

More type inference in Java 8

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

12. November 2013

Overview

Introduction

- Lambda-expressions

- Java 8 features

- Type-inference in Java 8

Type inference algorithm TI

- The function TYPE

 - The sub-function TYPEExpr

 - The sub-function TYPEStmt

- The function SOLVE

- The whole algorithm TI

Implementation

Summary and Outlook

History of Java type system

Version 1:

- ▶ Subtyping on classes (without parameters)

Ex.: `Integer \leq^* Object`

History of Java type system

Version 1:

- ▶ Subtyping on classes (without parameters)
Ex.: `Integer ≤* Object`

Version 5:

- ▶ Parametrized classes (type constructors)
Ex.: `Vector<X>`
- ▶ Subtyping extension
Ex.: `Matrix ≤* Vector<Vector<Integer>>`
- ▶ Wildcards (with lower and upper bounds)
Ex.: `Vector<? extends Integer>`

History of Java type system

Version 1:

- ▶ Subtyping on classes (without parameters)
Ex.: `Integer ≤* Object`

Version 5:

- ▶ Parametrized classes (type constructors)
Ex.: `Vector<X>`
- ▶ Subtyping extension
Ex.: `Matrix ≤* Vector<Vector<Integer>>`
- ▶ Wildcards (with lower and upper bounds)
Ex.: `Vector<? extends Integer>`

Version 8: ▶ Lambda-expressions

- ▶ Functional interfaces, **but no function types**

Motivating Example: *External Iteration*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  
}
```

Motivating Example: *External Iteration*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
    if (s.gradYear == 2013) {
        if (s.score > highestScore) {
            highestScore = s.score;
        }
    }
}
```

Motivating Example: *External Iteration*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
    if (s.gradYear == 2013) {
        if (s.score > highestScore) {
            highestScore = s.score;
        }
    }
}
```

- ▶ external iteration
- ▶ serial iteration
- ▶ not thread-safe

Motivating Example: *Inner classes*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  

```

```
SomeCoolList<Student> students = ...  
double highestScore =  
    students.filter(new Predicate<Student>() {  
        public boolean op(Student s) {  
            return s.getGradYear() == 2013; }  
    })
```

Motivating Example: *Inner classes*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  
}
```

```
SomeCoolList<Student> students = ...  
double highestScore =  
    students.filter(new Predicate<Student>() {  
        public boolean op(Student s) {  
            return s.getGradYear() == 2013; }  
    }).map(new Mapper<Student, Double>() {  
        public Double extract(Student s) {  
            return s.getScore(); }  
    })
```

Motivating Example: *Inner classes*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  
}
```

```
SomeCoolList<Student> students = ...  
double highestScore =  
    students.filter(new Predicate<Student>() {  
        public boolean op(Student s) {  
            return s.getGradYear() == 2013; }  
    }).map(new Mapper<Student, Double>() {  
        public Double extract(Student s) {  
            return s.getScore(); }  
    }).max();
```

Motivating Example: *Inner classes*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  
}
```

```
SomeCoolList<Student> students = ...  
double highestScore =  
    students.filter(new Predicate<Student>() {  
        public boolean op(Student s) {  
            return s.getGradYear() == 2013; }  
    }).map(new Mapper<Student, Double>() {  
        public Double extract(Student s) {  
            return s.getScore(); }  
    }).max();
```

- ▶ internal iteration by inner classes
- ▶ traversal may be done in parallel
- ▶ but ugly syntax

Motivating Example: *Inner classes*

```
public class Student {
    String name;
    int graduationYear;
    double score; }
```

```
SomeCoolList<Student> students = ...
double highestScore =
    students.filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.getGradYear() == 2013; }
    }).map(new Mapper<Student, Double>() {
        public Double extract(Student s) {
            return s.getScore(); }
    }).max();
```

- ▶ internal iteration by inner classes
- ▶ traversal may be done in parallel
- ▶ but ugly syntax

Idea: Lambda-expressions

Lambda-expression

Well-known from functional programming languages.

Example: Identity-function

```
\x -> x -- HASKELL
```

```
fn x => x -- SML
```

```
(lambda (x) x) -- SCHEME
```

Lambda-expression

Well-known from functional programming languages.

Example: Identity-function

