Typeless Programming in Java 5 and 7

Martin Plümicke

Baden-Württemberg Cooperative State University
Stuttgart/Horb

20. September 2010

Overview

Introduction

Extensions of the Java type–system

- parametrized types, type variables, type terms, wildcards
  e.g.

  `Vector<? extends AbstractList<? super Integer>>`

Introduction

Extensions of the Java type–system

- parametrized types, type variables, type terms, wildcards
  e.g.

  `Vector<? extends AbstractList<? super Integer>>`

Complex typings

- Often it is not obvious, which are the *best* types for methods and variables
- Sometimes principal types in Java are *intersection types*, which are not expressible (contradictive of writing re-usable code)

Introduction

Extensions of the Java type–system

- parametrized types, type variables, type terms, wildcards
  e.g.

`Vector<? extends AbstractList<? super Integer>>`

Complex typings

- Often it is not obvious, which are the *best* types for methods and variables
- Sometimes principal types in Java are *intersection types*, which are not expressible (contradictive of writing re-usable code)

$\Longrightarrow$ Developing a type–inference–system, which determines principal types

Example: Multiplication of matrices

```
class Matrix extends Vector<Vector<Integer>> {
    Matrix mul(Matrix m) {
        Matrix ret = new Matrix();
        int i = 0;
        while(i <size()) {
            Vector<Integer> v1 = this.elementAt(i);
            Vector<Integer> v2 = new Vector<Integer>();
            int j = 0;
            while(j < size()) {
                int erg = 0;
                int k = 0;
                while(k < v1.size()) {
                    erg = erg + v1.elementAt(k)
                        * m.elementAt(k).elementAt(j); k++; }
                v2.addElement(new Integer(erg)); j++; }
            ret.addElement(v2); i++; }
        return ret; }}
```

Alternative Typing

```
class Matrix extends Vector<Vector<Integer>> {
    Matrix/Vector<Vector<Integer>> mul(Matrix/Vector<Vector<Integer>> m) {
        Matrix/Vector<Vector<Integer>> ret = new Matrix();
        int i = 0;
        while(i <size()) {
            Vector<Integer> v1 = this.elementAt(i);
            Vector<Integer> v2 = new Vector<Integer>();
            int j = 0;
            while(j < size()) {
                int erg = 0;
                int k = 0;
                while(k < v1.size()) {
                    erg = erg + v1.elementAt(k)
                        * m.elementAt(k).elementAt(j); k++; }
                v2.addElement(new Integer(erg)); j++; }
            ret.addElement(v2); i++; }
        return ret; }}
```

Purpose: Typless

```
class Matrix extends Vector<Vector<Integer>> {
    mul(m) {
        ret = new Matrix();
        i = 0;
        while(i <size()) {
            v1 = this.elementAt(i);
            v2 = new Vector<Integer>();
            j = 0;
            while(j < size()) {
                erg = 0;
                k = 0;
                while(k < v1.size()) {
                    erg = erg + v1.elementAt(k)
                        * m.elementAt(k).elementAt(j); k++; }
                v2.addElement(new Integer(erg)); j++; }
            ret.addElement(v2); i++; }
        return ret; }}
```

System determines the principal typing(s)

```
mul: Matrix → Matrix &
     Matrix → Vector<Vector<Integer>>
     & ... &
     Vector<? extends Vector<? extends Integer>>
                                  → Vector<? super Vector<Integer>>
```

Type inference algorithm for Java 5[2]

The Idea

---

[1] L. Damas, R. Milner. Principal type-schemes for functional programs.

[2] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. PPPJ 2007

Type inference algorithm for Java 5[2]

The Idea

Hindley/Milner Type inference [Damas, Milner 1982][1]

---

[1] L. Damas, R. Milner. Principal type-schemes for functional programs.

[2] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. PPPJ 2007

Type inference algorithm for Java 5[2]

The Idea

Hindley/Milner Type inference [Damas, Milner 1982][1]

– function type constructor → (no higher–order functions)

---

[1] L. Damas, R. Milner. Principal type-schemes for functional programs.

[2] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. PPPJ 2007

Type inference algorithm for Java 5[2]

The Idea

Hindley/Milner Type inference [Damas, Milner 1982][1]

– function type constructor → (no higher–order functions)

$+$ function template $(ty_1 \times \ldots ty_n) \rightarrow ty_0$
(first–order functions)

---

[1] L. Damas, R. Milner. Principal type-schemes for functional programs.

[2] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. PPPJ 2007

Type inference algorithm for Java 5[2]

The Idea

Hindley/Milner Type inference [Damas, Milner 1982][1]

– function type constructor → (no higher–order functions)

+ function template $(ty_1 \times \ldots ty_n) \rightarrow ty_0$
(first–order functions)

+ subtyping

---

[1] L. Damas, R. Milner. Principal type-schemes for functional programs.

[2] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. PPPJ 2007

Type inference algorithm for Java 5[2]

The Idea

Hindley/Milner Type inference [Damas, Milner 1982][1]

– function type constructor → (no higher–order functions)

\+ function template $(ty_1 \times \ldots ty_n) \to ty_0$
(first–order functions)

\+ subtyping

\+ data and function polymorphism (overloading)

---

[1] L. Damas, R. Milner. Principal type-schemes for functional programs.

[2] M. Plümicke. Typeless Programming in Java 5.0 with wildcards. PPPJ 2007

Simple types $\mathrm{SType}_{TS}(\,BTV\,)$

- $BTV^{(ty)} \subseteq \mathrm{SType}_{TS}(\,BTV\,)$ (bounded type variables)
- $TC^{()} \subseteq \mathrm{SType}_{TS}(\,BTV\,)$ (0-ary type constructors/classes)
- For $ty_i \in \mathrm{SType}_{TS}(\,BTV\,)$

$$\begin{aligned} & \cup \{\, ? \,\} \\ & \cup \{\, ?\ \texttt{extends}\ \tau \mid \tau \in \mathrm{SType}_{TS}(\,BTV\,) \,\} \\ & \cup \{\, ?\ \texttt{super}\ \tau \mid \tau \in \mathrm{SType}_{TS}(\,BTV\,) \,\} \end{aligned}$$

and $C \in TC^{(a_1|_{b_1} \ldots a_n|_{b_n})}$ it holds

$$C\texttt{<}ty_1, \ldots, ty_n\texttt{>} \in \mathrm{SType}_{TS}(\,BTV\,)$$

if for $CC(\,C\texttt{<}ty_1, \ldots, ty_n\texttt{>}\,) = C\texttt{<}\overline{ty_1}, \ldots, \overline{ty_n}\texttt{>}$ holds:

$$\overline{ty_i} \leq^* b_i[a_j \mapsto \overline{ty_j} \mid 1 \leqslant j \leqslant n],$$

where

- $CC(\,\ldots\,)$ denotes the capture conversion
- $\leq^*$ is the subtyping ordering.

