Type inference in Java 8

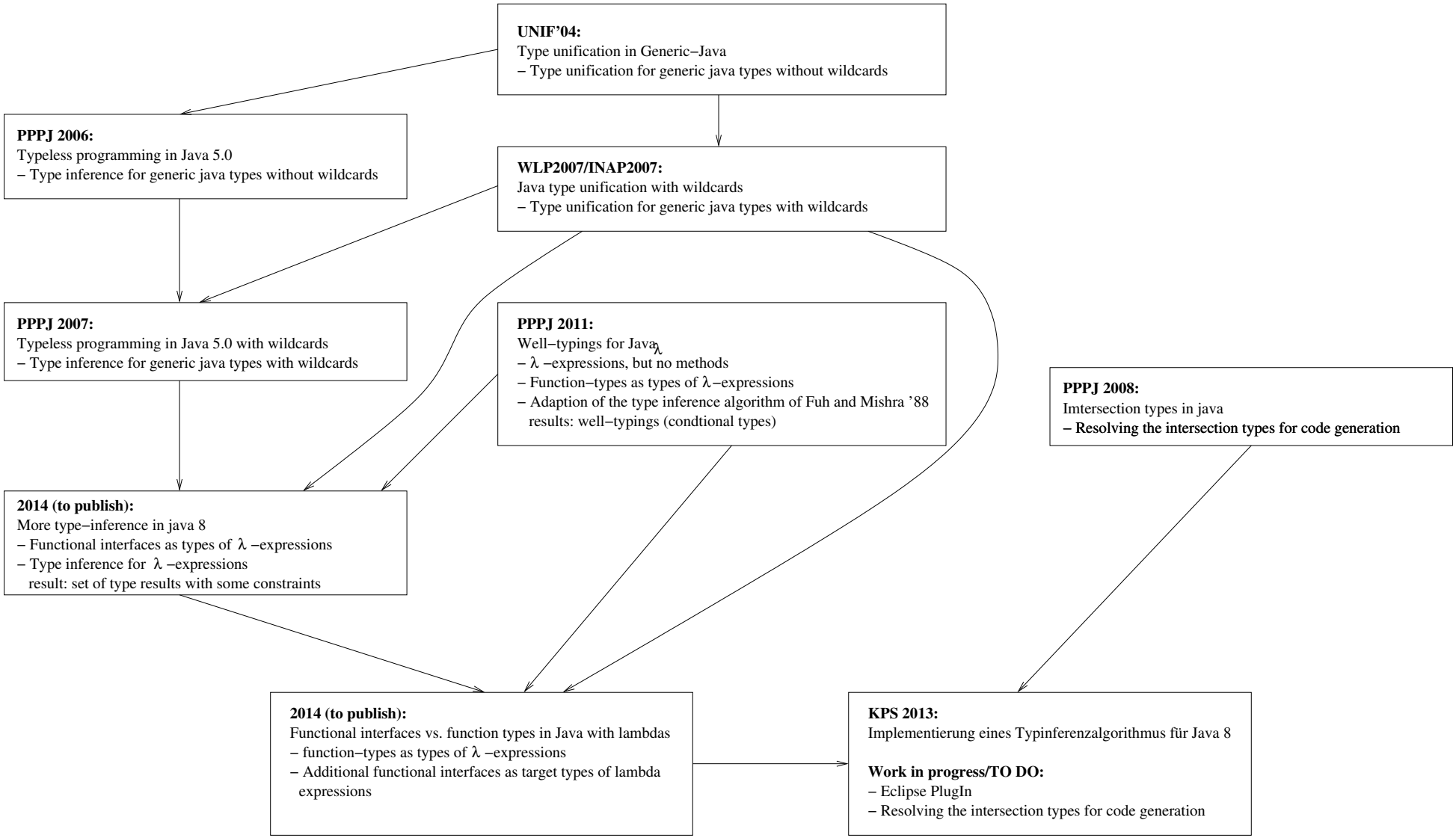
Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

