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
\[ E \text{ m}(B \ x)\{ \text{return } x.f1.f2; \} \]
is given. The compilation algorithm generates the polymorphic typed Java expressions
\[ E \text{ m}(B \ x)\{ \text{return } [[x:B].f1:α].f2:β; \}, \]
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 β ≤∗ E. In this approach B and E are still nominal types.

We generalize this approach, such that also untyped methods like
\[ m(x)\{ \text{return } x.f1.f2; \} \]
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 process is reduced to a simple check 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.

```java
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

Input syntax:

\[
\begin{align*}
L & ::= \text{class } C \text{ extends } (CT)^* \{ T; R \} \\
M & ::= m(x) \{ \text{return } e; \} \\
e & ::= x | e.f | e.m(\pi) | \text{new } NCT(\pi) | (CT)e \\
NCT & ::= CT | C<TVar = T> \\
T & ::= CT | TVar \\
CT & ::= C<TVar>
\end{align*}
\]

Output syntax:

\[
\begin{align*}
L_t & ::= I^* \ CL_t \\
CL_t & ::= \text{class } C<TVar>[CONS] \text{ extends } (CT)^* \{ T_T, M_t \} \\
CONS & ::= T \text{ extends } T \\
MH & ::= T m(\overline{T} x); \\
M_t & ::= MH \{ \text{return } e_t; \} \\
e_t & ::= x : T | e.f : T | e.m(\overline{\pi}) : T | \text{new } NCT(\overline{\pi}) : CT | (CT)e : CT \\
I & ::= \text{interface } I<TVar>[\overline{T}, \overline{MH}]
\end{align*}
\]

Fig. 1. The syntax

Java program \(L\) and the output is a typed Java program \(L_t\), including generated interfaces. There are some extensions in comparison to usual Java. The class declarations in the output syntax have the form \textit{class } C<TVar> [CONS]. \(TVar\) are the generics
and \([\text{CONS}]\) is a set of subtype constraints \(T \text{ extends} \ T'\), 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 \text{extends} and \text{implements} declarations. Both are declared by \text{extends}. Interfaces can have fields. Furthermore, the use of the new-statement is allowed without assigning all generics. This is done by the syntax \(C<T\text{Var} = CT>\). The not assigned generics are derived by the type inference algorithm.

3 The algorithm

The algorithm \(\text{TI}\) consists of three parts. First the function \(\text{TYPE}\) inserts types (usually type variables) to all sub-terms and collects constraints. Second, the function \(\text{construct}\) generates the interfaces and completes the constraints. Finally, the function \(\text{solve}\) 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 \(\text{TYPE}\). This definition is oriented at \([\text{ADDZ05}]\):

**Definition 1. (Type constraints)**

- \(c \ll c'\) means \(c\) has to be a subtype of \(c'\).
- \(\phi(c, f, c')\) means \(c\) provides a field \(f\) with type \(c'\).
- \(\mu(c, m, \pi, (c', \pi'))\) means \(c\) provides a method \(m\) applicable to arguments of type \(\pi\), with return type \(c'\) and parameters of type \(\pi'\).

Note that \(\mu(c, m, \pi, (c', \pi'))\) implicitly includes the constraints \(\pi \leq \pi'\).

Let \(<\) be the extends relation defined by the \text{Java} declarations und \(\leq^*\) the corresponding subtyping relation.

