Typ–Inferenz in Java 5.0

Martin Plümicke

Berufsakademie Stuttgart

5. Mai 2006

Überblick

1. Problemstellung
2. Java 5.0 Typssystem
3. Typ–Unifikation
4. Typ–Inferenz
5. Zusammenfassung, Fazit und Ausblick

1. Problemstellung

Erweiterung des **Java 5.0** Typsystems

- Parametrisierte Typen, Typvariablen, Typterme, Wildcards
- Beispiele:
  - Klassenparameter:

```
class Vector<E> extends AbstractList<E> { ... }
```

    oder

```
class Pair<E,F> { ... }
```

  - Methodenparameter:

```
<E> E id (E arg) { return arg; }
```

  - Typterme:

```
Vector<Vector<Pair<Integer, Integer>>>
```

  - Wildcards:

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

Problem: Komplexe Typisierung

- Oftmals ist es nicht offensichtlich, wie ein sinnvoller Typ aussehen müsste.
- Java 5.0 Code hätte als allgemeinste Typen oftmals *Durchschnittstypen*, die in Java 5.0 nicht ausdrückbar sind.

$\Longrightarrow$ Entwicklung eines Typ–Inferenz–Systems, das den Programmierer unterstützt und allgemeinst mögliche Typisierungen berechnet.

Beispiel: Matrix–Multiplikation

```
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 Typisierung

```
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; } }
```

Ziel: Typlos programmieren

