KOLLOQUIUM PROGRAMMIERUNG

Ein parametrisch polymorphes Typ–System für **Java**

Martin Plümicke

30. Januar 2004

Überblick

1. Parametrisch polymorphe Typen in Java
2. Vererbungshierarchie
3. Typ–System und –Inferenzregeln
4. Ansätze für einen Typrekonstruktionsalgorithmus

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 `Vector`s:

```
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<String>();
   v.add("Hallo");                     //v.add("Hallo");

   ((Integer)v.get(0)).intValue()      //((Integer)v.get(0)).intValue()
   ```

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

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

Parameter Constraints

- Parameter muss bestimmtes Interface implementieren (bestimmte Methoden enthalten)

Bsp.:

```
class ReprChange<A implements ConvertibleTo<B>,
                 B implements ConvertibleTo<A>> {
    A a;

    void set(B x) { a = x.convert(); }
    B get() { return a.convert(); }
}
```

mit dem Interface

```
interface ConvertibleTo<A> {
    A convert();
}
```

Instanzierte Parameter Constraints

```
class INT implements ConvertibleTo<FLOAT> {
    int i;

    INT(int i) { this.i = i; }
    public FLOAT convert() {
       return new FLOAT((new Float((float)i)).floatValue()); }
}

class FLOAT implements ConvertibleTo<INT> {
    float i;

    FLOAT(float i) { this.i = i; }
    public INT convert() { return new INT((new Integer((int)i)).intValue()); }
}

class Main {
    public static void main(String[] args) {
        ReprChange<INT, FLOAT> r = new ReprChange<INT, FLOAT> ();
        r.set(new FLOAT(2));
        System.out.println(r.get().i);
    }

}
```

Typterme in G–JAVA

Menge der Typvariablen: TV

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

Die Klassendeklaration

$$\texttt{class } C\texttt{<}a_1 \texttt{ [implements } I_1\texttt{]}, \ldots, a_n \texttt{ [implements } I_n\texttt{]>}$$