The type–inference algorithm

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

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

- \(\delta^f_A\): Type variable for the field \(f\) in the class \(A\).
- \(\alpha^{n,1}_A, \beta^{n,1}_A\): Type variable for the \(i\)-th argument of the method \(m\) in the class \(A\).
- \(\alpha^1_A, \beta^1_A\): is an abbreviation for the tuple \(\alpha^{n,1}_A, \ldots, \alpha^{n,m}_A\) respectively \(\beta^{n,1}_A, \ldots, \beta^{n,m}_A\).
- \(\gamma^m_A\): 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 L_t is the typed Java class extended by a set of interfaces.

TI: TypeAssumptions × L → L_t

TI (Ass, class A extends B {f; M}) =
let
  (cl_t, C) = Type(Ass, cl)
  (I1...I_m cl_t) = construct(cl_t, C)
in
  (I1...I_m solve(cl_t))

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

TYPE: TypeAssumptions × L → L_t × ConstraintsSet

TYPE(Ass, class A extends B {f; M}) = let
  fass = {this.f : δ_f A | f ∈ T} ∪ {this.f : T | T f ∈ fields(B)}
  mass = {this.m: α_m A → γ_m A | m(φ) { return e; } ∈ R} ∪
  {this.m: aty → rty | mtype(m, B) = aty → rty}
  AssAll = Ass ∪ fass ∪ mass ∪ {this : A}
  For m(φ) { return e; } ∈ R {
    Ass = AssAll ∪ {x_j : α_j A | x = x_1...x_n}
    (e_t : rty, C') = TYPEExpr(Ass, e)
    C = (C ∪ C') [γ_m A → rty]
  }R_t = {rty m(φ) x | m(φ) { return e; } ∈ R}
  in(class A extends B {δ_f A; K_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: TypeAssumptions × e → e_t × ConstraintsSet

TYPEExpr(Ass, x) = let (x : θ) ∈ Ass in (x : θ, φ)

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, e.f) =
let
  (e_t : ty, C) = TYPEExpr(Ass, e)
in
  if (ty is no type variable) & & (ty ∈ Ass) & & (rty f ∈ fields(ty))
  then (((e_t : ty).f) : rty, C)
  else (((e_t : ty).f) : δ_f, {φ(ty, f, δ_f) | 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 subtyperelation is introduced. Otherwise a constraint is generated that demands a corresponding method in the type.

\[
\text{TYPEExpr} \left( \text{Ass}, e_0, m(e) \right) = \begin{cases} 
\text{let} & (e_0, : ty_0, C_0) = \text{TYPEExpr}(\text{Ass}, e_0) \\
(e_i, : ty_i, C_i) = \text{TYPEExpr}(\text{Ass}, e_i), \forall 1 \leq i \leq n 
\end{cases}
\]

if \( (ty_0 \text{ is no type variable}) \& \& (ty_0 \in \text{Ass}) \& \& (\text{mtype}(m, ty_0) = aty \to rty) \)

then \( ((e_0, : ty_0).m(e_1, : ty_1, \ldots, e_n, : ty_n) : rty, (C_0 \cup \bigcup_i C_i) \cup \{ ty \leq aty \}) \)

else \( ((e_0, : ty_0).m(e_1, : ty_1, \ldots, e_n, : ty_n) : \gamma^a_{ty_0}, (C_0 \cup \bigcup_i C_i) \cup \{ \mu(ty_0, m, \gamma^a_{ty_0}, \beta^a_{ty_0}) \}) \)

where \( \beta^a_{ty_0} \) and \( \gamma^a_{ty_0} \) 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 \( \text{C } \text{TVar } = \text{CT} \). 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.

\[
\text{TYPEExpr} \left( \text{Ass} \cup \{ \text{class } A_<T>[C_A] \ \text{extends } B \{ \text{T}_A T; \text{K}_A \} \}, \text{new } A<S>(\sigma) \right) = \begin{cases} 
\text{let} & S' = S[\text{Ty} \mapsto \nu] \\
(e_i, : ty_i, C_i) = \text{TYPEExpr}(\text{Ass}, e_i), \forall 1 \leq i \leq m 
\end{cases}
\]

