Structural type inference in Java-like languages – Extended abstract –

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
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

Let us consider an example that shows the idea. In [Plü15] 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.

In this paper we give an algorithm that infers for `v` a structural type α, which has a method `elementAt`. Our algorithm 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 $\beta \leq^* \texttt{E}$. 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.

The idea

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

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 $\texttt{L}_t$, including generated interfaces.

Input syntax :

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

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

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

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

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

Output syntax :

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

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

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

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

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

$\texttt{M}_t ::= \texttt{MH}\ \{\ \texttt{return e}_\texttt{t};\ \}$

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

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

Figure 1: The syntax

There are some extensions in comparison to usual Java. The class declarations in the output syntax have the form `class C<`$\overline{\texttt{TVar}}$`> [CONS]`. $\overline{\texttt{TVar}}$ are the generics and `[CONS]` is a set of subtype constraints `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 `C<`$\overline{\texttt{TVar = CT}}$`>`. 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 is 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:

$\delta^{\texttt{f}}_{\texttt{A}}$**:** Type variable for the field `f` in the class `A`.

$\alpha^{\texttt{m,i}}_{\texttt{A}}, \beta^{\texttt{m,i}}_{\texttt{A}}$**:** Type variable for the i-th argument of the method `m` in the class `A`.

$\overline{\alpha^{\texttt{m}}_{\texttt{A}}}, \overline{\beta^{\texttt{m}}_{\texttt{A}}}$**:** is an abbreviation for the tuple $\alpha^{\texttt{m,1}}_{\texttt{A}}, \ldots, \alpha^{\texttt{m,n}}_{\texttt{A}}$ respectively $\beta^{\texttt{m,1}}_{\texttt{A}}, \ldots, \beta^{\texttt{m,n}}_{\texttt{A}}$.

$\gamma^{\texttt{m}}_{\texttt{A}}$**:** 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 $(\ Ass, \mathtt{class\ A\ extends\ \overline{B}\ \{\ \overline{f};\ \overline{M}\ \}\ }) =$
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: $\mathtt{TypeAssumptions} \times \mathtt{L} \to \mathtt{L}_t \times \mathtt{ConstraintsSet}$

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

The function **TYPEExpr** inserts types into the expressions and collects the corresponding constraints. The function **TYPEExpr** is given for all cases of expressions `e`. In the following we present **TYPEExpr** for the both most important cases, the method-call and the new-statement.

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.

$\mathbf{TYPEExpr}(\ Ass, \mathtt{e_0.m}(\overline{e})\ ) = \mathbf{let}$
$(\mathtt{e}_{0_t} : ty_0, C_0) = \mathbf{TYPEExpr}(\ Ass, \mathtt{e_0}\ )$
$(\mathtt{e}_{i_t} : ty_i, C_i) = \mathbf{TYPEExpr}(\ Ass, \mathtt{e_i}\ ), \forall 1 \leqslant i \leqslant n$
in
if $(ty_0$ *is no type variable*$)\ \&\&\ (ty_0 \in Ass)\ \&\&\ (mtype(\ \mathtt{m}, ty_0\ ) = \overline{aty} \to rty)$ **then**
$((\mathtt{e}_{0_t} : ty_0).\mathtt{m}(\mathtt{e}_{1_t} : ty_1, \ldots, \mathtt{e}_{n_t} : ty_n) : rty, (C_0 \cup \bigcup_i C_i) \cup \{\ \overline{ty \lessdot aty}\ \})$
else
$((\mathtt{e}_{0_t} : ty_0).\mathtt{m}(\mathtt{e}_{1_t} : ty_1, \ldots, \mathtt{e}_{n_t} : ty_n) : \gamma^{\mathtt{m}}_{\mathtt{ty}_0}, (C_0 \cup \bigcup_i C_i) \cup \{\ \mu(\ ty_0, \mathtt{m}, \overline{ty}, (\gamma^{\mathtt{m}}_{\mathtt{ty}_0}, \overline{\beta^{\mathtt{m}}_{\mathtt{ty}_0}})\ )\ \})$

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.

$\mathbf{TYPEExpr}(\ Ass \cup \{\ \mathtt{class\ A<}\overline{\mathtt{T}}\mathtt{>[}C_{\mathtt{A}}\mathtt{]\ extends\ \overline{B}\ \{}\overline{\mathtt{T_A\ f}};\ \overline{\mathtt{M_t}}\mathtt{\}\ }\}, \mathtt{new\ A<}S\mathtt{>}(\ \overline{e}\ )\ ) =$
where $S = [\mathtt{T}_{\pi(1)} = \tau_1, \ldots, \mathtt{T}_{\pi(\mathtt{k})} = \tau_k]$ with $k \leq n$ for $|\overline{\mathtt{T}}| = n$
let
$\overline{\nu}$ fresh type variables, that substitute $\overline{\mathtt{T}}$ in class `A`
$S' = S[\overline{\mathtt{T} \mapsto \nu}]$

$(\texttt{e}_{\texttt{i}_\texttt{t}} : ty_i, C_i) = \textbf{TYPEExpr}(\, Ass, e_i\,), \forall 1 \leqslant i \leqslant m$

in

$(\texttt{new A<S'>(}\, \texttt{e}_{\texttt{1}_\texttt{t}}\texttt{:ty}_\texttt{1}, \ldots, \texttt{e}_{\texttt{m}_\texttt{t}}\texttt{:ty}_\texttt{m}\,\texttt{)}\,\texttt{:A<}\overline{\nu}[\nu \mapsto \tau \mid \nu = \tau \in S']\texttt{>}, (\bigcup_i C_i) \cup C_\texttt{A}[\overline{\nu \mapsto \tau}] \cup \{\, \overline{ty} \lessdot \overline{\texttt{T}_{\overline{\texttt{B}}}\,\texttt{T}_\texttt{A}[\overline{\nu \mapsto \tau}]}\,\}$
where $\mathit{fields}(\,\overline{\texttt{B}}\,) = \overline{\texttt{T}_{\overline{\texttt{B}}}\,\texttt{g}}$

