Type inference for Generic Featherweight Java

ANDREAS STADELMEIER, DHBW Stuttgart, Campus Horb, Germany

Type inference for Generic Featherweight Java

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: typeinference, java, compiler

ACM Reference Format:
Andreas Stadelmeier. 2018. Type inference for Generic Featherweight Java. *J. ACM* 37, 4, Article 111 (August 2018), 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

2 PRELIMINARIES

2.1 Input assumptions

The input is a GFJ program lacking the type assignments for method parameters and method return types.

The Typeless Generic Featherweight Java (TGFJ) syntax is different in that from normal Generic Featherweight Java (GFJ) that it is possible to omit the type annotations for methods.We declare the syntax for TGFJ as follows:

$$
\begin{aligned}
T &::= X \mid N \\
N &::= \texttt{C<}\overline{\texttt{T}}\texttt{>} \\
L &::= \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \ \triangleleft\ N\{\overline{T}\ \overline{f};\ K\ \overline{M}\} \\
K &::= C(\overline{T}\ \overline{f})\{\texttt{super}(\overline{f});\ \texttt{this}.\overline{f} = \overline{f};\} \\
M &::= \texttt{m}(\overline{x})\{\texttt{return}\ e;\} \\
e &::= \texttt{this} \mid x \mid e.f \mid e.\texttt{m}(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \mid (C)e
\end{aligned}
$$

All type annotations in our TGFJ language can be omitted ($T = \epsilon$). The only exception are fields which must be given a concrete type.

Another difference to the syntax of GFJ is that we added the special variable `this` to the syntax. FJ treats `this` as a normal variable but our algorithm treats it as a special variable which always has a predetermined type; the type of the class it is used in.

We assume every method name is only used once in the input program.

Author's address: Andreas Stadelmeier, DHBW Stuttgart, Campus Horb, Tannenweg 4, TÃijbingen, Germany, a.stadelmeier@hb.dhbw-stuttgart.de.

0004-5411/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Type inference for polymorphic recursion is undecidable. Therefore we have to alter the GFJ typing rules to exclude polymorphic recursion in method calls:

(1) The MT-CLASS rule is removed.
(2) We change the GT-METHOD rule: GT-METHOD:

$$\frac{\triangle \vdash \overline{X} <: \overline{N}, \overline{Y} <: \overline{P} \qquad \triangle \vdash \overline{T}, T \text{ ok} \qquad \triangle; \overline{x} : \overline{T}, this : \mathtt{C<\overline{X}>} \vdash e_0 : S \qquad \triangle \vdash S <: T \qquad \mathtt{class\ C<\overline{X} \triangleleft \overline{N}>} \triangleleft N\{\ldots\} \qquad mtype(m, \mathtt{C<\overline{X}>}) := \overline{T} \to T}{mtype(m, \mathtt{C<\overline{Z}>}) = [\overline{Z}/\overline{X}](\mathtt{<\overline{Y}>}\ \overline{T} \to T)}$$

2.2 Principal Type

GFJ supports two kinds of types; Type variables and nonvariable types. Nonvariable types have the form $\mathtt{C<\overline{T}>}$ and can contain multiple type variables. The type $T_2 = [N/X]\mathtt{C<\overline{T}>}$ is a generic instantiation, where all occurences of the type variable X are replaced by the nonvariable type N.

Principal Type: A nonvariable type for a declaration is a principal nonvariable type, if any other type-scheme for the declaration is a subtype of a generic instance of the type-scheme.

Additionally we define the subtype relation of methods as follows:

$$\frac{\overline{T} \leq^* \overline{U} \qquad T \leq^* U}{\overline{T} \to T \ \leq^* \ \overline{U} \to U}$$

Theorem 1. ***There is one unique principal type for every method***

Proof:
We exclude polymorphic recursion, remove overloading and assume every method and field name to be unique. Now every method $T\ \mathtt{m}(\overline{T}\ \overline{x})\{\ \mathtt{return}\ e;\ \}$ has one principal type. This is when its return type T is the same type as its return expression e. For the parameter types $\overline{T}$ the principal type is chosen.