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>>
```

Vererbungshierarchie

Definition: Die Vererbungshierarchie wird bestimmt als partielle Ordnung $\leq^*$ durch *Hüllenbildung über Substitution, Reflexivität und Transitivität* der durch folgende Regeln bestimmten Relation $\leq$:

1. `class` C`<`$a_1, \ldots, a_n$`>` `extends` τ' ergibt

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

2. `class` C`<`$a_1, \ldots, a_n$`>` `implements` $\tau'_1, \ldots, \tau'_p$ ergibt

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

3. `class` C`<`a_1 `implements` I_1`<`$b_{1,1}, \ldots, b_{1,o_1}$`>`$, \ldots,$
   a_n `implements` I_n`<`$b_{n,1}, \ldots, b_{n,o_n}$`>>`
   `extends` τ' `implements` $\tau'_1, \ldots, \tau'_p$

   ergibt

$$C\texttt{<}I_1\texttt{<}b_{1,1}, \ldots, b_{1,o_1}\texttt{>}, \ldots, I_n\texttt{<}b_{n,1}, \ldots, b_{n,o_n}\texttt{>>} \leq \tau'$$

   und für $0 \leqslant j \leqslant p$:

$$C\texttt{<}I_1\texttt{<}b_{1,1}, \ldots, b_{1,o_1}\texttt{>}, \ldots, I_n\texttt{<}b_{n,1}, \ldots, b_{n,o_n}\texttt{>>} \leq \tau'_j,$$

   wobei für $1 \leqslant i \leqslant n$: $\{\, b_{i,1}, \ldots, b_{i,o_i} \,\} \subseteq \{\, a_1, \ldots, a_n \,\}$.

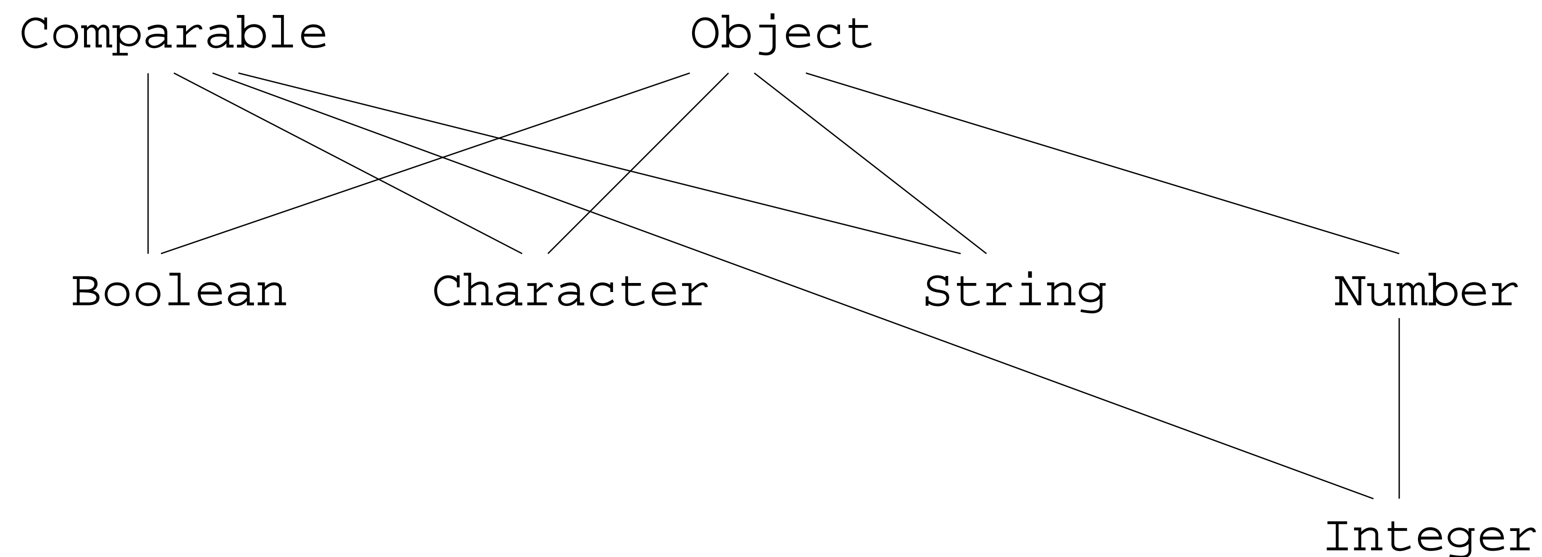
Matrix–Multiplikation **G–JAVA**

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

 Matrix mul(Matrix m) {
    Matrix ret = new Matrix ();
    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).intValue()
                    * (m.elementAt(k)).elementAt(j).intValue();
                k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; }
```

Vererbungshierarchie von G–JAVA

Relation $\leq$ als Hasse–Diagramm:

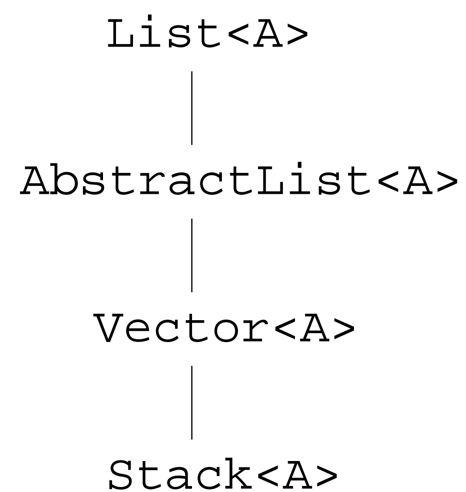


Programmdeklarationen:

```
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>>` $\leq^*$ `Vector<Vector<Integer>>`

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 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**).

5. Wenn $ty \in \mathsf{Type}(T_\Theta(TV))$ kein intersection type ist und $\mathtt{A} \in TV$ dann gilt:
   $\forall \mathtt{A}.ty \in \mathsf{Type}(T_\Theta(TV))$ (**type scheme**).

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)

Idee des Typ–Inferenz Systems

Ziel: Bestimmung der Typen aller Methoden einer Klasse.

Aufbau einer Methode:

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

- Methodenname `name`
- Rückgabetyp `Rtyp`
- typisierte Parameter `(ty_1 p_1 , ..., ty_n p_n)`
- Block bestehend aus Statements `stmts`

Vorgehen bei der Typ–Inferenz

Bestimmen der Typen `Rtyp` und `ty_1 , ..., ty_n`:

Rückgabetyp `Rtyp`:

- Der `Rtyp` ergibt sich aus dem Statement `return` *expr*
- Zunächst wird der Typ von *expr* bestimmt
- Dieser Typ wird dem Return–Statement zugeordnet
- Der Typ eines Statement wird an eine Liste von Statements propagiert

So erhält man schließlich den Typ `Rtyp` von `stmts`

Parametertypen `ty_1 , ..., ty_n`:

- Die Typen `ty_1 , ..., ty_n` werden durch die Methodenanwendung auf die Variablen in Expressions bestimmt.

⇒ Wir benötigen Regel für die Typherleitungen von

- Expressions
- Statements
- Identifier

Grundsätzlicher Aufbau von Typ–Inferenz Regeln

Idee von Typ–Inferenz Regeln: Es werden bestimmte Typ–Annahmen für Identifier gemacht. Diese werden dann mit Hilfe der gegebenen Regeln bewiesen.

Typ–Annahmen

Definition: Die Menge der Typ–Annahmen O ist eine Zuordnung der Identifier (Variable, Methodennamen) zu G–JAVA–Typen.

Bsp.:

$$O_{\texttt{Math}} = \{\, \texttt{+} : \texttt{int} \times \texttt{int} \to \texttt{int},\\ \texttt{sqr} : \texttt{int} \to \texttt{int} \,\}$$

und

$$O_{\texttt{Vector}} = \{\, \texttt{add} : \texttt{Object} \to \texttt{void},\\ \texttt{len} : \to \texttt{int} \,\}$$

bilden die Menge von Typ–Annahmen

$$O = \{\, O_{\texttt{Math}}, O_{\texttt{Vector}} \,\}.$$

Expression Typ–Inferenz Regeln

[**Add**]

$$\frac{(O,\tau,\tau') \;\rhd_{Expr}\; e_1 : \texttt{int}, \quad (O,\tau,\tau') \;\rhd_{Expr}\; e_2 : \texttt{int}}{(O,\tau,\tau') \;\rhd_{Expr}\; \mathsf{Add}(e_1, e_2) : \texttt{int}}$$

[**Method-Call**]

$$\frac{(O,\tau,\tau') \;\rhd_{Expr}\; re : \overline{\theta} \quad \forall 1 \leqslant i \leqslant n : \; (O,\tau,\tau') \;\rhd_{Expr}\; e_i : \theta_i, \quad O_{\overline{\theta}'} \;\rhd_{Id}\; f : \theta_1 \times \ldots \times \theta_n \to \theta}{(O,\tau,\tau') \;\rhd_{Expr}\; \mathsf{MethodCall}(re, f(e_1, \ldots, e_n)) : \theta} \quad \overline{\theta}' = \texttt{lub}(f_{\theta_1 \ldots \theta_n}, \overline{\theta})$$

[**Sub**]

$$\frac{(O,\tau,\tau') \;\rhd_{Expr}\; e : \tau, \quad \tau \leq^* \tau'}{(O,\tau,\tau') \;\rhd_{Expr}\; e : \tau'}$$

