Global Type Inference for Java using ASP

Andreas Stadelmeier[1], Martin Plümicke[1]

[1]Duale Hochschule Baden-Württemberg, Stuttgart, Germany

Abstract

Global type inference for Java is able to compute correct types for an input program which has no type annotations at all, but turns out to be NP-hard. Former implementations of the algorithm therefore struggled with bad runtime performance. In this original paper, we translate the type inference algorithm for Featherweight Generic Java to an Answer Set Program. Answer Set Programming (ASP) promises to solve complex computational search problems as long as they are finite domain. This paper shows that it is possible to implement global type inference for Featherweight Generic Java (FGJ) as an ASP program. Afterwards we compared the ASP implementation with our own implementation in Java showing promising results. Another advantage is that the algorithm's specification can be mapped directly to ASP code, which reduces the likelihood of errors in the implementation.

Full Version

A version with all proofs and a reference implementation is available at: doi:10.5281/zenodo.16836767.

Keywords

type inference, java, programming languages, answer set programming

1. Global Type Inference

Java is a strictly typed programming language. The current versions come with a local type inference feature that allows the programmer to omit type annotations in some places like argument types of lambda expressions for example. But methods still have to be fully typed including argument and return types. We work on a global type inference algorithm for Java which is able to compute correct types for a Java program with no type annotations at all.

```
class C1 {
    C1 self(){
        return this;
    }
}
class C2 {
    C2 self(){
        return this;
    }
}
class Example {
    untypedMethod(var){
        return var.self();    //  ⟹ {var ⩽ C1, ret ≐ C1} || {var ⩽ C2, ret ≐ C2}
    }
}
```

Figure 1: Java program where a method is missing type annotations

A possible input for our algorithm is shown in Figure 1. Here the method `untypedMethod` is missing its argument type for `var` and its return type. Global type inference works in two steps. First we generate a set of subtype constraints, which are later unified resulting in a type solution consisting of type unifiers σ. Every missing type is replaced by a type placeholder. In this example the type

Joint Proceedings of the Workshops and Doctoral Consortium of the 41st International Conference on Logic Programming, September 9–13, 2025, Rende, Italy

placeholder var is a placeholder for the type of the `var` variable. The ret placeholder represents the return type of the `untypedMethod` method. Also shown as a comment behind the respective method call. There are two types of type constraints:

- A subtype constraint such as $var \lessdot \texttt{C1}$ indicates that the unification algorithm must determine a type substitution for var that is a subtype of `C1`. One possible solution would be $\sigma(var) = \texttt{C1}$.
- A equals constraint $var \doteq \texttt{C1}$, which results in $\sigma(ret) = \texttt{C1}$.

Our type inference algorithms for Java are described in a formal manner in [1],[2]. The first prototype implementation put the focus on a correct implementation rather than fast execution times. Depending on the input, it currently takes up to several minutes or even days to compute all or even one possible type solution. To make them usable in practice, we now focus on decreasing the runtime to a feasible level.

2. Motivation

The nature of the global type inference algorithm causes the search space of the unification algorithm to increase exponentially with every ambiguous method call. Java allows overloaded methods, causing our type inference algorithm to create so-called Or-Constraints. This happens if multiple classes implement a method with the same name and the same amount of parameters. Given the input program in Figure 1, we do not know, which method `self` is meant for the method call `var.self()`, because there is no type annotation for `var`.

An Or-Constraint like $\{var \lessdot \texttt{C1}, ret \doteq \texttt{C1}\} \,||\, \{var \lessdot \texttt{C2}, ret \doteq \texttt{C2}\}$ means that either the constraint set $\{var \lessdot \texttt{C1}, ret \doteq \texttt{C1}\}$ or $\{var \lessdot \texttt{C2}, ret \doteq \texttt{C2}\}$ has to be satisfied. If we set the type of `var` to `C1`, then the return type of the method will be `C1` as well. If we set it to `C2`, then the return type will also be `C2`. Since two distinct possibilities exist, an Or-Constraint with both variants is required.

```
untypedMethod(var){
    return var.self().self().self().self();
}
```

Figure 2: Java program with stacked ambiguous method calls

If we chain multiple overloaded method calls together, we end up with multiple Or-Constraints. The type unification algorithm **Unify** only sees the supplied constraints and has to potentially try all combinations of them. This is proven in [2] and is the reason why type inference for Featherweight Generic Java is NP-hard.