in \( (\text{new } A<S>(e_1, : ty_1, \ldots, e_n, : ty_n) : A<S>[\nu \mapsto \tau] \mid \nu = \tau \in S'), \bigcup_i C_i \cup \bigcup \{ \text{fields}(B) = T_F \} \)

where fields(\( B \)) = \( T_F \)

The function construct The function construct takes the result from TYPE, a typed class and a set of constraints. It generates for any type \( ty_1, ty_2 \) occurring
in constraints \( \phi(ty_1, f, \delta) \) or \( \mu(ty_2, m, \overline{\alpha}, (\gamma, \overline{\beta})) \) corresponding interfaces with the demanded fields and methods.

**construct** : \( L_t \times \text{ConstraintsSet} \rightarrow L_t \)

\[
\text{construct}(\text{class} A \text{ extends } B \{ \delta_A f; \bar{\alpha} \}, C) \{
C_A = \{ \alpha < \beta | \alpha < \beta \in C \} \cup \{ \alpha < \beta | \mu(ty, m, \overline{\alpha}, (\gamma, \overline{\beta})) \in C \}
\]

if \( C_A \) contains a constraint \( ty \leq ty' \), where \( ty \) and \( ty' \)
are not type variables and \( ty \not\leq \ast ty' \) then fail exit

new_interf = \{ \iota | \phi(\iota, f, \gamma) \in C \} \cup \{ \iota | \mu(\iota, m, \overline{\beta}, (\gamma, \overline{\beta}')) \in C \}

For \( \iota \in \text{new_interf} \{
I_\iota = \text{interface } \iota < > \{ \} \) is generated

inh_tyterm = "\iota < >"

For \( f \) with \( \phi(\iota, f, \delta) \in C \{
An field \( f \) and a fresh type variable \( T \) is added to the interface \( \iota \):

\[
I_\iota = \text{interface } \iota < \text{args, } T > \{ \ldots \ T f; \ldots \}
\]

\( T \) is instantiated by \( \delta \) in inh_tyterm: inh_tyterm = "\iota < \text{args}', \delta >"

For \( m \) with \( \mu(\iota, m, \overline{\beta}, (\gamma, \overline{\beta}'_1), \ldots, \overline{\beta}'_n) \in C \{
A method signature for \( m \) and fresh type variable \( T \), \( T \) are introduced into the interface \( \iota \):

\[
I_\iota = \text{interface } \iota < \text{args, } T, T_1, \ldots, T_n > \{
\text{fields}
\text{meth_sigs}
T \ m(T_1, \ldots, T_n);
\}
\]

\( T \) and \( T \) are instantiated by \( \gamma \) and \( \overline{\beta}' \) in inh_tyterm:

inh_tyterm = "\iota < \text{args}', \gamma, \overline{\beta}' >"

\[
C_h = C_A[\iota \mapsto X] \cup \{ X < \text{inh_tyterm} \}, \text{where } X \text{ is a fresh type variable}
\]

\[
\sigma = \sigma \cup \{ \iota \mapsto X \}
\]

\[
tv = \text{typevar}(\overline{\sigma(\delta_A)}) \cup \text{typevar}(\overline{\sigma(M_t)})
\]

return \( I_\iota \) class \( A<tv>[C_A] \) extends \( B \{ \delta_A f; \bar{\alpha} \} \)

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 \( T < ty \) are solved by substituting \( T \) by all subtypes of \( ty \). Now we give a different version of the algorithm where \( T < ty \) is not solved.
Lemma 1 (Termination). The algorithm \( \text{TUnify}(C) \) is given by the rules (Figure 2) application the most often as possible. If \( C \) contains finally of pairs \( T \doteqTy \), \( T \lt Ty \), or \( Ty \lt T \) then \( C \) is the result, otherwise the algorithm fails.

Lemma 2 (Soundness of \( \text{TUnify} \)). If a substitution \( \sigma \) is a solution of a constraint set \( C \) then \( \sigma \) is also a solution of \( \text{TUnify}(C) \).
