More type inference in Java 8

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

24. June 2014

Overview

Type inference in Java 8

Type inference for lambda expressions:

```
(ty1 a1, ..., ty1 a1) -> 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);
```

$T \mapsto \texttt{Integer}$

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 Fun$N$<R,T_1 , . . . , T_N> as a representative of each equivalence class

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

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

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

The type inference algorithm

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

$$\textbf{TI}(\, \mathit{Ass}, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls} \,) \,) =$$

let

$$\underline{(\mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}_t \,), \mathit{ConS})} = \textbf{TYPE}(\, \mathit{Ass}, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls} \,) \,)$$

$$\underline{\{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \}} = \textbf{SOLVE}(\, \mathit{ConS} \,)$$

in

$$\{ (cs_i, \sigma_i(\, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}_t \,) \,)) \mid 1 \leqslant i \leqslant n \}$$

The type inference algorithm

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

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

let

$$\underline{(\text{Class}(\tau, \text{extends}(\tau'), \mathit{fdecls}_t), \mathit{ConS})} = \textbf{TYPE}(\mathit{Ass}, \text{Class}(\tau, \text{extends}(\tau'), \mathit{fdecls}))$$

$$\underline{\{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \}} = \textbf{SOLVE}(\mathit{ConS})$$

in

$$\{ (cs_i, \sigma_i(\text{Class}(\tau, \text{extends}(\tau'), \mathit{fdecls}_t))) \mid 1 \leqslant i \leqslant n \}$$

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

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

let

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

$$\underline{\{\ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n)\ \}} = \textbf{SOLVE}(\ ConS\ )$$

in

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

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 : a_m,
  Lam(f : a_f,
    MCall(LoVar(f) : a_f,
      apply, (this : Matrix,
        LoVar(m) : a_m)) : a_3) : Fun1<a_{app}, a_f>) : Fun1<$a_{\lambda f}$, a_m>

Set of constraints:

$$\{(\texttt{Fun1<}a_{\lambda f}, a_m\texttt{>} \lessdot a_{\text{op}}), (\texttt{Fun1<}a_{app}, a_f\texttt{>} \lessdot a_{\lambda f}), (a_f \doteq \texttt{Fun2<}a_3, a_1, a_2\texttt{>}),$$
$$(\texttt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{app}) \}.$$

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

SOLVE:

$$\{\,(\{\, a_m \lessdot a_2, a_3 \lessdot a_{app} \,\},$$
$$\{\, a_{\mathtt{op}} \doteq \mathtt{Fun1<Fun1<}a_{app}, \mathtt{Fun2<}a_3, X, a_2\mathtt{>>}, a_{\mathtt{m}}\mathtt{>},$$
$$a_{\lambda f} \doteq \mathtt{Fun1<}a_{app}, \mathtt{Fun2<}a_3, X, a_2\mathtt{>>}, a_f \doteq \mathtt{Fun2<}a_3, X, a_2\mathtt{>},$$
$$a_1 \doteq X \,\})\,\} \;\mid\; \mathtt{Matrix}\text{ is a subtype of } X \,\}$$

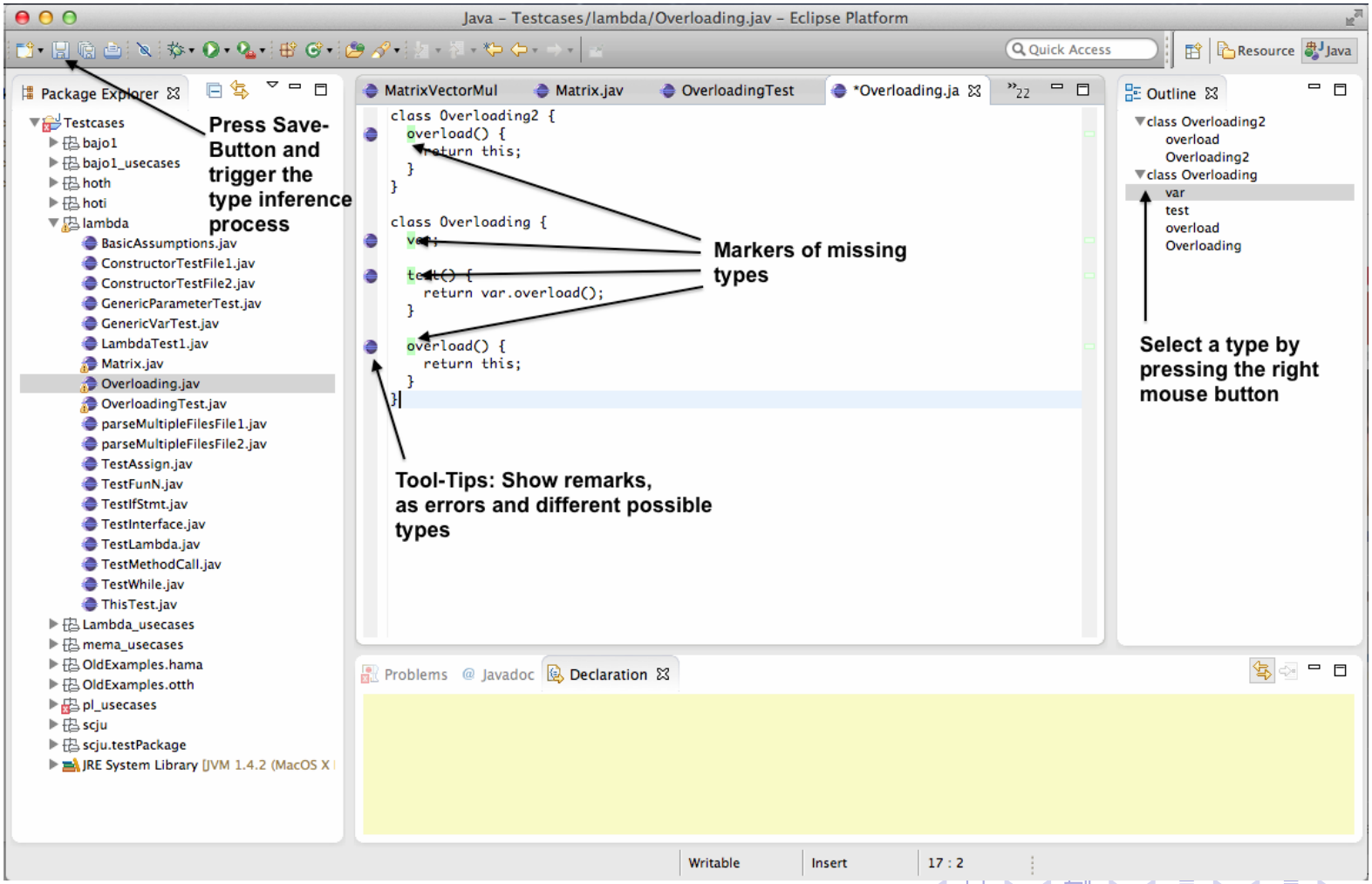
Example: `op = (m) -> (f) -> f.apply(this, m);`

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 }
```

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

Eclipse plugin: User type selection



Summary and Future work

Summary:

- Introduction of functional interfaces Fun$N$ as representatives of the types of lambda expressions.
- Type inference for lambda expressions and recursive functions
- Results are sets of typed programs

[2] [Pluemicke 08: Intersection Types in Java]

Summary and Future work

Summary:
- Introduction of functional interfaces Fun$N$ 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 buliding from set of results (cp. [Pluemicke 08][2])
- Byte-code generation for functions with intersection types

$\Longrightarrow$ User type selection is not any longer necessary

[2] [Pluemicke 08: Intersection Types in Java]