

Kapitel 18

Palsberg-Schwartzbach Typinferenz

- Ziel: Optimierung des Methodenaufrufs (insbesondere dynamische Bindung)
- Technik: Berechnung der möglichen Typen (Klassen) jedes Ausdrucks
- Idee: berechne *Mengen* möglicher Typen
- falls Menge leer: Typfehler
- falls Menge einelementig: keine dynamische Bindung
- *Typinferenz*: Herleitung der (kleinsten !) Typmengen aus den *Variablenverwendungen*
- Typinferenz kann für typfreie Sprachen (Smalltalk) auch die notwendigen Laufzeittests bestimmen

Schreibweise:

$$\llbracket expr \rrbracket = \{\tau_1, \dots, \tau_n\}$$

Typconstraints:

$\{\tau_1, \dots, \tau_n\} \subseteq X$	(explizites Constraint)
$X \subseteq Y$	(Propagationsconstraint)
$\tau \in X \Rightarrow Y \subseteq Z$	(Bedingtes Constraint)
$X \subseteq \{\tau_1, \dots, \tau_n\}$	(Sicherheitsconstraint)
$X = Y$	(Mengengleichheit)

weitere Schreibweise:

$$\downarrow C = \{\tau \mid \tau \leq C\}$$

Dies nennt man ein *Ideal* in der Klassenhalbordnung

18.1 Elementare Regeln

Wir betrachten zunächst nur Kernsprache ohne Überladungen, Konversionen

- $k \in Const$:

$$\llbracket k \rrbracket = \{int\} = \downarrow int$$

$$\llbracket true \rrbracket = \{bool\}$$

ähnlich für andere Konstanten

- `null`, `this`, `new`:

$$\llbracket null \rrbracket = \emptyset = \downarrow void$$

$$\llbracket this_C \rrbracket = \llbracket new C \rrbracket = \{C\}$$

- $expr_1 + expr_2$:

$$\llbracket expr_1 \rrbracket = \llbracket expr_2 \rrbracket = \llbracket expr_1 + expr_2 \rrbracket = \{int\}$$

ähnlich für andere arithmetische/logische Operatoren

- Zuweisung $Id = expr$: Typ-Konformanz!

$$\llbracket expr \rrbracket \subseteq \llbracket Id \rrbracket$$

- `if (expr1) expr2 else expr3`:

$$\llbracket expr_1 \rrbracket = \{bool\}, \llbracket if... \rrbracket = \llbracket expr_2 \rrbracket \cup \llbracket expr_3 \rrbracket$$

ähnlich für andere Anweisungen

- Deklaration $C Id$:

$$\llbracket Id \rrbracket = \downarrow C$$

- Methodenaufruf $expr.Id(expr_1, \dots, expr_n)$:

$$\llbracket expr \rrbracket \subseteq \{C \mid C \text{ implementiert } Id\}$$

Sei ferner die Implementierung

type Id ($type_1 Id_1, \dots, type_n Id_n$) $\{\dots \text{return } expr_0 \dots\}$ in Klasse C vorhanden. Falls $C \in \llbracket expr \rrbracket$ muß gelten:

$$\llbracket expr_i \rrbracket \subseteq \llbracket Id_i \rrbracket, \quad i = 1..n$$

$$\llbracket expr.Id(expr_1, \dots, expr_n) \rrbracket \ni \downarrow type \ni \llbracket expr_0 \rrbracket$$

da es mehrere Implementierungen der Methode geben kann, handelt es sich tatsächlich um bedingte Constraints.
Vollständige Form:

$$C \in \llbracket expr \rrbracket \Rightarrow \\ \left(\llbracket expr_i \rrbracket \subseteq \llbracket Id_i \rrbracket \wedge \llbracket expr.Id(\dots) \rrbracket \ni \llbracket expr_0 \rrbracket \right)$$