Let's have a look at the following alternation of our example shown in Figure 2. Now there are four chained method calls leading to four Or-Constraints:

$$\{var \lessdot \texttt{C1}, r1 \doteq \texttt{C1}\} \,||\, \{var \lessdot \texttt{C2}, r1 \doteq \texttt{C2}\}$$
$$\{r1 \lessdot \texttt{C1}, r2 \doteq \texttt{C1}\} \,||\, \{r1 \lessdot \texttt{C2}, r2 \doteq \texttt{C2}\}$$
$$\{r2 \lessdot \texttt{C1}, r3 \doteq \texttt{C1}\} \,||\, \{r2 \lessdot \texttt{C2}, r3 \doteq \texttt{C2}\}$$
$$\{r3 \lessdot \texttt{C1}, ret \doteq \texttt{C1}\} \,||\, \{r3 \lessdot \texttt{C2}, ret \doteq \texttt{C2}\}$$

The placeholder $r1$ stands for the return type of the first call to `self` and $r2$ for the return type of the second call and so on. It is obvious that this constraint set only has two solutions. The variable `var` and the return type of the method as well as all intermediate placeholders $r1 - r3$ get either the type `C1` or `C2`. A first prototype implementation of the **Unify** algorithm [3] simply created the cartesian product of all Or-Constraints, 16 possibilities in this example, and iteratively tried all of them. This

leads to a exponential runtime increase with every added overloaded method call. Even though the current algorithm is equipped with various optimizations as presented in [4, 5] there are still runtime issues when dealing with many Or-Constraints.

Our global type inference algorithm should construct type solutions in real time. Then, it can effectively be used as a Java compiler which can deal with large inputs of untyped Java code. Another use case scenario is as an editor plugin which helps a Java programmer by enabling him to write untyped Java code and letting our tool insert the missing type statements. For both applications, a short execution time is vital.

Answer Set programming promises to solve complex search problems in a highly optimized way. The idea in this paper is to implement the algorithm presented in [2] as an ASP program and see how well it handles our type unification problem.

3. Existing Type Unification Algorithm

The original global type inference algorithm used by the Java-TX[1] project proceeds in two phases: (1) generate a set of subtype constraints, and (2) unify them to obtain a substitution σ mapping type placeholders to types.

This type unification for Java has been studied in several works. Plümicke [6] presented the basic ideas and an initial algorithm for GJ [7], which assumed invariant type parameters without wildcards. Later extensions added wildcards [1] and implemented a restricted form [8, 3] where bounds were either `Object` or another type parameter. The work in [2] generalized this so that type parameters can have arbitrary bounds.

Type unification is the process of finding substitutions for type variables so that two given type terms become compatible under a specified compatibility relation. Formally, given two type terms θ_1 and θ_2, the type unification problem is to find a substitution σ mapping type placeholders to type terms such that:

$$\sigma(\theta_1) \ \leq^* \ \sigma(\theta_2).$$

where $\leq^*$ is the transitive closure of the Java subtype relation.

4. ASP Implementation

Our ASP implementation realizes the algorithm of [2], meaning that Featherweight Generic Java serves as our basis, with a type system featuring invariant generics, no wildcards, and no F-bounded type parameters. The implementation replaces the unification step of the type inference algorithm. So the input is a set of constraints c and the output is a set of unifiers σ.

Answer Set Programming has a declarative way of formulating search problems. In ASP you encode the problem as a set of logical rules and facts and let an ASP solver (like `clingo`[2]) compute one or more stable models (also called answer sets). A *stable model* is a self-justifying, minimal[3], and internally consistent interpretation of the program [9].

