Type inference for functional interfaces in Java 8

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

6. Mai 2013

Overview

Two approaches [Plümicke, Bad Honnef 2012]

- Martin's approach: Java$_\lambda$ [Project Lambda 2010, version 0.1.5; Martin Plümicke, PPPJ 2011]
  - closures (λ–expressions)
  - function types
  - higher-order functions
- Brian's approach: Java 8 [Project Lambda 2011 version 0.4.2, Brian Goetz 2011]
  - closures (λ–expressions)
  - functional interfaces
  - method references
  - default methods

Function types vs. Functional interfaces

Function types (Java$_\lambda$)

```
void doAddition(#int(int, int) o)
  { ... }
```

Functional interfaces (Java 8)

```
interface Operation {
 public int op (int x, int y);
}
```

```
void doAddition(Operation o)
  { ... }
```

Functional interfaces as compatible target types

A lambda expression is *compatible* with a type T, if

- T is a functional interface type
- The lambda expression has the same number of parameters as T's method, and those parameters' types are the same
- Each expression returned by the lambda body is compatible with T's method's return type
- Each exception thrown by the lambda body is allowed by T's method's throws clause

Functional interfaces as compatible target types

A lambda expression is *compatible* with a type T, if

- T is a functional interface type
- The lambda expression has the same number of parameters as T's method, and those parameters' types are the same
- Each expression returned by the lambda body is compatible with T's method's return type
- Each exception thrown by the lambda body is allowed by T's method's throws clause

Lemma: There is an equivalence class of compatible target types for a lambda expression.

Canonical representation

For the equivalence class of the compatible target types of a lambda expression, there is a canonical representation

$$\texttt{Fun}N\texttt{<}R, T1, \ldots, TN\texttt{>}$$

with

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

if the type of the single method of a compatible target type is

$$(T1, \ldots, TN) \rightarrow R$$

Matrices in Java 8

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (Matrix m) -> (Fun2<Matrix, Matrix,Matrix> f) ->
       f.apply(this, m);
```

Matrices in Java 8

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (Matrix m) -> (Fun2<Matrix, Matrix,Matrix> f) ->
       f.apply(this, m);
  //(Matrix, Matrix) -> Matrix
  Fun2<Matrix, Matrix,Matrix>
  mul = (Matrix m1, Matrix m2) -> {
    Matrix ret = new Matrix ();
    ...                 //matrix multiplication
    return ret; }
```

Matrices in Java 8

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (Matrix m) -> (Fun2<Matrix, Matrix,Matrix> f) ->
       f.apply(this, m);
  //(Matrix, Matrix) -> Matrix
  Fun2<Matrix, Matrix,Matrix>
  mul = (Matrix m1, Matrix m2) -> {
    Matrix ret = new Matrix ();
    ...                 //matrix multiplication
    return ret; }
  public static void main(String[] args) {
      Matrix m1 = new Matrix(...);
      Matrix m2 = new Matrix(...);
      (m1.op.apply(m2)).apply(m1.mul);} } }
```

Goal

