1. Fachtagung Informationstechnik und Informatik
an der Berufsakademie Baden–Württemberg

Untyped methods in Generic Java

Martin Plümicke

Berufsakademie Stuttgart – Außenstelle Horb

7. Juli 2004

Überblick

1. Generische Typen in Java
2. Vererbungshierarchie
3. Typ–System und –Inferenzregeln
4. Zusammenfassung und Ausblick

1. Generische Typen in Java

Erweitertes Typsystem von Java 1.5 (G–JAVA)

- Typvariablen
- parametrisierte Typen

Bsp.:

```
Seq<A>
Vector<A>
Pair<A,B>
```

Beispiel: parametrisierter `Vector`

```
//G–JAVA                          //Java

class Vector<A> {                 //class Vector {
                                  //
    void add (A elem) { ... }     //    void add (Object elem) { ... }
    A get () { ... }              //    Object get () { ... }
}                                 //}
```

Deklaration eines `Vectors`:

```
Vector<Integer> v = new Vector<Integer>();
```

Bedeutung: Der Vector `v` enthält nur Objekte des Typs `Integer`.

Verbesserung

1. **Vermeidung von Type–Casts:**

   Statt in Java

   ```
   Integer i = (Integer)v.get(6);
   ```

   genügt in G–JAVA

   ```
   Integer i = v.get(6);
   ```

2. **Laufzeitfehler werden bereits zur Compilationszeit festgestellt:**

   ```
   //Java                                  //G-Java
   Vector                                  //Vector<String>
       v = new Vector();                   //    v = new Vector<Integer>();
   v.add(new Integer(27));                 //v.add(new Integer(27));

   ((String)v.get(0)) ++ "Hallo";          //((String)v.get(0)) ++ "Hallo";
   ```

   In Java hat `v.get(0)` zur Compilationszeit den Typ `Object`
   und zur Laufzeit den Typ `Integer`
   ⇒ Type–Cast `((String)v.get(0))` führt zu Laufzeitfehler

   In G–JAVA hat `v.get(0)` bereits zur Compilationszeit den Typ `Integer`
   ⇒ Type–Cast `((String)v.get(0))` führt zu Compilationsfehler.

Typterme in G–JAVA

Menge der Typvariablen: TV

Rangalfabet der Typen: $\Theta = (\Theta^{(n)})_{n \in \mathbb{N}}$

Die Klassendeklaration

$$\texttt{class } C\texttt{<}a_1, \ldots, a_n\texttt{>}$$

ergibt

$$C \in \Theta^{(n)}$$

Bsp.:

- $\texttt{Seq}, \texttt{Vector} \in \Theta^{(1)}$
- $\texttt{Pair} \in \Theta^{(2)}$

Definition: Die Menge aller Terme über Θ (Bez. $T_\Theta(TV)$) sind die Typen des G–JAVA–Programms.

Bsp.:

```
Pair<Seq<Integer>, Vector<A>>
```

2. Vererbungshierarchie

Typterm Ordnung $\leq$: Die *extends* bzw. *implements* Declaration

(`class | interface`) $C\texttt{<}a_1, \ldots, a_n\texttt{>}$ (`extends | implements`) $\tau'_1 \ldots \tau'_m$

definiert

$$C\texttt{<}a_1, \ldots, a_n\texttt{>} \leq \tau'_1$$

$$\ldots$$

$$C\texttt{<}a_1, \ldots, a_n\texttt{>} \leq \tau'_m.$$

Hülle der Typterm Ordnung $\leq^*$: Kleinste Ordnung mit den Bedingungen:

- $(\theta, \theta') \in T_\Theta(TV) \times T_\Theta(TV)$ ist Element der reflexiven und transitiven Hülle von $\leq$
- Wenn $\theta_1 \leq^* \theta_2$ dann $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ für alle Substitutionen σ_1, σ_2, mit $\sigma_1(a) = \sigma_2(a)$ für alle $a \in \mathsf{TVar}(\theta_2)$.