Translated to our algorithm, this means that the input set of constraints C are the facts and the implication rules shown in Figure 5 are the logic rules. Thus, a stable model must be a constraint set that satisfies all the rules, contains the input set of constraints, and is minimal (see definition 1). Furthermore, the rules of our algorithm Ω are formulated in a way that each input set of constraints C either ends in a *fail* condition — meaning there is no possible solution — or leads to a stable model C_S that contains correct unifiers of the form $\sigma(a) = \texttt{G}$ for every type placeholder a in the input constraints. We prove those statements in chapter 5.

[1] https://www.dhbw-stuttgart.de/horb/forschung-transfer/forschungsschwerpunkte/typsysteme-fuer-objektorientierte-programmiersprachen/

[2] https://potassco.org/clingo/

[3] all rules are satisfied and no proper subset of it satisfies the same conditions

c	::=	$\texttt{T} \lessdot \texttt{T} \mid \texttt{T} \doteq \texttt{T}$	Constraint
$\texttt{T}, \texttt{S}$	::=	$a \mid \texttt{N}$	Type placeholder or Type
$\texttt{N}$	::=	$\texttt{C<}\overline{\texttt{P}}\texttt{>}$	Class Type containing type placeholders
$\texttt{G}$	::=	$\texttt{C<}\overline{\texttt{G}}\texttt{>}$	Class Type not containing type placeholders
$\texttt{P}$	::=	$a \mid \texttt{G}$	Well-Formed Parameter

Figure 3: Syntax of types and constraints

G are types that contain no type placeholders. A type P is either a type placeholder or a type not containing type placeholders at all. The reason for those definitions is that they restrict type placeholder to only appear in the uppermost level of a type parameter list.

Definition 1 (Stable Model)**.** *A stable model (or answer set) of an input set C is a set of constraints C_S with $C \subseteq C_S$ that:*

- *Satisfies all the rules*
- *Is complete: If the premise of any rule in Ω is satisfied by elements of C_S, then the conclusion of that rule must also be included in C_S.*
- *Is minimal (no unnecessary atoms)*

All `extends` relations in the input program have to satisfy the WF-Class rule. This restriction is necessary because our algorithm only supports type placeholders in the uppermost layer of a type parameter list. WF-Class rule ensures that supertypes generated by the Super rule are well-formed. For example the class definition **`class`** `Matrix<A>` **`extends`** `List<List<A>>` is not supported by our algorithm Ω. If an input C contains a constraint $\texttt{Matrix<}a\texttt{>} \lessdot b$, then the Super rule would require for a $b \doteq \texttt{List<List<}a\texttt{>>}$ constraint where the type $\texttt{List<List<}a\texttt{>>}$ is **not well-formed** according to the rules in Figure 4 and therefore not supported by our algorithm.

$$\frac{\overline{\texttt{S}_1, \texttt{S}_2, \texttt{T}_1, \texttt{T}_2 \text{ ok}} \qquad C = \{\overline{\texttt{S}_1 \doteq \texttt{S}_2}, \overline{\texttt{T}_1 \lessdot \texttt{T}_2}\}}{C \text{ ok}}\ \text{WF-Constraints} \qquad \frac{}{\texttt{C<}\overline{\texttt{P}}\texttt{> ok}}\ \text{WF-Type} \qquad \frac{}{a \text{ ok}}\ \text{WF-Var} \qquad \frac{\text{for any } \overline{\texttt{a}}: \qquad ([\overline{\texttt{X}}/\overline{\texttt{a}}]\texttt{N}) \text{ ok}}{\texttt{class C<}\overline{\texttt{X}}\texttt{>} \triangleleft \texttt{N ok}}\ \text{WF-Class}$$

Figure 4: Well-formedness condition

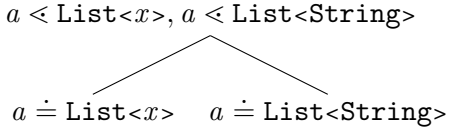
4.1. Unification Algorithm

All the implication rules depicted in this chapter can be translated to ASP statements as described in chapter 6. The whole algorithm is depicted in Figure 5

We need the Solution-Requirement rule to force at least one type solution for every type placeholder in the input constraints (see example 4.1).