Lemma 3 (Completeness). Let C be a set of constraints C and α' a solution. For σ = { T ↦ ty | T ⊆ ty ∈ TUnify(C) } there are substitutions σ'' and σ_{rest}, such that σ' = σ'' ∘ ((σ_{rest} ∘ σ) ∪ σ_{rest}).

Remark 1 (Most general unifier of TUnify). The substitution σ_{rest} is a solution of the remaining pairs T << ty and ty << T. For any solution σ' there is substitution σ'_{rest}, such that σ'_{rest} ∘ σ 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 application are the same.

adapt1: From the type term construction follows iff application are the same.

adapt2: As ≼ * 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 ≼ * 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: L_t → L_t

solve(\texttt{class A<}\texttt{T}>[C_k] \text{ extends B } \{ \texttt{ty f} ; \texttt{M_c} \} ) =

\begin{align*}
\text{let} & \quad \text{subst} = \text{TUnify}(C_k) \\
& \quad \sigma = \{ T ↦ ty | T ⊆ ty ∈ \text{subst} \} \\
& \quad cs = \{ T << ty | T ⊆ ty ∈ \text{subst} \} ∪ \{ ty << T | ty ⊆ T ∈ \text{subst} \} \\
& \quad T_{\text{new}} = \bigcup \{ \text{\texttt{return o; \text{ty}} | return o; \text{ty} ∈ \sigma(\text{M_c})} \} (\text{\texttt{TVar(\text{ty})}} \cup \text{\texttt{TVar(\text{aty})}}) \\
\text{in} & \quad \text{if is\_solvable}(cs) \text{ then} \\
& \quad \text{class A<}\texttt{T}_{\text{new}>}[cs] \text{ extends B } \{ \texttt{\sigma (ty) f}; \texttt{\sigma (M_c)} \} \\
& \quad \text{else fail}
\end{align*}
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

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

First **TYPE** is applied:

**TYPE**($\emptyset$, A)

$$ mass = \{ \text{this} \times mt: (\alpha_A^{nt.1}, \alpha_A^{nt.2}, \alpha_A^{nt.3}) \rightarrow \gamma \} $$

AssAll = mass $\cup$ \{ this: A \}

$$ \text{mt} = \text{AssAll} \cup \{ x: \alpha_A^{nt.1}, y: \alpha_A^{nt.2}, z: \alpha_A^{nt.3} \} $$

**TYPEExpr**(Ass, x.sub(y).add(z))

**TYPEExpr**(Ass, x)

$$ \text{Result} (x: \alpha_A^{nt.1}, 0) $$

**TYPEExpr**(Ass, y)

$$ \text{Result} (y: \alpha_A^{nt.2}, 0) $$

$$ C = \{ \mu(\alpha_A^{nt.1}, \text{sub}, \alpha_A^{nt.2}, (\gamma^{sub}_{\alpha_A^{nt.1}}, \beta^{sub}_{\alpha_A^{nt.1}})) \} $$

$$ \text{Result} ([x: \alpha_A^{nt.1}, \text{sub}([y: \alpha_A^{nt.2}]), \gamma^{sub}_{\alpha_A^{nt.1}], C]) $$

**TYPEExpr**(Ass, z)

$$ \text{Result} (z: \alpha_A^{nt.3}, 0) $$

$$ C = C \cup \{ \mu(\gamma^{sub}_{\alpha_A^{nt.1}}, \text{add}, \alpha_A^{nt.3}, (\gamma^{add}_{\alpha_A^{nt.1}}, \beta^{add}_{\alpha_A^{nt.1}})) \} $$

$$ \text{Result} ([[[x: \alpha_A^{nt.1}, \text{sub}([y: \alpha_A^{nt.2}]), \gamma^{sub}_{\alpha_A^{nt.1}], \text{add}(z: \alpha_A^{nt.3}), \gamma^{add}_{\alpha_A^{nt.1}]}, C)] =: e_t $$

Result($c_t$, C)

with $c_t := \text{class A}$

$$ \gamma_{\alpha_A^{nt.1}}^{add}_{\alpha_A^{nt.1}} \times (\alpha_A^{nt.2}, \alpha_A^{nt.3}, z) \{ \text{return } e_t; \} $$

and

$$ C = \{ \mu(\alpha_A^{nt.1}, \text{sub}, \alpha_A^{nt.2}, (\gamma^{sub}_{\alpha_A^{nt.1}}, \beta^{sub}_{\alpha_A^{nt.1}})), $$

$$ \mu(\gamma^{sub}_{\alpha_A^{nt.1}}, \text{add}, \alpha_A^{nt.3}, (\gamma^{add}_{\alpha_A^{nt.1}}, \beta^{add}_{\alpha_A^{nt.1}})) \} $$