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 $\phi(\, ty1, \texttt{f}, \delta\,)$ or $\mu(\, ty2, \texttt{m}, \overline{\alpha}, (\gamma, \overline{\beta})\,)$ corresponding interfaces with the demanded fields respectively methods:

```
interface ty1< ...,δ, ...>  {
    δ f;
}

interface ty2 <...,γ, β̄, ...> {
    γ m(β̄ x1);
},
```

introduces fresh type variables `X1`, `X2` and constraints, that have to implement these interfaces: `X1` $\lessdot$ `ty1<...>`, `X2` $\lessdot$ `ty2<...>`. Finally, the occuring type variables are introduced as generics of the class.

The function solve

The function **solve** takes the result of **construct** and solves the constraints of the class by the type unification algorithm from [Plü09], such that the constraints contains only pairs with at least one type variable.

Now 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 2 In this example we print the input syntax (user written) in black and the output syntax (automatically generated) in gray. Let the following class be given

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

The result of **TYPE** is: `class A {` $\gamma^{\texttt{add}}_{\gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}}$ `mt(`$\alpha^{\texttt{mt},1}_\texttt{A}$ `x,` $\alpha^{\texttt{mt},2}_\texttt{A}$ `y,` $\alpha^{\texttt{mt},3}_\texttt{A}$ `z) { return` $\texttt{e}_\texttt{t}$`; } }`, with

$\texttt{e}_\texttt{t} = [[[\texttt{x} : \alpha^{\texttt{mt},1}_\texttt{A}].\texttt{sub}([\texttt{y} : \alpha^{\texttt{mt},2}_\texttt{A}]) : \gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}].\texttt{add}(\texttt{z} : \alpha^{\texttt{mt},3}_\texttt{A}) : \gamma^{\texttt{add}}_{\gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}}]$ and

$C = \{\, \mu(\, \alpha^{\texttt{mt},1}_\texttt{A}, \texttt{sub}, \alpha^{\texttt{mt},2}_\texttt{A}, (\gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}, \beta^{\texttt{sub},1}_{\alpha^{\texttt{mt},1}_\texttt{A}})\,),\quad \mu(\, \gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}, \texttt{add}, \alpha^{\texttt{mt},3}_\texttt{A}, (\gamma^{\texttt{add}}_{\gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}}, \beta^{\texttt{add},1}_{\gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}})\,)\,)\,\}$

The result of **construct**$(\, cl_t, C\,)$ is (with fresh type variables):

```
interface α^mt,1_A <Gamma_m, Beta_m> { Gamma_m sub(Beta_m x); }

interface γ^sub_{α^mt,1_A}<Gamma_n, Beta_n> { Gamma_n add(Beta_n x); }

class A <ν1,ν2,ν3,ν4,ν5,ν6,ν7>[ν3 extends ν5, ν4 extends ν7, ν1 extends α^mt,1_A<ν2,ν5>, ν2 extends γ^sub_{α^mt,1_A}<ν6,ν7>]{
    ν6 mt(ν1 x, ν3 y, ν4 z) { return x.sub(y).add(z); }
}
```

As the application of **solve** changes nothing, this is the result of **TI**'s application.

In the following we show, as an instance of the type inferred class can be used.
For this implementations of $\alpha^{\texttt{mt},1}_\texttt{A}$ and $\gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_\texttt{A}}$ must be given:

```
class myInteger extends α^{mt,1}_A<myInteger, myInteger>, γ^{sub}_{α^{mt,1}_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<ν1=myInteger, ν2=myInteger>()
                    .mt(new myInteger(2), new myInteger(1), new myInteger(3)); } }
```

The mappings ν_1=`myInteger`, ν_2=`myInteger` means that ν_1 and ν_2 are instantiated and all other parameters of `A` should be inferred by the type inference algorithm **TI**. We call **TI** for `Main` with the set of assumptions consisting of the class `A` and the class `myInteger`.

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

$$C_{\texttt{main}} = \{\ \nu_3 \lessdot \nu_5, \nu_4 \lessdot \nu_7, \texttt{myInteger} \lessdot \alpha^{\texttt{mt},1}_{\texttt{A}}\texttt{<myInteger},\nu_5\texttt{>}, \texttt{myInteger} \lessdot \gamma^{\texttt{sub}}_{\alpha^{\texttt{mt},1}_{\texttt{A}}}\texttt{<}\nu_6,\nu_7\texttt{>},$$
$$\texttt{myInteger} \lessdot \nu_3, \texttt{myInteger} \lessdot \nu_4,\ \}$$

The functions **construct** adds no interfaces, as there is no call of abstract fields or methods.
In **solve** $C_{\texttt{main}}$ is unified. The result of the unification is:

$$\sigma = \{\, \nu_5 \mapsto \texttt{myInteger}, \nu_6 \mapsto \texttt{myInteger}, \nu_7 \mapsto \texttt{myInteger}, \nu_3 \mapsto \texttt{myInteger}, \nu_4 \mapsto \texttt{myInteger} \,\}$$

The resulting Java class is given as:

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

4 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.

[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.