Abbreviation for wildcard–types

Instead of `A<? extends B>` we write

$$\mathtt{A<_{?}B>}$$

and instead of `C<? super D>` we write

$$\mathtt{C<^{?}D>}.$$

Subtyping ordering $\leq^*$

Reflexive and transitive closure of

- if θ `extends` θ' then $\underline{\theta \leq^* \theta'}$.
- if $\theta_1 \leq^* \theta_2$ then $\underline{\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)}$, where for each type variable a of θ_2 holds $\sigma_1(a) = \sigma_2(a) \in \mathrm{SType}_{TS}(BTV)$ (soundness condition).
- $\underline{a \leq^* \theta'}$ for $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ where $\exists \theta_i : \theta_i \leq^* \theta'$.
- It holds $\underline{C{<}\theta_1, \ldots, \theta_n{>} \leq^* C{<}\theta'_1, \ldots, \theta'_n{>}}$ if for θ_i and θ'_i either
    - $\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ and $\overline{\theta}_i \leq^* \overline{\theta}'_i$ or
    - $\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ and $\overline{\theta}'_i \leq^* \overline{\theta}_i$ or
    - θ_i, θ'_i are no wildcard arguments and $\theta_i = \theta'_i$ or
    - $\theta'_i = {}_?\theta_i$ or
    - $\theta'_i = {}^?\theta_i$
- From $C{<}\overline{\theta}_1, \ldots, \overline{\theta}_n{>} \leq^* C{<}\theta'_1, \ldots, \theta'_n{>}$ follows with $C{<}\overline{\theta}_1, \ldots, \overline{\theta}_n{>} = CC(\, C{<}\theta_1, \ldots, \theta_n{>}\,)$: $\underline{C{<}\theta_1, \ldots, \theta_n{>} \leq^* C{<}\theta'_1, \ldots, \theta'_n{>}}$
- $\underline{T|^{(\theta_1 \& \ldots \& \theta_n)} \leq^* \theta_i}$ for any $1 \leqslant i \leqslant n$.

Type unification [Pluemicke 2009][3]

Subtyping relation for type terms: $\leq^*$

Type Unification problem:

For two type terms θ_1 and θ_2 a substitution σ is demanded such that:

$$\sigma(\,\theta_1\,) \leq^* \sigma(\,\theta_2\,).$$

Base: Unification algorithm [Martelli, Montanari 1982][4]

---

[3] M. Pluemicke. *Java type unification with wildcards*, INAP 07. LNAI 5437.

[4] A. Martelli, U. Montanari. An efficient unification algorithm.

Example

Subtyping relation:

$\texttt{Integer} \leq^* \texttt{Number}$

$\texttt{Stack<a>} \leq^* \texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}$

Example

Subtyping relation:

$$\texttt{Integer} \leq^* \texttt{Number}$$

$$\texttt{Stack<a>} \leq^* \texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}$$

Application of the algorithm:

$$\{ (\texttt{Stack<a>} \lessdot \texttt{Vector<}_{?}\texttt{Number>}), (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>}) \}$$

Example

Subtyping relation:

$$\mathtt{Integer} \leq^* \mathtt{Number}$$

$$\mathtt{Stack<a>} \leq^* \mathtt{Vector<a>} \leq^* \mathtt{AbstractList<a>} \leq^* \mathtt{List<a>}$$

Application of the algorithm:

$$\{\, (\mathtt{Stack<a>} \lessdot \mathtt{Vector<}{}_?\mathtt{Number>}), (\mathtt{AbstractList<Integer>} \lessdot \mathtt{List<a>}) \,\}$$

$$\Longrightarrow \{\, \mathtt{a} \lessdot_? {}_?\mathtt{Number},\ \mathtt{Integer} \lessdot_? \mathtt{a} \,\}$$

Example

Subtyping relation:

$$\texttt{Integer} \leq^* \texttt{Number}$$

$$\texttt{Stack<a>} \leq^* \texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}$$

Application of the algorithm:

$$\{\, (\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_{?}\texttt{Number>}), (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>}) \,\}$$

$$\Longrightarrow \{\, \texttt{a} \lessdot_{?} {}_{?}\texttt{Number},\ \texttt{Integer} \lessdot_{?} \texttt{a} \,\}$$

$$\begin{aligned}
\Longrightarrow \{\, & \{\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq \texttt{Integer}\}, \{\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\}, \\
& \{\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\}, \{\texttt{a} \doteq {}_{?}\texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\}, \\
& \{\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer}\}, \{\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Number}\}, \\
& \{\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_{?}\texttt{Integer}\}, \{\texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^{?}\texttt{Integer}\}, \\
& \{\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\}, \{\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\}, \\
& \{\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\}, \{\texttt{a} \doteq {}_{?}\texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\}, \\
& \{\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer}\}, \{\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Number}\}, \\
& \{\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_{?}\texttt{Integer}\}, \{\texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^{?}\texttt{Integer}\} \,\}
\end{aligned}$$

Example cont.

$$\begin{array}{l}
\Longrightarrow \\
\{\, \{\, \mathtt{Integer} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq \mathtt{Integer} \,\}, \underline{\{\, {}_{?}\mathtt{Number} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}}, \\
\quad \{\, {}_{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}, \{\, {}^{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}, \\
\quad \{\, \mathtt{Integer} \doteq \mathtt{Number}, \mathtt{a} \doteq \mathtt{Integer} \,\}, \{\, {}_{?}\mathtt{Number} \doteq \mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}, \\
\quad \{\, {}_{?}\mathtt{Integer} \doteq \mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}, \{\, {}^{?}\mathtt{Integer} \doteq \mathtt{Number}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}, \\
\quad \{\, \mathtt{Integer} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq \mathtt{Integer} \,\}, \{\, {}_{?}\mathtt{Number} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}, \\
\quad \underline{\{\, {}_{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}}, \\
\quad \{\, {}^{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}, \\
\quad \underline{\{\, \mathtt{Integer} \doteq \mathtt{Integer}, \mathtt{a} \doteq \mathtt{Integer} \,\}}, \{\, {}_{?}\mathtt{Number} \doteq \mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}, \\
\quad \{\, {}_{?}\mathtt{Integer} \doteq \mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}\{\, {}^{?}\mathtt{Integer} \doteq \mathtt{Integer}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\} \,\}
\end{array}$$

Example cont.