Statement Typ–Inferenz Regeln

[**StmtInit**]

$$\frac{stmt \in \{\, s \mid s \ \textit{is a statement}, s \neq \mathsf{Return}(e) \,\}}{(O,\tau,\tau') \ \rhd_{Stmt} \ stmt : \texttt{void}}$$

[**BlockInit**]

$$\frac{(O,\tau,\tau') \ \rhd_{Stmt} \ stmt : \theta}{(O,\tau,\tau') \ \rhd_{Stmt} \ \mathsf{Block}(stmt) : \theta}$$

[**Block**]

$$\frac{(O,\tau,\tau') \ \rhd_{Stmt} \ s_1 : \texttt{void}, \quad (O,\tau,\tau') \ \rhd_{Stmt} \ \mathsf{Block}(s_2; \ldots; s_n;) : \theta}{(O,\tau,\tau') \ \rhd_{Stmt} \ \mathsf{Block}(s_1; \ldots; s_n;) : \theta}$$

[**Return**]

$$\frac{(O,\tau,\tau') \ \rhd_{Expr} \ e : \theta}{(O,\tau,\tau') \ \rhd_{Stmt} \ \mathsf{Return}(e) : \theta}$$

Identifier Typ–Inferenz Regeln

$$[\textbf{Ident}] \quad \frac{}{(O \cup \{\, id : ty \,\})_\theta \;\rhd_{Id}\; id : \sigma|_{\{\, A_1,\ldots,A_n \,\}}(ty)} \quad \sigma \in \textbf{subst}(T_\Theta(TV))$$

where $ty = \forall A_1.\ldots.\forall A_n.\tau$

and τ *is no type scheme*

- $\textbf{subst}(T_\Theta(TV))$ ist die Menge aller Substitutionen, die Variablen in Typen durch andere Typen ersetzt

Klassen Typ–Inferenz Regel

$$
\textbf{[Class]}\;
\dfrac{
\begin{array}{l}
p \blacktriangleright O \\
\forall 1 \leqslant i \leqslant m : \\
\quad (O \cup \{ \tau \mapsto (\overline{O}_\tau \cup O_{field} \cup \{ x_{i,1} : \theta_{i,1}, \ldots, x_{i,k_i} : \theta_{i,k_i} \}) \}, \tau, \tau') \\
\hfill \triangleright_{Stmt} \mathsf{Block}(B_i) : \theta_i \\
\text{where} \\
\overline{O}_\tau = \\
\quad \{ M_1 : \theta_{1,1} \times \ldots \times \theta_{1,k_1} \to \theta_1, \ldots, M_m : \theta_{m,1} \times \ldots \times \theta_{1,k_m} \to \theta_m \} \\
O_{field} = \{ v_1 : \tau_1, \ldots, v_n : \tau_n \}
\end{array}
}{
\begin{array}{l}
(p \cup \mathsf{Class}(\mathsf{ClassName}(\tau), \mathsf{extends}(\tau') \\
\qquad \mathsf{InstVarDecl}(v_1, \tau_1), \ldots, \mathsf{InstVarDecl}(v_n, \tau_n) \\
\qquad \mathsf{Method}(M_1, \theta_1, (x_{1,1} : \theta_{1,1}, \ldots, x_{1,m_1} : \theta_{1,k_1}), \mathsf{Block}(B_1)) \\
\qquad , \ldots , \\
\qquad \mathsf{Method}(M_m, \theta_m, (x_{m,1} : \theta_{m,1}, \ldots, x_{m,m_m} : \theta_{m,k_m}), \\
\hfill \mathsf{Block}(B_m))) \\
\quad \blacktriangleright O \cup \{ \tau \mapsto \mathtt{gen}(\overline{O}_\tau \cup O_{field}) \}
\end{array}
}
$$

Intersection Typ–Inferenz Regel

$$[\textbf{IntSec}]\ \frac{\begin{array}{c} p\ \blacktriangleright O \cup \{\, \tau \mapsto \{\, M_1 : ty_1, \ldots, M_k : ty_k, \ldots, M_m : ty_m \,\}\,\} \\ p\ \blacktriangleright O \cup \{\, \tau \mapsto \{\, M_1 : ty_1, \ldots, M_k : ty'_k, \ldots, M_m : ty_m \,\}\,\} \end{array}}{p\ \blacktriangleright O \cup \{\, \tau \mapsto \{\, M_1 : ty_1, \ldots, M_k : ty_k \wedge ty'_k, \ldots, M_m : ty_m \,\}\,\}}$$

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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; }
     return ret; }
```