Beispiel:

```
class C {
  int n(int i) {
    return i+1
  }
}
C x = new C();
int k = x.n(true);
```

$$\llbracket i \rrbracket = \llbracket 1 \rrbracket = \llbracket i + 1 \rrbracket = \{\text{int}\}, \llbracket \text{true} \rrbracket = \{\text{bool}\}$$

$$\llbracket x \rrbracket = \llbracket \text{new } C \rrbracket = \{C\}, \llbracket k \rrbracket = \{\text{int}\} \supseteq \llbracket x.n(\text{true}) \rrbracket$$

Für den Methodenaufruf muß gelten:

$$\llbracket x \rrbracket \subseteq \{C\}, \llbracket \text{true} \rrbracket \subseteq \llbracket i \rrbracket, \llbracket x.n(\text{true}) \rrbracket \supseteq \{\text{int}\} \supseteq \llbracket i + 1 \rrbracket$$

Daraus folgt

$$\{\text{bool}\} \subseteq \{\text{int}\}$$

was nicht gilt: TYPFEHLER

18.2 Typinferenz

- beliebige Typmengen statt Ideale
Typen in Deklarationen werden ignoriert
- $\llbracket e \rrbracket$ beschreibt *minimale Anforderungen* an die Typen von e , berechnet aus den *Verwendungen* von Variablen
 $\emptyset =$ „keine Anforderungen“
- Korrektheitskriterium: falls e zur Laufzeit Typ τ haben kann, so $\tau \in \llbracket e \rrbracket$
- \Rightarrow möglichst kleine Typmengen
 $\Rightarrow \supseteq$ statt $=$ in den Constraints!
- Aus den Constraints wird durch *Fixpunktiteration* der minimale Fixpunkt berechnet
- es gibt immer trivialen Fixpunkt $\llbracket x \rrbracket = \downarrow$ *Object* für alle Variablen/Ausdrücke x
- Fixpunktiteration startet für alle x mit der leeren Menge und vergrößert diese dann durch Anwenden von Constraints
- effiziente Implementierung für volle Sprache nichttrivial!

Beispiel 1:

```

class C {
  ? n(? i) {
    return i+1
  }
}
? x = new C();
? k = x.n(true);

```

$$\begin{aligned}
\llbracket i \rrbracket &\supseteq \{\text{int}\}, \llbracket 1 \rrbracket \supseteq \{\text{int}\}, \llbracket i + 1 \rrbracket \supseteq \{\text{int}\}, \\
\llbracket \text{true} \rrbracket &\supseteq \{\text{bool}\}, \llbracket x \rrbracket \supseteq \{C\}, \llbracket \text{new } C \rrbracket \supseteq \{C\}, \\
k &\supseteq \llbracket x.n(\text{true}) \rrbracket \supseteq \llbracket \text{return } i + 1 \rrbracket \supseteq \{\text{int}\}
\end{aligned}$$

Für den Methodenaufruf muß wie gehabt gelten:

$$\llbracket x \rrbracket \subseteq \{C\}, \llbracket \text{true} \rrbracket \subseteq \llbracket i \rrbracket, \llbracket x.n(\text{true}) \rrbracket \supseteq \llbracket i + 1 \rrbracket$$

demnach $\llbracket i \rrbracket \supseteq \{\text{int}\}$ und $\llbracket i \rrbracket \supseteq \{\text{bool}\}$

Die minimale Lösung ist:

$$\begin{aligned}
\llbracket i \rrbracket &= \{\text{int}, \text{bool}\} (!), \llbracket 1 \rrbracket = \{\text{int}\}, \\
\llbracket i + 1 \rrbracket &= \{\text{int}\}, \llbracket \text{true} \rrbracket = \{\text{bool}\} \\
\llbracket x \rrbracket &= \{C\}, \llbracket \text{new } C \rrbracket = \{C\}, \\
\llbracket k \rrbracket &= \{\text{int}\}, \llbracket x.n(\text{true}) \rrbracket = \{\text{int}\}
\end{aligned}$$

Mithin hat i keinen eindeutigen Typ; eine Deklaration für i läßt sich nicht angeben.