$$\Longrightarrow$$

$$\begin{array}{l}
\{\, \{\, \mathtt{Integer} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq \mathtt{Integer} \,\}, \underline{\{\, {}_{?}\mathtt{Number} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}}, \\
\quad \{\, {}_{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}, \{\, {}^{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Number}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}, \\
\quad \{\, \mathtt{Integer} \doteq \mathtt{Number}, \mathtt{a} \doteq \mathtt{Integer} \,\}, \{\, {}_{?}\mathtt{Number} \doteq \mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}, \\
\quad \{\, {}_{?}\mathtt{Integer} \doteq \mathtt{Number}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}, \{\, {}^{?}\mathtt{Integer} \doteq \mathtt{Number}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}, \\
\quad \{\, \mathtt{Integer} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq \mathtt{Integer} \,\}, \{\, {}_{?}\mathtt{Number} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}, \\
\quad \underline{\{\, {}_{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}}, \\
\quad \{\, {}^{?}\mathtt{Integer} \doteq {}_{?}\mathtt{Integer}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}, \\
\quad \underline{\{\, \mathtt{Integer} \doteq \mathtt{Integer}, \mathtt{a} \doteq \mathtt{Integer} \,\}}, \{\, {}_{?}\mathtt{Number} \doteq \mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Number} \,\}, \\
\quad \{\, {}_{?}\mathtt{Integer} \doteq \mathtt{Integer}, \mathtt{a} \doteq {}_{?}\mathtt{Integer} \,\}\{\, {}^{?}\mathtt{Integer} \doteq \mathtt{Integer}, \mathtt{a} \doteq {}^{?}\mathtt{Integer} \,\}\,\}
\end{array}$$

$$\Longrightarrow \{\, \{\, \mathtt{a} \mapsto {}_{?}\mathtt{Number} \,\}, \{\, \mathtt{a} \mapsto {}_{?}\mathtt{Integer} \,\}, \{\, \mathtt{a} \mapsto \mathtt{Integer} \,\}\,\}$$

The language Java$_{core}$

$$
\begin{array}{lll}
\mathit{Source} & := & \mathit{class}* \\
\mathit{class} & := & \text{Class}(\mathit{stype}, [\ \text{extends}(\ \mathit{stype}\ ),]\mathit{IVarDecl}*, \mathit{Method}*) \\
\mathit{IVarDecl} & := & \text{InstVarDecl}(\ \mathit{stype}, \mathit{var}\ ) \\
\mathit{Method} & := & \text{Method}(\ \mathit{mname}, [\mathit{stype}], (\mathit{var}[, \mathit{stype}])*, \mathit{block}\ ) \\
\mathit{block} & := & \text{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block}\ \mid\ \text{Return}(\ \mathit{expr}\ )\ \mid\ \text{While}(\ \mathit{bexpr}, \mathit{block}\ ) \\
 & \mid & \text{LocalVarDecl}(\ \mathit{var}[, \mathit{stype}]\ )\ \mid \text{If}(\ \mathit{bexpr}, \mathit{block}[, \mathit{block}]\ ) \\
 & \mid & \mathit{stmtexpr} \\
\mathit{stmtexpr} & := & \text{Assign}(\ \mathit{var}, \mathit{expr}\ )\ \mid\ \text{New}(\ \mathit{stype}, \mathit{expr}*\ ) \\
 & \mid & \text{MethodCall}(\ [\mathit{expr},]\mathit{f}, \mathit{expr}*\ ) \\
\mathit{expr} & := & \mathit{stmtexpr}\ \mid\ \text{this}\ \mid\ \text{super} \\
 & \mid & \text{LocalOrFieldVar}(\ \mathit{var}\ )\ \mid\ \text{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \\
 & \mid & \mathit{bexp}\ \mid\ \mathit{sexp}
\end{array}
$$

The algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

The algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

Run over the abstract syntax tree: During the run over the abstract syntax tree of the coresponding java class the types are calculated gradually by type unification.

The algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

Run over the abstract syntax tree: During the run over the abstract syntax tree of the coresponding java class the types are calculated gradually by type unification.

Multiplying the assumptions: If the result of a type unification contains more than one result or if there is data polymorphism, the set of type assumptions is multiplied.

The algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

Run over the abstract syntax tree: During the run over the abstract syntax tree of the coresponding java class the types are calculated gradually by type unification.

Multiplying the assumptions: If the result of a type unification contains more than one result or if there is data polymorphism, the set of type assumptions is multiplied.

Erase type assumptions: If the type unification fails, the corresponding set of type assumptions is erased.

The algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

Run over the abstract syntax tree: During the run over the abstract syntax tree of the coresponding java class the types are calculated gradually by type unification.

Multiplying the assumptions: If the result of a type unification contains more than one result or if there is data polymorphism, the set of type assumptions is multiplied.

Erase type assumptions: If the type unification fails, the corresponding set of type assumptions is erased.

New method type parameters: At the end remained type-placeholders are replaced by new introduced method type parameters.

The algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

Run over the abstract syntax tree: During the run over the abstract syntax tree of the coresponding java class the types are calculated gradually by type unification.

Multiplying the assumptions: If the result of a type unification contains more than one result or if there is data polymorphism, the set of type assumptions is multiplied.

Erase type assumptions: If the type unification fails, the corresponding set of type assumptions is erased.

New method type parameters: At the end remained type-placeholders are replaced by new introduced method type parameters.

Intersection types: At the end each remained set of type assumptions forms one element of the result's intersection type.

Example: Multiplication of matrices: Type assumptions

```
class Matrix extends Vector<Vector<Integer>> {
   {α} mul({β} m) {
      {γ} ret = new Matrix();
      int i = 0;
      while(i <size()) {
         {ι} v1 = this.elementAt(i);
         {κ} v2 = new Vector<Integer>();
         int j = 0;
         while(j < size()) {
            {χ} erg = 0;
            int k = 0;
            while(k < v1.size()) {
               erg = erg + ({ξ}({ι} v1).elementAt(k))
                  * ({ψ}({φ} ({β} m).elementAt(k)).elementAt(j)); k++;}
            v2.addElement({χ} erg); j++; }
         ret.addElement({μ} v2); i++; }
      return ret; }}
```

ret = new Matrix ()

```
{$\alpha$} mul({$\beta$} m) {
    {$\gamma$} ret = {Matrix} new Matrix();
    ...
    return {$\gamma$} ret;
}
```

Unification: $\texttt{Matrix} \lessdot \gamma$

$\Rightarrow$

$$\begin{aligned}
\gamma &= \texttt{Matrix}\\
\gamma &= \texttt{Vector<Vector<Integer>>}\\
\gamma &= \texttt{Vector<}_{?}\texttt{Vector<Integer>>}\\
\gamma &= \texttt{Vector<}_{?}\texttt{Vector<}_{?}\texttt{Integer>>}\\
\gamma &= \texttt{Vector<}_{?}\texttt{Vector<}^{?}\texttt{Integer>>}\\
\gamma &= \texttt{Vector<}^{?}\texttt{Vector<Integer>>}
\end{aligned}$$