Typ–Annahmen

$$O_{\texttt{Vector<A>}} = \{ \quad \begin{array}{l} \texttt{elementAt} : \forall\texttt{A.A} \rightarrow \texttt{int} \\ \texttt{addElement} : \forall\texttt{A.A} \rightarrow \texttt{void} \} \end{array}$$

und

$$O_{\texttt{Integer}} = \{ \quad \begin{array}{l} \texttt{<init>}_{\texttt{Integer}} : \texttt{int} \rightarrow \texttt{Integer} \\ \texttt{intValue} : \rightarrow \texttt{int} \}. \end{array}$$

Die Menge $O_{\texttt{Matrix}}$ wird nun Schritt für Schritt hergeleitet.

- Typherleitung für den *Identifier* `ret` (Regel **Ident**)
- Bestimmung der Typ–Annahme `ret` : `Matrix`

  $O_{\texttt{Matrix}}$ wird um die Typ–Annahme ergänzt.

```
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).intValue()
                    * (m.elementAt(k)).elementAt(j).intValue();
                k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret : Matrix; }
```

- Typherleitung für die *Expression* `ret` (Regel **LocalOrFieldVar**)

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; }
     return ret : Matrix; }
```

- Typherleitung für das *Statement* `return ret` (Regel **Return**)

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; }
     return ret : Matrix; }
```

- Typherleitung für die Block–Initialisierung (Regel **BlockInit**)

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; }
     return ret : Matrix; }
```

- Typherleitung für die *Statements* `i++` und `ret.addElement(v2)` (Regeln **AssignStmt** und **MethodCall**)

```
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).intValue()
                    * (m.elementAt(k)).elementAt(j).intValue();
                k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2) : void;
        i++ : void; }
    return ret : Matrix; }
```

- Anwendung der Regel **Block**

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++ : void; }
     return ret : Matrix; }
```

- Typherleitung für *Statement* `k++` (Regel **AssignStmt**)
- Bestimmung der Typ–Annahme von m : `Vector<Vector<Integer>>`

  Begründung: ergibt sich aus `m.elementAt(k)).elementAt(j).intValue()`

  $O_{\texttt{Matrix}}$ wird um die Typ–Annahme ergänzt

```
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).intValue()
                    * (m.elementAt(k)).elementAt(j).intValue();
                k++ : void; }
            v2.addElement(new Integer(erg));
            j++ : void; }
        ret.addElement(v2);
        i++ : void; }
    return ret : Matrix; }
```

- Anwendung verschiedener Regeln

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; } : void
     return ret : Matrix; }
```

- Anwendung der Block–Regel

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; }
     return ret : Matrix; }
```

Analog ergibt sich:

```
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).intValue()
                     * (m.elementAt(k)).elementAt(j).intValue();
                 k++; }
             v2.addElement(new Integer(erg));
             j++; }
         ret.addElement(v2);
         i++; }
     return ret; } : Matrix
```

Ergebnis

- Typ–Annahme `Matrix` für den methodendefinierenden Block ist bewiesen
- Typ–Annahme `m` : `Vector<Vector<Integer>>` ist ebenfalls bewiesen.

Daraus ergibt sich:

`mul` : `Vector<Vector<Integer>>` ⟶ `Matrix`.

Anwendung der Intersection–Regel

Erinnerung an das Anfangsbeispiel. Die Typ–Annahme `m : Matrix` ist ebenfalls korrekt. (lässt sich auch herleiten)

Daraus ergibt sich:

$$\texttt{mul} : \texttt{Matrix} \longrightarrow \texttt{Matrix}.$$

mit der Intersection–Regel erhalten wir:

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

Typrekonstruktionsalgorithmus

Idee: Typberechnung einer Methode

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

1. Man belegt die gesuchten Typen zunächst mit Typvariablen:
   - `ty_1 : A_1`
   - `ty_2 : A_2`

     ...
   - `ty_n : A_n`
   - `name` : `A_1` × ... × `A_n` → `A_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:

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

Gert Smolka (Diss): Offenes Problem!

Ansätze:

1. Upper Matcher ($t_1 \leq^* \sigma(t_2)$), Lower Matcher ($\sigma(t_1) \leq^* t_2$) (Smolka)
2. Algorithmus, der nicht alle Unifikatoren findet.

$\Rightarrow$ Beide Ansätze werden u.U. nicht den allgemeinsten Typ finden (Es werden einzelne Elemente des Intersection–Typs nicht gefunden).

Zusammenfassung

1. Java 1.5 (G–JAVA) wird Typvariablen und generische Typen enthalten.
2. Die Vererbungshierarchie lässt sich als partielle Ordnung beschreiben.
3. Allgemeinste Typen von G–JAVA lassen sich als Intersection Typ-Schemata beschreiben.
4. Mit Hilfe von Typ–Inferenz Regeln für Identifier, Expressions, Statements kann man verschiedene korrekte Typen einer Methode herleiten.
5. Die Typrekonstruktion führt auf ein Unifikationsproblem, das zumindest teilweise lösbar ist und damit Elemente des Intersection Typ-Schemas der Methode bestimmt.