Example: A method `m` can have multiple possible types due to the fact that GFJ has subtypes.

```
class C1 {
  C1 f;
}
class C2 extends C1{
  m(x){
    return x.f;
  }
}
```

The method `m` can either have the type `C1 m(C1 x)` or `C1 m(C2 x)`. In this case the principal type would be `C1 m(C1 x)`. This type has the type of the return expression as its return type and for every parameter type it has the principal type.

3 TYPE INFERENCE ALGORITHM

In this chapter we present our type inference algorithm. The algorithm is split into following parts:

(1) Create assumptions and subtype relation
(2) Constraint generation with **GFJTYPE**
(3) Unification of those constraints
(4) Set in principal type solution

The Unify algorithm returns a set of possible type solutions. This means that there are possibly multiple type solutions for each method. The last step has to choose the principal type out of those possibilities.

3.1 Process multiple classes

The algorithm processes only one class at a time. Only the first step creating the type assumptions is able to consider other classes as well.

Nevertheless we allow the input to consist out of multiple classes. But in that case there are some additional requirements for the input.

We assume that the algorithm are given the input classes in the correct order $C_1, \ldots C_2$. Hereby there must exist a correct typisation for the class C_1 when existing on its own. When there are two classes which depend on one another this would be invalid input for our algorithm.

```
class C1 extends Object {
  C1(){ super(); }
  m1(){ return new C2(); }
}
class C2 extends C1{
  C2(){ super(); }
}
```

Here the class `C2` extends `C1`, but the class `C1` cannot be processed first because it already uses `C2`. Without the method `m1` the class `C1` would not reference any other class and stand on its own.

```
class C1 extends Object {
  C1(){ super(); }
}
class C2 extends C1{
  C2(){ super(); }
  m1(){ return new C2(); }
}
```

By moving `m1` to class `C2` we create a valid input for our algorithm.

When processing the second class the first step of the algorithm (create assumptions) also considers the first and already compiled class. A class C_n therefore can use and reference all the classes before it $(C_1, \ldots, C_{n-1})$.

3.2 Generate Assumptions

Generating assumptions consists of two parts. At first we add type variables to the untyped class. The second part generates the assumption set. This is the same algorithm for the already typed classes as for the new untyped class, which is now equipped with type variables.

(1) Every missing type in the input class gets assigned a fresh type variable. For methods:

$$\frac{m(\overline{x})\{\ldots\} \qquad A \cup \overline{A} \text{ are fresh type variables}}{Am(\overline{A}\,\overline{x})\{\ldots\}}$$

For fields:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{>}\ \{\overline{f};\quad \ldots\} \qquad \overline{F} \text{ are fresh type variables}}{\texttt{class C<}\overline{\texttt{X}}\texttt{>}\ \{\overline{F}\ \overline{f};\quad \ldots\}}$$

(2) We define the two functions $ftype_{Ass}$ and $mtype_{Ass}$. Both functions return a set of all types for a method `m` or a field `f`. This is due to the fact that there can be multiple methods and fields with the same name.

$$\frac{class\ \texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\ \{\ \overline{N}\ \overline{f};\ K\ \overline{M}\ \}\qquad \texttt{<}\overline{\texttt{Y}}\texttt{>}\ U\ \texttt{m}(\overline{U}\ \overline{x})\{\ldots\}\in\overline{M}}{mtype_{Ass}(\texttt{m}) = \texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\ \rightarrow\ \texttt{<}\overline{\texttt{Y}}\texttt{>}\ (\overline{U}\rightarrow U)}$$

$$\frac{class\ \texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\ \{\ \overline{T}\ \overline{f};\ K\ \overline{M}\ \}\qquad T\ \texttt{f}\in\overline{f}}{ftype_{Ass}(\texttt{f}) = \texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\ \rightarrow T}$$

(3) We do not include casts in the syntax and therefore remove the `GT-DCAST`, `GT-UCAST` and `GT-SCAST` typing rules.

3.3 GFJTYPE

The algorithm **GFJTYPE** is given as follows:

FJTYPE: $\texttt{TypeAssumptions}\times\texttt{Class}\rightarrow\texttt{Constraints}$
FJTYPE$(Ass, \texttt{class}\ C\ \texttt{extends}\ D\{\overline{T}\ \overline{f};\ K\ \overline{M}\}) =$
$\quad\{\ \textbf{TYPEMethod}(Ass\cup\{\texttt{this}:C\}, m_i)\quad|\quad m_i\in\overline{M}\ \}$

The **FJTYPE** function gets called for every class in the input. This function accumulates all the constraints generated from calling the **TYPEMethod** function for each method declared in the given class.