Type assumptions after the first unification

```
class Matrix extends Vector<Vector<Integer>> {
   { α, α, α, α, α, α } mul({ β, β, β, β, β, β } m) {
      { Matrix, Vector<Vector<Integer>>, Vector<_?Vector<Integer>>,
       Vector<_?Vector<_?Integer>>, Vector<_?Vector<^?Integer>>,
       Vector<^?Vector<Integer>> } ret = new Matrix();
      int i = 0; while(i <size()) {
         { ι, ι, ι, ι, ι, ι } v1 = this.elementAt(i);
         { κ, κ, κ, κ, κ, κ } v2 = new Vector<Integer>();
         int j = 0; while(j < size()) {
            { χ, χ, χ, χ, χ, χ } erg = 0;
            int k = 0; while(k < v1.size()) {
               erg = erg +({ ξ,ξ,ξ,ξ,ξ,ξ }({ ι,ι,ι,ι,ι,ι } v1).elementAt(k))
                * ({ ψ,ψ,ψ,ψ,ψ,ψ }
                    ({ ϕ,ϕ,ϕ,ϕ,ϕ,ϕ }
                     ({ β,β,β,β,β,β } m).elementAt(k)).elementAt(j)); k++;}
            v2.addElement({ χ, χ, χ, χ, χ, χ } erg); j++; }
         ret.addElement({ μ, μ, μ, μ, μ, μ } v2); i++; }
      return ret; }}
```

v1 = this.elementAt(i);

```
{α} mul({β} m) {
    ...
    {ι} v1 = ({Matrix} this).elementAt(i);
    ...
}
```

Unification: $\texttt{Matrix} \lessdot \texttt{Vector<}\iota\texttt{>}$

$\Rightarrow$

$$\iota = \texttt{Vector<Integer>}$$
$$\iota = \texttt{Vector<}_?\texttt{Integer>}$$
$$\iota = \texttt{Vector<}^?\texttt{Integer>}$$

return ret;

```
{ α } mul({ β } m) {
    ...
    return { γ } ret;
}
```

Unification: $\gamma \lessdot \alpha$ for

$\gamma = \texttt{Matrix}$

$\gamma = \texttt{Vector<Vector<Integer>>}$

$\gamma = \texttt{Vector<}^{?}\texttt{Vector<Integer>>}$

Result:

$\alpha = \texttt{Matrix}$

$\alpha = \texttt{Vector<Vector<Integer>>}$

$\alpha = \texttt{Vector<}^{?}\texttt{Vector<Integer>>}$

$\alpha = \texttt{Vector<}_{?}\texttt{Vector<Integer>>}$

$\alpha = \texttt{Vector<}_{?}\texttt{Vector<}_{?}\texttt{Integer>>}$

$\alpha = \texttt{Vector<}_{?}\texttt{Vector<}^{?}\texttt{Integer>>}$

Result:

$$\texttt{mul}: \&_{\beta,\alpha}(\beta \rightarrow \alpha),$$

where

$\beta \leq^* \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>}$,
$\texttt{Matrix} \leq^* \alpha$

Implementation

- Overloading–Example
- Return–Example
- Matrix–Example

Implementation

- Overloading–Example
- Return–Example
- Matrix–Example

Purpuse: Byte-code generation for methods with intersection types

Code generation for method with intersection types[5]

- Byte-code allows no intersection types
- First approach: generate for each element of the intersection type an own method

---

[5] M. Pluemicke, *Intersection types in java*. PPPJ 2008.

Example: class `OL` I

```
class OL {
     Integer m(x) { return x + x; }   //Integer → Integer
     Boolean m(x) { return x || x; }  //Boolean → Boolean
}

class Main {
    main(x) {    //   Integer → Integer & Boolean → Boolean
       ol;
       ol = new OL();
       return ol.m(x);
    }
}
```

Example: class OL II

Result for `Main`:

```
class Main {
    Integer main(Integer x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }

    Boolean main(Boolean x) {
        OL ol;
        ol = new OL();
        return ol.m(x);
    }
}
```

Example: class `OL` II

Result for `Main`:

```
class Main {
    Integer main(Integer x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }

    Boolean main(Boolean x) {
        OL ol;
        ol = new OL();
        return ol.m(x);
    }
}
```

Java program is correct

Example: Multiplication of matrices

$$\texttt{mul:}\ \&_{\beta,\alpha}(\beta\rightarrow\alpha),$$

where

$\beta \leq^{*}$ `Vector<? extends Vector<? extends Integer>>`,
`Matrix` $\leq^{*} \alpha$

Example: Multiplication of matrices

$$\texttt{mul:}\ \&_{\beta,\alpha}(\beta\longrightarrow\alpha),$$

where

$\beta \leq^* \texttt{Vector<? extends Vector<? extends Integer>>}$,
$\texttt{Matrix} \leq^* \alpha$

```
class Matrix extends Vector<Vector<Integer>> {
  Matrix mul(Vector<? extends Vector<? extends Integer>> m) { ... }
  Matrix mul(Vector<? extends Vector<Integer>> m) { ... }
  Matrix mul(Vector<Vector<Integer>> m) { ... }
  ...
  Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m) { ... }
  ...
  Vector<? extends Vector<? extends Integer>> mul(Matrix m) { ... }
```

Example: Multiplication of matrices

$$\texttt{mul:}\ \&_{\beta,\alpha}(\beta \rightarrow \alpha),$$

where

$\beta \leq^* \texttt{Vector<? extends Vector<? extends Integer>>}$,

$\texttt{Matrix} \leq^* \alpha$

```
class Matrix extends Vector<Vector<Integer>> {
  Matrix mul(Vector<? extends Vector<? extends Integer>> m) { ... }
  Matrix mul(Vector<? extends Vector<Integer>> m) { ... }
  Matrix mul(Vector<Vector<Integer>> m) { ... }
  ...
  Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m) { ... }
  ...
  Vector<? extends Vector<? extends Integer>> mul(Matrix m) { ... }
```

Not a correct Java program

Group elements of the intersection type

Idea:

1. Group all elements which
   - executes the same code
   - have a common subtype
2. Generate new methods only for the groups

Group elements of the intersection type

Idea:

1. Group all elements which
    - executes the same code
    - have a common subtype
2. Generate new methods only for the groups

Code–execution: Callgraph of the method declarations

$$\mathcal{CG}(\,cl.m : \tau\,)$$

Callgraph of the method m in the class cl with the typing τ.

Group elements of the intersection type

Idea:

1. Group all elements which
   - executes the same code
   - have a common subtype
2. Generate new methods only for the groups

Code–execution: Callgraph of the method declarations

$$\mathcal{CG}(\, cl.m : \tau \,)$$

Callgraph of the method m in the class cl with the typing τ.