Example 4.1: Solution-Requirement

$a \lessdot \texttt{List<}x\texttt{>}, a \lessdot \texttt{List<String>}$. We want our program to create a unifier for a which requires at least one constraint of the form $a \doteq \ldots$. Solution-Requirement forces the ASP solver to choose one of the following possibilities:



In this case, it does not matter which one gets chosen as they will both lead to the same result. Let's take the $a \doteq \texttt{List<}x\texttt{>}$ for example. Then we end up with a constraint $\texttt{List<}x\texttt{>} \lessdot \texttt{List<String>}$

Subtyping:

S-Refl
$\mathtt{T} <: \mathtt{T}$

S-Trans
$$\frac{\mathtt{T}_1 <: \mathtt{T}_2 \qquad \mathtt{T}_2 <: \mathtt{T}_3}{\mathtt{T}_1 <: \mathtt{T}_3}$$

S-Class
$$\frac{\texttt{class C<}\overline{\mathtt{X}}\texttt{>} \triangleleft \mathtt{N}}{\texttt{C<}\overline{\mathtt{T}}\texttt{>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}}$$

Unify:

Subst-L
$$\frac{a \doteq \mathtt{T}_1 \qquad a \lessdot \mathtt{T}_2}{\mathtt{T}_1 \lessdot \mathtt{T}_2}$$

Subst-R
$$\frac{a \doteq \mathtt{T}_1 \qquad \mathtt{T}_2 \lessdot a}{\mathtt{T}_2 \lessdot \mathtt{T}_1}$$

Subst-Equal
$$\frac{a \doteq \mathtt{T}_1 \qquad a \doteq \mathtt{T}_2}{\mathtt{T}_1 \doteq \mathtt{T}_2}$$

Swap
$$\frac{\mathtt{T}_1 \doteq \mathtt{T}_2}{\mathtt{T}_2 \doteq \mathtt{T}_1}$$

Subst-Param
$$\frac{a \doteq \mathtt{G} \qquad \mathtt{T} \doteq \texttt{C<}\mathtt{P}_1 \ldots, a, \ldots \mathtt{P}_\mathtt{n}\texttt{>}}{\mathtt{T} \doteq \texttt{C<}\mathtt{P}_1, \ldots \mathtt{G}, \ldots \mathtt{P}_\mathtt{n}\texttt{>}}$$

Subst-Param-Right
$$\frac{a \doteq \mathtt{G} \qquad b \lessdot \texttt{C<}\mathtt{T}_1 \ldots, a, \ldots \mathtt{T}_\mathtt{n}\texttt{>}}{b \lessdot \texttt{C<}\mathtt{P}_1, \ldots \mathtt{G}, \ldots \mathtt{P}_\mathtt{n}\texttt{>}}$$

Subst-Param-Left
$$\frac{a \doteq \mathtt{G} \qquad \texttt{C<}\mathtt{P}_1 \ldots, a, \ldots \mathtt{P}_\mathtt{n}\texttt{>} \lessdot b}{\texttt{C<}\mathtt{P}_1, \ldots \mathtt{G}, \ldots \mathtt{P}_\mathtt{n}\texttt{>} \lessdot b}$$

S-Object
$a \lessdot \texttt{Object}$

Adapt
$$\frac{\mathtt{N} \lessdot \texttt{C<}\mathtt{P}_1 \ldots \mathtt{P}_\mathtt{n}\texttt{>} \qquad \mathtt{N} <: \texttt{C<}\mathtt{P'}_1 \ldots \mathtt{P'}_\mathtt{n}\texttt{>}}{\mathtt{P}_1 \doteq \mathtt{P'}_1 \ \ldots \ \mathtt{P}_n \doteq \mathtt{P'}_n}$$

