Structural type inference in Java-like languages

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. In the past we considered type inference for Java with generics and lambdas. Our type inference algorithm determines nominal types in subjection to a given environment. This is a hard restriction as separate compilation of Java classes without relying on type informations of other classes is impossible. In this paper we present a type inference algorithm for a Java-like language, that infers structural types without a given environment. This allows separate compilation of Java classes without relying on type informations of other classes.

1 Introduction

In this paper we give an algorithm which is a generalization of an idea, that is given in [ADDZ05]. In the introducing example from [ADDZ05] the method `E m(B x){ return x.f1.f2; }` is given. The compilation algorithm generates the polymorphic typed Java expressions

E m(B x){ return [[x:B].f1:α].f2:β; },

where α and β are type variables. In this system m is applicable to instances of the class `B` with the field `f1` with the type α, where α must have a field `f2` with the type β and the constraint $\beta \leq^*$ `E`. In this approach `B` and `E` are still nominal types.

We generalize this approach, such that also untyped methods like `m(x){ return x.f1.f2; }` can be compiled, that means the type of `x` and the return type are type variables, too.

Furthermore the results of our algorithms are Java-like programs, not byte-code, as in [ADDZ05], such that the typed programs are readable and the linking prcoess is reduced to a simple ckeck if a class implements a given interface.

This algorithm can be considered as generalization of our type inference algorithms [Plü15,Plü07].

Let us consider an example that shows the differences. For the following program no type can be inferred, as there is no type assumption for `elementAt`.

```
class A { m (v) { return v.elementAt(0); } }
```

Only with an import declaration `import java.util.Vector;` a type can be inferred.

The generalized algorithm which we give in this paper infers for `v` a structural type α, which has a method `elementAt`.

The basic idea

The result of our type inference algorithm is a parameterized class, where each inferred type is represented by a parameter that implements a new generated interfaces.

The paper is organized as follows. In the next section the language is introduced. In the third section we give the algorithm. Then we present a large example. Finally we close with a summary.

2 The language

We consider a core of a Java-like language without lambdas. In Figure 1 the syntax of the language is given. It is an extension of Featherweight Java [IPW01]. The syntax is differed between input and output syntax. The input is an untyped Java program L and the output is a typed Java program $\mathtt{L}_t$, including generated interfaces.

Input syntax :

$\mathtt{L} ::= \mathtt{class\ C\ extends\ (CT)}^*\ \{\overline{\mathtt{f}};\ \overline{\mathtt{M}}\}$

$\mathtt{M} ::= \mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \}$

$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ NCT}(\overline{\mathtt{e}}) \mid \mathtt{(CT)e}$

$\mathtt{NCT} ::= \mathtt{CT} \mid \mathtt{C}\texttt{<}\overline{\mathtt{TVar\ =\ T}}\texttt{>}$

$\mathtt{T} ::= \mathtt{CT} \mid \mathtt{TVar}$

$\mathtt{CT} ::= \mathtt{C}\texttt{<}\overline{\mathtt{CT}}\texttt{>}$

Output syntax :

$\mathtt{L}_t ::= \mathtt{I}^*\ \mathtt{CL}_t$

$\mathtt{CL}_t ::= \mathtt{class\ C}\texttt{<}\overline{\mathtt{TVar}}\texttt{>}[\overline{\mathtt{CONS}}]\ \mathtt{extends\ (CT)}^*\ \{\overline{\mathtt{T\ f}};\overline{\mathtt{M_t}}\}$

$\mathtt{CONS} ::= \mathtt{T\ extends\ T}$

$\mathtt{MH} ::= \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}});$

$\mathtt{M}_t ::= \mathtt{MH}\ \{\ \mathtt{return\ e_t};\ \}$

$\mathtt{e}_t ::= \mathtt{x} : \mathtt{T} \mid \mathtt{e.f} : \mathtt{T} \mid \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T} \mid \mathtt{new\ NCT}(\overline{\mathtt{e}}) : \mathtt{CT} \mid \mathtt{(CT)e} : \mathtt{CT}$

$\mathtt{I} ::= \mathtt{interface\ I}\texttt{<}\overline{\mathtt{TVar}}\texttt{>}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\overline{\mathtt{MH}}\}$

Fig. 1. The syntax

There are some extensions in comparison to usual Java. The class declarations in the output syntax have the form $\mathtt{class\ C}\texttt{<}\overline{\mathtt{TVar}}\texttt{>}\ [\mathtt{CONS}]$. $\overline{\mathtt{TVar}}$ are the generics and `[CONS]` is a set of subtype constraints $\texttt{T extends T}'$, that must fulfill all instances of the class. In any class there is an implicit constructor with all fields (including them from the superclasses) as arguments. There is no differentiation between `extends` and `implements` declarations. Both are declared by `extends`. Interfaces can have fields. Furthermore, the use of the new-statement is allowed without assigning all generics. This is done by the syntax $\texttt{C<}\overline{\texttt{TVar = CT}}\texttt{>}$. The not assigned generics are derived by the type inference algorithm.

3 The algorithm

The algorithm **TI** consists of three parts. First the function **TYPE** inserts types (usually type variables) to all sub-terms and collects constraints. Second, the function **construct** generates the interfaces and completes the constraints. Finally, the function **solve** unifies the type constraints and applies the unifier to the class.