Second, **construct** is applied to **TYPE**'s result:
\textbf{construct}(c_i, C) :
\[ C_A = \{ \alpha_{mt}^{2} < \beta_{sub}^{1}, \alpha_{mt}^{3} < \beta_{add}^{1}, \alpha_{sub}^{1} < \alpha_{sub}^{1} \} \]

\text{new\_interf} = \{ \alpha_{mt}^{1}, \alpha_{sub}^{1} \}

\alpha_{mt}^{1} : \text{For } \mu(\alpha_{mt}^{1}, \beta_{sub}^{1}, \alpha_{mt}^{2}, \beta_{add}^{1}) \in C: \text{the following interface is constructed:}

\text{interface } \alpha_{mt}^{1} < T, T1 > \{ T \text{sub}(T1 x); \}

\text{inh\_tyterm} = \alpha_{mt}^{1} < \gamma_{sub}^{1}, \beta_{sub}^{1} > \alpha_{mt}^{1} \gamma_{sub}^{1} \beta_{sub}^{1}

C_A = C_A \cup \{ X1 < \text{inh\_tyterm} \}

\sigma = \{ \alpha_{mt}^{1} \mapsto X1 \}

\text{interface } \gamma_{sub}^{1} < T, T1 > \{ T \text{add}(T1 x); \}

\gamma_{sub}^{1} : \text{For } \mu(\gamma_{sub}^{1}, \beta_{add}^{1}, \gamma_{sub}^{1}) \in C: \text{the following interface is constructed:}

\text{interface } \gamma_{sub}^{1} < T, T1 > \{ T \text{add}(T1 x); \}

\text{inh\_tyterm} = \gamma_{sub}^{1} < \gamma_{add}^{1}, \beta_{add}^{1} > \gamma_{sub}^{1} \beta_{add}^{1}

C_A = C_A \cup \{ X2 < \text{inh\_tyterm} \}

\sigma = \sigma \cup \{ \gamma_{sub}^{1} \mapsto X2 \}

tv = \{ X1, \alpha_{mt}^{2}, \gamma_{sub}^{1} \}

\text{It holds } \sigma = \{ \alpha_{mt}^{1} \mapsto X1, \gamma_{sub}^{1} \mapsto X2 \}

\text{and } C_A = \{ \alpha_{mt}^{2} < \beta_{sub}^{1}, \alpha_{mt}^{3} < \beta_{add}^{1}, \gamma_{sub}^{1} < \gamma_{sub}^{1} \}

X1 < \alpha_{mt}^{1} < X2, \beta_{sub}^{1}, \gamma_{sub}^{1} \beta_{add}^{1}

X2 < \gamma_{sub}^{1} < \gamma_{add}^{1}, \gamma_{sub}^{1} \gamma_{sub}^{1} \}

The result class is given as:

\textbf{class } A < X1, \alpha_{mt}^{2}, \alpha_{mt}^{3}, \alpha_{sub}^{1}, \gamma_{sub}^{1} >\[
\alpha_{mt}^{2} \text{ extends } \beta_{sub}^{1} \alpha_{mt}^{1}
\alpha_{mt}^{3} \text{ extends } \beta_{add}^{1} \gamma_{sub}^{1} \alpha_{mt}^{1}
X1 \text{ extends } \alpha_{mt}^{1} < X2, \beta_{sub}^{1} \alpha_{mt}^{1} \}
X2 extends \( \gamma_{\text{add}}^{\text{sub}} \alpha_{\text{mt},1} \gamma_{\text{add}}^{\text{sub}} > \)

\[
\gamma_{\text{add}}^{\text{sub}} \alpha_{\text{mt},1} \gamma_{\text{add}}^{\text{sub}} >
\]

\[
\alpha_{\text{mt},1} \gamma_{\text{add}}^{\text{sub}} >
\]

\[
\alpha_{\text{mt},1} \gamma_{\text{add}}^{\text{sub}} >
\]

\[
\alpha_{\text{mt},1} \gamma_{\text{add}}^{\text{sub}} >
\]

As

\[
C_A = \left\{ \begin{array}{l}
[\alpha_{\text{mt},2} < \beta_{\text{sub},1}]_A, \\
[\alpha_{\text{mt},3} < \beta_{\text{add},1}]_A, \\
[X1 < \alpha_{\text{mt},1} \beta_{\text{add},1}]_A,
\end{array} \right\}
\]

\[
[X2 < \gamma_{\text{add}}^{\text{sub}} \alpha_{\text{mt},1} \gamma_{\text{add}}^{\text{sub}} >]_A
\]

is in solved form, \textbf{T}Unify respectively solve changes nothing. This means the result of \textbf{construct} is the result of \textbf{TI}.

In the following we extend the example, such that an instance of class \( A \) is used. For this implementations of the inerfaces \( \alpha_{\text{mt},1} \) and \( \gamma_{\text{sub}}^{\text{add}} \) must be given. We give one class \textit{myInteger}, which implements both interfaces:

```java
class myInteger extends \alpha_{\text{mt},1} A <\text{myInteger, myInteger}, \\
\gamma_{\text{sub}}^{\text{add}} \alpha_{\text{mt},1} >
{
    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 \textit{Main} an instance of \( A \) is used and the method \texttt{mt} is called.

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

We call \textbf{TI} for \textit{Main} with the set of assumptions \( \text{Ass} \) consisting of the class \( A \) and the class \textit{myInteger}.

In \textbf{TYPEExpr} \( \langle \text{Ass, new A<>() } \rangle \) the class \( A \) gets fresh type variables:

```java
class A \langle \nu_1, \nu_2, \nu_4, \nu_6 \rangle
{ \nu_3 \text{ extends } \nu_3,
    \nu_4 \text{ extends } \nu_7,
    \nu_1 \text{ extends } \alpha_{\text{mt},1} \nu_2, \nu_6,
    \nu_2 \text{ extends } \gamma_{\text{add}}^{\text{sub}} \alpha_{\text{mt},1} \nu_3, \nu_7 \rangle{
    \nu_6 \text{ mt}(\nu_1 x, \nu_3 y, \nu_4 z) { \return x \text{.sub}(y).add(z); }
}
```
The result of \texttt{TYPEExpr(\texttt{Ass, new A<>()})} is: \texttt{(new A<>():A<\nu_1, \nu_3, \nu_4, \nu_6>, C_{\text{new}})}
with
\[
C_{\text{new}} = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \alpha_{nt,1}^A < \nu_2, \nu_5>, \nu_2 < \gamma_{nt,1}^{\text{sub}} < \nu_6, \nu_7> \} \]

The constraint set of the result of \texttt{TYPE} is given as
\[
C_{\text{main}} = \{ \nu_3 < \nu_5, \nu_4 < \nu_7, \nu_1 < \alpha_{nt,1}^A < \nu_2, \nu_5>, \nu_2 < \gamma_{nt,1}^{\text{sub}} < \nu_6, \nu_7>,
\quad \text{myInteger} \preceq \nu_1, \text{myInteger} \preceq \nu_3, \text{myInteger} \preceq \nu_4 \} \]

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

```java
class Main<\nu_6> [C_{\text{main}}] {
    \nu_6 main() {
        return new A<>()
            .mt(new myInteger(2), new myInteger(1), new myInteger(3));
    }
}
```

In \texttt{solve} \texttt{C_{\text{main}}} is unified: The class declarations implies
\[
\text{myInteger} \preceq \alpha_{nt,1}^A < \text{myInteger, myInteger} \]
and
\[
\text{myInteger} \preceq \gamma_{nt,1}^{\text{sub}} < \text{myInteger, myInteger}. \]

With the \texttt{adapt2}-rule follows from \texttt{myInteger} \preceq \nu_1, \nu_1 < \alpha_{nt,1}^A < \nu_2, \nu_5:
\[
\text{myInteger} \preceq \nu_1, \nu_1 < \alpha_{nt,1}^A < \text{myInteger, myInteger},
\quad \nu_2 \doteq \text{myInteger}, \nu_5 \doteq \text{myInteger}. \]

From this follows with the \texttt{subst-rule}
\[
\text{myInteger} \preceq \gamma_{nt,1}^{\text{sub}} < \nu_6, \nu_7
\]
and with the \texttt{adapt1}-rule:
\[
\gamma_{nt,1}^{\text{sub}} < \text{myInteger, myInteger} \doteq \gamma_{nt,1}^{\text{sub}} < \nu_6, \nu_7. \]

With the \texttt{reduce1}- and the \texttt{swap}-rule we get:
\[
\nu_6 \doteq \text{myInteger}, \nu_7 \doteq \text{myInteger}. \]

With the \texttt{subst-rule} follows from
\[
\text{myInteger} \preceq \nu_3, \nu_3 \preceq \text{myInteger} \text{ and myInteger} \preceq \nu_4, \nu_4 \preceq \text{myInteger} \]
and from this with the refl-rule:
\[ \nu_3 = \text{myInteger}, \nu_4 = \text{myInteger}. \]

The result of \texttt{solve} is given as:
\[
\begin{align*}
\nu_1 \vdash \text{myInteger}, & \nu_1 < \alpha_{\text{nt},1} \times \text{myInteger, myInteger} \\
\nu_2 \vdash \text{myInteger}, & \nu_5 \vdash \text{myInteger}, \nu_6 \vdash \text{myInteger}, \\
\nu_7 \vdash \text{myInteger}, & \nu_3 \vdash \text{myInteger}, \nu_4 \vdash \text{myInteger} \\
\end{align*}
\]

The resulting \texttt{Java} class is given as:
\[
\text{class Main [ myInteger extends } \nu_1, \\
& \quad \nu_1 \text{ extends } \alpha_{\text{nt},1} \times \text{myInteger, myInteger} \text{ ]}
\{
\begin{align*}
\text{myInteger } \text{main}() \{} \\
& \quad \text{return new A<>()\text{.mt(new myInteger(2),} \\
& \qquad \text{new myInteger(1),} \\
& \qquad \text{new myInteger(3))}; \}
\end{align*}
\}
\]

There is one remaining type variable \(\nu_1\), that is not used in a argument- or return-type of a method. Therefore \(\nu_1\) is no class-parameter of \texttt{Main}. The two remaining bounds of \(\nu_1\) are consistent. This means \texttt{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 \texttt{Java} classes independently from any environment. This allows separate compilation of \texttt{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


[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.