```
class Matrix extends Vector<Vector<Integer>> {


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


  mul = (m1, m2) -> {
    ret = new Matrix ();
    ...                  //matrix multiplication
    return ret; }
  public static void main(String[] args) {
      Matrix m1 = new Matrix(...);
      Matrix m2 = new Matrix(...);
      (m1.op.apply(m2)).apply(m1.mul);} } }
```

The language

Source	:=	*class*$*$
class	:=	Class(*stype*, [extends(*stype*),]*FieldDecl*$*$)
FieldDecl	:=	Field([*type*,]*var*[, *expr*])
block	:=	Block(*stmt*$*$)
stmt	:=	*block* | Return(*expr*) | While(*bexpr*, *block*)
	|	LocalVarDecl(*var*[, *type*]) | If(*bexpr*, *block*[, *block*])
	|	*stmtexpr*
lambdaexpr	:=	Lambda(((*var*[, *type*]))$*$, (*stmt* | *expr*))
stmtexpr	:=	Assign(*var*, *expr*) | New(*stype*, *expr*$*$)
	|	MethodCall(*expr*, apply, *expr*$*$)
vexpr	:=	LocalVar(*var*) | InstVar(*iexpr*, *var*)
iexpr	=	*vexpr* | *stmtexpr* | Cast(*type*, *iexpr*) | this | super
expr	:=	*lambdaexpr* | *iexp* | *bexp* | *sexp*

Sketch of the algorithm TI

TYPE: Introduces fresh type variables and collects the constraints

SOLVE: Solves the constraints by type unification

Data-structures

- **Set of type assumptions:**
  $v : \theta$: Assumptions for fields or local variables of the actual class.
  $\tau.v : \theta$: Assumptions for fields of the class τ.
  $\texttt{Fun}N\texttt{<R,T1,...,T}N\texttt{>.apply} : (\texttt{T1},\ldots,\texttt{T}N) \rightarrow \texttt{R}$

Data-structures

- **Set of type assumptions:**
  $v : \theta$: Assumptions for fields or local variables of the actual class.
  $\tau.v : \theta$: Assumptions for fields of the class τ.
  $\texttt{Fun}N\texttt{<R, T1, ..., T}N\texttt{>.apply} : (\texttt{T1}, \ldots, \texttt{T}N) \rightarrow \texttt{R}$
- **Set of constraints:**

$$\{\, \theta \, R \, \theta' \mid R \in \{\, <\!\cdot, <\!\cdot_? , \doteq \,\} \,\}$$

TYPE: $\texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \times \texttt{ConstraintsSet}$

TYPE: $\texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \times \texttt{ConstraintsSet}$

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

$\mathit{fdecls} = [\mathsf{Field}(\ f_1, \mathit{lexpr}_1\ ), \ldots, \mathsf{Field}(\ f_n, \mathit{lexpr}_n\ )]$

$\mathit{ftypeass} = \{\ \texttt{this}.f_i : a_i \mid a_i \textit{ fresh type variables}\ \}$
$\quad \cup \{\ \texttt{this} : \tau, \texttt{super} : \tau'\ \}$
$\quad \cup \{\ \textit{visible types of fields of } \tau'\ \}$

$\mathit{AssAll} = \mathit{Ass} \cup \mathit{ftypeass}$

TYPE: $\texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \times \texttt{ConstraintsSet}$

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

$fdecls = [\text{Field}(\ f_1, lexpr_1\ ), \ldots, \text{Field}(\ f_n, lexpr_n\ )]$

$ftypeass = \{\ \texttt{this}.f_i : a_i \mid a_i \text{ fresh type variables}\ \}$
$\cup \{\ \texttt{this} : \tau, \texttt{super} : \tau'\ \}$
$\cup \{\ \text{visible types of fields of } \tau'\ \}$

$AssAll = Ass \cup ftypeass$

Forall $1 \leqslant i \leqslant n$