TYPEMethod : $\texttt{TypeAssumptions}\times\texttt{Method}\rightarrow\texttt{Constraints}$
TYPEMethod$(Ass, T_r\ \texttt{m}(\overline{T}\ \overline{x})\{\texttt{return}\ e;\}) =$
$\quad$**let** $Ass_m = Ass\cup\{\overline{T}:\overline{x}\}$
$\quad\quad(e:rty, ConS) =$ **TYPEExpr**(Ass_m, e)
$\quad$**in** $(ConS\cup(rty\lessdot T_r))$

The **TYPEMethod** function for methods just calls the **TYPEExpr** function with the return expression. It is significant to note that it adds the assumptions for the method parameters to the global assumptions before passing them to **TYPEExpr**.

In the following we define the **TYPEExpr** function for every possible expression:

TYPEExpr : $\texttt{TypeAssumptions}\times\texttt{Expression}\rightarrow\texttt{Type}\times\texttt{Constraints}$
TYPEExpr$(Ass, \texttt{this}) = (t, \{\})$ **with** $(\texttt{this}:t)\in Ass$
TYPEExpr$(Ass, x) = (t, \{\})$ **with** $(x:t)\in Ass$

TYPEExpr$(Ass, e.f) =$
$\quad$**let** $(rty, ConS) =$ **TYPEExpr**(Ass, e),
$\quad\quad$**fresh** = a mapping from each variable in $\overline{X}$ to a fresh type variable,
$\quad\quad Ass_f = ftype_{Ass}(f) = \texttt{C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\ \rightarrow T$
$\quad\quad Cons_f = \{\ rty \doteq \texttt{C<}$**fresh**$(\overline{\texttt{X}})\texttt{>}, a \doteq$ **fresh**$(T)\}$,
$\quad$**in** $(a, ConS\cup Cons_f)$
$\quad\quad$*where a is a fresh type variable*

TYPEExpr$(Ass, e_r.\mathtt{m}(\overline{e})) =$
  let$(rty, ConS) =$ **TYPEExpr**(Ass, e_r),
    $\forall e_i \in \overline{e} : (pt_i, ConS_i) =$ **TYPEExpr**(Ass, e_i),
    fresh = a mapping from each variable in $\overline{X}$ to a fresh type variable,
    $Ass_m = mtype_{Ass}(m) = \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\rangle \rightarrow \langle\overline{\mathtt{Y}}\rangle\ (\overline{T} \rightarrow T)$
    $Cons_m = \{\ rty \doteq \mathtt{C}\langle$**fresh**$(\overline{\mathtt{X}})\rangle, a \doteq$ **fresh**$(T),$
      $\bigcup_{T_i \in \overline{T}} (pt_i \lessdot$ **fresh**$(T_i))\}$
  in $(a, ConS \cup Cons_m \cup \bigcup_i ConS_i)$
    where a is a fresh type variable

TYPEExpr$(Ass, \mathtt{new}\ N(\overline{e})) =$
  let$\forall e_i \in \overline{e} : (pt_i, ConS_i) =$ **TYPEExpr**(Ass, e_i),
    $Cons = \{\bigcup_{T_i \in \overline{T}} (pt_i \lessdot T_i) \mid \mathtt{constructor\ C}\langle\overline{\mathtt{X}}\rangle\ (\overline{T}\overline{x}) \in Ass\}$,
  in $(C, ConS \cup \bigcup_i ConS_i)$

3.3.1 Completeness. Theorem: The Unify algorithm is complete Theorem: **GFJTYPE** generates the principal type Proof: The **Unify** algorithm is complete, so the principal type is included in the solution set. We only have to choose the principal type out of those solutions.

All types that are possible under the GFJ typing rules, plus our additional assumptions, also comply with the generated constraints.

We match every generated constraint with the respective type rule to show completeness of our **GFJTYPE** algorithm. This shows that none of the generated constraints remove a type which otherwise would be possible under the GFJ typing rules. The constraints are generated on expression statements. We now compare the constraints for each expression with the appropriate type rule from GFJ:

this has always the type of the surrounding class and generates no constraints.

Local var No constraints are generated.