In the following definition we give the different forms of constraints, that are collected in **TYPE**. This definition is oriented at [ADDZ05]:

Definition 1. (Type constraints)

- $c \lessdot c'$ *means* c has to be a subtype of c'*.*
- $\phi(\, c, f, c' \,)$ *means* c provides a field f with type c'*.*
- $\mu(\, c, m, \overline{c}, (c', \overline{c}')\,)$ *means* c provides a method m applicable to arguments of type $\overline{c}$, with return type c' and parameters of type $\overline{c}'$*.*

Note that $\mu(\, c, m, \overline{c}, (c', \overline{c}')\,)$ implicitly includes the constraints $\overline{c} \leq \overline{c}'$.

Let $<$ be the extends relation defined by the `Java` declarations und $\leq^*$ the corresponding subtyping relation.

The type–inference algorithm

Let `TypeAssumptions` be a set of assumptions, that can consists of assumptions for fields, methods and whole classes with fields and methods. The functions *fields* and *mtype* extracts the typed fields respectively the typed methods from a given class, as in [IPW01].

In the type inference algorithm we use the following name conventions for type variables:

$\delta^{\mathtt{f}}_{\mathtt{A}}$**:** Type variable for the field `f` in the class `A`.
$\alpha^{\mathtt{m,i}}_{\mathtt{A}}, \beta^{\mathtt{m,i}}_{\mathtt{A}}$**:** Type variable for the i-th argument of the method `m` in the class `A`.
$\overline{\alpha^{\mathtt{m}}_{\mathtt{A}}}, \overline{\beta^{\mathtt{m}}_{\mathtt{A}}}$**:** is an abbreviation for the tuple $\alpha^{\mathtt{m,1}}_{\mathtt{A}}, \ldots, \alpha^{\mathtt{m,n}}_{\mathtt{A}}$ respectively $\beta^{\mathtt{m,1}}_{\mathtt{A}}, \ldots, \beta^{\mathtt{m,n}}_{\mathtt{A}}$.
$\gamma^{\mathtt{m}}_{\mathtt{A}}$**:** Type variable for the return type of the method `m` in the class `A`.

The main function TI The main function **TI** calls the three functions **TYPE**, **construct**, and **solve**. The input is a set of type assumptions `TypeAssumptions` and an untyped `Java` class `L`. The result $\mathtt{L}_t$ is the typed `Java` class extended by a set of interfaces.

```
TI: $\mathtt{TypeAssumptions} \times \mathtt{L} \to \mathtt{L}_t$
TI  $(\, Ass, \mathtt{class\ A\ extends\ } \overline{\mathtt{B}}\ \{\, \overline{\mathtt{f}};\ \overline{\mathtt{M}} \,\}\,) =$
  let
     $(cl_t, C) = \mathbf{Type}(\, Ass, cl \,)$
     $(I_1 \ldots I_m\ cl_t) = \mathbf{construct}(\, cl_t, C \,)$
  in
     $(I_1 \ldots I_m\ \mathbf{solve}(\, cl_t \,))$
```

The function TYPE The function **TYPE** inserts types (usually type variables) to all sub-terms and collects the constraints.

```
TYPE: $\mathtt{TypeAssumptions} \times \mathtt{L} \to \mathtt{L}_t \times \mathtt{ConstraintsSet}$
TYPE$(\, Ass, \mathtt{class\ A\ extends\ } \overline{\mathtt{B}}\ \{\, \overline{\mathtt{f}};\ \overline{\mathtt{M}} \,\}\,) = \mathbf{let}$
   $fass := \{\, \mathtt{this.f} : \delta^{\mathtt{f}}_{\mathtt{A}} \mid \mathtt{f} \in \overline{\mathtt{f}} \,\} \cup \{\, \mathtt{this.f} : \mathtt{T} \mid \mathtt{T\ f} \in fields(\, \overline{\mathtt{B}} \,) \,\}$
   $mass := \{\, \mathtt{this.m} : \overline{\alpha^{\mathtt{m}}_{\mathtt{A}}} \to \gamma^{\mathtt{m}}_{\mathtt{A}} \mid\ \mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}} \,\} \cup$
           $\{\, \mathtt{this.m} : \overline{aty} \to rty \mid\ mtype(\, \mathtt{m}, \overline{\mathtt{B}} \,) = \overline{aty} \to rty \,\}$
   $AssAll = Ass \cup fass \cup mass \cup \{\, \mathtt{this} : \mathtt{A} \,\}$
   For $\mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}}$ {
      $Ass = AssAll \cup \{\, \mathtt{x}_j : \alpha^{\mathtt{m,j}}_{\mathtt{A}} \mid \overline{\mathtt{x}} = \mathtt{x}_1 \ldots \mathtt{x}_n \,\}$
      $(\mathtt{e}_t : rty, C') = \mathbf{TYPEExpr}(\, Ass, \mathtt{e} \,)$
      $C = (C \cup C')[\gamma^{\mathtt{m}}_{\mathtt{A}} \mapsto rty]\}$
   $\overline{\mathtt{M}}_t = \{\, rty\ \ \mathtt{m}(\overline{\alpha^{\mathtt{m}}_{\mathtt{A}}\ \ \mathtt{x}})\{\ \mathtt{return\ e_t};\ \} \mid \mathtt{m}(\overline{\mathtt{x}})\{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}} \,\}$
 in$(\mathtt{class\ A\ extends\ } \overline{\mathtt{B}}\ \{\, \overline{\delta^{\mathtt{f}}_{\mathtt{A}}\mathtt{f}}; \overline{\mathtt{M_t}} \,\}, C)$
```

The function **TYPEExpr** inserts types into the expressions and collects the corresponding constraints. It is given for all cases of expressions `e`.