Beispiel 2:

```

class A {
  ? x; ? y;
  ? m(? f) {
    return f.m(x);
  }
}
class B {
  ? m(? g) {
    return this;
  }
}
? u = (new A).m(new B).m(null);

```

1. Constraints für $f.m(x)$:

Wg. Bedingung $\llbracket f \rrbracket \subseteq \{A, B\} = \{c \mid c \text{ hat } m\}$ je ein bedingtes Constraint für jede mögliche Bindung von m , nämlich $A :: m$ und $B :: m$

a) eventuell ist $m = A :: m$:

$$A \in \llbracket f \rrbracket \Rightarrow \llbracket x \rrbracket \subseteq \llbracket f \rrbracket \text{ und } \llbracket f.m(x) \rrbracket \supseteq \llbracket f.m(x) \rrbracket$$

b) oder es ist $m = B :: m$:

$$B \in \llbracket f \rrbracket \Rightarrow \llbracket x \rrbracket \subseteq \llbracket g \rrbracket \text{ und } \llbracket f.m(x) \rrbracket \supseteq \llbracket \text{this}_B \rrbracket$$

2. Constraints für `this`, `new A`, `new B`:

$$\llbracket \text{this}_B \rrbracket \supseteq \{B\}, \llbracket \text{new } A \rrbracket \supseteq \{A\}, \llbracket \text{new } B \rrbracket \supseteq \{B\}$$

3. Constraints für $(\text{new } A).m(\text{new } B)$: wiederum 2 Fälle

- a)** $A \in \llbracket \text{new } A \rrbracket \Rightarrow \llbracket \text{new } B \rrbracket \subseteq \llbracket f \rrbracket$ und
 $\llbracket (\text{new } A).m(\text{new } B) \rrbracket \supseteq \llbracket f.m(x) \rrbracket$
- b)** $B \in \llbracket \text{new } A \rrbracket \Rightarrow \llbracket \text{new } B \rrbracket \subseteq \llbracket g \rrbracket$ und
 $\llbracket (\text{new } A).m(\text{new } B) \rrbracket \supseteq \llbracket \text{this}_B \rrbracket$

4. Constraints für $(\text{new } A).m(\text{new } B).m(\text{null})$:

- a)** $A \in \llbracket (\text{new } A).m(\text{new } B) \rrbracket \Rightarrow \llbracket \text{null} \rrbracket \subseteq \llbracket f \rrbracket$ und
 $\llbracket (\text{new } A).m(\text{new } B).m(\text{null}) \rrbracket \supseteq \llbracket f.m(x) \rrbracket$
- b)** $B \in \llbracket (\text{new } A).m(\text{new } B) \rrbracket \Rightarrow \llbracket \text{null} \rrbracket \subseteq \llbracket g \rrbracket$ und
 $\llbracket (\text{new } A).m(\text{new } B).m(\text{null}) \rrbracket \supseteq \llbracket \text{this}_B \rrbracket$

5. Constraint für die Zuweisung:

$$\llbracket u \rrbracket \supseteq \llbracket (\text{new } A).m(\text{new } B).m(\text{null}) \rrbracket$$

Wir haben also

$$\begin{aligned} \{B\} &\subseteq \llbracket \text{new } B \rrbracket \subseteq \llbracket f \rrbracket \\ \{B\} &\subseteq \llbracket \text{this}_B \rrbracket \subseteq \llbracket f.m(x) \rrbracket \subseteq \llbracket (\text{new } A).m(\text{new } B) \rrbracket \\ \{B\} &\subseteq \llbracket (\text{new } A).m(\text{new } B).m(\text{null}) \rrbracket \end{aligned}$$

Andere Anforderungen gibt es nicht; insbesondere wird $A \in \llbracket f \rrbracket, \llbracket f.m(x) \rrbracket, \llbracket (\text{new } A).m(\text{new } B) \rrbracket, \llbracket (\text{new } A).m(\text{new } B).m(\text{null}) \rrbracket$ nicht erzwungen!

