Optimization of the Java Type Unification
– Short paper –

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 does not reduce the solutions. The second kind restricts the solutions which means that we have to guarantee that the algorithm determines the relevant solutions.

We present two benchmarks: The scalar product and the matrix multiplication. The types of the matrix multiplication cannot be determined with the original algorithm with an ordinary computer because of memory and runtime lack. But with the optimization the types of the matrix multiplication can be determined with an ordinary computer in acceptable runtime.

1 Introduction

Java-TX (TX standing for **T**ype e**X**tended) is an extension of Java. The predominant new features are global type inference and real function types for lambda expressions [PS17, PZ22]. Global type inference enables to write Java programs without any type annotation. The type inference algorithm introduces the types during compile time. This allows us to omit the types without losing the static type property. The type inference problem can be reduced to a type unification problem [Plü15]. One of the major problems is the runtime of the type unification algorithm. The type unification problem is NP-hard [Sta18]. The algorithm iterates all elements of a cartesian product of all possible type annotations. This leads to an exponential running time in terms of the number of omitted types n and the height of the subtyping hierarchy k. The algorithm is given and proven as correct and complete in [Plü09]. In this paper we present four optimizations for the algorithm: *Evaluation of the Cartesian product in a backtracking approach, evaluation of only one inequation for each iteration*, *considering dependent mappings of error-pairs*, *considering only relevant solutions*.

In nearly all cases, theses optimizations reduces space and runtime enormously. We show this for two examples (scalar product and matrices multiplication) with the height of the subtyping hierarchy $k = 3, 4, 5$, respectively.

The paper is organized as follows: After a brief summary of the main part of the type unification algorithm and an example in Section 2, we present the different optimizations in Section 3. We present the improvement results for the two examples in Section 3. Finally, 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 [Plü09, SP18].
In the following let θ, θ', θ_i be 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 σ 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:
– $\theta \doteq \theta'$ means that θ and θ' should be unified such that $\sigma(\, \theta \,) = \sigma(\, \theta' \,)$.
– $\theta \lessdot \theta'$ means that θ and θ' should be unified such that $\sigma(\, \theta \,) \leq^* \sigma(\, \theta' \,)$.
– $\theta \lessdot_? \theta'$ depicts a subtype relation between type arguments. The type unification algorithm uses it internally to store intermediate results of the algorithm.
– We write A<$_?B$> for A<? extends B> and C<$^?D$> for C<? super D>.
– A set of equations is in *solved form*, if all pairs are either of the form $a_i \doteq \theta_i$ where a_i are pairwise different type variables and for all i, j holds true that a_i do not occur in θ_j or of the form $a \lessdot b$ and $a \lessdot_? b$ (consists only of type variables), respectively.

In the following we give a brief summary of the type unification algorithm. For details we refer to [Plü09].

Input: Set of equations $Eq = \{ \theta_1 \lessdot \theta'_1, \ldots, \theta_n \lessdot \theta'_n \}$

Precondition: For all $i \in 1 \leqslant i \leqslant n : \theta_i, \theta'_i$ are Java types.

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

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

Algorithm:

1. Repeated application of reduction rules. Afterwards at least one side of each pair must consist of a type variable, otherwise the pairs are unsolvable.

2. Generate every possible solution. For every θ' with θ' being a subtype of θ: Replace each pair $a \lessdot \theta$ and $a \lessdot_? \theta$ by a set of pairs $a \doteq \theta'$. For every θ' with θ' being a supertype of θ: Replace each pair $\theta \lessdot a$ and $\theta \lessdot_? a$ by a set of pairs $a \doteq \theta'$.

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 θ.

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 $a \doteq \theta$, $a \lessdot b$, or $a \lessdot_? b$, 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 `Number`, `Integer`, `Stack`, `Vector`, `AbstractList`, and `List`. It holds

$$\texttt{Integer} \leq^* \texttt{Number} \text{ and } \texttt{Stack<a>} \leq^* \texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}.$$

As a start configuration we use

$$\{\, (\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_?\texttt{Number>}),\ (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>}) \,\}.$$

In the first step the reduction rules are applied twice:

$$\{\, \texttt{a} \lessdot_? {}_?\texttt{Number},\ \texttt{Integer} \lessdot_? \texttt{a} \,\}$$

With the second step we receive in step three:

$$\begin{aligned}
\{&\{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\},\\
&\{\, \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\
&\{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\},\\
&\{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\}, \{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\},\\
&\{\, \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\},\\
&\{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\}, \{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\}\}
\end{aligned}$$