$(lexp_{i_t} : rtyF_i, ConSF_i) =$ **TYPEExpr**$(\ AssAll, lexpr_i\ )$

TYPE: $\texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \times \texttt{ConstraintsSet}$

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

$fdecls = [\text{Field}(\ f_1, lexpr_1\ ), \ldots, \text{Field}(\ f_n, lexpr_n\ )]$

$ftypeass = \{\ \texttt{this}.f_i : a_i \mid a_i \text{ fresh type variables}\ \}$
$\cup \{\ \texttt{this} : \tau, \texttt{super} : \tau'\ \}$
$\cup \{\ \text{visible types of fields of } \tau'\ \}$

$AssAll = Ass \cup ftypeass$

Forall $1 \leqslant i \leqslant n$

$(lexp_{i_t} : rtyF_i, ConSF_i) =$ **TYPEExpr**$(\ AssAll, lexpr_i\ )$

$fdecls_t =$

$[\text{Field}(\ a_1, f_1, lexpr_{1_t} : rtyF_1\ ), \ldots, \text{Field}(\ a_n, f_n, lexpr_{n_t} : rtyF_{n_t}\ )]$

TYPE: $\texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \texttt{TClass} \times \texttt{ConstraintsSet}$

TYPE$(\ Ass, \text{Class}(\ \tau, \text{extends}(\ \tau'\ ), fdecls\ )\ ) =$
let
$fdecls = [\text{Field}(\ f_1, lexpr_1\ ), \ldots, \text{Field}(\ f_n, lexpr_n\ )]$
$ftypeass = \{\ \texttt{this}.f_i : a_i \mid a_i \text{ fresh type variables}\ \}$
$\cup \{\ \texttt{this} : \tau, \texttt{super} : \tau'\ \}$
$\cup \{\ \text{visible types of fields of } \tau'\ \}$
$AssAll = Ass \cup ftypeass$
Forall $1 \leqslant i \leqslant n$
$(lexp_{i_t} : rtyF_i, ConSF_i) =$ **TYPEExpr**$(\ AssAll, lexpr_i\ )$
$fdecls_t =$
$[\text{Field}(\ a_1, f_1, lexpr_{1_t} : rtyF_1\ ), \ldots, \text{Field}(\ a_n, f_n, lexpr_{n_t} : rtyF_{n_t}\ )]$
in
$(\text{Class}(\ \tau, \text{extends}(\ \tau'\ ), fdecls_t\ ),$
$(\bigcup_i ConSF_i \cup \{\ (rTyF_i \lessdot a_i) \mid 1 \leqslant i \leqslant n\ \}))$

TYPEExpr: $\texttt{TypeAssumptions} \times \texttt{Expr} \rightarrow \texttt{TExpr} \times \texttt{ConstraintsSet}$

TYPEExpr: $\texttt{TypeAssumptions} \times \texttt{Expr} \rightarrow \texttt{TExpr} \times \texttt{ConstraintsSet}$

For Lambda:

TYPEExpr(Ass, Lambda($(x_1, \ldots, x_N)$, $expr | stmt$)) =
let
$AssArgs = \{ x_i : a_i \mid a_i \text{ fresh type variables} \}$
$(expr_t : rty, ConS) =$ **TYPEExpr**($Ass \cup AssArgs, expr$)
$|\,(stmt_t : rty, ConS) =$ **TYPEStmt**($Ass \cup AssArgs, stmt$)
in
(Lambda($(x_1 : a_1, \ldots, x_N : a_N)$, $expr_t : rty | stmt_t : rty$) $: a$,
$ConS \cup \{ (\texttt{Fun}N\texttt{<}rty, a_1, \ldots, a_N\texttt{>} \lessdot a) \}$),
where a is a fresh type variable

TYPEStmt: $\texttt{TypeAssumptions} \times \texttt{Stmt} \rightarrow \texttt{TStmt} \times \texttt{ConstraintsSet}$

TYPEStmt: $\texttt{TypeAssumptions} \times \texttt{Stmt} \rightarrow \texttt{TStmt} \times \texttt{ConstraintsSet}$

For Return:

TYPEStmt(*Ass*, Return(*e*)) =

let

$(e_t : rty, ConS) =$ **TYPEExpr**(*Ass*, *e*)

in

$(\text{Return}(\ e_t : rty\ ) : a, ConS \cup \{\ (rty \lessdot a)\ \})$

where a is a fresh type variable

For Block:

$$\textbf{TYPEStmt}(\, \mathit{Ass}, \mathrm{Block}(\, s\,)\,) =$$
$$\quad \textbf{let}$$
$$\quad\quad (s_t : \mathit{rty}, \mathit{ConS}) = \textbf{TYPEStmt}(\, \mathit{Ass}, s\,)$$
$$\quad \textbf{in}$$