Method invocation By direct comparison we show that each of the generated constraints do not apply more restrictions than the GT-INVK rule. The GT-INVK rule states the condition $mtype(m, bound_\triangle(T_0)) = \langle\overline{\mathtt{Y}}\rangle\ \overline{U} \rightarrow U$. In our version of typeless GFJ every method name is unique and there is only one class with that particular method. The constraint $rty \lessdot \mathtt{C}\langle$**fresh**$(\overline{\mathtt{X}})\rangle$ assures that the type of the expression e_0 contains the method m.

GFJ Type rule	**Constraints**
$\triangle; \Gamma \vdash e_o : T_0$	$(rty, ConS) =$ **TYPEExpr**(Ass, e_r)
$\quad mtype(m, bound_\triangle(T_0)) = \langle\overline{\mathtt{Y}}\rangle\ \overline{U} \rightarrow U$	$rty \lessdot \mathtt{C}\langle$**fresh**$(\overline{\mathtt{X}})\rangle$
$\triangle; \Gamma \vdash \overline{e} : \overline{S}$	$\forall e_i \in \overline{e} : (pt_i, ConS_i) =$ **TYPEExpr**(Ass, e_i)
$\triangle \vdash \overline{S} <: \overline{U}$	$\bigcup_{T_i \in \overline{T}} (pt_i \lessdot$ **fresh**$(T_i))$
$\triangle; \Gamma \vdash \mathtt{e_0.m}(\overline{\mathtt{e}}) : \mathtt{U}$	$a \doteq$ **fresh**(T)

Note: The **TYPEExpr** function only generates constraints which apply to our assumption.

Field access Mostly the same as method invocation. Fieldnames by default are unique in the GFJ language.

GFJ Type rule	**Constraints**
$\Gamma \vdash e_0 : T_0$	$(rty, ConS) =$ **TYPEExpr**(Ass, e_r)
$\quad fields(bound_\triangle(T_0)) = \overline{T}\ \overline{f}$	$rty \doteq \mathtt{C}\langle$**fresh**$(\overline{\mathtt{X}})\rangle$
$\triangle; \Gamma \vdash \overline{e} : \overline{S}$	$\forall e_i \in \overline{e} : (pt_i, ConS_i) =$ **TYPEExpr**(Ass, e_i)
$\triangle \vdash \overline{S} <: \overline{U}$	$\bigcup_{T_i \in \overline{T}} (pt_i \lessdot$ **fresh**$(T_i))$
$\triangle; \Gamma \vdash \mathtt{e_0.m}(\overline{\mathtt{e}}) : \mathtt{U}$	$a \doteq$ **fresh**(T)

$$\textbf{Constructor} \quad \begin{array}{l|l} \triangle; \Gamma \vdash \overline{e} : \overline{S} & \forall e_i \in \overline{e} : (pt_i, ConS_i) = \textbf{TYPEExpr}(Ass, e_i) \\ \triangle \vdash \overline{S} <: \overline{T} & \bigcup_{T_i \in \overline{T}} (pt_i \lessdot T_i) \end{array}$$

4 UNIFY

This chapter describes the **GenericUnify** algorithm which is used to find type solutions for the constraints generated by **FGJType**.

input A set of type constraints $Cons_{in}$ and a set of subtype relationships $S_{\leq}$
output A set of type unifiers Uni or fail $Uni = \emptyset$.

The algorithm starts by setting $Eq_{set} = \{Cons_{in}\}$. Afterwards the following steps are repeatedly executed on Eq_{set} until the algorithm termiates:

(1) Repeated application of the rules depicted in figure 1 and 2. The end configuration of Eq is reached if for each element no rule is applicable.
(2) (The function **fresh**(i) returns an array of i fresh type variables.)