Reduce
$$\frac{\texttt{C<}\mathtt{P}_1 \ldots \mathtt{P}_\mathtt{n}\texttt{>} \doteq \texttt{C<}\mathtt{P'}_1 \ldots \mathtt{P'}_\mathtt{n}\texttt{>}}{\mathtt{P}_i \doteq \mathtt{P'}_i}$$

Super
$$\frac{\mathtt{N} \lessdot a \qquad \mathtt{N} <: \mathtt{N}_1, \ldots, \mathtt{N} <: \mathtt{N}_n \qquad \mathtt{N}_1, \ldots, \mathtt{N}_n \text{ ok}}{\text{for one } m \in \{1, \ldots, n\} : \quad a \doteq \mathtt{N}_m}$$

Solution:

Solution
$$\frac{a \doteq \mathtt{G}}{\sigma(a) = \mathtt{G}}$$

Solution-Gen
$$\frac{a \lessdot \mathtt{N}}{a \doteq \mathtt{N} \text{ or } a \lessdot \mathtt{N}}$$

Solution-Requirement
$$\frac{a \lessdot \mathtt{N}_1, \ldots, a \lessdot \mathtt{N}_n}{\text{there must exist at least one : } \quad a \doteq \mathtt{N}}$$

Fail Conditions:

Fail-Subtype
$$\frac{\texttt{C<}\ldots\texttt{>} \lessdot \texttt{D<}\ldots\texttt{>} \qquad \texttt{C not extends D}}{\emptyset}$$

Fail-Equals
$$\frac{\texttt{C<}\ldots\texttt{>} \doteq \texttt{D<}\ldots\texttt{>} \qquad \texttt{C} \neq \texttt{D}}{\emptyset}$$

Anti Circle Conditions:

Build
$$\frac{a \doteq \texttt{C<}\ldots, b, \ldots\texttt{>}}{a \to b}$$

Connect
$$\frac{a \to b \qquad b \to c}{a \to c}$$

Fail-Circle
$$\frac{a \to a}{\textit{fail}}$$

Figure 5: Unify Algorithm Ω

due to Subst-L. Adapt creates a `List<`x`>` $\doteq$ `List<String>` constraint and from there we get $x \doteq$ `String` by Reduce. Together with $a \doteq$ `List<`x`>` this leads to $a \doteq$ `List<String>` (by Subst-Param), which is a correct solution for the given constraints.

4.2. Differences to the Featherweight Generic Java Type Unification

The ASP unify algorithm Ω presented here is a toned down version of the algorithm presented in [2]. The main idea of our ASP prototype is to measure runtime performance especially in relation to Or-Constraints. When it comes to those core concepts, the ASP algorithm has the same capabilities as the Type Inference algorithm for Generic Featherweight Java. Though unlike the original algorithm, we do not create Generic Type Variables. Instead, every type variable is directly assigned a concrete type. The prototype is not complete in the sense that it cannot find solutions that require polymorphic methods. However, it still finds all solutions that are possible without involving polymorphism.

The Scala prototype of the **Unify** algorithm, provided with [2], solves the problem through successive constraint-set transformations. For example if a constraint $a \doteq \texttt{N}$ is found, all occurences of a are replaced by `N` via substitution.

Our ASP implementation on the other hand does not use any state and therefore cannot change an existing constraint set. All possible substitutions are laid out by the grounding process. Afterwards all incorrect solutions are ruled out. This allows for a more declarative programming style, where we specify the type rules and let ASP Figure out a solution for itself.

5. Proofs

Given a set of constraints C and our algorithm Ω as input, the ASP solver attempts to find the smallest possible set of constraints C_S. This set contains the input constraints C and satisfies all implication rules shown in Figure 5. The constraint set C_S is also called a stable model (see Definition 1). We proved two key properties of C_S: soundness and completeness.

Soundness All produced typings respect the original constraints; the algorithm never returns an incorrect assignment.

Completeness If a solution exists, the procedure finds one, as long as the input set does not contain circles (Def. 2)

See the Full Paper for a detailed version of the proofs.