```
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; } }
```

Ziel: System bestimmt Typen

1. System berechnet allgemeinsten (Durchschnitts)typ:

   `mul:` `Matrix` → `Matrix` &

   `Matrix` → `Vector<Vector<Integer>>` &

   `Vector<Vector<Integer>>` → `Matrix` &

   `Vector<Vector<Integer>>` → `Vector<Vector<Integer>>`

2. Weiterverarbeitung:

   Eine der folgenden Alternativen:

   - Benutzer wählt gewünschten Typ aus.
   - Codeerzeugung für jedes Elements des Durchschnittstyps (Generierung von überladenen Funktionen)
   - Erweiterung der JVM um Durchschnittstypen

Idee

Grundlage: Polymorphically oder-sorted types

- Typsystem eine logischen Programmiersprache [Smolka 1989]
  - **Offenes Problem:** Ist die Typ–Unifikation entscheidbar?
- Typsystem von OBJ–P einer Erweiterung von OBJ–3 [Plümicke 1999]

2. Typsystem (ohne Wildcards)

Definition 1 (Simpletypes)

- *Typvariablen sind Simpletypes*
- *Sei* $\Theta = (\Theta^{(n)})_{n \in \mathbb{N}}$ *Rangalphabet der Klassen/Interface–Namen*
  $\Rightarrow$ *Terme über* Θ $(T_\Theta(TV))$ *sind Simpletypes*

Definition 2 (Extends relation $\leq$)

1. `class`/`interface` `C<`$a_1, \ldots, a_n$`>` `extends` τ' *definiert*

$$C\texttt{<}\mathtt{a}_1, \ldots, \mathtt{a}_n\texttt{>} \leq \tau'$$

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

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

$$, \ldots,$$

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

Definition 3 (Typtermordnung $\leq^*$)

- *alle Elemente der reflexiven und transitiven Hülle von* $\leq$.
- *wenn* $\theta_1 \leq^* \theta_2$, *dann gilt für Substitutionen* σ_1, σ_2

$$\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2),$$

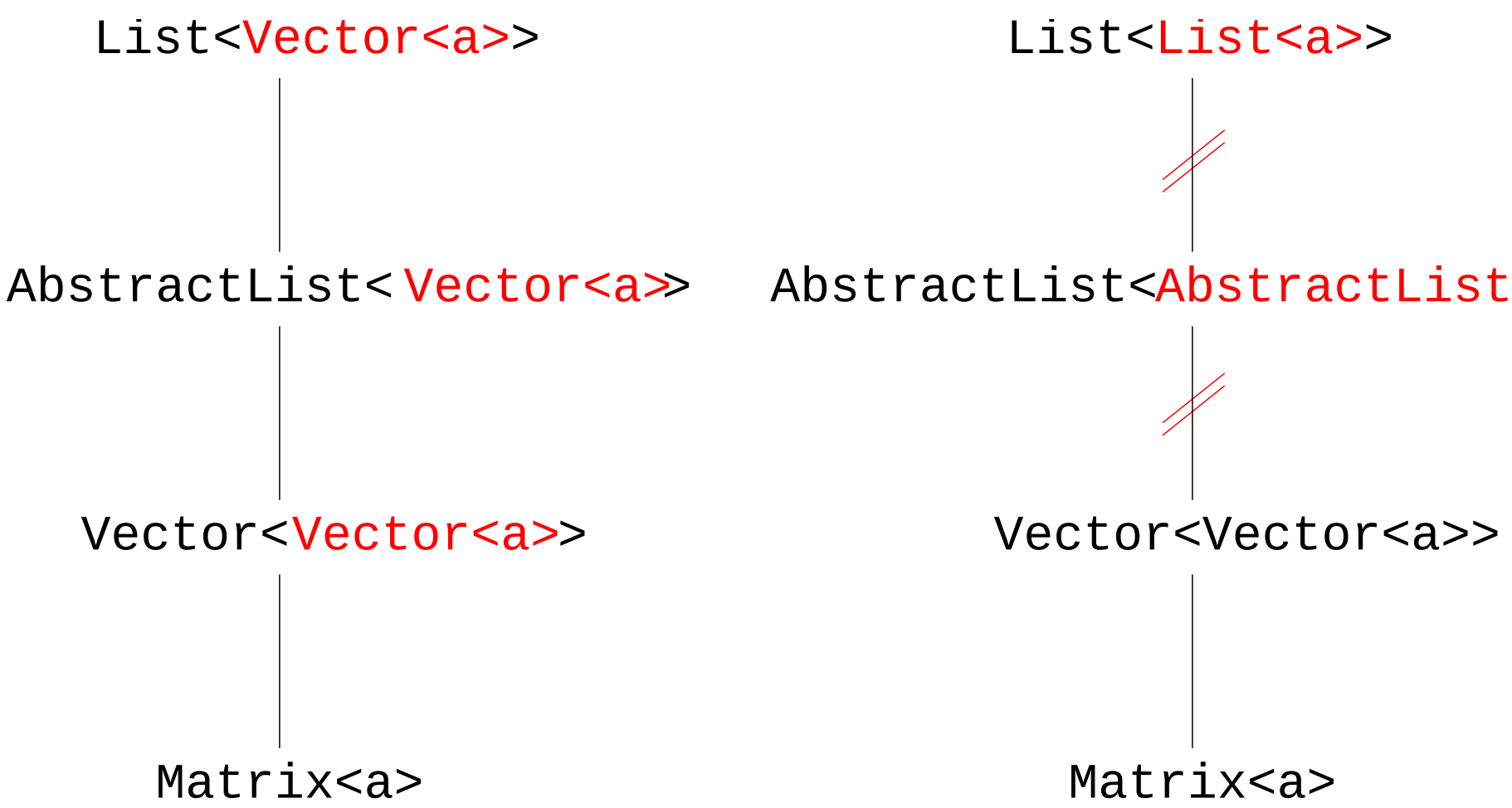
wenn für alle Typvariablen aus θ_2 *gilt:*

$$\sigma_1(a) = \sigma_2(a) \text{ (Contravariance Bedingung).}$$

Lemma 4 **Es gibt keine unendlichen Ketten in $\leq^*$.**

Beweis: Folgt direkt aus der Contravariance Bedingung. ■

Beispiel: Typtermordnung $\leq^*$



3. Typ–Unifikation

Typ–Unifikations Problem

Für zwei Typterme θ_1 und θ_2 ist eine Substitution σ gesucht, so dass gilt:

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

Lemma 5 **Das Typ–Unifikations Problem ist finitär.**

Beweis: Folgt aus:

- der Tatsache, dass es in $\leq^*$ keine unendlichen Ketten gibt bzw.
- der Contravariance Bedingung

■

Typ-Unifikations–Algorithmus[a]

(Grundlage [Martelli, Montanari 1982])

1. $Eq_1 = \{\, \theta \lessdot \theta' \mid (\theta \lessdot \theta') \in Eq \wedge ((\theta, \theta' \notin TV) \vee (\theta, \theta' \in TV)) \,\}$
2. $Eq_2 = Eq \backslash Eq_1$
3. $Eq_{set} = \{\, Eq_1 \,\} \times \Big(\bigotimes_{(a \lessdot \theta') \in Eq_2} \{\, a \doteq \sigma(\theta) \mid (\theta \leq^* \overline{\theta}') \in \mathbf{FC}(\leq), \sigma = \mathsf{Unify}(\theta', \overline{\theta}') \,\} \Big)$
   $\times \Big(\bigotimes_{(\theta \lessdot a) \in Eq_2} \{\, a \doteq \theta' \mid \theta \leq^* \theta' \,\} \Big)$
4. Wiederholte Anwendung der Regeln *reduce1*, *reduce2*, *erase*, *swap* und *adapt*.
5. Anwendung der Regel *subst*
6. (a) Für jede in letzten Durchlauf veränderte Menge von Paaren erneut starten