TYPEExpr: $\mathtt{TypeAssumptions} \times \mathtt{e} \to \mathtt{e}_t \times \mathtt{ConstraintsSet}$

TYPEExpr$(\, Ass, \mathtt{x} \,) =$ **let** $(\mathtt{x} : \theta) \in Ass$ **in** $(\mathtt{x} : \theta, \emptyset)$

TYPEExpr for field-application: First, the type of the receiver is determined. Then it is differed between fields with and without known receiver types. In the known case the types are introduced. Otherwise a constraint is generated that demands a corresponding field in the type.

```
TYPEExpr$(\, Ass, \mathtt{e.f} \,) =$
  let
     $\underline{(\mathtt{e_t} : ty, C)} = \mathbf{TYPEExpr}(\, Ass, \mathtt{e} \,)$
  in
     if $(ty\ \textit{is no type variable}\,)\ \&\&\ (ty \in Ass)\ \&\&(rty\ \mathtt{f} \in fields(\, ty \,))$
     then $(((\mathtt{e_t} : ty).\mathtt{f}) : rty, C)$
     else $(((\mathtt{e_t} : ty).\mathtt{f}) : \delta^{\mathtt{f}}_{\mathtt{ty}}, \{\, \phi(\, ty, \mathtt{f}, \delta^{\mathtt{f}}_{\mathtt{ty}} \,) \,\} \cup C)$
```

TYPEExpr for method-call: First, the types of the receiver and the arguments are determined, recursively. Then it is differed between methods with and without known receiver types. In the known case a subtype relation is introduced. Otherwise a constraint is generated that demands a corresponding method in the type.

```
$\mathbf{TYPEExpr}(\,Ass, \mathtt{e}_0.\mathtt{m}(\overline{e})\,) = \mathbf{let}$
    $(\mathtt{e}_{0_t} : ty_0, C_0) = \mathbf{TYPEExpr}(\,Ass, \mathtt{e}_0\,)$
    $(\mathtt{e}_{\mathtt{i}_t} : ty_i, C_i) = \mathbf{TYPEExpr}(\,Ass, \mathtt{e}_\mathtt{i}\,), \forall 1 \leqslant i \leqslant n$
  $\mathbf{in}$
    $\mathbf{if}\ (ty_0\ \textit{is no type variable})\ \&\&\ (ty_0 \in Ass)\ \&\&\ (mtype(\,\mathtt{m}, ty_0\,) = \overline{aty} \to rty)$
    $\mathbf{then}$
      $((\mathtt{e}_{0_t} : ty_0).\mathtt{m}(\mathtt{e}_{1_t} : ty_1, \ldots, \mathtt{e}_{\mathtt{n}_t} : ty_n) : rty, (C_0 \cup \bigcup_i C_i) \cup \{\, \overline{ty \lessdot aty}\,\})$
    $\mathbf{else}$
      $((\mathtt{e}_{0_t} : ty_0).\mathtt{m}(\mathtt{e}_{1_t} : ty_1, \ldots, \mathtt{e}_{\mathtt{n}_t} : ty_n) : \gamma^{\mathtt{m}}_{\mathtt{ty_0}},$
       $(C_0 \cup \bigcup_i C_i) \cup \{\, \mu(\,ty_0, \mathtt{m}, \overline{ty}, (\gamma^{\mathtt{m}}_{\mathtt{ty_0}}, \overline{\beta^{\mathtt{m}}_{\mathtt{ty_0}}})\,)\,\})$
       where $\overline{\beta^{\mathtt{m}}_{\mathtt{ty_0}}}$ and $\gamma^{\mathtt{m}}_{\mathtt{ty_0}}$ are fresh type variables.
```

TYPEExpr for the new-statement: The use of the new-statement is allowed without assigning all generics. This is done by the syntax `C<`$\overline{\mathtt{TVar = CT}}$`>`. First, fresh type variables are introduced in the assumptions of the corresponding class. Then the types of the arguments are determined. Finally, the assigned generics are introduced and the subtype relations between the argument types and the fields of the class and its super classes are added.

```
$\mathbf{TYPEExpr}(\,Ass \cup \{\,\mathtt{class\ A}{<}\overline{\mathtt{T}}{>}[C_\mathtt{A}]\ \mathtt{extends}\ \overline{\mathtt{B}}\ \{\overline{\mathtt{T_A\ f;}}\ \overline{\mathtt{M_t}}\}\,\}, \mathtt{new\ A}{<}S{>}(\,\overline{e}\,)\,) =$
    where $S = [\mathtt{T}_{\pi(\mathtt{1})} = \tau_1, \ldots, \mathtt{T}_{\pi(\mathtt{k})} = \tau_k]$ with $k \leq n$ for $|\overline{\mathtt{T}}| = n$
    $\mathbf{let}$
        $\overline{\nu}$ fresh type variables, that substitute $\overline{\mathtt{T}}$ and all type variables of $C_A$
        in class $\mathtt{A}$
        $S' = S[\overline{\mathtt{T} \mapsto \nu}]$
        $(\mathtt{e}_{\mathtt{i}_t} : ty_i, C_i) = \mathbf{TYPEExpr}(\,Ass, e_i\,), \forall 1 \leqslant i \leqslant m$
    $\mathbf{in}$
        $(\mathtt{new\ A}{<}\mathtt{S'}{>}(\,\mathtt{e}_{1_t} : \mathtt{ty_1}, \ldots, \mathtt{e}_{\mathtt{m}_t} : \mathtt{ty_m}\,) : \mathtt{A}{<}\overline{\nu}[\nu \mapsto \tau \mid \nu = \tau \in S']{>},$
         $(\bigcup_i C_i) \cup C_\mathtt{A}[\overline{\nu \mapsto \tau}] \cup \{\,\overline{ty} \lessdot \overline{\mathtt{T}_{\overline{\mathtt{B}}}\,\mathtt{T_A}[\overline{\nu \mapsto \tau}]}\,\}$
         where $\mathit{fields}(\,\overline{\mathtt{B}}\,) = \overline{\mathtt{T}_{\overline{\mathtt{B}}}\,\mathtt{g}}$
```

```
$\mathbf{TYPEExpr}(\,Ass, (\mathtt{A})\mathtt{e}\,) =$
    $\mathbf{let}$
        $(ty, C) = \mathbf{TYPEExpr}(\,Ass, \mathtt{e}\,)$
    $\mathbf{in}$
        $((\mathtt{A})\mathtt{e} : \mathtt{A}, C)$
```