Definition 2 (Circular Constraint)**.** *A set of constraints* $\overline{a \doteq \texttt{N}}$ *where* $\forall \texttt{N} \in \overline{\texttt{N}} : fv(\texttt{N}) \neq \emptyset$ *and* $\overline{a}$ *contains all type placeholders used in* $\overline{\texttt{N}}$ *(meaning* $\overline{a} = fv(\overline{\texttt{N}})$*) is called a circle.*

6. ASP Encoding

Note: See the Full Paper for the complete program including an example input

ASP statements consist of a head and a body separated by a implication operator ":-": `head :- body`. This statement is interpreted as an implication rule. If the premises in the body are satisfied, the facts stated in the head are deducted. The implication rules defined in Figure 5 are formulated in a way that can be translated to an ASP program. This chapter explains how to code our algorithm Ω in ASP. At first, we give a ASP representation for each syntax element used in Figure 3:

Syntax element	ASP representation
$\ldots \lessdot \ldots$	`subcons(. . . , . . . )`
$\ldots \doteq \ldots$	`equalcons(. . . , . . . )`
a	`tph("a")`
`C<...>`	`type("C", params(. . . ))`
`Object`	`type("Object", null)`

For further clarification we will present some examples:

$$\texttt{C<Object>} \implies \texttt{type("C", params(Object,null))}$$
$$a \lessdot b \implies \texttt{subcons(tph("a"), tph("b"))}$$
$$\texttt{List<}a\texttt{>} \implies \texttt{type("List", params(tph("a")))}$$

In addition to the constraint set C, our algorithm requires the class inheritance information of all classes that appear in the constraints. This information is essential for evaluating subtype relationships and is encoded in the form of ASP rules. Specifically, for every class declaration in the Java type system of the form `class C<X> extends D<X>` we generate a corresponding ASP rule:

Listing 1: Extends Relation as ASP rule

```
subtype(type("C", params(X)), type("D", params(X)))
    :- subtype(type("C", params(X))).
```

These rules capture the inheritance hierarchy and serve as a static input to the algorithm, just like the constraint set C. Thus, the complete input to our algorithm consists of:

- The set of type constraints C
- The set of ASP rules encoding all relevant `extends` relations

This inheritance information is especially relevant for the Super and Adapt rules, both of which contain premises of the form `N` $<:$ `D<...>`. For example, when the input set contains a constraint like $a \lessdot$ `N`, the algorithm must determine the supertypes of `N` in order to apply the Super rule. This is exactly where the encoded inheritance rules come into play. The following example 6.1 illustrates this process in practice.

Example 6.1: Creating Supertypes

Assume two classes `Vector` and `List` are declared such that

$$\texttt{class Vector<X> extends List<X>},$$

and let the constraint set be

$$C = \{\, \texttt{Vector<String>} \lessdot a \,\}.$$

We need a helper rule in our ASP program that looks like this:

```
subtype(type(A, P)) :- subcons(type(A, P), _).
```

This deduces the fact:

```
subtype(type("Vector", params(type("String", null)))).
```

from the input constraint

```
subcons(type("Vector", params(type("String", null))),
        tph("a")).
```

Now, the rule from Listing 1 generates the corresponding supertype:

```
subtype(type("Vector", params(type("String", null))),
        type("List", params(type("String", null)))).
```

This reflects the subtyping relationship

$$\texttt{Vector<String>} <: \texttt{List<String>}.$$

With this information, the Super rule (see Listing 2) can select a valid supertype for a. In this case, it yields the ASP fact:

```
equalcons(tph("a"),
          type("List", params(type("String", null)))).
```

which corresponds to the constraint

$$a \doteq \texttt{List<String>}.$$

Listing 2: Super-Rule

```
{equalcons(tph(A), type(D,DP)): subtype(type(C,CP), type(D,DP))}=1
    :- subcons(type(C, CP), tph(A)).
```

The vital part is the generation of the Cartesian product of the Or-Constraints.
`subcons(A,B); orCons(C,D) :- orCons(subcons(A,B), subcons(C,D)).`
The operator ";" tells ASP to consider either the left or the right side. By supplying multiple Or-Constraints this way, the ASP interpreter will consider all combinations, the Cartesian product of the Or-Constraints.