$$\begin{aligned} Eq_1 &= \text{Subset of pairs where both type terms are type variables} \\ Eq_2 &= Eq / Eq_1 \\ Eq_{set} &= \times \Big(\bigotimes_{(a<\mathtt{C}<\overline{\mathtt{X}}>) \in Eq'_2} \{ (a \doteq \mathtt{D}<\overline{\mathtt{A}}>) \cup (\overline{X} \doteq \overline{Y'}) \mid (\mathtt{D}<\overline{\mathtt{Z}}> \leq^* \mathtt{C}<\overline{\mathtt{Y}}>) \in S_{\leq}, \\ &\qquad \overline{A} = \textbf{fresh}(\#(\overline{Z})), \\ &\qquad \overline{Y'} = [\overline{A}/\overline{Z}]\overline{Y} \} \Big) \\ &\quad \times \Big(\bigotimes_{(\mathtt{C}<\overline{\mathtt{T}}> <a) \in Eq'_2} \{ (a \doteq [\overline{T}/\overline{X}]N) \mid (\mathtt{C}<\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}> \leq N) \in S_{\leq} \} \Big) \\ &\quad \times \{ [a \doteq \theta \mid (a \doteq \theta) \in Eq'_2] \} \times Eq_1 \end{aligned}$$

(3) Application of the following *subst* rule

$$\text{(subst)} \quad \frac{Eq'' \cup \{a \doteq \theta\}}{Eq''[a \mapsto \theta] \cup \{a \doteq \theta\}} \quad a \text{ occurs in } Eq'' \text{ but not in } \theta$$

for each $a \doteq \theta$ in each element of $Eq' \in Eq'_{set}$.
(4)(a) Foreach $Eq \in Eq_{set}$ which has changed in the last step start again with the first step.
(b) Build the union Eq_{set} of all results of (a) and all $Eq' \in Eq'_{set}$ which has not changed in the last step.
(5)(a) Filter all constraint sets which are in solved form:
$Eq_{solved} = \{Eq \mid Eq \in Eq_{set}, Eq \text{ is in solved form}\}$
(b) We apply the following rule to every constraint set in Eq_{solved}:

$$\frac{Eq \cup \{a \lessdot b\}}{Eq \cup \{a \doteq b\}}$$

(c) $Uni = \{\sigma \mid Eq \in Eq_{solved}, \ \sigma = \{a \mapsto \theta \mid (a \doteq \mathtt{T}) \in Eq\}\}$

4.1 Unify proof

Theorem 2. ***(Soundness):*** *If the* ***Unify*** *algorithm finds a solution it does not contradict any of the input constraints:* $\nexists (a \lessdot b) \in Cons_{in}$ *where* $\sigma(a) \not\leq \sigma(b)$

$$\text{(adapt)}\quad \frac{Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{C<}\overline{\texttt{B}}\texttt{>}\}}{Eq \cup \{\texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>} \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}\}} \quad (\texttt{D<}\overline{\texttt{X}}\texttt{>} \leq^* \texttt{C<}\overline{\texttt{Y}}\texttt{>}) \in S_{\leq}$$

$$\text{(reduce1)}\quad \frac{Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{B}}\texttt{>}\}}{Eq \cup \{\overline{A} \doteq \overline{B}\}}$$

$$\text{(reduce2)}\quad \frac{Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \doteq \texttt{D<}\overline{\texttt{B}}\texttt{>}\}}{Eq \cup \{\overline{A} \doteq \overline{B}\}}$$

Fig. 1. Reduce and adapt rules

$$\text{(erase1)}\quad \frac{Eq \cup \{C \lessdot D\}}{Eq} \quad C \leq^* D \in S_{\leq}$$

$$\text{(erase2)}\quad \frac{Eq \cup \{C \doteq C\}}{Eq}$$

$$\text{(swap)}\quad \frac{Eq \cup \{C \doteq a\}}{Eq \cup \{a \doteq C\}}$$

Fig. 2. Erase and swap rules

Proof: We show theorem 2 by going backwards over every step of the algorithm. We assume there exists a unifier $\sigma = \{a_1 \mapsto \theta_1, \ldots, a_n \mapsto \theta_n\}$ for the input constraints, which is the result of the **Unify** algorithm. This means for every constraint in the input set $(a \lessdot b) \in Cons_{in}$ and $(c \doteq d) \in Cons_{in}$ this unifier will substitute all variables in a way that all constraints are satisfied: $\sigma(a) \leq \sigma(b)$, $\sigma(c) = \sigma(d)$

We now look at each step of the **Unify** algorithm which transforms the input set of constraints Eq to a set Eq'. If we assume the unifier σ is correct for the set Eq', then we can show that it will also be correct for the constraints Eq.

Step 5 c) The last step of the algorithm transforms a set of constraints Eq of the the form $Eq = \{a_1 \doteq \theta_1, \ldots, a_n \doteq \theta_n\}$ to the unifier σ. Trivial.