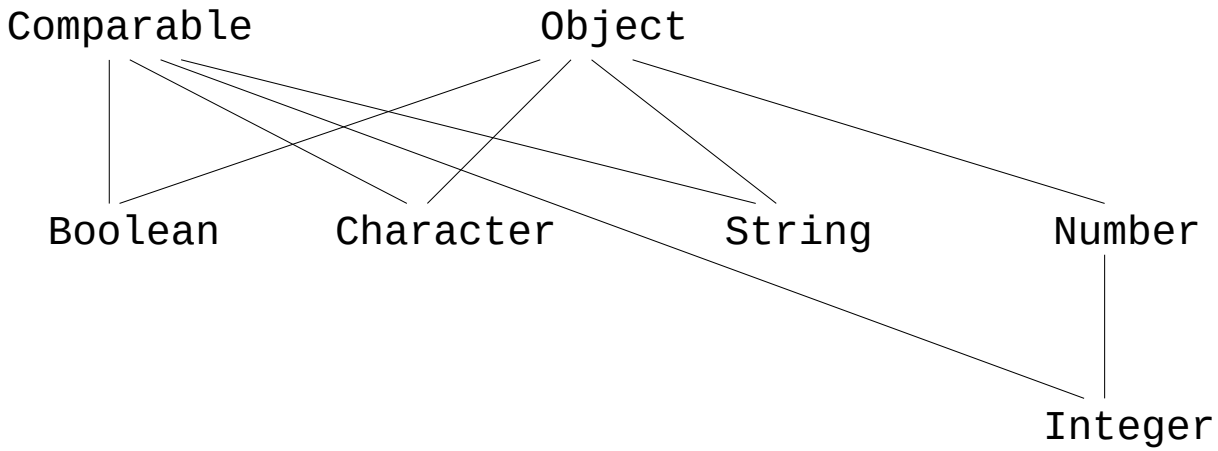
Contravariance Problem

```
class Super { ...}

class Sub extends Super { ...}

class Vector<a> {

    void add (a elem) { ...}
}

class Main {
  public static void main(String[] args) {
    Vector<Super> v;
    v = new Vector<Sub> ();
    v.add(new Super());
  }
}
```

$\text{Super} \not\leq^* \text{Sub}$

Vererbungshierarchie von G–JAVA

Relation $\leq$ als Hasse–Diagramm:

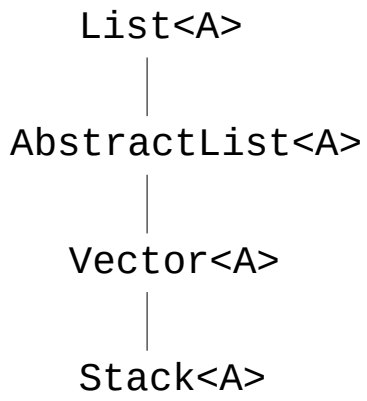


Programmdeklarationen:

```
class Object { ... }
class Boolean extends Object { ... }
interface Comparable { ... }
class Character extends Object implements Comparable { ... }
class Number extends Object { ... }
class Integer extends Number implements Comparable { ... }
class String extends Object implements Comparable { ... }
```

Parametrisierte Listen

Relation $\leq$ als Hasse–Diagramm:



Programmdeklarationen:

```
interface List<A> { ... }
abstract class AbstractList<A> implements List<A> { ... }
class Vector<A> extends AbstractList<A> { ... }
class Stack<A> extends Vector<A> { ... }
```

Beispielelemente der zugehörigen partiellen Ordnung $\leq^*$:

- `Stack<Integer>` $\leq^*$ `Vector<Integer>`
- `Stack<Vector<Integer>>` $\leq^*$ `Vector<Vector<Integer>>`
- `Stack<Stack<Integer>>` $\leq^*$ `List<Stack<Integer>>`

3. Typsystem und –Inferenz

Beispiel: Matrix–Multiplikation **G–JAVA**

```
class Matrix extends Vector<Vector<Integer>> {

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

G–JAVA Typ–System

Definition: Die Menge der *G–JAVA–Typen* $\mathsf{Type}(T_\Theta(TV))$ ist definiert als kleinste Menge mit folgenden Eigenschaften:

1. Sei $\mathsf{Basetype}$ die Menge G–JAVA Basistypen.
   Dann gilt $\mathsf{Basetype} \subseteq \mathsf{Type}(T_\Theta(TV))$ (**basetype**).
2. $T_\Theta(TV) \subseteq \mathsf{Type}(T_\Theta(TV))$ (**simple type**).
3. Für $0 \leqslant i \leqslant n$: $\theta_i \in (T_\Theta(TV) \cup \mathsf{basetype})$ gilt
   $\theta_1 \times \ldots \times \theta_n \to \theta_0 \in \mathsf{Type}(T_\Theta(TV))$ (**function type**).
4. Für ty_1, $ty_2 \in \mathsf{Type}(T_\Theta(TV))$, gilt $ty_1 \wedge ty_2 \in \mathsf{Type}(T_\Theta(TV))$
   (**intersection type**).

Typ–Inferenz

Grundsätzliches Ziel: Bestimmen von Typen für ungetypt programmierte Ausdrücke.

Problem in G–JAVA: Bei der klassischen Typisierung (λ–Kalkül bis Haskel) kann man alle Sprach–Konstrukte als Ausdrücke ansehen. Dies macht bei G–JAVA keinen Sinn.

Man unterscheidet in G–JAVA die Sprach–Konstrukte in:

- Identifier (Variablen- und Funktionsnamen)
- Expressions (z.B. `2 + 5`)
- Statements (`while`, `for`, ...)
- Expressionstatements (Zuweisung, `new`–Operator, Methodenaufruf)

Typ–Annahmen

Definition: Die Menge der Typ–Annahmen A ist eine durch Klassennamen indizierte Menge in der Identifier (Variable, Methodennamen) G–JAVA–Typen zugeordnet werden.

Bsp.:

$$A_{\texttt{Math}} = \{\, \texttt{+} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int},\ \texttt{sqr} : \texttt{int} \rightarrow \texttt{int} \,\}$$

und

$$A_{\texttt{Vector}} = \{\, \texttt{add} : \texttt{Object} \rightarrow \texttt{void},\ \texttt{len} :\rightarrow \texttt{int} \,\}$$

bilden die Menge von Typ–Annahmen

$$A = \{\, A_{\texttt{Math}}, A_{\texttt{Vector}} \,\}.$$

Typ–Inferenz–Algorithmus

Idee: Typberechnung einer Methode