The function construct The function **construct** takes the result from **TYPE**, a typed class and a set of constraints. It generates for any type `ty1`, `ty2` occuring in constraints $\phi(\,ty1, \mathtt{f}, \delta\,)$ or $\mu(\,ty2, \mathtt{m}, \overline{\alpha}, (\gamma, \overline{\beta})\,)$ corresponding interfaces with the demanded fields and methods.

construct : $\mathtt{L}_t \times \mathtt{ConstraintsSet} \rightarrow \mathtt{L}_t$

```
construct( class A extends $\overline{\mathtt{B}}$ { $\overline{\delta_{\mathtt{A}}\mathtt{f};}$ $\overline{\mathtt{M_t}}$ }, $C$ ) {
    $C_{\mathtt{A}} = \{\, \alpha \lessdot \beta \mid \alpha \lessdot \beta \in C \,\} \cup \{\, \overline{\alpha \lessdot \beta} \mid \mu(\,ty, \mathtt{m}, \overline{\alpha}, (\gamma, \overline{\beta})\,) \in C \,\}$
    if $C_A$ contains a constraint $ty \lessdot ty'$, where $ty$ and $ty'$
        are not type variables and $ty \not\leq^* ty'$ then fail exit
    new_interf $= \{\, \iota \mid \phi(\,\iota, \mathtt{f}, \gamma\,) \in C \,\} \cup \{\, \iota \mid \mu(\,\iota, \mathtt{m}, \overline{\beta}, (\gamma, \overline{\beta'})\,) \in C \,\}$
    For $\iota \in$ new_interf {
        $I_\iota$ = interface $\iota$ < > { } is generated
        inh_tyterm = "$\iota$ < >"
        For f with $\phi(\,\iota, \mathtt{f}, \delta\,) \in C$ {
            An field f and a fresh type variable T is added to the interface $\iota$:

            $I_\iota$ = interface $\iota$ <args , T> {... T f; ... }

            T is instantiated by $\delta$ in inh_tyterm: inh_tyterm = "$\iota$<args' ,$\delta$>" }
        For m with $\mu(\,\iota, \mathtt{m}, \overline{\beta}, (\gamma, \beta'_1, \ldots, \beta'_n)\,) \in C$ {
            A method signature for m and fresh type variable T, $\overline{\mathtt{T}}$
            are introduced into the interface $\iota$:

            $I_\iota$ = interface $\iota$ <args ,T,T$_1$, ..., T$_n$> {
                    fields
                    meth_sigs
                    T m(T$_1$, ..., T$_n$);
                }

            T and $\overline{\mathtt{T}}$ are instantiated by $\gamma$ and $\overline{\beta'}$ in inh_tyterm:
                inh_tyterm = "$\iota$<args' ,$\gamma$,$\overline{\beta'}$>" }
        $C_{\mathtt{A}} = C_{\mathtt{A}}[\iota \mapsto X] \cup \{\, X \lessdot inh\_tyterm \,\}$, where $X$ is a fresh type variable
        $\sigma = \sigma \cup \{\, \iota \mapsto X \,\}$ }
    tv = typevar( $\overline{\sigma(\,\delta^{\mathtt{A}}_{\mathtt{f}}\,)}$ ) $\cup$ typevar( $\overline{\sigma(\,\mathtt{M_t}\,)}$ )
    return $\overline{I_\iota}$ class A<tv>[$C_{\mathtt{A}}$] extends $\overline{\mathtt{B}}$ { $\overline{\sigma(\,\delta^{\mathtt{A}}_{\mathtt{f}}\,)\ \mathtt{f};}$ $\overline{\sigma(\,\mathtt{M_t}\,)}$ }
```

The function solve The function **solve** takes the result of **construct** and solves the constraints of the class by a type unification algorithm, such that the constraints contains only pairs with at least one type variable.