Subtype of function types: Subtyping ordering
$\theta_i \leq^* \theta'_i, \theta \leq^* \theta' \Rightarrow$

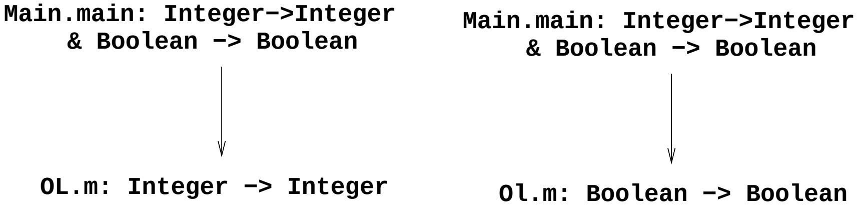
$$\theta'_1 \times \ldots \times \theta'_n \rightarrow \theta \leq^* \theta_1 \times \ldots \times \theta_n \rightarrow \theta'$$

Example class OL I

Callgraph

$\mathcal{CG}(\,\texttt{Main.main} : \texttt{Integer} \rightarrow \texttt{Integer}\,)$ $=$

$\mathcal{CG}(\,\texttt{Main.main} : \texttt{Boolean} \rightarrow \texttt{Boolean}\,)$ $=$



`Main.main: Integer->Integer & Boolean -> Boolean`

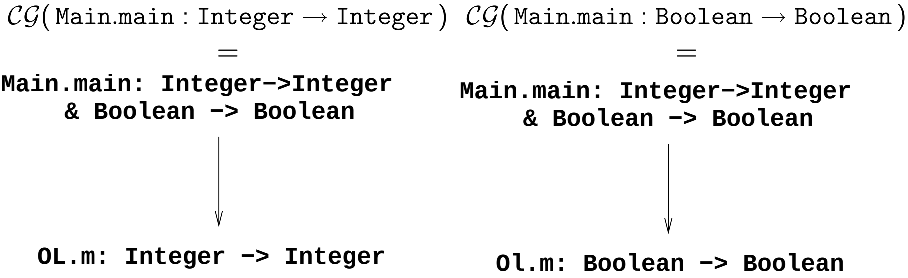
`OL.m: Integer -> Integer`

`Main.main: Integer->Integer & Boolean -> Boolean`

`Ol.m: Boolean -> Boolean`

Example class OL I

Callgraph



$\mathcal{CG}(\,\texttt{Main.main} : \texttt{Integer} \to \texttt{Integer}\,)$

$=$

Main.main: Integer->Integer & Boolean -> Boolean

OL.m: Integer -> Integer

$\mathcal{CG}(\,\texttt{Main.main} : \texttt{Boolean} \to \texttt{Boolean}\,)$

$=$

Main.main: Integer->Integer & Boolean -> Boolean

Ol.m: Boolean -> Boolean

Subtype

$\texttt{Integer} \to \texttt{Integer}$

$\texttt{Boolean} \to \texttt{Boolean}$

Example class OL II

Code generation

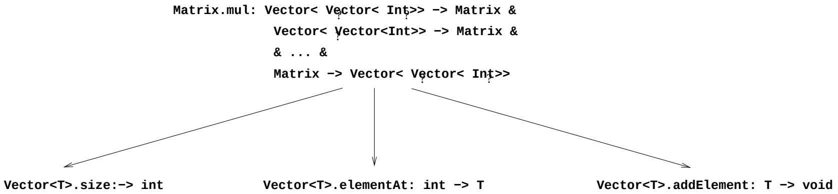
```
class Main {
    Integer main(Integer x) {
        OL ol;
        ol = new OL();
        return ol.m(x); }

    Boolean main(Boolean x) {
        OL ol;
        ol = new OL();
        return ol.m(x);
    } }
```

Code is unchanged in comparison to the first approach

Example class `Matrix` I

Callgraph $\mathcal{CG}($ `Matrix.mul` : τ $)$ for all τ



```
Matrix.mul: Vector< Vector< Int>> -> Matrix &
            Vector< Vector<Int>> -> Matrix &
            & ... &
            Matrix -> Vector< Vector< Int>>
```

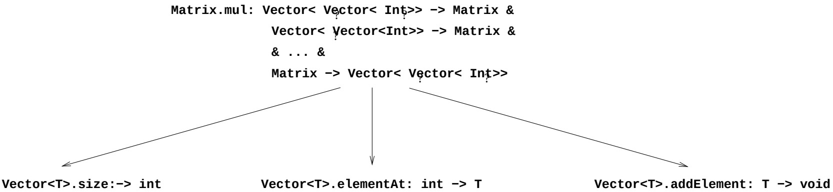
`Vector<T>.size:-> int`

`Vector<T>.elementAt: int -> T`

`Vector<T>.addElement: T -> void`

Example class `Matrix` I

Callgraph $\mathcal{CG}($ `Matrix.mul` $: \tau\,)$ for all τ



```
Matrix.mul: Vector< ?Vector< ?Int>> -> Matrix &
            Vector< ?Vector<Int>> -> Matrix &
            & ... &
            Matrix -> Vector< ?Vector< ?Int>>

Vector<T>.size:-> int      Vector<T>.elementAt: int -> T      Vector<T>.addElement: T -> void
```

Subtype:

$$\texttt{Vector<? extends Vector<? extends Integer>>} \longrightarrow \texttt{Matrix}$$

Example class Matrix II

Code generation (only one method!)

```
Matrix mul(Vector<? extends Vector<? extends Integer>> m) {
   Matrix ret = new Matrix();
   int i = 0;
   while(i <size()) {
       Vector<Integer> v1 = this.elementAt(i);
       Vector<Integer> v2 = new Vector<Integer>();
       int j = 0;
       while(j < size()) {
           int erg = 0;
           int k = 0;
           while(k < v1.size()) { erg = erg + ...; k++; }
           v2.addElement(new Integer(erg)); j++; }
       ret.addElement(v2); i++; }
   return ret; }}
```

The algorithm

Input: A Java program p with inferred (intersection) types.

Output: A Java program p', where the methods have standard Java types. The semantics of p and p' are equal.

1. Step: For every class cl in p consider for each method m the intersection type ty_m:
   - Build the callgraph $\mathcal{CG}(\, cl.m : \tau\,)$ for each function type τ of the intersection type ty_m.
   - Group all elements τ of ty_m, where $\mathcal{CG}(\, cl.m : \tau\,)$ is the same graph and there is a subtype.

The algorithm

Input: A Java program p with inferred (intersection) types.

Output: A Java program p', where the methods have standard Java types. The semantics of p and p' are equal.

1. Step: For every class cl in p consider for each method m the intersection type ty_m:
   - Build the callgraph $\mathcal{CG}(\, cl.m : \tau\,)$ for each function type τ of the intersection type ty_m.
   - Group all elements τ of ty_m, where $\mathcal{CG}(\, cl.m : \tau\,)$ is the same graph and there is a subtype.
2. Step: Determine the subtype of the respective group.

The algorithm

Input: A Java program p with inferred (intersection) types.