```
Rtyp name (ty_1 p_1 , ..., ty_n p_n) { stmts }.
```

1. Man belegt die gesuchten Typen zunächst mit Typvariablen:
   - `p_1` : α_1
   - `p_2` : α_2
   - ...
   - `p_n` : α_n
   - `name` : $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_R$
2. Man fügt diese Typ–Annahmen zu den Typ–Annahmen der bekannten Methoden hinzu.
3. Man geht durch den abstrakten Syntaxbaum der Klasse durch und unifiziert die Typ–Annahmen miteinander.
4. Als Ergebnis erhält man (mehrere) Unifikatoren. Diese wendet man auf die angenommenen Typvariablen an und erhält so, (mehrere) Typen der Methode. Diese fasst man zu einem Intersection–Typ zusammen.

Unifikationsproblem

Für zwei Typterme t_1 und t_2 ist eine Substitution σ mit folgenden Eigenschaften gesucht:

$$\sigma(t_1) \leq^* \sigma(t_2)$$

Ist durch einen Algorithmus [Plümicke, Unif'04] gelöst.

Eingeschränkungen: G–JAVA–Programme ohne

- Interfaces
- F–bounded Parameter
- *extends*–Deklarationen mit zusätzlichen Typvariablen

Typ–Inferenz Beispiel

```
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(erg);
             j++; }
         ret.addElement(v2);
         i++; }
     return ret; }
```

Typ–Annahmen

$$O_{\texttt{Vector<A>}} = \{\, \texttt{elementAt} : \texttt{A} \rightarrow \texttt{int} \quad \texttt{addElement} : \texttt{A} \rightarrow \texttt{void} \,\}$$

und

$$V_1 = \{\, \texttt{mul} : \alpha_2 \rightarrow \alpha_1, \texttt{m} : \alpha_2 \,\}.$$

Die Typen für α_2, α_1 werden nun Schritt für Schritt hergeleitet.

– $V_1 = \{\texttt{mul} : \alpha_2 \to \alpha_1, \texttt{m} : \alpha_2, \texttt{ret} : \texttt{Matrix}\}$.

– $V_2 = \{\texttt{mul} : \alpha_2 \to \alpha_1, \texttt{m} : \alpha_2, \texttt{ret} : \texttt{Vector<Vector<Integer>>}\}$.

```
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(erg);
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; }
```

– $V_1 = \{\texttt{mul} : \alpha_2 \to \alpha_1, \texttt{m} : \alpha_2, \texttt{ret} : \texttt{Matrix}, \texttt{i} : \texttt{int}\}$.

– $V_2 = \{\texttt{mul} : \alpha_2 \to \alpha_1, \texttt{m} : \alpha_2, \texttt{ret} : \texttt{Vector<Vector<Integer>>}, \texttt{i} : \texttt{int}\}$.

```
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(erg);
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; }
```

- $V_1 = \{$ mul : $\alpha_2 \rightarrow \alpha_1$, m : α_2, ret : Matrix, i : int,
  v1 : Vector<Integer>, v2 : Vector<Integer> $\}$.
- $V_2 = \{$ mul : $\alpha_2 \rightarrow \alpha_1$, m : α_2, ret : Vector<Vector<Integer>>, i : int,
  v1 : Vector<Integer>, v2 : Vector<Integer> $\}$.