First we consider the type unification. In [Plü09] we gave a type unification for Java 5.0 types. This algorithm is finitary but not unitary, as pairs $\mathtt{T} \lessdot ty$ are solved by substituting `T` by all subtypes of ty. Now we give a different version of the algorithm where $\mathtt{T} \lessdot ty$ is not solved.

$$\text{(reduce1)}\quad \frac{C \cup \{\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{C \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

where $C\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D\texttt{<}T_1, \ldots, T_n\texttt{>}$ with T_i are type variables

$$\text{(reduce2)}\quad \frac{C \cup \{\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \doteq C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}}{C \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

$$\text{(adapt1)}\quad \frac{C \cup \{\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}}{C \cup \{\, D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}}$$

where $(D\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>})$ with T_i are type variables

$$\text{(adapt2)}\quad \frac{C \cup \{\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot S_1, S_1 \lessdot S_2, \ldots, S_{k-1} \lessdot S_k, S_k \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}}{\begin{array}{l} C \cup \{\, \sigma(\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}\,) \lessdot S_1, S_1 \lessdot S_2, \ldots, \\ \hfill S_{k-1} \lessdot S_k, S_k \lessdot \sigma(\, D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>}\,) \cup \sigma \,\} \end{array}}$$

where
- $k \geq 1$
- $S_i \in TV$ and
- $(D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \not\leq^* D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>})$
  but $(D\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>})$ with $T_i \in TV$
- $\sigma = \mathbf{Unify}^1(\, \{\, D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}\,)$

$$\text{(erase1)}\quad \frac{C \cup \{\, \theta \lessdot \theta' \,\}}{C}\; \theta \leq^* \theta'$$

$$\text{(erase2)}\quad \frac{C \cup \{\, \theta \doteq \theta' \,\}}{C}\; \theta = \theta'$$

$$\text{(swap)}\quad \frac{C \cup \{\, \theta \doteq T \,\}}{C \cup \{\, T \doteq \theta \,\}}\; \theta \notin TV,\ T \in TV$$

$$\text{(subst)}\quad \frac{C \cup \{\, T \doteq \theta \,\}}{C[T \mapsto \theta] \cup \{\, T \doteq \theta \,\}}\; T \in TV \text{ and } T \text{ occurs in } C \text{ but not in } \theta$$

$$\text{(refl)}\quad \frac{C \cup \{\, \theta \lessdot T_1, T_1 \lessdot T_2, \ldots, T_{n-1} \lessdot T_n, T_n \lessdot \theta \,\}}{C \cup \{\, T_i = \theta \mid 1 \leqslant i \leqslant n \,\}}$$

Fig. 2. Java type unification

The algorithm $\mathbf{TUnify}(\, C\,)$ is given by the rules (Figure 2) application the most often as possible. If C contains finally of pairs $\mathtt{T} \doteq ty$, $\mathtt{T} \lessdot ty$, or $ty \lessdot \mathtt{T}$ then C is the result, otherwise the algorithm fails.

Lemma 1 (Termination). The algorithm **TUnify** terminates.

Lemma 2 (Soundness of TUnify). If a substitution σ is a solution of a constraint set C then σ is also a solution of $\mathbf{TUnify}(\, C\,)$.

Lemma 3 (Completeness). Let C be a set of constraints C and σ' a solution. For $\sigma = \{\, \mathtt{T} \mapsto ty \mid \mathtt{T} \doteq ty \in \mathbf{TUnify}(\, C\,) \,\}$ there are substitutions σ'' and σ_{rest}, such that $\sigma' = \sigma'' \circ ((\sigma_{rest} \circ \sigma) \cup \sigma_{rest})$.

Remark 1 (Most general unifier of TUnify). The substitution σ_{rest} is a solution of the remaining pairs $\mathtt{T} \lessdot ty$ and $ty \lessdot \mathtt{T}$. For any solution σ' there is substitution σ'_{rest}, such that $\sigma'_{rest} \circ \sigma$ is a most general unifier.

In the following we prove the two lemmata of *soundness* and *completeness*.

Proof. We do the prove by showing soundness and completeness for all type unification rules. We do this as we show that the solutions before and after application are the same.

reduce1: From the type term construction follows, if and only if σ is a solution of $\{\, C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>} \,\}$ it is also a solution of $\{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}$, if $C\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D\texttt{<}T_1, \ldots, T_n\texttt{>}$.

adapt1: From the type term construction follows iff

$$D\texttt{<}T_1, \ldots, T_n\texttt{>} \leq^* D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}, \text{ where } T_i \text{ are type variables,}$$

then

$$D\texttt{<}T_1, \ldots, T_n\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \leq^* D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n].$$

This means iff σ is a solution of $\{\, D\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \lessdot D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}$ it is also a solution of $\{\, D'\texttt{<}\tilde{\theta}'_1, \ldots, \tilde{\theta}'_m\texttt{>}[T_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \doteq D'\texttt{<}\theta'_1, \ldots, \theta'_m\texttt{>} \,\}$

adapt2: As $\leq^*$ is a partial ordering and a partial ordering is transitiv from the soundness and completeness **adapt1** follows the soundness und completeness of **adapt2**.

refl: As $\leq^*$ is a partial ordering and a partial ordering is reflexive soundness and completeness follows directly.

erase1: obvious.

reduce2, erase2, swap, subst: These rule corresponds to the rules in [MM82]. Therefore soundness and completeness is given.