$$\quad\quad (\mathrm{Block}(\, s_t : \mathit{rty}\,) : \mathit{rty}, \mathit{ConS})$$

For Block:

TYPEStmt(*Ass*, Block(s)) =
  let
    $(s_t : rty, ConS) =$ **TYPEStmt**(*Ass*, s)
  in
    (Block($s_t : rty$) : $rty, ConS$)

TYPEStmt(*Ass*, Block($s_1, \ldots, s_n$)) =
  let
    $(s_{1_t} : rty_1, ConS_1) =$ **TYPEStmt**(*Ass*, s_1)
    $(\text{Block}(s_{2_t}, \ldots, s_{n_t}) : rty_2, ConS_2) =$
      TYPEStmt(*Ass*, Block($s_2, \ldots, s_n$))
  in
    $(\text{Block}(s_{1_t} : rty_1, s_{2_t}, \ldots, s_{n_t}) : a,$
     $ConS_1 \cup ConS_2 \cup \{ (rty_1 \lessdot a), (rty_2 \lessdot a) \})$
    where a is a fresh type variable

Example

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

Example

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

The result contains:

$lexpr_{1_t} =$

$$\begin{aligned}
&\text{Lam}(\,\texttt{m} : a_m,\\
&\quad \text{Lam}(\,\texttt{f} : a_f,\\
&\quad\quad \text{MCall}(\,\text{LoVar}(\,\texttt{f}\,) : a_f,\\
&\quad\quad\quad \texttt{apply}(\,\texttt{this} : \texttt{Matrix},\\
&\quad\quad\quad\quad \text{LoVar}(\,\texttt{m}\,) : a_m\,)\,) : a_{app}\,) : a_{\lambda f}\,) : a_{\lambda \texttt{m}}
\end{aligned}$$

Example

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

The result contains:

$lexpr_{1_t} =$

$\mathrm{Lam}(\,\mathtt{m} : a_m,$
$\quad \mathrm{Lam}(\,\mathtt{f} : a_f,$
$\quad\quad \mathrm{MCall}(\,\mathrm{LoVar}(\,\mathtt{f}\,) : a_f,$
$\quad\quad\quad \mathtt{apply}(\,\mathtt{this} : \mathtt{Matrix},$
$\quad\quad\quad\quad \mathrm{LoVar}(\,\mathtt{m}\,) : a_m\,)\,) : a_{app}\,) : a_{\lambda f}\,) : a_{\lambda \mathtt{m}}$

and the set of constraint:

$\{\,(a_{\lambda \mathtt{m}} \lessdot a_{\mathrm{op}}), (\mathtt{Fun1}\langle a_{\lambda f}, a_m\rangle \lessdot a_{\lambda \mathtt{m}}),$
$(\mathtt{Fun1}\langle a_{app}, a_f\rangle \lessdot a_{\lambda f}), (a_f \lessdot \mathtt{Fun2}\langle a_3, a_1, a_2\rangle),$
$(\mathtt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{app})\,\}$

SOLVE: $\texttt{ConstraintsSet} \to \texttt{Constraints\_SubstSet} \cup \{ \mathit{fail} \}$

SOLVE(*cs*) =
  let *subs* = **TUnify**(*cs*)
  in
    if (there are $\sigma \in \mathit{subs}$ in solved form) **then**
      $\{ \sigma \in \mathit{subs} \mid \sigma \text{ is in solved form} \}$
    if (there are $\sigma \in \mathit{subs}$, which has the form
      $\{ v \, R \, v' \mid v, v' \text{are type vars} \} \cup \{ v \doteq \theta \mid v \text{ is a type vars} \}$)
    then
      $\{ v \, R \, v' \mid v, v' \text{are type vars} \} \cup \{ v \doteq \theta \mid v \text{ is a type vars} \}$
    else *fail*

The sub-function **TUnify**[1]

TUnify: $\texttt{ConstraintsSet} \to \{\, \texttt{ConstraintsSet} \,\} \cup \{\, \mathit{fail} \,\}$

Input: $\{\, (\theta_1 \lessdot \theta'_1), \ldots, (\theta_n \lessdot \theta'_n) \,\}$

Output: $\{\, \sigma_1, \ldots, \sigma_m \,\}$

Post-condition: $\forall_j \{\, (\sigma_j(\,\theta_1\,) \leq^* \sigma_j(\,\theta'_1\,)), \ldots, (\sigma_j(\,\theta_n\,) \leq^* \sigma_j(\,\theta'_n\,)) \,\}$