```
\x -> x -- HASKELL
```

```
fn x => x -- SML
```

```
(lambda (x) x) -- SCHEME
```

Application: $(\lambda x \rightarrow x) 1 = 1$

Lambda-expression

Well-known from functional programming languages.

Example: Identity-function

```
\x -> x                -- HASKELL
```

```
fn x => x              -- SML
```

```
(lambda (x) x)        -- SCHEME
```

Application: $(\lambda x \rightarrow x) 1 = 1$

Function-types: $(\lambda x \rightarrow x) :: a \rightarrow a$

Motivating Example: *Lambda-expressions*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  
}
```

```
SomeCoolList<Student> students = ...  
double highestScore =  
    students.filter(Student s -> s.getGradYear() == 2013)  
        .map(Student s -> s.getScore())  
        .max();
```

Motivating Example: *Lambda-expressions*

```
public class Student {  
    String name;  
    int graduationYear;  
    double score; }  
}
```

```
SomeCoolList<Student> students = ...
```

```
double highestScore =
```

```
    students.filter(Student s -> s.getGradYear() == 2013)  
        .map(Student s -> s.getScore())  
        .max();
```

- ▶ Lambda-expressions
- ▶ more readable

Java 8 features

- ▶ Lambda-expressions
- ▶ functional interfaces
- ▶ restricted type inference
- ▶ method references
- ▶ default methods

Functional interfaces (SAM-Types)

An interface with one method

Example:

```
interface Operation {  
    public int op (int x, int y);  
}  
  
int doOp(Operation o, int a, int b)  
{  
    return o.op(a, b);  
}  
...
```

Functional interfaces (SAM-Types)

An interface with one method

Example:

```
interface Operation {  
    public int op (int x, int y);  
}  
  
int doOp(Operation o, int a, int b)  
{  
    return o.op(a, b);  
}  
...  
  
doOp((int x, int y) -> x + y, 5, 7);
```