10. Januar 2013

Overview

Martin Pluemicke's research on type inference for Java with generics and lambda expressions (version 5 – 8)



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` $\leq^*$ `Object`

Version 5:

- Parametrized classes (type constructors)
  Ex.: `Vector<X>`
- Subtyping extension
  Ex.: `Matrix` $\leq^*$ `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` $\leq^*$ `Object`

Version 5:

- Parametrized classes (type constructors)
  Ex.: `Vector<X>`
- Subtyping extension
  Ex.: `Matrix` $\leq^*$ `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: `(\x -> 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: `(\x -> x) 1 = 1`

Function-types: `(\x -> x) :: a -> 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 in the Java library

```
interface Comparator<T>  { int compare(T x, T y); }
interface FileFilter     { boolean accept(File x); }
interface DirectoryStream.Filter<T> { boolean accept(T x); }
interface Runnable       { void run(); }
interface ActionListener { void actionPerformed(...); }
interface Callable<T>    { T call(); }
```

Target types

- lambda expressions have no explicit type
- a target type is deduced from the context

Target types

- lambda expressions have no explicit type
- a target type is deduced from the context

Example:

```
Operation op = (int x, int y) -> x + y
```

⇒ Target type: Operation

```
Runnable op = (int x, int y) -> x + y
```

⇒ Target type: Runnable

Target types

- lambda expressions have no explicit type
- a target type is deduced from the context

Example:

```
Operation op = (int x, int y) -> x + y
```

⇒ Target type: Operation

```
Runnable op = (int x, int y) -> x + y
```

⇒ Target type: Runnable

Which is a correct target type?

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 representative

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

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

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(*iexpr*, apply, *expr*∗)
vexpr	:=	LocalVar(*var*) | InstVar(*iexpr*, *var*)
iexpr	=	*vexpr* | *stmtexpr* | Cast(*type*, *iexpr*) | this | super
expr	:=	*lambdaexpr* | *iexp* | *bexp* | *sexp*

Type–inference in Java 8

- Type parameter instatiation (Java 5.0):
  `id: a → 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, ..., TN xN) → h(x1, ..., XN)`

  The types `T1, ..., TN` can be inferred:

  `(x1, ..., xN) → h(x1, ..., 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.

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.
- **Set of constraints:**

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

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

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

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

$AssAll = Ass \cup ftypeass$

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: $\texttt{TypeAssumptions} \times \texttt{Class} \to \texttt{TClass} \times \texttt{ConstraintsSet}$

TYPE$(\ Ass, \text{Class}(\ \tau, \text{extends}(\ \tau'\ ), fdecls\ )\ ) =$
let
$fdecls = [\text{Field}(\ f_1, lexpr_1\ ), \ldots, \text{Field}(\ f_n, lexpr_n\ )]$
$ftypeass = \{\ \texttt{this}.f_i : a_i \mid a_i \textit{ fresh type variables}\ \}$
$\cup \{\ \texttt{this} : \tau, \texttt{super} : \tau'\ \}$
$\cup \{\ \textit{visible types of fields of } \tau'\ \}$
$AssAll = Ass \cup ftypeass$
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($Ass, \text{Lambda}((x_1, \ldots, x_N), expr | stmt)) =$

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

$(\text{Lambda}((x_1 : a_1, \ldots, x_N : a_N), expr_t : rty | stmt_t : rty)$

$: \texttt{Fun}N\texttt{<}a, a_1, \ldots, a_N\texttt{>},$

$ConS \cup \{ (rty \lessdot a) \}),$

where a *is a fresh type variable*

For MethodCall:

$$\textbf{TYPEExpr}(\, Ass, \text{MethodCall}(\, re, \texttt{apply}(\, e_1, \ldots, e_n \,)\,)\,) =$$
let
$$(re_t : rty, ConS) = \textbf{TYPEExpr}(\, Ass, re \,)$$
$$(e_{i_t} : rty_i, ConS_i) = \textbf{TYPEExpr}(\, Ass, e_i \,), \forall 1 \leqslant i \leqslant n$$
in
$$(\text{MethodCall}(\, re_t : rty, \texttt{apply}(\, e_{1_t} : rty_1, \ldots, e_{n_t} : rty_n \,)\,) : a,$$
$$(ConS \cup \bigcup_i ConS_i) \cup \{\, rty \lessdot \texttt{Fun}N\texttt{<}a, a_1, \ldots, a_N\texttt{>} \,\}$$
$$\cup \{\, rty_i \lessdot a_i \mid 1 \leqslant i \leqslant N \,\}$$
where $a_1, \ldots, a_N$ *and* a *are fresh type variables*

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
    (Return(e_t : *rty*) : *rty*, *ConS*))

For Block:

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

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)
    (Block($s_{2_t}, \ldots, s_{n_t}$) : $rty_2, ConS_2$) =
      TYPEStmt(Ass, Block($s_2, \ldots, s_n$))
  in
    (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:

$lexpr_{1_t} =$

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

Example

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

The result contains:

$$
\begin{aligned}
&\mathit{lexpr}_{1_t} = \\
&\quad \mathrm{Lam}(\,\mathtt{m} : a_m, \\
&\qquad \mathrm{Lam}(\,\mathtt{f} : a_f, \\
&\qquad\quad \mathrm{MCall}(\,\mathrm{LoVar}(\,\mathtt{f}\,) : a_f, \\
&\qquad\qquad\quad \mathtt{apply}(\,\mathtt{this} : \mathtt{Matrix}, \\
&\qquad\qquad\qquad\quad \mathrm{LoVar}(\,\mathtt{m}\,) : a_m\,)\,) : a_3\,) : \mathtt{Fun1}\langle a_{app}, a_f\rangle\,) : \mathtt{Fun1}\langle a_{\lambda f}, a_m\rangle
\end{aligned}
$$

and the set of constraint:

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

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

SOLVE(ConS) =
  let *subs* = **TUnify**(*ConS*)
  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

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

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

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

Example

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

Example

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

The result of SOLVE (without wildcard-types):

$$\{\ (\{\ a_m \lessdot a_2, a_3 \lessdot a_{app}\ \} \cup \{\ a_{\text{op}} \doteq \texttt{Fun1<Fun1<a}_{app}\texttt{,Fun2<a}_3\texttt{,Matrix,a}_2\texttt{>>,a}_\texttt{m}\texttt{>},\ a_{\lambda f} \doteq \texttt{Fun1<a}_{app}\texttt{,Fun2<a}_3\texttt{,Matrix,a}_2\texttt{>>},\ a_f \doteq \texttt{Fun2<a}_3\texttt{,Matrix,a}_2\texttt{>}, a_1 \doteq \texttt{Matrix}\ \})$$

$$(\{\ a_m \lessdot a_2, a_3 \lessdot a_{app}\ \} \cup \{\ a_{\text{op}} \doteq \texttt{Fun1<Fun1<a}_{app}\texttt{,Fun2<a}_3\texttt{,Vec<Vec<Int>>,a}_2\texttt{>>,a}_\texttt{m}\texttt{>},\ a_{\lambda f} \doteq \texttt{Fun1<a}_{app}\texttt{,Fun2<a}_3\texttt{,Vec<Vec<Int>>,a}_2\texttt{>>},\ a_f \doteq \texttt{Fun2<a}_3\texttt{,Vec<Vec<Int>>,a}_2\texttt{>}, a_1 \doteq \texttt{Vec<Vec<Int>>}\ \})\ \}$$

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 ...

```
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.
   $\Rightarrow$ standard generation of byte-code
2. Extend Java type-system by a restricted form of intersection types.
   $\Rightarrow$ a changed generation of byte-code is necessary[2]

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

Implementation

Java 8:

- `ThisTest`
- `TestLambda`
- `Matrix` (TestFunN)

Outlook: Methods

- `TestMethodCall`

Java 7: Overloading

- `OL`

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.
- Introduction of intersection types in the code generation