The rules implying a failure $\emptyset$ are translated by leaving the head of the ASP statement empty. If the body of such rules is satisfied, the algorithm considers the deduction as incorrect. See an example in Listing 3 that shows an ASP encoding of the Fail-Equals rule.

Listing 3: Fail-Equals rule in ASP

```
:- equalcons(type(A, _), type(B, _)), A != B.
```

7. Benchmark

To evaluate the performance and practical applicability of our ASP implementation of the Unify algorithm for FGJ, we conducted two benchmarks:

1. **Massive Or-Constraints**: Tests Unify's ability to create and handle Cartesian products of big Or-Constraints.
2. **Matrix Example**: Typical benchmark used to test Java-TX type inference, where the Unify algorithm has to consider many possible variations of supertypes of the class `Matrix`.

Even though import statements in standard Java are purely syntactic and used only to avoid writing fully qualified class names, the current Java-TX implementation uses them to limit the search space for missing type annotations. The algorithm only considers "imported" types. This makes Java-TX type inference unsuitable for typical Java development, where users expect to reuse classes without needing to import them in advance. In contrast, our ASP-based Unify algorithm aims to lift this restriction by considering the entire standard Java library as potential type candidates. To demonstrate this, we introduce the following benchmark program shown in Listing 4:

Listing 4: Benchmark

```
public class ThreeAdds {
    void addTest(a, b, c, d){
        a.add(b);
        b.add(c);
        c.add(d);
        d.add(a);
    }
}
```

The `addTest` declaration in Listing 4 is missing type annotations for its parameters `a`, `b`, `c`, `d`. We want our ASP implementation of the Unify algorithm to calculate all possible typings for the missing type annotations. Therefore, we have to first generate a set of constraints representing the input problem. The method `addTest` has three calls `a.add(...)`, which leads to three large Or-Constraints. We use the Open-JDK standard library of the version 21, which contains nearly twenty thousand classes, including inner classes. They have 200 different `add` methods all together. This leads to four Or-Constraints with 200 possibilities, meaning a Cartesian product with 1.6 billion elements. `clingo` finds a solution in under half a second.

Listing 5: Solution for Benchmark 4

```
public class ThreeAdds {
  void addTest(javax.naming.directory.Attribute a,
               java.util.concurrent.DelayQueue b,
```

```
                java.beans.beancontext.BeanContextSupport c,
                javax.naming.directory.BasicAttribute d){
    a.add(b);
    b.add(c);
    c.add(d);
    d.add(a);
  }
}
```

The program in Listing 4 admits an unexpectedly large number of valid type solutions, as many `add` methods in the Java standard library accept `Object` as an argument. One such solution is shown in Listing 5 and compiles under OpenJDK 17. We assume `clingo` spends most time in grounding, since finding a single valid typing is trivial once all possibilities are enumerated.

The second benchmark, the `Matrix` example shown in Listing 6, is also used by the Java-TX project as a functionality demo and stress test. The grayed-out types in the method header ❶ are to be inferred by our algorithm, including those omitted via Java's `var` keyword [10]. Multiple valid solutions exist beyond `Matrix mul(Matrix m)`, since supertypes of `Matrix` and even wildcard types are permissible. For instance, `Vector<Vector<?extendsInteger>>` is a valid parameter type for `m`. Which makes this benchmark especially hard for the original **Unify** algorithm, because adding wildcards enables a lot of new typing possibilities and increases the search space. Our prototype **Unify** implementation does not yet support wildcards, making direct comparison to the original Java-TX implementation (in Java) imperfect. Nevertheless, while Java-TX often needs minutes to enumerate all solutions, our ASP-based approach finds them in milliseconds.