Step 5 b) A unifier which is correct for $a \doteq b$ is also correct for $a \lessdot b$.

Step 5 a) Trivial, we do not alter the constraint set which lateron leads to the unifier.

Step 4 Trivial, the constraint sets are not altered here.

Step 3 An unifier σ that is correct for a constraint set $Eq[a \rightarrow \theta] \cup (a \doteq \theta)$ is also correct for the set $Eq \cup (a \doteq \theta)$. From the constraint $(a \doteq \theta)$ it follows that $\sigma(a) = \theta$. This means that $\sigma(Eq) = \sigma(Eq[a \rightarrow \theta])$, because every occurence of a in Eq will be raplaced by θ anyways when using the unifier σ.

Step 2 This step transforms constraints of the form $(\texttt{C<}\overline{\texttt{X}}\texttt{>} \lessdot a)$ and $(a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>})$ into sets of constraints and builds the cartesian product with the remaining constraints. We can show that if there is a resulting set of constraints which has σ as its correct unifier then σ also has to be a correct unifier for the constraints before this transformation. Proof:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \triangleleft\ \overline{\texttt{N}}\texttt{>}\ \triangleleft\ N\{\ldots\}}{\triangle \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ <: [\overline{T}/\overline{X}]N}$$

Fig. 3. Generic Featherweight Java subtyping rules

($a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}$) If σ is a correct unifier for a set containing ($a \doteq \texttt{D<}\overline{\texttt{A}}\texttt{>}$) and ($\overline{X} \doteq \overline{Y'}$) , then σ is also a correct unifier for the set containing ($a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}$). The reason is the S-CLASS subtype rule of GFJ.
When substituting ($a \rightarrow \texttt{D<}\overline{\texttt{A}}\texttt{>}$) and ($\overline{X} \rightarrow \overline{Y'}$) and finally ($\overline{Y'} \rightarrow [\overline{A}/\overline{Z}]\overline{Y}$) in the constraint ($a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>}$) we get: ($\texttt{D<}\overline{\texttt{A}}\texttt{>}\ \lessdot [\overline{A}/\overline{Z}]\texttt{C<}\overline{\texttt{Y}}\texttt{>}$ which is correct under the S-CLASS rule (see figure 3).
($\texttt{C<}\overline{\texttt{T}}\texttt{>}\ \lessdot a$) If $\texttt{C<}\overline{\texttt{X}}\texttt{>}\ leq\theta$ and σ is correct for ($a \doteq [\overline{T}/\overline{X}]N$) then σ is also correct for ($\texttt{C<}\overline{\texttt{T}}\texttt{>} \lessdot a$)). When substitutin a for $[\overline{T}/\overline{X}]N$ we get ($\texttt{C<}\overline{\texttt{T}}\texttt{>}\ \lessdot [\overline{T}/\overline{X}]N$) , which is correct because ($\texttt{C<}\overline{\texttt{X}}\texttt{>}\ \leq \texttt{C<}\overline{\texttt{Y}}\texttt{>}$ (see S-CLASS rule).
[1]

Step 1 erase-rules remove correct constraints from the constraint set. A unifier σ that is correct for the constraint set Eq is also correct for $Eq \cup \{\theta \doteq \theta\}$ and $Eq \cup \{\theta \lessdot \theta'\}$, when $\theta \leq \theta'$.

swap-rule does not change the unifier for the constraint set. $\doteq$ is a symmetric operator and parameters can be swapped freely.

adapt If there is a σ which is a correct unifier for a set $Eq \cup \{\texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>}\ \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}\ \}$ then it is also a correct unifier for the set $Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>}\ \lessdot \texttt{C<}\overline{\texttt{B}}\texttt{>}\ \}$, if there is a subtype relation $\texttt{D<}\overline{\texttt{X}}\texttt{>}\ \leq^{*} \texttt{C<}\overline{\texttt{Y}}\texttt{>}$. To make the set $Eq \cup \{[\overline{A}/\overline{X}]\texttt{C<}\overline{\texttt{Y}}\texttt{>}\ \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}\ \}$ the unifier σ must satisfy the condition $\sigma([\overline{A}/\overline{X}]\overline{Y}) = \sigma(\overline{B})$. By substitution we get $Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>}\ \lessdot \texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>}\ \}$ which is correct under the S-CLASS rule.

reduce The reduce1 and reduce2 rules are obviously correct under the FJ typing rules.