where $\leq^*$ is the sub-typing relation

[1] [Pluemicke 2009]: *Java type unification with wildcards*, INAP 07

$$\textbf{TI}\colon \texttt{TypeAssumptions} \times \texttt{Class} \to \{\,(\texttt{Constraints}, \texttt{TClass})\,\} \cup \{\,\mathit{fail}\,\}$$

$$\mathbf{TI}\colon \texttt{TypeAssumptions} \times \texttt{Class} \to \{\,(\texttt{Constraints}, \texttt{TClass})\,\} \cup \{\,\mathit{fail}\,\}$$

$$\begin{aligned}
&\mathbf{TI}(\,\mathit{Ass}, \mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}, \mathit{mdecls}\,)\,) = \\
&\quad \mathbf{let} \\
&\qquad (\mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}_t, \mathit{mdecls}_t\,), \mathit{ConS}) = \\
&\qquad\quad \mathbf{TYPE}(\,\mathit{Ass}, \mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}, \mathit{mdecls}\,)\,) \\
&\qquad \{\,(\mathit{cs}_1, \sigma_1), \ldots, (\mathit{cs}_n, \sigma_n)\,\} = \mathbf{SOLVE}(\,\mathit{ConS}\,) \\
&\quad \mathbf{in} \\
&\qquad \{\,(\mathit{cs}_i, \sigma_i(\,\mathsf{Class}(\,\tau, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}_t, \mathit{mdecls}_t\,)\,)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid 1 \leqslant i \leqslant n\,\}
\end{aligned}$$

Example

$$cs = \{\ (a_{\lambda \mathtt{m}} \lessdot a_{\mathtt{op}}), (\mathtt{Fun1}\langle a_{\lambda f}, a_m\rangle \lessdot a_{\lambda \mathtt{m}}),$$
$$(\mathtt{Fun1}\langle a_{app}, a_f\rangle \lessdot a_{\lambda f}), (a_f \lessdot \mathtt{Fun2}\langle a_3, a_1, a_2\rangle),$$
$$(\mathtt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{app})\ \}$$

Example

$$cs = \{\ (a_{\lambda \mathtt{m}} \lessdot a_{\mathtt{op}}), (\mathtt{Fun1<}a_{\lambda f}, a_m\mathtt{>} \lessdot a_{\lambda \mathtt{m}}),$$
$$(\mathtt{Fun1<}a_{app}, a_f\mathtt{>} \lessdot a_{\lambda f}), (a_f \lessdot \mathtt{Fun2<}a_3, a_1, a_2\mathtt{>}),$$
$$(\mathtt{Matrix} \lessdot a_1), (a_m \lessdot a_2), (a_3 \lessdot a_{app})\ \}$$

The result of SOLVE:

$$\{ (\{\ a_m \lessdot a_2, a_3 \lessdot a_{app}\ \} \cup$$
$$\{\ a_{\mathtt{op}} \doteq \mathtt{Fun1<Fun1<}a_{app}, \mathtt{Fun2<}a_3, \mathtt{Matrix}, a_2\mathtt{>>}, a_{\mathtt{m}}\mathtt{>},$$
$$a_{\lambda \mathtt{m}} \doteq \mathtt{Fun1<Fun1<}a_{app}, \mathtt{Fun2<}a_3, \mathtt{Matrix}, a_2\mathtt{>>}, a_{\mathtt{m}}\mathtt{>},$$
$$a_{\lambda f} \doteq \mathtt{Fun1<}a_{app}, \mathtt{Fun2<}a_3, \mathtt{Matrix}, a_2\mathtt{>>},$$
$$a_f \doteq \mathtt{Fun2<}a_3, \mathtt{Matrix}, a_2\mathtt{>}, a_1 \doteq \mathtt{Matrix}\ \})$$
$$(\{\ a_m \lessdot a_2, a_3 \lessdot a_{app}\ \} \cup$$
$$\{\ a_{\mathtt{op}} \doteq \mathtt{Fun1<Fun1<}a_{app}, \mathtt{Fun2<}a_3, \mathtt{Vec<Vec<Int>>}, a_2\mathtt{>>}, a_{\mathtt{m}}\mathtt{>},$$
$$a_{\lambda \mathtt{m}} \doteq \mathtt{Fun1<Fun1<}a_{app}, \mathtt{Fun2<}a_3, \mathtt{Vec<Vec<Int>>}, a_2\mathtt{>>}, a_{\mathtt{m}}\mathtt{>},$$
$$a_{\lambda f} \doteq \mathtt{Fun1<}a_{app}, \mathtt{Fun2<}a_3, \mathtt{Vec<Vec<Int>>}, a_2\mathtt{>>},$$
$$a_f \doteq \mathtt{Fun2<}a_3, \mathtt{Vec<Vec<Int>>}, a_2\mathtt{>}, a_1 \doteq \mathtt{Vec<Vec<Int>>}\ \})\ \}$$

Example cont.

