Optimization of the Java type unification

Martin Plümicke
Duale Hochschule Baden-Württemberg, Campus Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. The global type inference problem in Java can be reduced to the type Java unification problem. In a former article we have presented a Java type unification algorithm. The algorithm works fine for small examples. But for larger examples the used memory and the runtime is not acceptable.

In this paper we give a short summary of our algorithm and present some optimizations. We present two kinds of optimizations. The first kind do not reduce the solutions. The second kind restricts the solutions which means that we have guarantee that the algorithm determine the relevant solutions.

We present as a benchmark the matrix multiplication. The types cannot be determined with the original algorithm because of memory and runtime lack. But with the optimizations the types of the matrix multiplication can be determined with ordinary computer in acceptable runtime.

1 Introduction

Global type inference enables to write Java programs without any type annotation. The type inference algorithm introduces the types during compile time. This allows to omit the types without loosing the static type property. Such an algorithm is developed in the Java-TX project [Plü15,PS17].

The type inference problem can be reduced to a type unification problem. One of the major problems is the runtime of the type unification algorithm. The type unification problem is NP-hard. The algorithm described in [Plü09,SP18] iterates all elements of a cartesian product of all possible type annotations. This leads to an exponentiell runtime in comparison to the number of omitted types. There are examples, where the algorithm terminates after hours.

In the worst-case this is probably unchangeable, but in many cases the runtime can be reduced enormously.

In this paper we present after a brief summary of the main part of the type unification algorithm in Section 2 and a large example in Section 3 which we use as a benchmark, different optimizations in Section 4.

Early filtering of not solved forms: After the reduction steps the sets which includes error-pairs are erased.
Reduced evaluation of cartesian product elements: This leads to enormous reduction of memory use.

Evaluation of only one inequation each recursion iteration: This reduces the number of incorrect pairs.

Considering dependent mappings of error-pairs: Many branches in the backtracking, which results in errors, are not considered.

Considering only relevant solutions: This leads to an enormous reduction of the number of calculated solutions.

We close with a summary and an outlook.

2 The type unification algorithm

In this section we give a brief summary of the type unification algorithm. For more details we refer to the [Plü09,SP18].

In the following let \( \theta, \theta' \), \( \theta_i \) are Java types and \( \leq^* \) the subtyping relation of Java types.

The type unification problem is given as: For a set of type term pairs \( \{ (\theta_1, \theta'_1), \ldots, (\theta_n, \theta'_n) \} \) a substitution \( \sigma \) is demanded, such that

\[
\sigma(\theta_1) \leq^* \sigma(\theta'_1), \ldots, \sigma(\theta_n) \leq^* \sigma(\theta'_n)
\]

Some more notations are used in the following:

- \( \theta, \theta' \), \( \theta_i \) are Java types.
- The subtyping relation is denoted by \( \leq^* \).
- \( \theta \cong \theta' \) means that \( \theta \) and \( \theta' \) should be unified such that \( \theta = \theta' \).
- \( \theta < \theta' \) means that \( \theta \) and \( \theta' \) should be unified such that \( \theta \leq^* \theta' \).
- \( \theta \triangleleft \theta' \) means that \( \theta \) and \( \theta' \) should be unified such that \( \theta \) and \( \theta' \) are in subtype relation in argument position.
- We write \( A <\? B \) for \( A \cong \theta' \text{ extends } B \).
- We write \( C <\? D \) for \( C \cong \theta' \text{ super } D \).
- A set of equations is in solved form, if all pairs \( a_i \cong \theta_i \) where \( a_i \) are pairwise different type variables and for all \( i, j \) \( a_i \) does not occur in \( \theta_j \) and all pairs

\[
a < b, a \triangleleft b
\]

consists only of type variables.

The type unification algorithm is given as:

**Input:** Set of equations \( Eq = \{ \theta_1 \cong \theta'_1, \ldots, \theta_n \cong \theta'_n \} \)

**Precondition:** \( \theta_i, \theta'_i \) are Java types.

**Output:** Set of all general type unifiers \( Uni = \{ \sigma_1, \ldots, \sigma_m \} \)

**Postcondition:** For all \( 1 \leq j \leq m \) and for all \( 1 \leq i \leq n \) holds \( (\sigma_j(\theta_i) \leq^* \sigma_j(\theta'_i)) \).