   (b) Alle Mengen von Paaren zusammenfassen

[a] [Plümicke 2004, Unif'04, Cork]

4. Typ–Inferenz

Abstrakte Syntax der Sprache **Core Java 5.0**

$$
\begin{array}{lcl}
Source & := & (class|interface)* \\
class & := & \mathsf{Class}(type, [\mathsf{extends}(type),]\ [\mathsf{implements}(type+),] \\
 & & \qquad IVarDecl*, Method*) \\
interface & := & \mathsf{interface}(type, [\mathsf{extends}(type),] MHeader*) \\
IVarDecl & := & \mathsf{InstVarDecl}(type, var) \\
MHeader & := & \mathsf{MethodHeader}(mname, type, (var : type)*) \\
Method & := & \mathsf{Method}(mname, type, (var : type)*, block) \\
block & := & \mathsf{Block}(stmt*) \\
stmt & := & block \mid \mathsf{Return}(expr) \mid \mathsf{while}(expr, block) \\
 & \mid & \mathsf{LocalVarDecl}(var, [type]) \mid \mathsf{If}(expr, block[, block]) \mid stmtexpr \\
stmtexpr & := & \mathsf{Assign}(var, expr) \\
 & \mid & \mathsf{New}(type, expr*) \\
 & \mid & \mathsf{NewArray}(type, expr) \\
 & \mid & \mathsf{MethodCall}([expr,] f(expr*)) \\
expr & := & stmtexpr \mid \mathsf{this} \mid \mathsf{super} \mid \mathsf{LocalOrFieldVar}(var) \\
 & \mid & \mathsf{InstVar}(expr, var) \mid \mathsf{ArrayAcc}(expr, expr)
\end{array}
$$

Typ–Inferenz Regeln

$$p \blacktriangleright O$$

$$\forall 1 \leqslant i \leqslant m :$$

$$(O \cup \{\, \tau \mapsto (\overline{O}_\tau \cup O_{field}) \,\} \cup$$

$$\{\, \texttt{<local>} \mapsto (O_{field} \cup \{\, v_{i,1} : \theta_{i,1}, \ldots, v_{i,k_i} : \theta_{i,k_i} \,\}) \,\}), \tau, \tau')$$

$$\rhd_{Stmt} \mathsf{Block}(B_i) : \theta_i$$

where

$$\overline{O}_\tau = \{\, f_1 : \theta_{1,1} \times \ldots \times \theta_{1,k_1} \to \theta'_1, \ldots, f_m : \theta_{m,1} \times \ldots \times \theta_{1,k_m} \to \theta'_m \,\}$$

$$O_{field} = \{\, x_1 : \tau_1, \ldots, x_n : \tau_n \,\}$$

$$\textbf{[Class]} \quad \frac{\text{for } 1 \leqslant i \leqslant m : \;\; \theta_i \leq^* \theta'_i}{\begin{array}{l}(p \cup \mathsf{Class}(\mathsf{ClassName}(\tau), \mathsf{extends}(\tau') \\ \qquad \mathsf{InstVarDecl}(x_1, \tau_1), \ldots, \mathsf{InstVarDecl}(x_n, \tau_n) \\ \qquad \mathsf{Method}(f_1, \theta_1, (v_{1,1} : \theta_{1,1}, \ldots, v_{1,m_1} : \theta_{1,k_1}), \mathsf{Block}(B_1)) \\ \qquad , \ldots , \\ \qquad \mathsf{Method}(f_m, \theta_m, (v_{m,1} : \theta_{m,1}, \ldots, v_{m,m_m} : \theta_{m,k_m}), \mathsf{Block}(B_m))) \\ \blacktriangleright O \cup \{\, \mathsf{gen}(\tau) \mapsto \overline{O}_\tau \cup O_{field} \,\}\end{array}}$$

Wenn Typen für Methoden herleitbar sind,
so sind sie generalisiert auch ableitbar.

IntSec Regel

$$[\textbf{IntSec}]\ \frac{p \blacktriangleright O \cup \{\, \tau \mapsto \{\, f_1 : ty_1, \ldots, f_k : ty_k, \ldots, f_m : ty_m \,\}\,\} \quad p \blacktriangleright O \cup \{\, \tau \mapsto \{\, f_1 : ty_1, \ldots, f_k : ty'_k, \ldots, f_m : ty_m \,\}\,\}}{p \blacktriangleright O \cup \{\, \tau \mapsto \{\, f_1 : ty_1, \ldots, f_k : ty_k \,\&\, ty'_k, \ldots, f_m : ty_m \,\}\,\}}$$

Wenn zwei unterschiedliche Typen für eine Methode herleitbar sind, so ist auch der Durchschnitt dieser Typen herleitbar

Ident Regel