In the forth step the rule *subst* is applied:

$$\begin{aligned}
\{\,&\{\, \texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \underline{\{\, {}_?\texttt{Number} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\}},\\
&\{\, {}_?\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, {}^?\texttt{Integer} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\
&\{\, \texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, {}_?\texttt{Number} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\
&\{\, {}_?\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}, \{\, {}^?\texttt{Integer} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\
&\{\, \texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\}, \{\, {}_?\texttt{Number} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\
&\underline{\{\, {}_?\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\}}, \{\, {}^?\texttt{Integer} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\},\\
&\underline{\{\, \texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \,\}}, \{\, {}_?\texttt{Number} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \,\},\\
&\{\, {}_?\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \,\} \{\, {}^?\texttt{Integer} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \,\} \,\}
\end{aligned}$$

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:

$$\{\, \{\, a \mapsto {}_?\texttt{Number} \,\}, \{\, a \mapsto {}_?\texttt{Integer} \,\}, \{\, a \mapsto \texttt{Integer} \,\} \,\}.$$

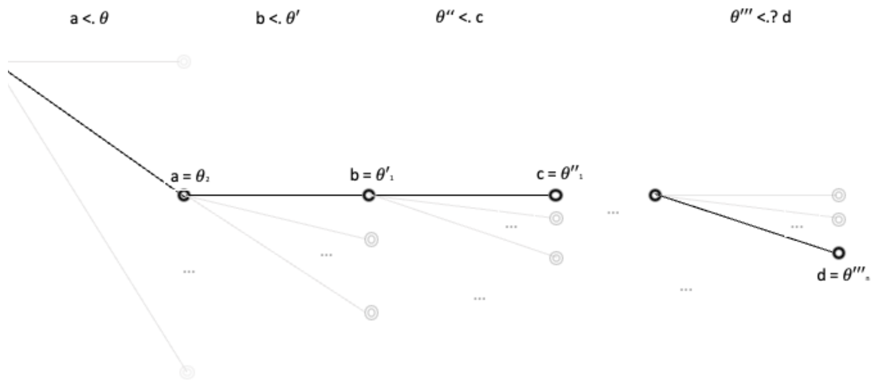

Figure 1: Backtracking strategy

3 Optimizations

3.1 Evaluation of the cartesian product in a backtracking strategy

In Fig. 1 the cartesian product of step 3 is visualized gray in the background. It is obvious that the number of equation sets $\{ a = \theta_i, b = \theta'_j, \ldots, \theta'''_k \}$ grows enormously. If we have n inequations $(a \lessdot \theta)$ and each inequation has m possible solutions $(a = \theta')$, then step 3 generates m^n equation sets.
We reduce the number of sets in storage space, as we build the cartesian product with a backtracking strategy (visualized in black in the foreground in Fig. 1). This means during the calculation of the unification algorithm only one set of equations, and $n \cdot m^1$ equations are in storage space. This is an enormous reduction of used storage space.

3.2 Lazy evaluation of the inequations

We change step 2 of the algorithm such that only one element $a \lessdot \theta$ is evaluated to a set $\{ a \doteq \theta_1, \ldots, a \doteq \theta_n \}$. All other pairs remain untouched. They will be evaluated not till a later iteration. This means the cartesian product in step 3 has only m elements (number of solutions of $a \lessdot \theta$) and $n + (m-1)$ equations/inequation in storage space. It is visualized in Fig. 2.
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. This means that for this element for the remaining pairs $(b \lessdot \theta'[a \to \theta_j], \theta''[a \to \theta_j] \lessdot c, \ldots, \theta'''[a \to \theta_j] \lessdot_? d)$ no cartesian product is built.

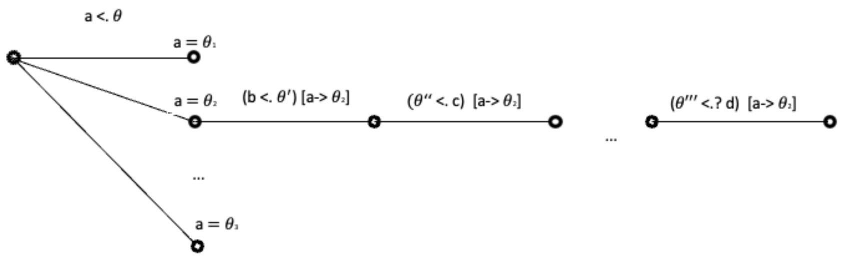

Figure 2: Evaluation of only one inequation

3.3 Considering dependent substitutions of error-pairs

Each pair of the form $\theta \lessdot \theta'$ stores a history of substitutions, which lead to its current state (e.g. $(\texttt{Integer} \lessdot \texttt{String})$ follows from $(a \lessdot b)$ with $[a \mapsto \texttt{Integer}]$ and $[b \mapsto \texttt{String}]$). If such a pair is the cause of an error in step 1, we now have to backtrack until at least one of the substitutions in its history is changed (e.g. in the above example we to backtrack until either a or b is changed).
This method allows the unification algorithm to safely backtrack multiple steps without excluding a solution. The optimization is visualized in Fig. 3.