1. Repeated application of reduction rules, such that either pairs consists of at least one type variable or the pairs are unsolvable.
2. For each pair \( a < \theta \) and \( a \triangleleft \theta \) for all subtypes \( \overline{\theta} \) of \( \theta \) pairs \( a \cong \overline{\theta} \) are built and for each pair \( \theta < a \) and \( \theta \triangleleft a \) for all supertypes \( \theta' \) of \( \theta \) pairs \( a \cong \theta' \) are built.
3. The cartesian product of the sets from step 2 is built.
4. Application of the subst rule which replaces for all pairs \( a \doteq \theta \) in all types all occurring \( a \) by \( \theta \).
5. For all changed sets of type terms start again with step 1.
6. Filter all results in solved form (either pairs of the form \( a \doteq \theta \), \( a \ll b \), or \( a \ll \gamma b \), where \( a \) and \( b \) are type variables) and unite them.

Let us consider an example.

**Example 1.** In this example we use the standard Java types \( \text{Number}, \text{Integer}, \text{Stack}, \text{Vector}, \text{AbstractList}, \text{and List} \). It holds \( \text{Integer} \ll \text{Number} \) and \( \text{Stack}<a> \ll \text{Vector}<a> \ll \text{AbstractList}<a> \ll \text{List}<a> \).

As a start configuration we use

\[
\{(\text{Stack}<a> \ll \text{Vector}<a>), \ (\text{AbstractList}<\text{Integer}> \ll \text{List}<a>)\}.
\]

In the first step the reduction rules are applied twice:

\[
\{(a \ll \gamma \text{Number}, a \ll \gamma \text{Integer}) \}
\]

With the second step we receive in step three:

\[
\begin{align*}
&\{(a \doteq \gamma \text{Number}, a \doteq \text{Integer})\}, \{(a \doteq \gamma \text{Number}, a \doteq \gamma \text{Number})\}, \\
&\{(a \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \{(a \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \\
&\{(a \doteq \text{Number}, a \doteq \gamma \text{Integer})\}, \{(a \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \\
&\{(a \doteq \gamma \text{Integer}, a \doteq \text{Integer})\}, \{(a \doteq \gamma \text{Integer}, a \doteq \gamma \text{Number})\}, \\
&\{(a \doteq \gamma \text{Integer}, a \doteq \gamma \text{Integer})\}, \{(a \doteq \gamma \text{Integer}, a \doteq \gamma \text{Integer})\}, \\
&\{(a \doteq \gamma \text{Integer}, a \doteq \gamma \text{Integer})\}, \{(a \doteq \gamma \text{Integer}, a \doteq \gamma \text{Integer})\}
\end{align*}
\]

In the forth step the rule \( \text{subst} \) is applied:

\[
\begin{align*}
&\{(\gamma \text{Integer} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Number})\}, \\
&\{(\gamma \text{Integer} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \{(\gamma \text{Integer} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \\
&\{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Number})\}, \{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Number})\}, \\
&\{(\gamma \text{Integer} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \{(\gamma \text{Integer} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \\
&\{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Number})\}, \{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Number})\}, \\
&\{(\gamma \text{Integer} \doteq \gamma \text{Integer}, a \doteq \gamma \text{Integer})\}, \{(\gamma \text{Integer} \doteq \gamma \text{Integer}, a \doteq \gamma \text{Integer})\}, \\
&\{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}, \{(\gamma \text{Number} \doteq \gamma \text{Number}, a \doteq \gamma \text{Integer})\}
\end{align*}
\]

The underlined sets of type term pairs lead to unifiers.

Now we have to continue with the first step (step 5). With the application of reduction rule (step 1) and step 6, we get three general type unifiers:

\[
\{(a \mapsto \gamma \text{Number}), (a \mapsto \gamma \text{Integer}), (a \mapsto \text{Integer})\}.
\]

### 3 Benchmark example

Let us have a look at another example which we use as benchmark for our optimizations. The class \( \text{Matrix} \) (Fig. 1) with the method \( \text{mul} \) implements the
```java
class Matrix extends Vector<Vector<Integer>> {
    mul(m) {
        var ret = new Matrix();
        var i = 0;
        while (i < size()) {
            var v1 = this.elementAt(i);
            var v2 = new Vector<Integer>();
            var j = 0;
            while (j < v1.size()) {
                var erg = 0;
                var 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.** Matrix multiplication