Now we give the function **solve**, where **TUnify** is called.

solve: $\mathtt{L}_t \to \mathtt{L}_t$

```
solve( class A<$\overline{\mathtt{T}}$>[$C_{\mathtt{A}}$] extends $\overline{\mathtt{B}}$ {$\overline{\mathtt{ty\ f;}}$ $\overline{\mathtt{M_t}}$ }) =
    let
        $subst = \mathbf{TUnify}(\, C_{\mathtt{A}} \,)$
        $\sigma = \{\, \mathtt{T} \mapsto ty \mid \mathtt{T} \doteq ty \in subst \,\}$
        $cs = \{\, \mathtt{T} \lessdot ty \mid \mathtt{T} \lessdot ty \in subst \,\} \cup \{\, ty \lessdot \mathtt{T} \mid ty \lessdot \mathtt{T} \in subst \,\}$
        $\overline{\mathtt{T}}_{new} = \bigcup_{[(\mathtt{rty\ m(}\overline{\mathtt{aty\ x}}\mathtt{)\{\ return\ e;\ \})} \in \overline{\sigma(\,\mathtt{M_t}\,)}]} (\mathsf{TVar}(\,\mathtt{rty}\,) \cup \mathsf{TVar}(\,\mathtt{aty}\,))$
    in
        if is_solvable( $cs$ ) then
            class A<$\overline{\mathtt{T}}_{new}$>[$cs$] extends $\overline{\mathtt{B}}$ { $\overline{\sigma(\,\mathtt{ty}\,)\ \mathtt{f;}}$ $\overline{\sigma(\,\mathtt{M_t}\,)}$ }
        else $fail$
```

4 Example

In this section we give an example, that shows first a structural typing of a class independent from any environment. Then a concrete implementation of this class is given.

Example 1. Let the following class be given

```
class A {
    mt(x, y, z) { return x.sub(y).add(z); }
}
```

First **TYPE** is applied:

$$
\begin{array}{l}
\mathbf{TYPE}(\,\emptyset, \mathtt{A}\,)\\
\quad \underline{\mathit{mass}} = \{\, \mathtt{this.mt} : (\alpha_{\mathtt{A}}^{\mathtt{mt},1}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}) \to \gamma_{\mathtt{A}}^{\mathtt{mt}} \,\}\\
\quad \underline{\mathit{AssAll}} = \mathit{mass} \cup \{\, \mathtt{this} : \mathtt{A} \,\}\\
\quad \underline{\mathtt{mt} \in \overline{M}}:\\
\qquad \underline{\mathit{Ass}} = \mathit{AssAll} \cup \{\, \mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \mathtt{z} : \alpha_{\mathtt{A}}^{\mathtt{mt},3} \,\}\\
\qquad \mathbf{TYPEExpr}(\, \mathit{Ass}, \mathtt{x.sub(y).add(z)} \,)\\
\qquad\quad \mathbf{TYPEExpr}(\, \mathit{Ass}, \mathtt{x.sub(y)} \,)\\
\qquad\qquad \mathbf{TYPEExpr}(\, \mathit{Ass}, \mathtt{x} \,)\\
\qquad\qquad\quad \mathbf{Result}\ (\mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \emptyset)\\
\qquad\qquad \mathbf{TYPEExpr}(\, \mathit{Ass}, \mathtt{y} \,)\\
\qquad\qquad\quad \mathbf{Result}\ (\mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}, \emptyset)\\
\qquad\qquad C = \{\, \mu(\, \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{sub}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, (\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}})\,)\,\}\\
\qquad\qquad \mathbf{Result}\ ([[\mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}].\mathtt{sub}([\mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}]) : \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}], C)\\
\qquad\quad \mathbf{TYPEExpr}(\, \mathit{Ass}, \mathtt{z} \,)\\
\qquad\qquad \mathbf{Result}\ (\mathtt{z} : \alpha_{\mathtt{A}}^{\mathtt{mt},3}, \emptyset)\\
\qquad\quad C = C \cup \{\, \mu(\, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \mathtt{add}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}, (\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}})\,)\,\}\\
\qquad\quad \mathbf{Result}\ ([[[\mathtt{x} : \alpha_{\mathtt{A}}^{\mathtt{mt},1}].\mathtt{sub}([\mathtt{y} : \alpha_{\mathtt{A}}^{\mathtt{mt},2}]) : \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}].\mathtt{add}(\mathtt{z} : \alpha_{\mathtt{A}}^{\mathtt{mt},3}) : \gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}],\\
\qquad\qquad\qquad\qquad C) =: \mathtt{e_t}\\
\quad \mathbf{Result}(cl_t, C)\\
\quad \text{with}\quad cl_t := \mathtt{class\ A\ \{}\\
\qquad\qquad\qquad\qquad \gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}\ \ \mathtt{mt(}\alpha_{\mathtt{A}}^{\mathtt{mt},1}\ \mathtt{x,}\ \alpha_{\mathtt{A}}^{\mathtt{mt},2}\ \mathtt{y,}\ \alpha_{\mathtt{A}}^{\mathtt{mt},3}\ \mathtt{z)\ \{\ return\ e_t;\ \}}\\
\qquad\qquad\qquad \mathtt{\}}\\
\quad \text{and}\quad C = \{\, \mu(\, \alpha_{\mathtt{A}}^{\mathtt{mt},1}, \mathtt{sub}, \alpha_{\mathtt{A}}^{\mathtt{mt},2}, (\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \beta^{\mathtt{sub},1}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}})\,),\\
\qquad\qquad\qquad \mu(\, \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}, \mathtt{add}, \alpha_{\mathtt{A}}^{\mathtt{mt},3}, (\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}})\,)\,\}
\end{array}
$$

Second, **construct** is applied to **TYPE**'s result:

construct$(\, cl_t, C\,)$:

$C_{\mathtt{A}} = \{\, \alpha^{\mathtt{mt},2}_{\mathtt{A}} \lessdot \beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}},\;\; \alpha^{\mathtt{mt},3}_{\mathtt{A}} \lessdot \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}} \,\}$

new_interf $= \{\, \alpha^{\mathtt{mt},1}_{\mathtt{A}}, \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}} \,\}$