```
class Matrix extends Vector<Vector<Integer>> {

 mul(m) { ...
    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(erg);
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; }
```

- $V_1 = \{$`mul : Vector<`α_{10}`> → `α_1`, m : Vector<`α_{10}`>,`
  `ret : Matrix,`
  `v1 : Vector<Integer>, v2 : Vector<Integer>`$\}$.
- $V_2 = \{$`mul : Vector<`α_{10}`> → `α_1`, m : Vector<`α_{10}`>,`
  `ret : Vector<Vector<Integer>>,`
  `v1 : Vector<Integer>, v2 : Vector<Integer>`$\}$.
- $V_3 = \{$`mul : Matrix → `α_1`, m : Matrix, ret : Matrix,`
  `v1 : Vector<Integer>, v2 : Vector<Integer>`$\}$.
- $V_4 = \{$`mul : Matrix → `α_1`, m : Matrix, ret : Vector<Vector<Integer>>,`
  `v1 : Vector<Integer>, v2 : Vector<Integer>`$\}$.

```
class Matrix extends Vector<Vector<Integer>> {
   mul(m) {
            ...
            while (k < v1.size()) {
                erg = erg + v1.elementAt(k)
                    * (m.elementAt(k)).elementAt(j);
                k++; }
            ...
          }
```

- $V_1 = \{$`mul : Vector<Vector<Integer>>` $\rightarrow \alpha_1$, `m : Vector<Vector<Integer>>`,
  `ret : Matrix`,
  `v1 : Vector<Integer>, v2 : Vector<Integer>` $\}$.
- $V_2 = \{$`mul : Vector<Vector<Integer>>` $\rightarrow \alpha_1$, `m : Vector<Vector<Integer>>`,
  `ret : Vector<Vector<Integer>>`,
  `v1 : Vector<Integer>, v2 : Vector<Integer>` $\}$.
- $V_3 = \{$`mul : Matrix` $\rightarrow \alpha_1$, `m : Matrix, ret : Matrix`,
  `v1 : Vector<Integer>, v2 : Vector<Integer>` $\}$.
- $V_4 = \{$ `mul : Matrix` $\rightarrow \alpha_1$, `m : Matrix, ret : Vector<Vector<Integer>>`,
  `v1 : Vector<Integer>, v2 : Vector<Integer>` $\}$.

```
class Matrix extends Vector<Vector<Integer>> {
   mul(m) {
            ...
            while (k < v1.size()) {
                erg = erg + v1.elementAt(k)
                    * (m.elementAt(k)).elementAt(j);
                k++; }
            ...
          }
```

- $V_1 = \{$`mul : Vector<Vector<Integer>>` $\rightarrow$ `Matrix`,
  `m : Vector<Vector<Integer>>`,
  `ret : Matrix` $\}$.
- $V_2 = \{$`mul : Vector<Vector<Integer>>` $\rightarrow$ `Vector<Vector<Integer>>`,
  `m : Vector<Vector<Integer>>`,
  `ret : Vector<Vector<Integer>>` $\}$.
- $V_3 = \{$`mul : Matrix` $\rightarrow$ `Matrix`, `m : Matrix`, `ret : Matrix` $\}$.
- $V_4 = \{$ `mul : Matrix` $\rightarrow$ `Vector<Vector<Integer>>`, `m : Matrix`,
  `ret : Vector<Vector<Integer>>` $\}$.

```
class Matrix extends Vector<Vector<Integer>> {

  mul(m) {
    ret = new Matrix ();
    while(i < size()) { ...
        while (j < v1.size()) { ...
            while (k < v1.size()) { ...
            }
        }
    }
    return ret; }
```

Ergebnis

Als Ergebnis ergibt sich der Intersection–Typ

$$
\begin{aligned}
\texttt{mul} \quad &: \quad \texttt{Vector<Vector<Integer>>} \longrightarrow \texttt{Matrix} \\
&\wedge \quad \texttt{Vector<Vector<Integer>>} \longrightarrow \texttt{Vector<Vector<Integer>>} \\
&\wedge \quad \texttt{Matrix} \longrightarrow \texttt{Matrix} \\
&\wedge \quad \texttt{Matrix} \longrightarrow \texttt{Vector<Vector<Integer>>}
\end{aligned}
$$

Vergleich mit dem Eingangsbeispiel

```
class Matrix extends Vector<Vector<Integer>> {

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

4. Zusammenfassung und Ausblick

Zusammenfassung

1. Java 1.5 (G–JAVA) enthält Typvariablen und generische Typen.
2. Die Vererbungshierarchie lässt sich als partielle Ordnung beschreiben.
3. Das Typinferenz–Problem in G–JAVA führt auf ein Unifikationsproblem, das für bestimmte Einschränkungen gelöst werden konnte.
4. Daraus ergibt sich ein Typinferenz–Algorithmus

Ausblick

- Implementierung des Typinferenz–Algorithmus
- Einbau des Systems in Programmierumgebungen wie Eclipse oder Netbeans.