follows:
"An intersection type-scheme for a declaration is a principal type-scheme*, if any other type-scheme for the declaration is a generic instance of one element of the intersection type-scheme."*
We proved for our algorithm:

Theorem 1 (Principle type property). *The type inference algorithm calculates a principle type.*

3 Conclusion and Outlook

We gave a type inference algorithm for Java 5.0. The algorithm calculates different method's parameter types and its corresponding return types. These types are connected to an intersection type. The inferred intersection type is a principle type. The algorithm bases on a type unification algorithm.
If we consider again the example in section 2.3, we can see that

$$\begin{aligned}&\texttt{mul} : \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>} \rightarrow \texttt{Matrix} \\ &\quad \& \, \texttt{Vector<Vector<}_?\texttt{Integer>>Matrix}\end{aligned} \tag{2}$$

is also a principle type, as all other types have smaller argument types and greater result types. This means that some inferred types are not necessary. Future work has to be done extending the algorithm, such that it calculates efficiently only a *reduced* principle type.
Further investigation are necessary to integrate intersection types into a Java 5.0 compiler. In the moment only for one element of an method's intersection type byte-code could be generated. The goal is to compile byte-code such that for each element of the intersection type byte-code is generated.

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. 3rd edn. The Java series. Addison-Wesley (2005)
2. Plümicke, M., Bäuerle, J.: Typeless Programming in Java 5.0. In Gitzel, R., Aleksey, M., Schader, M., Krintz, C., eds.: 4th International Conference on Principles and Practices of Programming in Java. ACM International Conference Proceeding Series, Mannheim University Press (August 2006) 175–181
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. Proc. 9th Symposium on Principles of Programming Languages (1982)
4. Palsberg, J., Schwartzbach, M.I.: Object-oriented type systems. John Wiley & Sons (1994)
5. Baader, F., Snyder, W.: Unification Theory. [6] chapter 8 447–533
6. Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier Science Publishers B.V. (2001)