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**

$$\text{Fun}N\langle R, T_1, \dots, T_N \rangle$$

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

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

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 | iexpr | bexpr | sexp
  
```

Type-inference in Java 8

- ▶ Type parameter instantiation (Java 5.0):

id: $a \rightarrow a$

id(1) : for a the type `Integer` is inferred.

- ▶ Diamond-operator (Java 7):

`Vector <Integer> v = new Vector <>`

- ▶ Parameter's type-inference in Lambda-expressions (Java 8):

$$(T1\ x1, \dots, TN\ xN) \rightarrow h(x1, \dots, xN)$$

The types $T1, \dots, TN$ can be inferred:

$$(x1, \dots, xN) \rightarrow h(x1, \dots, xN)$$

is a correct Lambda-expression in Java 8.

More type inference

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 .

$\text{Fun } N \langle R, T_1, \dots, T_N \rangle . \text{apply} : (T_1, \dots, T_N) \rightarrow R$

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 .

$\text{Fun } N \langle R, T1, \dots, TN \rangle . \text{apply} : (T1, \dots, TN) \rightarrow R$

► **Set of constraints:**

$$\{ \theta R \theta' \mid R \in \{ \leq, \leq?, \doteq \} \}$$

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

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

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

let

fdecls = [Field(f_1, lexpr_1), ..., Field(f_n, lexpr_n)]

fypeass = { *this.f_i* : *a_i* | *a_i* fresh type variables }

∪ { *this* : τ , *super* : τ' }

∪ { *visible types of fields of* τ' }

AssAll = *Ass* ∪ *fypeass*

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

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

let

fdecls = [Field(f_1, lexpr_1), ..., Field(f_n, lexpr_n)]

fypeass = { $\text{this}.f_i : a_i \mid a_i$ fresh type variables }

∪ { $\text{this} : \tau, \text{super} : \tau' \}$

∪ { visible types of fields of $\tau' \}$

AssAll = *Ass* ∪ *fypeass*

Forall $1 \leq i \leq n$

($\text{lexpr}_i : \text{rty}F_i, \text{ConSF}_i$) = **TYPEExpr**(*AssAll*, lexpr_i)

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

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

let

fdecls = $[\text{Field}(f_1, \text{lexpr}_1), \dots, \text{Field}(f_n, \text{lexpr}_n)]$

fypeass = $\{ \text{this}.f_i : a_i \mid a_i \text{ fresh type variables} \}$

$\cup \{ \text{this} : \tau, \text{super} : \tau' \}$

$\cup \{ \text{visible types of fields of } \tau' \}$

AssAll = *Ass* \cup *fypeass*

Forall $1 \leq i \leq n$

$(\text{lexpr}_{i_t} : \text{rty}F_i, \text{ConSF}_i) = \text{TYPEExpr}(\text{AssAll}, \text{lexpr}_i)$

*fdecls*_{*t*} =

$[\text{Field}(a_1, f_1, \text{lexpr}_{1_t} : \text{rty}F_1), \dots, \text{Field}(a_n, f_n, \text{lexpr}_{n_t} : \text{rty}F_{n_t})]$

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

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

let

fdecls = $[\text{Field}(f_1, \text{lexpr}_1), \dots, \text{Field}(f_n, \text{lexpr}_n)]$

ftypeass = $\{ \text{this}.f_i : a_i \mid a_i \text{ fresh type variables} \}$

$\cup \{ \text{this} : \tau, \text{super} : \tau' \}$

$\cup \{ \text{visible types of fields of } \tau' \}$

AssAll = *Ass* \cup *ftypeass*

Forall $1 \leq i \leq n$

$(\text{lexpr}_{i_t} : \text{rty}F_i, \text{ConSF}_i) = \text{TYPEExpr}(\text{AssAll}, \text{lexpr}_i)$

*fdecls*_{*t*} =

$[\text{Field}(a_1, f_1, \text{lexpr}_{1_t} : \text{rty}F_1), \dots, \text{Field}(a_n, f_n, \text{lexpr}_{n_t} : \text{rty}F_{n_t})]$

in

$(\text{Class}(\tau, \text{extends}(\tau'), \text{fdecls}_t),$

$(\bigcup_i \text{ConSF}_i \cup \{ (r\text{Ty}F_i \triangleleft a_i) \mid 1 \leq i \leq n \}))$)

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

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

For Lambda:

TYPEExpr(*Ass*, **Lambda**((x_1, \dots, x_N) , *expr* | *stmt*)) =

let

$\text{AssArgs} = \{ x_i : a_i \mid a_i \text{ fresh type variables} \}$

$(\text{expr}_t : \text{rty}, \text{ConS}) = \text{TYPEExpr}(\text{Ass} \cup \text{AssArgs}, \text{expr})$

$\mid (\text{stmt}_t : \text{rty}, \text{ConS}) = \text{TYPEStmt}(\text{Ass} \cup \text{AssArgs}, \text{stmt})$

in

$(\text{Lambda}((x_1 : a_1, \dots, x_N : a_N), \text{expr}_t : \text{rty} \mid \text{stmt}_t : \text{rty}) : a,$

$\text{ConS} \cup \{ (\text{Fun } N < \text{rty}, a_1, \dots, a_N > \leq a) \})$,

where *a* is a fresh type variable

TYPEstmt: TypeAssumptions \times Stmt \rightarrow TStmt \times ConstraintsSet

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

For Return:

TYPEStmt(*Ass*, **Return**(*e*)) =
 let
 (*e_t* : *rty*, *ConS*) = **TYPEExpr**(*Ass*, *e*)
 in
 (**Return**(*e_t* : *rty*) : *a*, *ConS* \cup { (*rty* \leq *a*) })
 where a is a fresh type variable

For Block:

```
TYPEstmt( Ass, Block( s ) ) =  
  let  
    (st : rty, ConS) = TYPEstmt( Ass, s )  
  in  
    (Block( st : rty ) : rty, ConS)
```

For Block:

$$\begin{aligned} \text{TYPEStmt}(\text{Ass}, \text{Block}(s)) = \\ \text{let} \\ (s_t : rty, \text{ConS}) = \text{TYPEStmt}(\text{Ass}, s) \\ \text{in} \\ (\text{Block}(s_t : rty) : rty, \text{ConS}) \end{aligned}$$

$$\begin{aligned} \text{TYPEStmt}(\text{Ass}, \text{Block}(s_1, \dots, s_n)) = \\ \text{let} \\ (s_{1_t} : rty_1, \text{ConS}_1) = \text{TYPEStmt}(\text{Ass}, s_1) \\ (\text{Block}(s_{2_t}, \dots, s_{n_t}) : rty_2, \text{ConS}_2) = \\ \text{TYPEStmt}(\text{Ass}, \text{Block}(s_2, \dots, s_n)) \\ \text{in} \\ (\text{Block}(s_{1_t} : rty_1, s_{2_t}, \dots, s_{n_t}) : a, \\ \text{ConS}_1 \cup \text{ConS}_2 \cup \{ (rty_1 \triangleleft a), (rty_2 \triangleleft a) \}) \\ \text{where } a \text{ is a fresh type variable} \end{aligned}$$

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:

$$\text{lexpr}_{1_t} =$$

$$\text{Lam}(m : a_m,$$

$$\quad \text{Lam}(f : a_f,$$

$$\quad \quad \text{MCall}(\text{LoVar}(f) : a_f,$$

$$\quad \quad \quad \text{apply}(\text{this} : \text{Matrix},$$

$$\quad \quad \quad \quad \text{LoVar}(m) : a_m)) : a_{app}) : a_{\lambda f}) : a_{\lambda m}$$

Example

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

The result contains:

$l_{expr1_t} =$

```
Lam(m : am,
  Lam(f : af,
    MCall(LoVar(f) : af,
      apply(this : Matrix,
        LoVar(m) : am)) : aapp) : aλf) : aλm
```

and the set of constraint:

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

SOLVE: $\text{ConstraintsSet} \rightarrow \text{Constraints_SubstSet} \cup \{ \textit{fail} \}$

SOLVE(cs) =

let $subs = \text{TUnify}(cs)$

in

if (there are $\sigma \in subs$ in solved form) **then**

$\{ \sigma \in subs \mid \sigma \text{ is in solved form} \}$

if (there are $\sigma \in subs$, 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: $\text{ConstraintsSet} \rightarrow \{ \text{ConstraintsSet} \} \cup \{ \text{fail} \}$

Input: $\{ (\theta_1 \triangleleft \theta'_1), \dots, (\theta_n \triangleleft \theta'_n) \}$

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

Post-condition: $\forall_j \{ (\sigma_j(\theta_1) \leq^* \sigma_j(\theta'_1)), \dots, (\sigma_j(\theta_n) \leq^* \sigma_j(\theta'_n)) \}$
where \leq^* is the sub-typing relation

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

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

T1: $\text{TypeAssumptions} \times \text{Class} \rightarrow \{ (\text{Constraints}, \text{TClass}) \} \cup \{ \text{fail} \}$

T1($\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls, mdecls)$) =
let
 $(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t, mdecls_t), \text{ConS}) =$
 TYPE($\text{Ass}, \text{Class}(\tau, \text{extends}(\tau'), fdecls, mdecls)$)
 $\{ (cs_1, \sigma_1), \dots, (cs_n, \sigma_n) \} = \text{SOLVE}(\text{ConS})$
in
 $\{ (cs_i, \sigma_i(\text{Class}(\tau, \text{extends}(\tau'), fdecls_t, mdecls_t)))$
 $\mid 1 \leq i \leq n \}$

Example

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

Example

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

The result of SOLVE (without wildcard-types):

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

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); }
```

Solution of multiple results

1. Select one correct type in an Eclipse-PlugIn.
⇒ standard generation of byte-code
2. Extend Java type-system by a restricted form of intersection types.
⇒ a changed generation of byte-code is necessary²

²[Pluemicke 2008] ,Intersection Types in Java, PPPJ'08]

Implementation

- ▶ ThisTest
- ▶ TestLambda
- ▶ Matrix (TestFunN)

Outlook: Methods

- ▶ TestMethodCall

Summary and Outlook

Summary

- ▶ Complete type inference for a core of Java 8.
- ▶ Prototypical implementation

Summary and Outlook

Summary

- ▶ Complete type inference for a core of Java 8.
- ▶ Prototypical implementation

Outlook

- ▶ Methods with overloading and overriding.
- ▶ EclipsePlugin for type selection
- ▶ Introduction of intersection types in the code generation