$\underline{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}$**:** For $\mu(\, \alpha^{\mathtt{mt},1}_{\mathtt{A}}, \mathtt{sub}, \alpha^{\mathtt{mt},2}_{\mathtt{A}}, (\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}, \beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}})\,) \in C$: the following interface is constructed:

`interface` $\alpha^{\mathtt{mt},1}_{\mathtt{A}}$ `<T, T1> { T sub(T1 x); }`

inh_tyterm $= \alpha^{\mathtt{mt},1}_{\mathtt{A}}\texttt{<}\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}, \beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}\texttt{>}$
$C_{\mathtt{A}} = C_{\mathtt{A}} \cup \{\, \mathtt{X1} \lessdot \text{inh\_tyterm} \,\}$
$\sigma = \{\, \alpha^{\mathtt{mt},1}_{\mathtt{A}} \mapsto \mathtt{X1} \,\}$

$\underline{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}$**:** For $\mu(\, \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}, \mathtt{add}, \alpha^{\mathtt{mt},3}_{\mathtt{A}}, (\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}})\,) \in C$ the following interface is constructed:

`interface` $\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}$`<T, T1> { T add(T1 x); }`

inh_tyterm $= \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}\texttt{<}\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}\texttt{>}$
$C_{\mathtt{A}} = C_{\mathtt{A}} \cup \{\, \mathtt{X2} \lessdot \text{inh\_tyterm} \,\}$
$\sigma = \sigma \cup \{\, \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}} \mapsto \mathtt{X2} \,\}$

tv $= \{\, \mathtt{X1}, \alpha^{\mathtt{mt},2}_{\mathtt{A}}, \alpha^{\mathtt{mt},3}_{\mathtt{A}}, \gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}} \,\}$

It holds $\quad \sigma = \{\alpha^{\mathtt{mt},1}_{\mathtt{A}} \mapsto \mathtt{X1},$
$\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}} \mapsto \mathtt{X2}\,\}$

and $\quad C_{\mathtt{A}} = \{\, \alpha^{\mathtt{mt},2}_{\mathtt{A}} \lessdot \beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}},$
$\alpha^{\mathtt{mt},3}_{\mathtt{A}} \lessdot \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}},$
$\mathtt{X1} \lessdot \alpha^{\mathtt{mt},1}_{\mathtt{A}}\texttt{<X2}, \beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}\texttt{>},$
$\mathtt{X2} \lessdot \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}\texttt{<}\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}, \beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}\texttt{>}\,\}$

The result class is given as:

`class A <X1,` $\alpha^{\mathtt{mt},2}_{\mathtt{A}}$`,` $\alpha^{\mathtt{mt},3}_{\mathtt{A}}$`,` $\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}$ `>[`

$\alpha^{\mathtt{mt},2}_{\mathtt{A}}$ `extends` $\beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}$
$\alpha^{\mathtt{mt},3}_{\mathtt{A}}$ `extends` $\beta^{\mathtt{add},1}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}}$ `,`

`X1 extends` $\alpha^{\mathtt{mt},1}_{\mathtt{A}}$`<X2,` $\beta^{\mathtt{sub},1}_{\alpha^{\mathtt{mt},1}_{\mathtt{A}}}$`>,`

```
    X2 extends $\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}$<$\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}}$,$\beta^{\mathtt{add,1}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}}$>
]
{
    $\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}}$ mt(X1 x, $\alpha^{\mathtt{mt,2}}_{\mathtt{A}}$ y, $\alpha^{\mathtt{mt,3}}_{\mathtt{A}}$ z) { return x.sub(y).add(z); }
}
```

As

$$C_{\mathtt{A}} = \{\, [\alpha^{\mathtt{mt,2}}_{\mathtt{A}} \lessdot \beta^{\mathtt{sub,1}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}],\ [\alpha^{\mathtt{mt,3}}_{\mathtt{A}} \lessdot \beta^{\mathtt{add,1}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}}],\ [\mathtt{X1} \lessdot \alpha^{\mathtt{mt,1}}_{\mathtt{A}}\mathtt{<X2}, \beta^{\mathtt{sub,1}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}\mathtt{>}],$$
$$[\mathtt{X2} \lessdot \gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}\mathtt{<}\gamma^{\mathtt{add}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}}, \beta^{\mathtt{add,1}}_{\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}}\mathtt{>}]\,\}$$

is in solved form, **TUnify** respectively **solve** changes nothing. This means the result of **construct** is the result of **TI**.

In the following we extend the example, such that an instance of class `A` is used. For this implementations of the inerfaces $\alpha^{\mathtt{mt,1}}_{\mathtt{A}}$ and $\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}$ must be given. We give one class `myInteger`, which implements both interfaces:

```
class myInteger extends $\alpha^{\mathtt{mt,1}}_{\mathtt{A}}$<myInteger, myInteger>,
                        $\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}$<myInteger, myInteger> {
    Integer i;
    myInteger sub(myInteger x) { return new myInteger(i - x.i); }
    myInteger add(myInteger x) { return new myInteger(i + x.i); } }
```

In the class `Main` an instance of `A` is used and the method `mt` is called.

```
class Main {
    main() { return new A<>()
                    .mt(new myInteger(2),
                        new myInteger(1),
                        new myInteger(3)); }
}
```

We call **TI** for `Main` with the set of assumptions *Ass* consisting of the class `A` and the class `myInteger`.