Listing 6: Matrix Benchmark

```
class Matrix extends Vector<Vector<Integer>> {
❶  Matrix mul(Matrix 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; } }
```

8. Outcome, Conclusion, and Future Work

ASP handles Or-Constraints efficiently. We evaluated our ASP-based unification algorithm on an extended version of the constraints from Figure 2, increasing the complexity by adding stacked method calls. Even with ten such calls, `clingo` succeeded in under 50 ms, whereas the current Java implementation[4] required several seconds for the same input.

[4] https://gitea.hb.dhbw-stuttgart.de/JavaTX/JavaCompilerCore

It is important to note that this benchmark avoids wildcard types to ensure a fair comparison, as the Java implementation supports them, whereas the ASP version does not. In this first study on applying ASP to Java type inference, wildcard types were deliberately omitted. Including wildcards makes subtype checking in Java Turing complete [11], which can pose challenges for `clingo`'s grounding process [12]. Future work could investigate grounding-free ASP solvers [13] to overcome these challenges and enable full wildcard support.

Declaration on Generative AI

In preparation of this work, the authors used AI-based tools for spell and grammar checking. All ideas and results presented are our own and we take full responsibility for the accuracy and integrity of the publication.

References

[1] M. Plümicke, Java type unification with wildcards, in: D. Seipel, M. Hanus, A. Wolf (Eds.), Applications of Declarative Programming and Knowledge Management, 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 Computer Science*, Springer, 2007, pp. 223–240. doi:10.1007/978-3-642-00675-3_15.

[2] A. Stadelmeier, M. Plümicke, P. Thiemann, Global type inference for featherweight generic java, CoRR abs/2205.08768 (2022). URL: https://doi.org/10.48550/arXiv.2205.08768. doi:10.48550/arXiv.2205.08768. arXiv:2205.08768.

[3] F. Steurer, M. Plümicke, Erweiterung und Neuimplementierung der Java Typunifikation, in: J. Knoop, M. Steffen, B. T. y Widemann (Eds.), Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts, number 482 in Research Report, Faculty of Mathematics and Natural Sciences, University of Oslo, 2018, pp. 134–149. ISBN 978-82-7368-447-9, (in German).

[4] M. Plümicke, Optimization of the Java type unification, in: J. Knoop, M. Steffen, B. T. y Widemann (Eds.), Proceedings of the 35th Annual Meeting of the GI Working Group Programming Languages and Computing Concepts, number 482 in Research Report, Faculty of Mathematics and Natural Sciences, UNIVERSITY OF OSLO, 2018, pp. 3–12. ISBN 978-82-7368-447-9.

[5] M. Plümicke, Optimization of the Java Type Unification, in: S. Schwarz, M. Wenzel (Eds.), Proceedings of the 37th Workshop on (Constraint) Logic Programming (WLP 2023), 2023. URL: https://dbs.informatik.uni-halle.de/wlp2023/WLP2023_Pl%C3%BCmicke_Optimization%20of%20the%20Java%20Type%20Unification.pdf.

[6] M. Plümicke, Type Unification in Generic–Java, in: M. Kohlhase (Ed.), Proceedings of 18th International Workshop on Unification (UNIF'04), Cork, 2004.

[7] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, GJ: Extending the Java Programming Language with type parameters, 1998.

[8] F. Steurer, Implementierung eines Typunifikationsalgorithmus für Java 8, DHBW Stuttgart, Campus Horb, Studienarbeit, 2016. (in German).

[9] P. Ferraris, V. Lifschitz, Mathematical foundations of answer set programming., in: We will show them!(1), 2005, pp. 615–664.

[10] B. Goetz, Jep 286: Local-variable type inference, https://openjdk.org/jeps/286, 2018. Accessed: 2025-07-26.

[11] R. Grigore, Java generics are turing complete, ACM SIGPLAN Notices 52 (2017) 73–85.

[12] B. Kaufmann, N. Leone, S. Perri, T. Schaub, Grounding and solving in answer set programming, AI magazine 37 (2016) 25–32.

[13] J. Arias Herrero, M. Carro Liñares, Z. Chen, G. Gupta, Constraint answer set programming without grounding and its applications (2019).