$$[\textbf{Ident}]\ \frac{(O \cup \{\, id : \&_{i \in I} \theta_i \,\})_{\forall a_1 .... \forall a_n . \theta'}}{(O \cup \{\, id : \bigwedge_{i \in I} \theta_i \,\})_{\sigma|_{\{\, a_1, \ldots, a_n \,\}}(\theta)} \ \rhd_{Id}\ id : \sigma|_{\{\, a_1, \ldots, a_n \,\}}(\theta_j)} \quad \theta \leq^* \theta', j \in I$$

- **Für (geerbete) Methoden und Attribute einer Klasse sind ihre Typen ableitbar.**
- **Generaliserte Typvariablen sind dabei instanzierbar.**

Weitere Regeln

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

$$[\textbf{BlockInit}] \quad \frac{(O, \tau, \tau') \rhd_{Stmt} stmt : \theta}{(O, \tau, \tau') \rhd_{Stmt} \mathsf{Block}(stmt) : \theta}$$

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

Statement Regeln: Jeder Block von Statements erhält einen Typ. Der Typ repräsentiert den Rückgabetype der zugehörigen Methode.

Expression Regeln: Jede Expression bekommt einen Typ.

Theorem 1 (Principal Type) *Für jede Methode einer Klasse in* Java 5.0 *gibt es einen allgemeinsten Typ. D.h. für*

$$p \blacktriangleright O$$

und

$$(O \cup \{ id : ty \})_\theta \rhd_{Id} id : \tau$$

gibt es

- *einen allgemeinsten Typ* $\overline{ty}$ *von id mit* $\overline{ty} = \bigwedge_{i \in I} \overline{\theta}_i$ *(I endliche Indexmenge),*
- *ein* $j \in I$
- *eine Substitution* σ

mit

$$\sigma(\overline{\theta}_j) = \tau.$$

Typ–Inferenz–Algorithmus

Typannahmen: Für jeden zu berechnenden Typ wird zunächst eine Typvariable angenommen.

Baumlauf: Der Algorithmus verfeinert durch Typ–Unifikation die Typannahmen sukzessive bei einem Lauf über den abstrakten Syntaxbaum.

Vervielfachung der Annahmen: Enthält das Ergebnis der Unifikation mehrere Unifikatoren, so wird für jeden Unifikator eine neue Menge von Typannahmen generiert.

Annahmen löschen: Schlägt die Unifikation fehl, wird die zugehörige Menge von Typannahmen gelöscht.

Generaliserung: Alle nach dem Baumlauf enthaltenen Typvariablen können entweder als Klassen– oder als Methodenparameter generalisiert werden.

Durchschnittstypen: Sind am Ende mehrere Mengen von Typannahmen enthalten, so hat den Methode einen Durchschnittstyp.

Tool–Demonstation

Zusammenfassung Typ–Inferenz–Algorithmus

Typ–Inferenz–Algorithmus kann als generischer Algorithmus parametrisiert durch eine Typtermordnung bzw. durch einen finitären Typ–Unifikations Algorithmus angesehen werden.

Mögliche Instanzen

- Typsystem Java < 1.4 (triviale Unifikation)
- Generische Typen, ohne bounded Variablen und ohne Wildcards (implementiert)
- Generische Typen, mit bounded Variablen aber ohne Wildcards (Theorie erarbeitet)
- Typsystem Java 5.0 (to do)
- Typsystem Erweiterung *Simple loose owership domains (TU Kaiserslautern)* (to do)

Zusammenfassung und Ausblick

Zusammenfassung

- Festlegung einer *Typtermordnung* in Java 5.0.
- Typ–Unifikation auf Typtermen ohne Wildcards ist finitär.
- Typ–Inferenz–Algorithmus aufbauend auf der Typ–Unifikation.

Ausblick

- **Erweiterung auf Wildcards:**
  - Es können unendliche Ketten in der *Typtermordnung* entstehen.
  - Typ–Unifikation ist nicht mehr finitär.
  - Berechenbarkeit der Typ–Unifikation klären.
  - Sinnvolle Strategie für die Typ–Inferenz entwickeln, wenn Typ–Unifikation unendliche viele Unifikatoren als Ergebnis hat.
- **Byte–Code Anpassung an Durchschnittstypen**