

More type inference in Java 8

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

24. June 2014

Overview

Introduction

The type inference algorithm

Example

Ongoing work: Eclipse plugin

Summary and Future work

Type inference in Java 8

Type inference for lambda expressions:

```
( $ty_1$   $a_1$ , ...,  $ty_1$   $a_1$ ) -> expr
```

Type inference for generic parameters (Diamond operator):

```
Vector<Integer> v = new Vector<> ();
```

Type instances:

```
<T> T id(T x) { return x; }
```

```
Integer i = id(1);
```

```
T  $\mapsto$  Integer
```

No type inference for lambda expressions

```
interface Fun1<R, T> { R apply(T arg); }

interface Fun2<R, T1, T2> { R apply(T1 arg1, T2 arg2); }

class Matrix extends Vector<Vector<Integer>> {

    Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
    op = (m) -> (f) -> f.apply(this, m);
}
```

Our goal

```
class Matrix extends Vector<Vector<Integer>> {  
  
    op = (m) -> (f) -> f.apply(this, m);  
}
```

The approach

1. Collecting all correct deducible functional interfaces as type for a lambda expression in **one equivalence class**.
2. Canonical functional interface **$\text{FunN}\langle R, T_1, \dots, T_N \rangle$** as a representative of each equivalence class

```
interface FunN<R, T1, ..., TN > {  
    R apply(T1 arg1, ..., TN argN );  
}
```

(cp. $(T_1, \dots, T_N) \rightarrow R$)

3. Type inference algorithm that determines the respective canonical interfaces.

The type inference algorithm

TI: $\text{TypeAssumptions} \times \text{Class} \rightarrow \{ (\text{Constraints}, \text{TClass}) \}$

TI($\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls)$) =

let

$\frac{(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t), \text{ConS})}{\text{TYPE}(\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls))}$ =

$\frac{\{ (cs_1, \sigma_1), \dots, (cs_n, \sigma_n) \}}{\text{SOLVE}(\text{ConS})}$

in

$\{ (cs_i, \sigma_i(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t))) \mid 1 \leq i \leq n \}$

The type inference algorithm

TI: $\text{TypeAssumptions} \times \text{Class} \rightarrow \{ (\text{Constraints}, \text{TClass}) \}$

TI($\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls)$) =

let

$(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t), \text{ConS}) =$
 $\text{TYPE}(\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls))$

$\{ (cs_1, \sigma_1), \dots, (cs_n, \sigma_n) \} = \text{SOLVE}(\text{ConS})$

in

$\{ (cs_i, \sigma_i(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t))) \mid 1 \leq i \leq n \}$

- TYPE:**
- ▶ Inserts type annotations, widely type variables as placeholders (cp. [Fuh/Mishra 88: Type Inference with Subtypes])
 - ▶ Determines the set of type constraints

The type inference algorithm

TI: $\text{TypeAssumptions} \times \text{Class} \rightarrow \{ (\text{Constraints}, \text{TClass}) \}$

TI($\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls)) =$

let

$\frac{(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t), \text{ConS})}{\text{TYPE}(\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls))} =$

$\frac{\{ (cs_1, \sigma_1), \dots, (cs_n, \sigma_n) \}}{\text{SOLVE}(\text{ConS})} =$

in

$\{ (cs_i, \sigma_i(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t))) \mid 1 \leq i \leq n \}$

- TYPE**: ▶ Inserts type annotations, widely type variables as placeholders (cp. [Fuh/Mishra 88: Type Inference with Subtypes])
- ▶ Determines the set of type constraints

SOLVE: Solves the constraints set by type unification [Plümicke 07]

Example: `op = (m) -> (f) -> f.apply(this, m);`

TYPE:

Annotated expression:

```
Lam(m : am,
    Lam(f : af,
        MCall(LoVar(f) : af,
            apply, (this : Matrix,
                LoVar(m) : am)) : a3) : Fun1<aapp, af>) : Fun1<aλf, am>
```

Set of constraints:

$$\{(\text{Fun1}\langle a_{\lambda f}, a_m \rangle \triangleleft a_{\text{op}}), (\text{Fun1}\langle a_{\text{app}}, a_f \rangle \triangleleft a_{\lambda f}), (a_f \doteq \text{Fun2}\langle a_3, a_1, a_2 \rangle), (\text{Matrix} \triangleleft a_1), (a_m \triangleleft a_2), (a_3 \triangleleft a_{\text{app}})\}.$$

Example: $op = (m) \rightarrow (f) \rightarrow f.apply(this, m);$

SOLVE:

$$\{ (\{ a_m \triangleleft a_2, a_3 \triangleleft a_{app} \},$$

$$\{ a_{op} \doteq \text{Fun1}\langle \text{Fun1}\langle a_{app}, \text{Fun2}\langle a_3, X, a_2 \rangle \rangle, a_m \rangle,$$

$$a_{\lambda f} \doteq \text{Fun1}\langle a_{app}, \text{Fun2}\langle a_3, X, a_2 \rangle \rangle, a_f \doteq \text{Fun2}\langle a_3, X, a_2 \rangle,$$

$$a_1 \doteq X \}) \} \mid \text{Matrix is a subtype of } X \}$$

Example: `op = (m) -> (f) -> f.apply(this, m);`

TI:

```
{  
  class Matrix extends Vector<Vector<Integer>> {  
    <a2, aapp, am extends a2, a3 extends aapp>1  
    Fun1<Fun1<aapp, Fun2<a3, X, a2>>, am>  
    op = (m) -> (f) -> f.apply(this, m);  
  }
```

| Matrix is a subtype of X }

¹The constraints are here given as bounded type variables for fields, which is in original Java only allowed for methods.

Eclipse plugin: User type selection

The screenshot shows the Eclipse IDE interface with the following components and annotations:

- Package Explorer (Left):** A tree view of the project structure. An arrow points to the `OverloadingTest.java` file with the text: **Press Save-Button and trigger the type inference process**.
- Code Editor (Center):** Displays the source code of `OverloadingTest.java`. Annotations include:
 - Three blue circular markers on the `Overloading2` class and its `overload()` method, with arrows pointing to the text: **Markers of missing types**.
 - A blue circular marker on the `Overloading` class with an arrow pointing to the text: **Tool-Tips: Show remarks, as errors and different possible types**.
- Outline (Right):** Shows a class hierarchy. The `Overloading` class is selected, with an arrow pointing to the text: **Select a type by pressing the right mouse button**.
- Problems (Bottom):** A yellow bar at the bottom of the IDE, currently empty.

Summary and Future work

- Summary:**
- ▶ Introduction of functional interfaces **FunN** as representatives of the types of lambda expressions.
 - ▶ Type inference for lambda expressions and recursive functions
 - ▶ Results are sets of typed programs

Summary and Future work

- Summary:**
- ▶ Introduction of functional interfaces **FunN** as representatives of the types of lambda expressions.
 - ▶ Type inference for lambda expressions and recursive functions
 - ▶ Results are sets of typed programs

- Future work:** Introduction of Intersection types
- ▶ Intersection type building from set of results (cp. [Pluemicke 08]²)
 - ▶ Byte-code generation for functions with intersection types

⇒ User type selection is not any longer necessary

²[Pluemicke 08: Intersection Types in Java]