Typeless Programming in Java 5.0 with Wildcards

Martin Plümicke

University of Cooperative Education
Stuttgart/Horb

PPPJ 2007, Portugal

6. September 2007

Overview

Problem

Extensions of the Java 5.0 type–system

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

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

Problem

Extensions of the Java 5.0 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 5.0 are *intersection types*, which are not expressible (contradictive of writing re-usable code)

Problem

Extensions of the Java 5.0 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 5.0 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 < v1.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 < v1.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 < v1.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>>
```

Related work

Two approaches

- OO–Type–inference [Palsberg, Schwartzbach, et.al.]:
  Precise types (no principal type)
- λ–Terms [Hindley/Milner, et.al.]:
  Principal type property

Our approach

[Hindley/Milner et al]

Our approach

[Hindley/Milner et al]

– function type constructor $\longrightarrow$ (no higher–order functions)

Our approach

[Hindley/Milner et al]

– function type constructor $\longrightarrow$ (no higher–order functions)

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

Our approach

[Hindley/Milner et al]

– function type constructor $\rightarrow$ (no higher–order functions)

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

\+ data and function polymorphism (overloading)

Abbreviation for wildcard–types

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

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

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

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

The algorithm

Type unification

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:

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

References

Base: Unification algorithm of Martelli, Montanari 1982

Our extensions:

- Type unification algorithm for Java 5.0 type terms without wildcards [Plümicke 2004, Unif'04, Cork]
- Type unification algorithm for Java 5.0 type terms with wildcards [Plümicke 2007, Unif'07, Paris]

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:

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

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

Application of the algorithm:

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

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

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:

$$\begin{aligned}
& \{ (\texttt{Stack<a>} \lessdot \texttt{Vector<}{}_?\texttt{Number>}), (\texttt{AbstractList<Integer>} \lessdot \texttt{List<a>}) \} \\
\Longrightarrow & \{ \texttt{a} \lessdot_? {}_?\texttt{Number},\ \texttt{Integer} \lessdot_? \texttt{a} \} \\
\Longrightarrow & \{ \{ \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq \texttt{Integer} \}, \{ \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \}, \\
& \{ \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \}, \{ \texttt{a} \doteq {}_?\texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \}, \\
& \{ \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq \texttt{Integer} \}, \{ \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Number} \}, \\
& \{ \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}_?\texttt{Integer} \}, \{ \texttt{a} \doteq \texttt{Number}, \texttt{a} \doteq {}^?\texttt{Integer} \}, \\
& \{ \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \}, \{ \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \}, \\
& \{ \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \}, \{ \texttt{a} \doteq {}_?\texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \}, \\
& \{ \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq \texttt{Integer} \}, \{ \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Number} \}, \\
& \{ \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}_?\texttt{Integer} \}, \{ \texttt{a} \doteq \texttt{Integer}, \texttt{a} \doteq {}^?\texttt{Integer} \} \}
\end{aligned}$$

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.

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

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

Type–inference–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 < v1.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 ()

```
{α} mul({β} m) {
    {γ} ret = {Matrix} new Matrix();
    ...
    return {γ} 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 < v1.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);

$\{\alpha\}$ mul($\{\beta\}$ m) {
    ...
    $\{\iota\}$ v1 = ($\{$Matrix$\}$ this).elementAt(i);
    ...
}

Unification: Matrix $\lessdot$ Vector<ι>

$\Rightarrow$

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

return ret;

$\{\alpha\}$ $\underline{\texttt{mul}}$($\{\beta\}$ m) {
    ...
    return $\{\gamma\}$ 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:

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

where

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

Principal type

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 type

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."

Generalization to the Java 5.0 type system

"An intersection type-scheme for a declaration is a *principal type-scheme*, if any (non–intersection) type-scheme for the declaration is a subtype of a generic instance of one element of the intersection type-scheme."

Principal type property

Theorem

The type inference algorithm calculates a principal type.

Reduced principal type:

The inferred type of the example

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

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

is a principal type.

Reduced principal type:

The inferred type of the example

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

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

is a principal type.

But there is also a *reduced* principal type:

$$\texttt{mul:}\ \texttt{Vector<}_?\texttt{Vector<}_?\texttt{Integer>>} \longrightarrow \texttt{Matrix}$$

Implementation

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

Conclusion and future work

Conclusion

- Type-inference-algorithm for Java 5.0
- Type unification
- Principal type property

Conclusion and future work

Conclusion

- Type-inference-algorithm for Java 5.0
- Type unification
- Principal type property

Future work

- Reduce the number of calculated typings.
    - Reduce the number of unnecessary unifications
    - Calculate the minimal number of types which are necessary for a reduced principal type
- Handling of interesection types (adaption of byte-code generation)