Typeless Programming in Java 5.0 with Wildcards

Martin Plümicke

University of Cooperative Education
Stuttgart/Horb

PPPJ 2007, Portugal
6. September 2007
Overview

Introduction
  Problem
  Related work

The Algorithm
  Type unification
  Type–inference–algorithm
  Principal type
  Implementation

Conclusion
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.

  \[
  \text{Vector}\langle ? \text{extends AbstractList}\langle ? \text{super Integer}\rangle \rangle
  \]

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.
  \[
  \text{Vector}\langle\text{? extends AbstractList}\langle\text{? super Integer}\rangle\rangle
  \]

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)

⇒ 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; } 
    }
}

Martin Plümicke
Typeless Programming in Java 5.0 with Wildcards
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;
    }
}

Martin Plümicke
Typeless Programming in Java 5.0 with Wildcards
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)

\[
\text{mul}: \text{Matrix} \rightarrow \text{Matrix} \& \text{Matrix} \rightarrow \text{Vector}\langle\text{Vector}\langle\text{Integer}\rangle\rangle \& \ldots \& \text{Vector}\langle? \text{extends} \text{Vector}\langle? \text{extends} \text{Integer}\rangle\rangle \rightarrow \text{Vector}\langle? \text{super} \text{Vector}\langle\text{Integer}\rangle\rangle
\]
Two approaches

- OO-Type-inference [Palsberg, Schwartzbach, et.al.]:
  Precise types (no principal type)

- \( \lambda \)-Terms [Hindley/Milner, et.al.]:
  Principal type property
Our approach

[Hindley/Milner et al]
Our approach

[Hindley/Milner et al]
- function type constructor $\rightarrow$ (no higher-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)
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 \( A<\? \text{ extends } B> \) we write

\[ A<\?B> \]

and instead of \( C<\? \text{ super } D> \) we write

\[ C<\?D>. \]
The algorithm

Type unification

Subtyping relation for type terms: \( \leq^* \)

Type Unification problem:

For two type terms \( \theta_1 \) and \( \theta_2 \) a substitution \( \sigma \) is demanded such that:

\[
\sigma(\theta_1) \leq^* \sigma(\theta_2).
\]
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]
Subtyping relation:

- $\text{Integer} \leq^* \text{Number}$
- $\text{Stack}<a> \leq^* \text{Vector}<a> \leq^* \text{AbstractList}<a> \leq^* \text{List}<a>$

Application of the algorithm:

\[
\{ \text{Stack}<a> \mathbin{\leq^*} \text{Vector}<\text{?}\text{Number}> \}, \{ \text{AbstractList}<\text{Integer}> \mathbin{\leq^*} \text{List}<a> \}\n\]

\[
\Rightarrow \{ \text{a} \mathbin{\leq^?} \text{?}\text{Number} \}, \{ \text{Integer} \mathbin{\leq^?} \text{a} \}, \{ \text{a} \mathbin{\leq^?} \text{Number} \}, \{ \text{a} \mathbin{\leq^?} \text{Number} \}, \{ \text{a} \mathbin{\leq^?} \text{Number} \}, \{ \text{a} \mathbin{\leq^?} \text{Number} \}, \{ \text{a} \mathbin{\leq^?} \text{Number} \}, \{ \text{a} \mathbin{\leq^?} \text{Number} \}
\]
Example

Subtyping relation:

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

Stack$\langle a \rangle 
\leq^* \text{Vector}\langle a \rangle 
\leq^* \text{AbstractList}\langle a \rangle 
\leq^* \text{List}\langle a \rangle$

Application of the algorithm:

$\{ (\text{Stack}\langle a \rangle 
\leq \text{Vector}\langle ? \text{Number} \rangle), (\text{AbstractList}\langle \text{Integer} \rangle 
\leq \text{List}\langle a \rangle) \}$
Example

Subtyping relation:

\[
\text{Integer} \leq^* \text{Number} \\
\text{Stack}<a> \leq^* \text{Vector}<a> \leq^* \text{AbstractList}<a> \leq^* \text{List}<a>
\]

Application of the algorithm:

\[
\{ (\text{Stack}<a> \llarrow \text{Vector}<_?\text{Number}>), (\text{AbstractList}<\text{Integer}> \llarrow \text{List}<a>) \} \\
\Rightarrow \{ a \llarrow_?_? \text{Number}, \ \text{Integer} \llarrow_? a \} 
\]
Example