```
[(Matrix ≪ E), (O ≪ java.lang.Integer), (java.lang.Boolean ≪ W),
(W ≪ java.lang.Boolean), (S ≪ java.lang.Integer),
(java.lang.Integer ≪ java.lang.Integer), (Z ≪ Matrix), (V ≪ U), (Z ≪ Matrix),
(Matrix ≪ Matrix), (F ≪ java.lang.Integer), (T ≪ S), (J ≪ java.lang.Integer),
(AVJ ≪ M), (P ≪ O), (R ≪ java.lang.Integer), (java.lang.Integer ≪ AE),
(java.util.Vector<java.lang.Integer> ≪ N),
(java.util.Vector<java.lang.Integer> ≪ java.util.Vector<java.lang.Integer>),
(AA ≪ java.lang.Integer), (java.lang.Integer ≪ AD), (AXP ≪ AB),
(U ≪ AF), (D ≪ java.util.Vector<AXP>), (L ≪ java.util.Vector<AVJ>),
(L ≪ Matrix), (U ≪ java.lang.Integer), (AE ≪ S), (java.lang.Boolean ≪ Q),
(Q ≪ java.lang.Boolean), (void ≪ AI), (M ≪ K), (java.lang.Integer ≪ AXR),
(void ≪ AM), (F ≪ AN), (AD ≪ java.lang.Integer), (N ≪ AXT),
(E ≪ java.util.Vector<AXU>), (E ≪ C), (N ≪ java.util.Vector<AXS>),
(X ≪ java.lang.Integer), (Y ≪ Matrix), (AB ≪ java.util.Vector<AXQ>),
(AC ≪ java.lang.Integer), (H ≪ java.lang.Boolean),
(java.lang.Boolean ≪ H), (O ≪ AJ), (G ≪ F), (K ≪ java.util.Vector<AXO>),
(AXQ ≪ AC), (Y ≪ Matrix), (AXO ≪ AA)]
```

**Fig. 2.** Type unification input from Matrix
matrices multiplication in Java-TX. The method `mul` has no declared argument type and no declared return type. The type inference algorithm shall determine them. The type inference problem is reduced to the type unification problem with the input given in Fig. 2.

We will use this example as a benchmark for the optimizations. Our measurement we have done with an eclipse Oxygen on Mac OS X with standard configurations. If we apply the algorithm to the input of Fig. 2 after many hours the program crashes with out of memory.

4 Optimizations

4.1 Early filtering of not solved forms

After step 1 pairs $\theta < \theta'$ or $\theta \equiv \gamma \theta'$ where $\theta$ and $\theta'$ is no type variable would let to errors. All sets of pairs which consists such pairs are erased.

4.2 Reduced evaluation of cartesian product elements

In Fig. 3 the cartesian product of step 3 is visualized. It is obvious that the number of pair sets grows enormously. E.g. the number of pairs in Fig. 2 is about 50. If foreach pair average 10 solutions are given, there are $10^{50}$ sets. We reduce the number of sets in memory, as we evaluate for each pair only one element at the same time. This means during the calculation of the unification algorithm only one set is in memory. If the set is unsolvable the set is erased. If it has a solution the solution is stored. In Fig. 4 we illustrate the optimization. This leads to an enormous reduction of used memory.
4.3 Evaluation of only one inequation each recursion iteration

We change step 2 of the algorithm such that only one element $a < \theta$ is evaluated to a set $\{a = \theta_1, \ldots, a = \theta_n\}$. All other pairs remain untouched. They will be evaluated in a later recursion iteration. This means the cartesian product in step 3 has only $n$ elements (number of solutions of $a < \theta$). It is visualized in Fig. 5.

Fig. 4. Reduced evaluation in step 2

Fig. 5. Evaluation of only one inequation
This leads to a reduction of the generated elements, as for all pairs \( a \doteq \theta_j \), which leads to errors, the respective element is removed after step 1 (cp. Section 4.1). This means that for this element for the remaining pairs no cartesian product is built.

4.4 Considering dependent substitutions of error-pairs

We change the data-structure of the pairs \( \theta \triangleleft \theta' \), such all substitutions which are applied to the respective pair are stored in the pairs. If an error occur after step 1 this pair would lead to an error as long as no dependent substitution of the pair is changed. This allows to go back until a dependent substitution is changed. All backtracking steps on the way back need not to be considered. The optimization is visualized in Fig. 6.

These four optimizations (Section 4.1 - 4.4) reduce runtime and used memory without reducing the number of solutions. Nevertheless, all these optimizations do not lead to satisfying results of runtime and memory use.

4.5 Considering relevant solutions

The following two approaches determines normally only one and in some case a few solutions. This means that many solutions are not determined. We have guarantee that the relevant solutions are calculated.
**One random solution** The naive approach to get only one solution is to stop the backtracking if one solution is reached. The runtime of this solution is very good. We get for the benchmark example a runtime of less than 10 seconds. The results are indeed correct but random and not unique.

**Covariance and contravariance of method argument and return types**

Before we will present the idea of relevant solutions let us have a look at the definition of principal type in this type system. In [Plü07] we made a proposal for a principal type definition.

Damas and Milner [DM82] define for ML-like languages a principal type: 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.

A generalization to the Java type system could be: An intersection type-scheme for a declaration is a principal type-scheme, if any (non-intersection) type-scheme for the declaration is a subtype of a generic instance of one element of the intersection type-scheme.

This idea leads to the formal definition:

**Definition 1.** An intersection type of a method $m$ 

$$m : ((\theta_1,1, \ldots, \theta_{1,n}) \rightarrow \theta_1) \& \ldots \& ((\theta_m,1, \ldots, \theta_{m,n}) \rightarrow \theta_m)$$

is called principal if for any correct type annotated method declaration

$$rty \ m(ty_1 a_1, \ldots, ty_1 a_n) \{ \ldots \}$$

there is an element $((\theta_1,1, \ldots, \theta_{1,n}) \rightarrow \theta_1)$ of the intersection type and there is a substitution $\sigma$, such that

$$\sigma(\theta_i) \leq^* rty, ty_1 \leq^* \sigma(\theta_1,1), \ldots, ty_n \leq^* \sigma(\theta_{i,n}).$$

This means that the argument types are contravariant and the return types are covariant. Therefore we make the following assumptions:

- Determine maximal (wrt. the subtyping relation) correct types for the arguments
- Determine minimal (wrt. the subtyping relation) correct types for the results.
- For all other types take the first solution.

Fig. 7 visualize this approach. This approach leads to suitable results. For the matrix benchmark the types

$$\text{Matrix mul(Vector<? extends Vector<? extends Integer>>) \{ \ldots \}}$$

are determined. The runtime of the matrix benchmark is lower that 20 seconds. For the matrix example indeed a principal type is determined. But there are some examples where not all elements of the principal intersection types are calculated.
Let us consider the next example