Output: A Java program p', where the methods have standard Java types. The semantics of p and p' are equal.

1. Step: For every class cl in p consider for each method m the intersection type ty_m:
   - Build the callgraph $\mathcal{CG}(\,cl.m : \tau\,)$ for each function type τ of the intersection type ty_m.
   - Group all elements τ of ty_m, where $\mathcal{CG}(\,cl.m : \tau\,)$ is the same graph and there is a subtype.
2. Step: Determine the subtype of the respective group.
3. Step: Generate for each group of function types the corresponding Java code with the subtype as standard typing in p'.

Principal Typing

Definition [Damas, Milner 1982]:

“A type-scheme for a declaration is a *principal type-scheme*, if any other type-scheme for the declaration is a generic instance of it.”

Principal Typing

Definition [Damas, Milner 1982]:

“A type-scheme for a declaration is a *principal type-scheme*, if any other type-scheme for the declaration is a generic instance of it.”

Generalized definition for Java:

“An intersection type-scheme for a declaration is a *principal type-scheme*, if any (non–intersection) type-scheme θ for the declaration is a supertype of a generic instance of one element of the intersection type-scheme τ and θ and τ have the same callgraph.”

Principal Typing

Definition [Damas, Milner 1982]:

"A type-scheme for a declaration is a *principal type-scheme*, if any other type-scheme for the declaration is a generic instance of it."

Generalized definition for Java:

"An intersection type-scheme for a declaration is a *principal type-scheme*, if any (non–intersection) type-scheme θ for the declaration is a supertype of a generic instance of one element of the intersection type-scheme τ and θ and τ have the same callgraph."

This refined definition guarantees, that for each method, which is generated by the resolving algorithm, at least one typing is contained in the principal type.

Conclusion of Java 5 type inference

Conclusion

- Type inference algorithm for Java 5, which determines intersection types
- Resolving algorithm of intersection types, which allows to generate byte code for intersection types
- Principal type property.

Conclusion of Java 5 type inference

Conclusion

- Type inference algorithm for Java 5, which determines intersection types
- Resolving algorithm of intersection types, which allows to generate byte code for intersection types
- Principal type property.

Outlook

At the moment: Type inference algorithm infers typings, which are later erased as supertypes by the resolving algorithm.

Purpuse: Type inference algorithm infers only subtypes, such that no typings are erased in the resolving algorithm.

Implementation: The resolving algorithm and optimize the type inference algorithm

Closures (λ–expressions) in Java 7?

Motivation: Bulk-data APIs like parallel arrays

- parallelism approach: sorting, searching, selection

Example:

```
public class Student {
    String name;
    int graduationYear;
    double gpa; //grade point average
}

ParallelArray<Student> students
    = new ParallelArray<Student>(fjPool, data);

double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();
```

Realization by helper objects

```
static final Ops.Predicate<Student> isSenior
    = new Ops.Predicate<Student>() {
        public boolean op(Student s) {
            return s.graduationYear == Student.THIS_YEAR;
        }
    };

static final Ops.ObjectToDouble<Student> selectGpa
    = new Ops.ObjectToDouble<Student>() {
        public double op(Student student) {
            return student.gpa;
        }
    };
```

Realization by closures (λ–expressions)

```
double bestGpa
    = students.withFilter(
                  #(Student s)(s.graduationYear == THIS_YEAR))
              .withMapping(#(Student s)(s.gpa))
              .max();
```

Different approaches

- Closures for the Java Programming Language: BGGA
  [Bracha, Gafter, Gosling, von der Ahé]
- Concise Instance Creation Expressions: Closures without Complexity: CICE
  [Lee, Lea, Bloch]
- First-class methods: Java-style closures: FCM
  [Colebourne, Schulz]

---

[6] http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt

Different approaches

- Closures for the Java Programming Language: BGGA
  [Bracha, Gafter, Gosling, von der Ahé]
- Concise Instance Creation Expressions: Closures without Complexity: CICE
  [Lee, Lea, Bloch]
- First-class methods: Java-style closures: FCM
  [Colebourne, Schulz]

Our approach is following:
Project Lambda[6] Java Language Specification draft (Version 0.1.5)

---

[6] http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt

Mark Reinhold's Blog (Principal Engineer Java SE and OpenJDK)

Two key features are needed:

- A literal syntax, for writing closures, and
- Function types, so that closures are first-class citizens in the type system.

To integrate closures with the rest of the language and the platform we need two additional features:

- Closure conversion to implement a single-method interface or abstract class and
- Extension methods

The language Java_λ

$$\begin{array}{lcl}
\mathit{Source} & := & \mathit{class}* \\
\mathit{class} & := & \mathrm{Class}(\mathit{stype},[\ \mathrm{extends}(\ \mathit{stype}\ ),]\mathit{IVarDecl}*, \mathit{FunDecl}*) \\
\mathit{IVarDecl} & := & \mathrm{InstVarDecl}(\ \mathit{stype}, \mathit{var}\ ) \\
\mathit{FunDecl} & := & \mathrm{Fun}(\ \mathit{fname}, \underline{[\mathit{type}]}, \mathit{lambdaexpr}\ ) \\
\mathit{block} & := & \mathrm{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block}\ \mid\ \mathrm{Return}(\ \mathit{expr}\ )\ \mid\ \mathrm{While}(\ \mathit{bexpr}, \mathit{block}\ ) \\
 & \mid & \mathrm{LocalVarDecl}(\ \mathit{var}\underline{[, \mathit{type}]}\ )\ \mid \mathrm{If}(\ \mathit{bexpr}, \mathit{block}[, \mathit{block}]\ ) \\
 & \mid & \mathit{stmtexpr} \\
\mathit{lambdaexpr} & := & \mathrm{Lambda}(\ ((\mathit{var}\underline{[, \mathit{type}])})*, (\mathit{stmt}\ \mid\ \mathit{expr})\ ) \\
\mathit{stmtexpr} & := & \mathrm{Assign}(\ \mathit{var}, \mathit{expr}\ )\ \mid\ \mathrm{New}(\ \mathit{stype}, \mathit{expr}*\ ) \\
 & \mid & \mathrm{Eval}(\ \mathit{expr}, \mathit{expr}*\ ) \\
\mathit{expr} & := & \mathit{lambdaexpr}\ \mid\ \mathit{stmtexpr}\ \mid\ \mathrm{this}\ \mid\ \mathrm{super} \\
 & \mid & \mathrm{LocalOrFieldVar}(\ \mathit{var}\ )\ \mid\ \mathrm{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \\
 & \mid & \mathrm{InstFun}(\ \mathit{expr}, \mathit{fname}\ )\ \mid\ \mathit{bexp}\ \mid\ \mathit{sexp}
\end{array}$$

Types $\mathrm{Type}_{TS}(\,BTV\,)$

- $\mathrm{SType}_{TS}(\,BTV\,) \subseteq \mathrm{Type}_{TS}(\,BTV\,)$
- For $\theta_i \in \mathrm{Type}_{TS}(\,BTV\,)$

$$\#\,\theta_0\,(\theta_1, \ldots, \theta_n) \in \mathrm{Type}_{TS}(\,BTV\,)$$

(*straw-man syntax*, correspond to $\theta_1 \times \ldots \times \theta_n \rightarrow \theta_0$)

Subtyping–relation

Let $\leq^*$ be the Java subtyping relation on simple types $\mathrm{SType}_{TS}(\, BTV \,)$.