Subtyping relation:
- \text{Integer} \leq ^* \text{Number}
- \text{Stack}<a> \leq ^* \text{Vector}<a> \leq ^* \text{AbstractList}<a> \leq ^* \text{List}<a>

Application of the algorithm:
{ (\text{Stack}<a> \bowtie \text{Vector}<\text{Number}>, (\text{AbstractList}<\text{Integer}> \bowtie \text{List}<a>)) }

\Rightarrow \{ a \bowtie \text{Number}, \text{Integer} \bowtie a \}

\Rightarrow \{
\{ a \bowtie \text{Number}, a \bowtie \text{Integer} \}, \{ a \bowtie \text{Number}, a \bowtie \text{Number} \},
\{ a \bowtie \text{Number}, a \bowtie \text{Integer} \}, \{ a \bowtie \text{Number}, a \bowtie \text{Number} \},
\{ a \bowtie \text{Number}, a \bowtie \text{Number} \}, \{ a \bowtie \text{Number}, a \bowtie \text{Integer} \},
\{ a \bowtie \text{Integer}, a \bowtie \text{Integer} \}, \{ a \bowtie \text{Integer}, a \bowtie \text{Number} \},
\{ a \bowtie \text{Integer}, a \bowtie \text{Integer} \}, \{ a \bowtie \text{Integer}, a \bowtie \text{Number} \},
\{ a \bowtie \text{Integer}, a \bowtie \text{Number} \}, \{ a \bowtie \text{Integer}, a \bowtie \text{Integer} \},
\{ a \bowtie \text{Integer}, a \bowtie \text{Number} \}, \{ a \bowtie \text{Integer}, a \bowtie \text{Number} \}
\}
Example cont.

\[
\begin{align*}
\{ \text{Integer} \doteq \text{Number}, a \doteq \text{Integer} \}, & \{ \text{?Number} \doteq \text{?Number}, a \doteq \text{?Number} \}, \\
\{ \text{?Integer} \doteq \text{?Number}, a \doteq \text{?Integer} \}, & \{ \text{?Integer} \doteq \text{?Number}, a \doteq \text{?Integer} \}, \\
\{ \text{Integer} \doteq \text{Number}, a \doteq \text{Integer} \}, & \{ \text{?Number} \doteq \text{Number}, a \doteq \text{?Number} \}, \\
\{ \text{?Integer} \doteq \text{Number}, a \doteq \text{?Integer} \}, & \{ \text{?Integer} \doteq \text{Number}, a \doteq \text{?Integer} \}, \\
\{ \text{Integer} \doteq \text{?Integer}, a \doteq \text{Integer} \}, & \{ \text{?Number} \doteq \text{?Integer}, a \doteq \text{?Number} \}, \\
\{ \text{?Integer} \doteq \text{?Integer}, a \doteq \text{?Integer} \}, & \\
\{ \text{Integer} \doteq \text{Integer}, a \doteq \text{Integer} \}, & \{ \text{?Number} \doteq \text{Integer}, a \doteq \text{?Number} \}, \\
\{ \text{?Integer} \doteq \text{Integer}, a \doteq \text{?Integer} \} & \{ \text{?Integer} \doteq \text{Integer}, a \doteq \text{?Integer} \}
\end{align*}
\]
Example cont.