```java
public class Lambda {
    m () {
        var lam1 = (x) -> {
            return x;
        };  
        return lam1 . apply (1);
    }
}
```

The algorithm determine `Object` as the result type of `m`, although `Integer` would also be correct and it is more general. The reason is that first for `lam1` a type variables $\alpha$ is determined for the argument type and a type variables $\beta$ is determined for the result type. Furthermore holds $\alpha \preceq \beta$ and $\beta \preceq \gamma$, where $\gamma$ is the return type of $m$.

In `lam1.apply(1)` `Integer \preceq \alpha` is generated. As $\alpha$ is an argument type the maximum for $\alpha$ is determined. This means $\alpha$ is instantiated by `Object`. Therefore $\gamma$ is instantiated by `Object`, too.

## 5 Summary and outlook

We have presented an optimization of our type unification algorithm. The algorithm determines a cartesian product of all possible typings. We reduce the number elements of the cartesian product as we remove wrong elements as early as possible. These optimization do not restrict the solutions. As we do not get
satisfying results by these optimizations, we restrict the solutions which are determined.

We considered the definition of a principal type which we gave in a former article. It is the goal to infer a principal type such that all other types can be derived from the inferred type. The principal type is the maximum of the argument types and the minimum of the result type of a function, as argument types are contravariant and result types are covariant. At the moment the algorithm determine in the most cases a principal type. Only in some cases not all types are determined. With our optimizations we can determine the types of matrix multiplication is acceptable runtime.

In future work we plan to optimize the algorithm again. At the moment many sets of sub- and supertypes are calculated often and some part-unification are determined often, too. We look forward to to get speed-ups by using hashtables. Furthermore we will change the optimized algorithm such that the algorithm determines in all cases principal types.

References