The continuation on $\mathrm{Type}_{TS}(\, BTV \,)$ is defined as:

$$\#\,\theta_0\,(\theta'_1, \ldots, \theta'_n) \leq^* \#\,\theta'_0\,(\theta_1, \ldots, \theta_n) \quad \textit{iff} \quad \theta_i \leq^* \theta'_i.$$

λ–Abstraction

[lambda$_{\textbf{stmt}}$]

$$\frac{(O \cup \{\, x_i : \theta_i \,\}, \tau, \tau') \ \triangleright_{Stmt} \ s : \theta}{(O, \tau, \tau') \ \triangleright_{Expr} \ \mathrm{Lambda}(\,(x_1, \ldots, x_n), s\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

λ–Abstraction

[lambda$_{\text{stmt}}$]

$$\frac{(O \cup \{\, x_i : \theta_i \,\}, \tau, \tau') \ \triangleright_{Stmt} \ s : \theta}{(O, \tau, \tau') \ \triangleright_{Expr} \ \text{Lambda}(\,(x_1, \ldots, x_n), s\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

[lambda$_{\text{expr}}$]

$$\frac{(O \cup \{\, x_i : \theta_i \,\}, \tau, \tau') \ \triangleright_{Expr} \ e : \theta}{(O, \tau, \tau') \ \triangleright_{Expr} \ \text{Lambda}(\,(x_1, \ldots, x_n), e\,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

Application

[App]

$$\frac{(O, \tau, \tau') \rhd_{Expr} \; e : \#\,\theta\,(\theta'_1, \ldots, \theta'_n), \quad (O, \tau, \tau') \rhd_{Expr} \; e_i : \theta_i}{(O, \tau, \tau') \rhd_{Expr} \; \mathrm{Eval}(\, e, e_1 \ldots e_n \,) : \theta} \; \theta_i \leq^* \theta'_i$$

Application

[App]

$$\frac{(O, \tau, \tau') \rhd_{Expr} \; e : \#\,\theta\,(\theta'_1, \ldots, \theta'_n), \quad (O, \tau, \tau') \rhd_{Expr} \; e_i : \theta_i}{(O, \tau, \tau') \rhd_{Expr} \; \text{Eval}(\, e, e_1 \ldots e_n \,) : \theta} \; \theta_i \leq^* \theta'_i$$

[InstFun]

$$\frac{(O, \tau, \tau') \rhd_{Expr} \; re : \overline{\theta}, \qquad O_{\overline{\theta}} \rhd_{Id} \; f : \#\,\theta\,(\theta_1, \ldots, \theta_n)}{(O, \tau, \tau') \rhd_{Expr} \; \text{InstFun}(\, re, f \,) : \#\,\theta\,(\theta_1, \ldots, \theta_n)}$$

Adapt Fuh and Mishra's algorithm[7].

- Java$_\lambda$ type system is equivalent
- subtyping, but
- no overloading

---

[7] Y.-C. Fuh, P. Mishra. Type inference with subtypes. ESOP '88

Adapt Fuh and Mishra's algorithm[7].

- Java$_\lambda$ type system is equivalent
- subtyping, but
- no overloading
- **Problem**: Fuh and Mishra's algorithm determines *well typings* (set of possibly not unique solvable unequations)

---

[7] Y.-C. Fuh, P. Mishra. Type inference with subtypes. ESOP '88

Fuh und Mishra's algorithm

$$\textbf{WTYPE} : \texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{\, \mathit{fail} \,\}$$

Fuh und Mishra's algorithm

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{\ \textit{fail}\ \}$

TYPE: $\texttt{TypeAssump} \times \texttt{Expr} \rightarrow \texttt{Type} \times \texttt{CoercionSet}$

Construct the set of coercions by running over the Expression.

Fuh und Mishra's algorithm

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \longrightarrow \texttt{WellTyping} + \{\, \mathit{fail} \,\}$

TYPE: $\texttt{TypeAssump} \times \texttt{Expr} \longrightarrow \texttt{Type} \times \texttt{CoercionSet}$

Construct the set of coercions by running over the Expression.

MATCH : $\texttt{CoercionSet} \longrightarrow \texttt{Substitution} + \{\, \mathit{fail} \,\}$

Extended unification to adopt the structure of the coercions.

Fuh und Mishra's algorithm

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{\, \mathit{fail} \,\}$

TYPE: $\texttt{TypeAssump} \times \texttt{Expr} \rightarrow \texttt{Type} \times \texttt{CoercionSet}$

Construct the set of coercions by running over the Expression.

MATCH : $\texttt{CoercionSet} \rightarrow \texttt{Substitution} + \{\, \mathit{fail} \,\}$

Extended unification to adopt the structure of the coercions.

SIMPLIFY : $\texttt{CoercionSet} \rightarrow \texttt{AtomicCoercionSet}$

Eliminate type constructors, especially the *functor* and the *tuple-construction*

Fuh und Mishra's algorithm

WTYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \rightarrow \texttt{WellTyping} + \{ \mathit{fail} \}$

TYPE: $\texttt{TypeAssump} \times \texttt{Expr} \rightarrow \texttt{Type} \times \texttt{CoercionSet}$

Construct the set of coercions by running over the Expression.

MATCH : $\texttt{CoercionSet} \rightarrow \texttt{Substitution} + \{ \mathit{fail} \}$

Extended unification to adopt the structure of the coercions.

SIMPLIFY : $\texttt{CoercionSet} \rightarrow \texttt{AtomicCoercionSet}$

Eliminate type constructors, especially the *functor* and the *tuple-construction*

CONSISTENT : $\texttt{AtomicCoercionSet} \rightarrow \texttt{Boolean} + \{ \mathit{fail} \}$

Consistence check, and additionally determination of possible solutions.

Adaption to the Java$_\lambda$ type system

SIMPLIFY: The [Martelli, Montanari 1982] unification is substituted by the [Pluemicke 2009] type unification[8]

CONSISTENCE: The functions *above* and *below* are substituted by the functions `greater` and `smaller`[8], as *above* and *below* are not finite in the Java$_\lambda$ type system.

---

[8] M. Pluemicke. *Java type unification with wildcards*, INAP 07. LNAI 5437.