Die minimale Lösung ist deshalb:

$$\begin{aligned} \llbracket x \rrbracket &= \llbracket y \rrbracket = \llbracket g \rrbracket = \llbracket \text{null} \rrbracket = \emptyset; & \llbracket \text{new } A \rrbracket &= \{A\} \\ \llbracket f \rrbracket &= \llbracket f.m(x) \rrbracket = \llbracket \text{this}_B \rrbracket = \llbracket \text{new } B \rrbracket = \{B\} \\ & & \llbracket (\text{new } A).m(\text{new } B) \rrbracket &= \{B\} \\ & & &= \llbracket (\text{new } A).m(\text{new } B).m(\text{null}) \rrbracket \end{aligned}$$

Damit ist die Mehrdeutigkeit der m -Aufrufe aufgelöst:

$(\text{new } A).m(\text{new } B)$ ruft mit Sicherheit $A :: m$;

$(\text{new } A).m(\text{new } B).m(\text{null})$ ruft mit Sicherheit $B :: m$

Ferner ist offenbar geworden, daß x, y beliebigen Typ haben können (minimale Anforderungen = leer)

18.3 Lösen von Mengengleichungssystemen

P/S-Typinferenz führt (wie etliche andere Analysen) letztendlich auf Mengengleichungssystem:

$$M_1 \subseteq N_1$$

$$M_2 \subseteq N_2$$

$$\vdots$$

$$M_k \subseteq N_k$$

Dabei kann durchaus $M_i = N_j$ sein (zB $M_i = N_j = \llbracket \text{this}_B \rrbracket$)

Wie löst man das System effektiv?

1. Wähle effiziente Mengenimplementierung. In P/S: nummeriere (endliche Zahl von) Klassen durch \Rightarrow Bitvektoren!
2. Initialisiere alle unbekanntes $M_i, N_i = \emptyset$ (falls zB $M_i = \{x_1, \dots, x_k\}$ natürlich nicht!)
3. Stelle *Abhängigkeitsgraph* auf:
Knoten = M_i, N_i , Kante $M_i \rightarrow N_i$ für jede Ungleichung $M_i \subseteq N_i$
4. Berechne die M_i, N_i in *topologischer Reihenfolge*: Berechnen N_i erst, wenn *alle* Vorgänger im Graphen berechnet wurden (\rightarrow topologische Sortierung)
5. Für Knoten X mit Vorgängern Y^1, \dots, Y^j berechne $X := X \cup Y^1 \cup Y^2 \cup \dots \cup Y^j$. Spezialfall: Knoten ohne Vorgänger sind unveränderlich.

6. falls Graph zyklensfrei: Berechnung fertig nach $2k$ Schritten
7. falls bedingte Ungleichungen $\tau_i \in U_i \Rightarrow M_i \subseteq N_i$: sobald $\tau_i \in U_i$, „feuert“ $M_i \subseteq N_i$
8. falls Zyklen: propagiere nur aus Zyklus heraus, wenn Zyklus stabil geworden ist!
9. Zyklen erfordern globale Iteration, bis die M_i, N_i stabil sind \Rightarrow typische Gesamtkomplexität $O(k^3)$
10. Verbesserung: ersetze Zyklus $X^1 \subseteq X^2 \subseteq \dots \subseteq X^j \subseteq X_1$ durch einen Knoten $X = X^1 \cup X^2 \cup \dots \cup X^j$, sobald Zyklus entdeckt wird. vermerke $X^i = X$ für alle i
11. weitere Optimierungen sind Gegenstand neuerer Forschung