The above optimizations reduce only the use of storage space and runtime, but preserve the number of solutions. The reason is that only fail-cases are omitted. All cases which leads to a solution are considered.

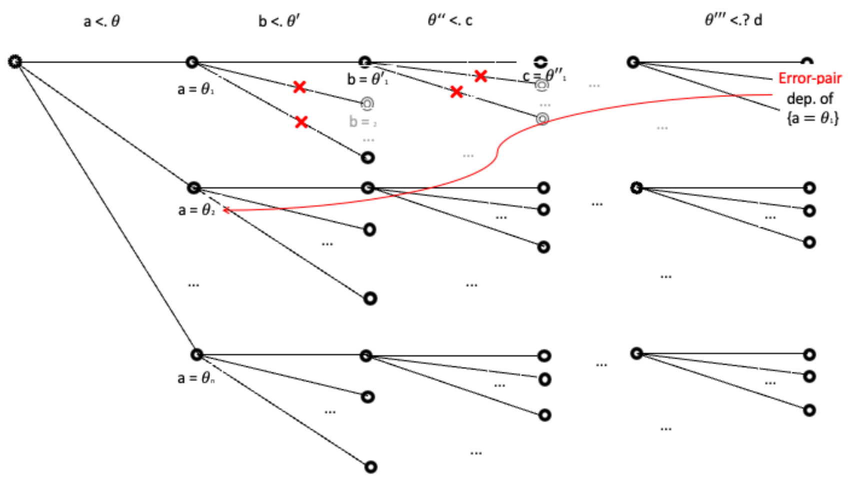

Figure 3: Dependent substitutions

The next optimization reduces the number of solutions, such that only solutions are determined which leads to principal types.

3.4 Considering relevant solutions

The argument types of methods 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.

This approach does not erase results that are elements of the principal type of Java-TX–programs [PZ22]. This approach leads to suitable results.

4 Results of the examples

We give results for the optimizations of two examples: *scalar multiplication* and *matrix multiplication*. The costs of the Java type unification algorithm are dependent of two parameters: the number of inequations n and the height of the subtyping hierarchy k. We give for both examples results of the subtyping height $k = 2, 3, 4$.
We consider the number of iterations for the original type unification algorithm (without optimizations) and for each optimization. It is obvious that the costs are reduced enormously if all three optimization are used (e.g. Scalar example: from 14680064 to 133; Matrix example: from $7,26E + 34$ to 10877).

	Scalar			Matrix		
Height of Subtyping Hierachy	k=2	k=3	k=4	k=2	k=3	k=4
Number of Constraints	n=18	n=18	n=18	n=33	n=32	n=34
No optimization						
Number of Iterations	14680064	74317824	469762048	2,03E+20	6,19E+24	7,26E+34
Lazy evaluation of the inequations						
Number of Results	6	10	14	240	3400	16576
Number of Iterations	2244	4666	7638	3264022	77591882	(*)
Dependent substitutions						
Number of Results	6	10	14	240	3400	16576
Number of Iterations	827	1606	3075	277333	11381413	66196815
Co-/Contravariance						
Number of Results	1	1	1	1	1	1
Number of Iterations	133	255	398	727	14555	10877

(*) could not be determined until WLP 23, will be handed later.

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 optimizations do not restrict the solutions. The fourth optimization restricts the solutions to the elements of the principal type.
In future work we plan to optimize the algorithm again. At the moment many sets of sub- and supertypes are calculated several times and some part-unification are determined several times, too. We look forward to to get speed-ups by using hashtables.
Furthermore, many different backtracking iterations can be done parallel.

References

[Plü09] Martin Plümicke. Java type unification with wildcards. *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, 5437:223–240, 2009.

[Plü15] Martin Plümicke. More type inference in Java 8. *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, 8974:248–256, 2015.

[PS17] Martin Plümicke and Andreas Stadelmeier. Introducing Scala-like function types into Java-TX. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 23–34, New York, NY, USA, 2017. ACM.

[PZ22] Martin Plümicke and Etienne Zink. Java-TX: The language. INSIGHTS – Schriftenreihe der Fakultät Technik 01/2022, DHBW Stuttgart, 2022.

[SP18] Florian Steurer and Martin Plümicke. Erweiterung und Neuimplementierung der Java Typunifikation. In Jens Knoop, Martin Steffen, and Baltasar Trancón y Widemann, editors, *Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts*, number 482 in Research Report, pages 134–149. Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO, 2018. ISBN 978-82-7368-447-9, (in german).

[Sta18] Andreas Stadelmeier. Java-Typinferenz mithilfe von logikbasierten Formalen Methoden. Master's thesis, Universität Tübingen, 2018. (in german).