Example

```
class Matrix extends Vector<Vector<Integer>> {
  //Matrix -> ((Matrix, Matrix) -> Matrix) -> Matrix
  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
    op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(this, m)))
  }
```

Example

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

Goal

Typed syntax:

```
##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(this, m)))
```

Goal

Typed syntax:

```
##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(this, m)))
```

Typeless syntax:

```
  op = #(m)(#(f)(f(this, m)))
```

Goal: The system determines the type

```
##Matrix(#Matrix(Matrix, Matrix))(Matrix)
```

automatically.

$WTYPE(\,\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m})))\,),$ (TYPE)

- $t_m \longrightarrow t_{\#f} \lessdot t_{\mathrm{op}}$
- $t_f \longrightarrow t_{f(\,this, m\,)} \lessdot t_{\#f}$
- $t_f \lessdot (t_1, t_2) \longrightarrow t_3$
- $\mathtt{Matrix} \lessdot t_1$
- $t_m \lessdot t_2$
- $t_3 \lessdot t_{f(\,this, m\,)}$

$WTYPE(\,\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m})))\,), \qquad (\mathrm{MATCH})$

- $t_m \to ((\epsilon_2, \epsilon'_2) \to \epsilon''_2) \to \gamma'_1 \lessdot \beta \to ((\epsilon_3, \epsilon'_3) \to \epsilon''_3) \to \gamma'_2,$

  $(t_{op} \mapsto \beta \to \beta'), (t_{\#f} \mapsto \gamma_1 \to \gamma'_1), (\beta' \mapsto \gamma_2 \to \gamma'_2),$
  $(\gamma_1 \mapsto (\epsilon_2, \epsilon'_2) \to \epsilon''_2), (\gamma_2 \mapsto (\epsilon_3, \epsilon'_3) \to \epsilon''_3)$

- $((\epsilon_1, \epsilon'_1) \to \epsilon''_1) \to t_{f(\,this, m\,)} \lessdot ((\epsilon_2, \epsilon'_2) \to \epsilon''_2) \to \gamma'_1$

  $(t_{\#f} \mapsto \gamma_1 \to \gamma'_1), (t_f \mapsto (\epsilon_1, \epsilon'_1) \to \epsilon''_1), (\gamma_1 \mapsto (\epsilon_2, \epsilon'_2) \to \epsilon''_2)$

- $(\epsilon_1, \epsilon'_1) \to \epsilon''_1 \lessdot (t_1, t_2) \to t_3$

  $(t_f \mapsto (\epsilon_1, \epsilon'_1) \to \epsilon''_1), (\gamma_1 \mapsto (\epsilon_2, \epsilon'_2) \to \epsilon''_2)$

- $\mathtt{Matrix} \lessdot t_1$
- $t_m \lessdot t_2$
- $t_3 \lessdot t_{f(\,this, m\,)}$

$WTYPE(\,\emptyset, \#(\mathtt{m})(\#(\mathtt{f})(\mathtt{f}.(\mathtt{this}, \mathtt{m})))\,)$, (SIMPLIFY)

- $\beta \lessdot t_m$,
  $\epsilon_2 \lessdot \epsilon_3,\ \epsilon'_2 \lessdot \epsilon'_3,\ \epsilon''_3 \lessdot \epsilon''_2$,
  $\gamma'_1 \lessdot \gamma'_2$
- $\epsilon_1 \lessdot \epsilon_2,\ \epsilon'_1 \lessdot \epsilon'_2,\ \epsilon''_2 \lessdot \epsilon''_1$,
  $t_{f(\,this,m\,)} \lessdot \gamma'_1$
- $t_1 \lessdot \epsilon_1$,
  $t_2 \lessdot \epsilon'_1$,
  $\epsilon''_1 \lessdot t_3$
- $\mathtt{Matrix} \lessdot t_1$
- $t_m \lessdot t_2$
- $t_3 \lessdot t_{f(\,this,m\,)}$

$WTYPE(\,\emptyset, \#(\texttt{m})(\#(\texttt{f})(\texttt{f}.(\texttt{this}, \texttt{m})))\,),\ (\mathrm{CONSISTENCE})$

It	$Coercion$	$l_{\texttt{Matrix}}$	l_{t_1}	l_{ϵ_1}	l_{ϵ_2}	l_{ϵ_3}	...
0		M	$*$	$*$	$*$	$*$	
1	$\texttt{M} \lessdot t_1$	M	M, V<V<Int>>	$*$	$*$	$*$	
1	$t_1 \lessdot \epsilon_1$	M	M, V<V<Int>>	M, V<V<Int>>	$*$	$*$	
1	$\epsilon_1 \lessdot \epsilon_2$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$	
1	$\epsilon_2 \lessdot \epsilon_3$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	
1	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	
2	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	

Result

$$\mathrm{op} : \beta \rightarrow ((\epsilon_3, \epsilon'_3) \rightarrow \epsilon''_3) \rightarrow \gamma'_2$$

with

$\epsilon_3 = \texttt{Matrix}, \texttt{Vector<Vector<Integer>>}$

$\beta \lessdot t_m \lessdot t_2 \lessdot \epsilon'_1 \lessdot \epsilon'_2 \lessdot \epsilon'_3$

$\epsilon''_3 \lessdot \epsilon''_2 \lessdot \epsilon''_1 \lessdot t_3 \lessdot t_{f(\,this, m\,)} \lessdot \gamma'_1 \lessdot \gamma'_2$

Result

$$\text{op} : \beta \rightarrow ((\epsilon_3, \epsilon_3') \rightarrow \epsilon_3'') \rightarrow \gamma_2'$$

with

$\epsilon_3 = \texttt{Matrix}, \texttt{Vector<Vector<Integer>>}$

$\beta \lessdot t_m \lessdot t_2 \lessdot \epsilon_1' \lessdot \epsilon_2' \lessdot \epsilon_3'$

$\epsilon_3'' \lessdot \epsilon_2'' \lessdot \epsilon_1'' \lessdot t_3 \lessdot t_{f(\,this, m\,)} \lessdot \gamma_1' \lessdot \gamma_2'$

This is a well-typing.

Compare to the goal:

$\texttt{Matrix} \rightarrow ((\texttt{Matrix}, \texttt{Matrix}) \rightarrow \texttt{Matrix}) \rightarrow \texttt{Matrix}$

The well-typing is more principal.

Conclusion and future work

Conclusion

- Type inference algorithm for Java 5
- Principal types are intersection types
- Fuh and Mishra's type inference algorithm could be adopt to Java 7.

Conclusion and future work

Conclusion

- Type inference algorithm for Java 5
- Principal types are intersection types
- Fuh and Mishra's type inference algorithm could be adopt to Java 7.

Future work

- Improve the type inference algorithm for Java 5, such that only principal types are inferred
- Integrate *well typings* into Java 7
- Implementation:
  - Code-generation for intersection types
  - Adopted Fuh and Mishra algorithm