OrConstraints If σ is a correct unifier for one of the constraint sets in Eq_{set} then it is also a correct unifier for the input set $Cons_{in}$. When building the cartesian product of the **OrConstraints** every possible combination for $Cons_{in}$ is build. No constraint is altered, deleted or modified during this step.

□

Theorem 3. ***(Completeness):*** *The* ***Unify*** *algorithm calculates the principal type solution for the input set of constraints ($Cons_{in}$). A unifier σ is a principal type solution for $Cons_{in}$ if it unifies $Cons_{in}$ and for every other unifier ω there is a unifier λ so that $\omega(x) = \lambda(\sigma(x))$.*

Proof:

We look at every step of the algorithm, which alters the set of constraints Eq, while assuming that there is at least one possible principal type solution σ for the input. We will show that the principal type is among them by proofing for every step of the algorithm that the principal type is never excluded.

[1] Discussion: Do we need to include the $\triangleleft\overline{N}$ bounds? The S-CLASS rule does not mention them.
When ignoring those rules this could lead to an error class T<X extends List> class S<X>
Constraint: S<String> <. a Unify: T<String> =. a // ERROR!
This is not a problem because no error is gonna result from this. TYPEExpr only hast to implement all of the typing rules of FJ and unify has to solely respect the subtyping rules.

Step 1: The first step applies the three rules from figure **??**. **erase-rules:** The erase2 rule from figure **??** removes a $\{\theta \leq \theta\}$ constraint from the constraint set. The erase1 rule removes a $\{\theta \leq \theta\}$ constraint, but only if the two types θ and θ' satisfy the constraint. Both rules do not change the set of possible solutions for the given constraint set.
swap-rule: $\doteq$ is a symmetric operator and parameters can be swapped freely. This operation does not change the meaning of the constraint set.
adapt-rule: Every solution which is correct for the constraints $Eq \cup \{\texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>} \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}\}$ is also a correct solution for the set $Eq \cup \{\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{C<}\overline{\texttt{B}}\texttt{>}\}$. According to the S-CLASS rule there can only be a possible solution for $\texttt{C<}[\overline{\texttt{A}}/\overline{\texttt{X}}]\overline{\texttt{Y}}\texttt{>} \doteq \texttt{C<}\overline{\texttt{B}}\texttt{>}$ if $\overline{B} = [\overline{A}/\overline{X}]\overline{Y}$. Therefore this transformation does not remove any possible solution from the constraint set.
reduce-rule: For a constraint $\texttt{D<}\overline{\texttt{A}}\texttt{>} \lessdot \texttt{D<}\overline{\texttt{A}}\texttt{>}$ the FJ subtyping rule S-REFL ($\triangle \vdash T <: T$) is the only one which applies. According to this rule the transformation to $\overline{A} \doteq \overline{B}$ is correct. Only D gets removed, which is not a type variable. Therefore this step does not remove a possible solution. This applies for both reduce rules **reduce1** and **reduce2**.

Step 2: The second step of the algorithm eliminates $\lessdot$-constraints by replacing them with $\doteq$-constraints. For each $(a \lessdot \texttt{C<}\overline{\texttt{X}}\texttt{>})$ constraint the algorithm builds a set with every possible subtype of $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ set in for a. So if there is a correct unifier σ for the constraints before this conversion there will be at least one set of constraints for which σ is a correct unifier.

Step 3: In the third step the **substitution**-rule is applied. If there is a constraint $a \doteq \theta$ then there is no other way to fulfill the constraint set than replacing a with θ. This does not remove a possible solution.

Step 4: None of the constraints get modified.