\[
\Rightarrow \\
\left\{ \begin{array}{ll} 
\text{Integer} \vdash \text{Number}, a \vdash \text{Integer} \},& \left\{ \begin{array}{ll} 
\text{Number} \vdash \text{Number}, a \vdash ?\text{Number} \}, \\
?\text{Integer} \vdash \text{Number}, a \vdash ?\text{Integer} \},& \left\{ ?\text{Integer} \vdash \text{Number}, a \vdash ?\text{Integer} \}, \\
\text{Integer} \vdash \text{Number}, a \vdash \text{Integer} \},& \left\{ ?\text{Number} \vdash \text{Number}, a \vdash ?\text{Number} \}, \\
?\text{Integer} \vdash \text{Number}, a \vdash ?\text{Integer} \},& \left\{ ?\text{Integer} \vdash \text{Number}, a \vdash ?\text{Integer} \}, \\
\text{Integer} \vdash ?\text{Integer}, a \vdash ?\text{Integer} \},& \left\{ ?\text{Number} \vdash ?\text{Integer}, a \vdash ?\text{Number} \}, \\
?\text{Integer} \vdash ?\text{Integer}, a \vdash ?\text{Integer} \},& \left\{ ?\text{Integer} \vdash ?\text{Integer}, a \vdash ?\text{Integer} \}, \\
\text{Integer} \vdash \text{Integer}, a \vdash \text{Integer} \},& \left\{ ?\text{Number} \vdash \text{Integer}, a \vdash ?\text{Number} \}, \\
?\text{Integer} \vdash \text{Integer}, a \vdash ?\text{Integer} \} & \left\{ ?\text{Integer} \vdash \text{Integer}, a \vdash ?\text{Integer} \} \right. \\
\end{array} \right. \\
\right. \\
\right. \\
\Rightarrow \left\{ \begin{array}{ll} 
\{ a \mapsto ?\text{Number} \},& \{ a \mapsto ?\text{Integer} \}, \{ a \mapsto \text{Integer} \} \right. \\
\end{array} \right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
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 corresponding 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} v1.elementAt(k))
                        * ({ψ}({φ} {β} m).elementAt(k)).elementAt(j)); k++;
                v2.addElement({χ} erg); j++;
            }
            ret.addElement({μ} v2); i++;
        }
        return ret; }
    }
```java
ret = new Matrix()

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

Unification: `Matrix < γ`

⇒

γ = Matrix
γ = Vector<Vector<Integer>>
γ = Vector<?Vector<Integer>>
γ = Vector<?Vector<?Integer>>
γ = Vector<?Vector<?Integer>>
γ = Vector<?Vector<Integer>>
```
Type assumptions after the first unification

class Matrix extends Vector<Vector<Integer>> {
    { α, α, α, α, α, α} mul({ β, β, β, β, β} m) {
        int i = 0; while(i <size()) {
            { l, l, l, l, l} 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}({ l, l, l, l, l} v1).elementAt(k))
                        * ( {ψ, ψ, ψ, ψ, ψ} 
                            ( {ϕ, ϕ, ϕ, ϕ, ϕ} 
                                ( { 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) {
    ...
    { i } v1 = (Matrix) this.elementAt(i);
    ...
}

**Unification:** Matrix ⋖ Vector<i>

⇒

i = Vector<Integer>
i = Vector<?Integer>
i = Vector<?Integer>
return ret;

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

**Unification:** γ ≼ α for

- γ = Matrix
- γ = Vector<Vector<Integer>>
- γ = Vector<?Vector<Integer>>

**Result:**

- α = Matrix
- α = Vector<Vector<Integer>>
- α = Vector<?Vector<Integer>>
- α = Vector<?Vector<?Integer>>
- α = Vector<?Vector<?Integer>>
- α = Vector<?Vector<?Integer>>
Result:

\[
\text{mul}: \mathcal\&_{\beta,\alpha}(\beta \rightarrow \alpha),
\]

where

\[
\beta \leq^* \text{Vector}<\text{Vector}<?\text{Integer}>>,
\]

Matrix \leq^* \alpha
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.”
The type inference algorithm calculates a principal type.
Reduced principal type:

The inferred type of the example

\[ \text{mul}: \&_{\beta,\alpha}(\beta \rightarrow \alpha), \]

where \( \beta \leq^* \text{Vector}\llcorner\text{Vector}\llcorner\text{Integer}\rrcorner\rrcorner \) and \( \text{Matrix} \leq^* \alpha. \)

is a principal type.
Reduced principal type:

The inferred type of the example

\[ \text{mul}: \beta,\alpha(\beta \rightarrow \alpha), \]

where \( \beta \leq^* \text{Vector}\langle \text{Vector}\langle ? \text{Integer} \rangle \rangle \) and \( \text{Matrix} \leq^* \alpha \).

is a principal type.

But there is also a reduced principal type:

\[ \text{mul}: \text{Vector}\langle \text{Vector}\langle ? \text{Integer} \rangle \rangle \rightarrow \text{Matrix} \]
Implementation

- Overloading–Example
- Return–Example
- Matrix–Example
Conclusion and future work

Conclusion

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

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 intersection types (adaption of byte-code generation).
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)