In **TYPEExpr**(Ass, `new A<>()`) the class `A` gets fresh type variables:

```
class A <$\nu_1$,$\nu_3$,$\nu_4$,$\nu_6$>
        [$\nu_3$ extends $\nu_5$,
         $\nu_4$ extends $\nu_7$,
         $\nu_1$ extends $\alpha^{\mathtt{mt,1}}_{\mathtt{A}}$<$\nu_2$,$\nu_5$>,
         $\nu_2$ extends $\gamma^{\mathtt{sub}}_{\alpha^{\mathtt{mt,1}}_{\mathtt{A}}}$<$\nu_6$,$\nu_7$>]{
     $\nu_6$ mt($\nu_1$ x, $\nu_3$ y, $\nu_4$ z) { return x.sub(y).add(z); }
}
```

The result of **TYPEExpr**(Ass, `new A<>()`) is: (`new A<>() :A<`$\nu_1, \nu_3, \nu_4, \nu_6$`>`, $C_{\mathtt{newA}}$)

with

$$C_{\mathtt{newA}} = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \alpha_{\mathtt{A}}^{\mathtt{mt},1}\texttt{<}\nu_2, \nu_5\texttt{>}, \nu_2 \lessdot \gamma^{\mathtt{m}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}\ \}$$

The constraint set of the result of **TYPE** is given as

$$\begin{gathered} C_{\mathtt{main}} = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \nu_1 \lessdot \alpha_{\mathtt{A}}^{\mathtt{mt},1}\texttt{<}\nu_2, \nu_5\texttt{>}, \nu_2 \lessdot \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}, \\ \mathtt{myInteger} \lessdot \nu_1, \mathtt{myInteger} \lessdot \nu_3, \mathtt{myInteger} \lessdot \nu_4\ \} \end{gathered}$$

The function **construct** adds no interfaces, as there is no call of abstract fields or methods. Therefore the result of **construct** is given as:

```
class Main<ν6> [C_Main] {
    ν6 main() {
        return new A<>()
               .mt(new myInteger(2), new myInteger(1), new myInteger(3));
    }
}
```

In **solve** $C_{\mathtt{main}}$ is unified: The class declarations implies

$$\mathtt{myInteger} \leq^* \alpha_{\mathtt{A}}^{\mathtt{mt},1}\texttt{<}\mathtt{myInteger}, \mathtt{myInteger}\texttt{>}$$

and

$$\mathtt{myInteger} \leq^* \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\texttt{<}\mathtt{myInteger}, \mathtt{myInteger}\texttt{>}.$$

With the *adapt2*-rule follows from $\mathtt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \alpha_{\mathtt{A}}^{\mathtt{mt},1}\texttt{<}\nu_2, \nu_5\texttt{>}$:

$$\begin{gathered} \mathtt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \alpha_{\mathtt{A}}^{\mathtt{mt},1}\texttt{<}\mathtt{myInteger}, \mathtt{myInteger}\texttt{>}, \\ \nu_2 \doteq \mathtt{myInteger}, \nu_5 \doteq \mathtt{myInteger}. \end{gathered}$$

From this follows with the *subst*-rule

$$\mathtt{myInteger} \lessdot \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}$$

and with the *adapt1*-rule:

$$\gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\texttt{<}\mathtt{myInteger}, \mathtt{myInteger}\texttt{>} \doteq \gamma^{\mathtt{sub}}_{\alpha_{\mathtt{A}}^{\mathtt{mt},1}}\texttt{<}\nu_6, \nu_7\texttt{>}.$$

With the *reduce1*- and the *swap*-rule we get:

$$\nu_6 \doteq \mathtt{myInteger}, \nu_7 \doteq \mathtt{myInteger}.$$

With the *subst*-rule follows from

$$\mathtt{myInteger} \lessdot \nu_3, \nu_3 \lessdot \mathtt{myInteger} \text{ and } \mathtt{myInteger} \lessdot \nu_4, \nu_4 \lessdot \mathtt{myInteger}$$

and from this with the *refl*-rule:

$$\nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger}.$$

The result of **solve** is given as:

$$\begin{array}{l} \{\, \texttt{myInteger} \lessdot \nu_1, \nu_1 \lessdot \alpha_{\texttt{A}}^{\texttt{mt,1}}\texttt{<myInteger,myInteger>} \\ \;\nu_2 \doteq \texttt{myInteger}, \nu_5 \doteq \texttt{myInteger}, \nu_6 \doteq \texttt{myInteger}, \\ \;\nu_7 \doteq \texttt{myInteger}, \nu_3 \doteq \texttt{myInteger}, \nu_4 \doteq \texttt{myInteger} \,\} \end{array}$$

The resulting Java class is given as:

```
class Main [ myInteger extends $\nu_1$,
             $\nu_1$ extends $\alpha_{\texttt{A}}^{\texttt{mt,1}}$<myInteger, myInteger> ]
{
    myInteger main() {
        return new A<>().mt(new myInteger(2),
                            new myInteger(1),
                            new myInteger(3)); }
}
```

There is one remaining type variable ν_1, that is not used in a argument- or return-type of a method. Therefore ν_1 is no class-parameter of `Main`. The two remaining bounds of ν_1 are consistent. This means `main` is executable. The result of the execution is 4.

5 Summary

We have presented a type inference algorithm for a Java-like language. The algorithm allows to declare type-less Java classes independently from any environment. This allows separate compilation of Java classes without relying on type informations of other classes. The algorithm infers structural types, that are given as generated interfaces. The instances have to implement these interfaces.

References

[ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 26–37, New York, NY, USA, 2005. ACM.

[IPW01] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *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 Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü15] Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.