Step 5 a): The removed sets do not have a possible unifier, therefore no possible solution is omitted in this step.
Proof: In step 5.a all constraint sets that have a unifier are in solved form. All other possibilities are eliminated in steps 1-4. There are 8 different variations of constraints:
$(a \doteq a), (a \doteq C), (C \doteq a), (C \doteq C), (a \lessdot a), (a \lessdot C), (C \lessdot a), (C \lessdot C)$
After step 1 there are no $(C \doteq C)$, $(C \lessdot C)$ and $(C \doteq a)$ constraints anymore, as long as the constraint set has a correct unifier. Because a constraint set that has a correct unifier cannot contain constraints of the form $\theta_1 \doteq \theta_2$ with $\theta_1 \neq \theta_2$ and $(\theta_1 \lessdot \theta_2)$ with $(\theta_1 \leq \theta_2) \notin S_{\leq}$. By removing $(\theta \doteq \theta)$ and $(\theta \lessdot \theta')$ with $(\theta \leq \theta') \notin S_{\leq}$ constraints no constraints of the form $(C \doteq C)$ and $(C \lessdot C)$ remain in a constraint set that has a correct unifier after step 1.
After step 2 there are no more $(a \lessdot C)$ constraints.
After step 3 there are no $(a \doteq C)$ anymore.
We only reach step 5 if the constraint set is not changed by the substitution (step 3).
If the constraint set has a correct unifier only $(a \lessdot a)$, $(a \doteq a)$ and $(a \doteq C)$ constraints are left at this point. The type variables in the $(a \lessdot a)$ and $(a \doteq a)$ constraints have to be independent type variables. If a type variable c is inside a $(c \doteq C)$ constraint it is not an independent type variable. But this variable c cannot be inside a $(a \doteq a)$ or $(a \lessdot a)$ constraint, because otherwise step 3 would have replaced it in there.

Step 5 b): If the algorithm advances to this step we further only work on constraint sets in solved form. This means there are only two kinds of constraints left. $(A \doteq \texttt{Typ})$, $(A \doteq B)$ and $(A \lessdot B)$ with A and B as type variables.
The GFJ language does not allow subtype constraints for generic types. A constraint like $(A \lessdot B)$ in a solution could be inserted as the typing shown in the example below. But this is not allowed by the syntax of GFJ. That is why we can treat this constraint as $(A \doteq B)$.
Example: This would be a valid Java program but is not allowed in GFJ:

Table 1. Two classes as input. Class1 is infered first (shown in red)

```
class Class1 extends Object {
  Class1() { super(); }
  id(a){
    return a;
  }
}
class Class2 extends Class1 {
  Class2() {
    super();
  }
  example(){
    return new Class1().id(this);
  }
}
```

```
class Class1 extends Object {
  Class1() { super(); }
   <A> A  id( A  a){
    return a;
  }
}
class Class2 extends Class1 {
  Class2() {
    super();
  }
   Class1 example(){
    return this.<Class1>id(this);
  }
}
```

```
class Example {
  <A extends Object, B extends A> A id(B a){
    return a;
  }
}
```

By replacing all $(A \lessdot B)$ constraints with $(A \doteq B)$ we do not remove a principal type solution.

Step 6: In the last step all the constraint sets, which are in solved form, are converted to unifiers. We see that only a constraint set which has no unifier does not reach solved form. We showed that in every step of the **Unify** algorithm we never exclude a possible unifier. Also we showed that after we reach step 5 only constraint sets with a correct unifier are in solved form. By removing all constraint sets which are not in solved form the algorithm does not remove a possible correct unifier.

If we assume that there is a possible principal type solution σ for the input set $Cons_{in}$ and the **Unify** algorithm does not exclude any of the possible unifiers, then the result **Unify** contains the principal type solution. □

5 EXAMPLES

Example 1

The algorithm is able to infer the types of multiple classes under specific circumstances. The individual classes must be given to him after one another. This comes with the restriction, that the first class is correct on its own and does not use any other class. The second class that gets compiled can use the first class and so on.

The following example shows how the algorithm infers and compiles multiple classes iteratively. The class Class1 is infered first. It has only one method which is the identity function, to which our algorithm allocates the type <A> $A \to A$. The next class Class2 is now able to use this generic method. The blue colored types are inferred in the next iteration of our algorithm.

Example 2

When compiling a class like the following we have to first split this class into two classes. The

TwoMethods class can be first split into the classes Class1 and Class2 and after being processed by the type inference algorithm it can be assembled back together again. This leads to a principal typing. When using our type inference algorithm on the class TwoMethods alone it would give the method id the type TwoMethods → TwoMethods, which is not the desired principal type.

```
class TwoMethods extends Object {
  TwoMethods() { super(); }
  id(a){
    return a;
  }
  example(){
    return this.id(this);
  }
}
```

6 RESTRICTIONS/PROBLEMS