```
class Matrix<a_2, a_m extends a_2, a_app, a_3 extends a_app>
extends Vector<Vector<Integer>> {
  Fun1<Fun1<a_app, Fun2<a_3, Matrix,a_2>>, a_m>
  op = (a_m m) -> (Fun2<a_3, Matrix,a_2> f) ->
                                    f.apply(this, m); }
```

Example cont.

```
class Matrix<a_2, a_m extends a_2, a_app, a_3 extends a_app>
extends Vector<Vector<Integer>> {
  Fun1<Fun1<a_app, Fun2<a_3, Matrix,a_2>>, a_m>
  op = (a_m m) -> (Fun2<a_3, Matrix,a_2> f) ->
                                    f.apply(this, m); }
```

more principal than ...

Example cont.

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

The algorithm WTYPE for Java$_\lambda$ [2]

WTYPE: $\texttt{TypeAssumptions} \times \texttt{class} \to \{\, \texttt{WellTyping} \,\} \cup \{\, \mathit{fail} \,\}$

WTYPE$(\, \mathit{Ass}, \mathsf{Class}(\, \mathit{cl}, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls} \,) \,) =$

let

$(\{\, f_1 : a_1, \ldots, f_n : a_n \,\}, \mathit{CoeS}) =$

TYPE$(\, \mathit{Ass}, \mathsf{Class}(\, \mathit{cl}, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}, \mathit{ivardecls} \,) \,)$

$(\sigma, \mathit{AC}) =$ **match**$(\, \mathit{CoeS} \,)$

$((\tau_1, \ldots, \tau_m), \mathit{AC}') =$ **TUnify**$(\, \mathit{AC} \,)$

in

$\{\, (\mathit{AC}', \mathit{Ass} \vdash \{\, f_i : \tau_j \circ \sigma(\, a_i \,) \mid 1 \leqslant i \leqslant n \,\}) \mid 1 \leqslant j \leqslant m \,\}$

[2] [Pluemicke 2012]: *Functional implementation of well-typings in Java$_\lambda$* , IFL 2012

The algorithm WTYPE for Java$_\lambda$ [2]

WTYPE: $\texttt{TypeAssumptions} \times \texttt{class} \to \{\,\texttt{WellTyping}\,\} \cup \{\,\mathit{fail}\,\}$

$$\begin{aligned}
&\textbf{WTYPE}(\,\mathit{Ass}, \mathsf{Class}(\,\mathit{cl}, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}, \mathit{ivardecls}\,)\,) = \\
&\quad \textbf{let} \\
&\qquad (\{\,f_1 : a_1, \ldots, f_n : a_n\,\}, \mathit{CoeS}) = \\
&\qquad\quad \textbf{TYPE}(\,\mathit{Ass}, \mathsf{Class}(\,\mathit{cl}, \mathsf{extends}(\,\tau'\,), \mathit{fdecls}, \mathit{ivardecls}\,)\,) \\
&\qquad (\sigma, \mathit{AC}) = \textbf{match}(\,\mathit{CoeS}\,) \\
&\qquad ((\tau_1, \ldots, \tau_m), \mathit{AC}') = \textbf{TUnify}(\,\mathit{AC}\,) \\
&\quad \textbf{in} \\
&\qquad \{\,(\mathit{AC}', \mathit{Ass} \vdash \{\,f_i : \tau_j \circ \sigma(\,a_i\,) \mid 1 \leqslant i \leqslant n\,\}) \mid 1 \leqslant j \leqslant m\,\}
\end{aligned}$$

match reduces the function types

[2] [Pluemicke 2012]: *Functional implementation of well-typings in Java$_\lambda$* , IFL 2012

Summary and Outlook

Summary

- Adoption of the Java$_\lambda$ type inference algorithm to Java 8
- Simplication as there are no function types

Summary and Outlook

Summary

- Adoption of the Java$_\lambda$ type inference algorithm to Java 8
- Simplication as there are no function types

Outlook

- Consideration of methods with overloading and overriding in Java$_\lambda$ and Java 8
- Introduction of intersection types in the code generation