Korrekte Software: Grundlagen und Methoden
Vorlesung 9 vom 06.06.19
Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2019

Fahrplan

Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;
//{X = x ∧ Y = z}
y = x;
// {X = y ∧ Y = z}
x = z;
//{X = y ∧ Y = x}
```

- Wir haben gesehen:
  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Verifikation kann **berechnet** werden.
- Muss das rückwärts sein? Warum nicht vorwärts? Was ist der Vorteil?

Nachteile der Rückwärtsberechnung

```
// {i ≠ 3}
.
. // 400 Zeilen, die
. // i nicht verändern
.
a[i]= 5;
// {a[3] = 7}
```

Errechnete Vorbedingung (AWP):

$$(a[3] = 7)[5/a[i]]$$

- Kann nicht vereinfacht werden, weil wir nicht wissen, ob $i \neq 3$
- AWP wird **sehr groß**.
- Das Problem wächst mit der Länge der Programme.

Der Floyd-Hoare-Kalkül Vorwärts

Vorwärtsanwendung der Regeln

- Zuweisungsregel kann nicht vorwärts angewandt werden, weil die Vorbedingung keine offene Variable ist:

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Die anderen Regeln passen:

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Zuweisungregel Vorwärts

- Alternative Zuweisungsregel (nach Floyd):

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = e[V/x] \wedge P[V/x]\}}$$

  - $FV(P)$ sind die **freien** Variablen in P.
- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden
- Gilt auch für die anderen Regeln.

Vorwärtsverkettung

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = (e[V/x]) \wedge P[V/x]\}}$$

```
// {0 ≤ x}
x= 2*y;
// {∃V1. 0 ≤ V1 ∧ x = 2 · y}
x= x+1;
// {∃V2. (∃V1. 0 ≤ V1 ∧ x = 2 · y)[V2/x] ∧ x = (x + 1)[V2/x]}
```

- **Vereinfachung** der letzten Nachbedingung:

$$\begin{aligned}
&\exists V_2.\, (\exists V_1. 0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x] \\
\Longleftrightarrow\ &\exists V_2.\, (\exists V_1. 0 \leq V_1 \wedge V_2 = 2 \cdot y) \wedge x = V_2 + 1 \\
\Longleftrightarrow\ &\exists V_2.\, \exists V_1. 0 \leq V_1 \wedge x = V_2 + 1 \wedge V_2 = 2 \cdot y \\
\Longleftrightarrow\ &\exists V_1. 0 \leq V_1 \wedge x = 2 \cdot y + 1
\end{aligned}$$

Regeln der Vorwärtsverkettung

1. Wenn x nicht in Vorbedingung auftritt, dann $P[V/x] \equiv P$.
2. Wenn x nicht in rechter Seite e auftritt, dann $e[V/x] \equiv e$.
3. Wenn beides der Fall ist, kann der Existenzquantor wegfallen:
$$V \notin FV(P) \Longrightarrow \exists V.\, P \equiv P$$
4. Wenn x vorher zugewiesen wurde, Vereinfachung mit
$$\exists V.\, P[V] \wedge V = t \Longrightarrow P[t/V]$$

Vorwärtsverkettung

- Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- Vorteil: Vorbedingung bleibt kleiner
- Nachteil: in der Anwendung **umständlicher**
- Vereinfachung benötigt Lemma: $\exists x.P(x) \wedge x = t \Longleftrightarrow P(t)$

Zwischenfazit: Der Floyd-Hoare-Kalkül ist symmetrisch

Es gibt zwei Zuweisungsregeln, eine für die **Rückwärtsanwendung** von Regeln, eine für die **Vorwärtsanwendung**

Vorwärtsberechnung von Verifikationsbedingungen

Stärkste Nachbedingung

- Vorwärtsberechnung von Verifikationsbedingungen: Nachbedingung
- Gegeben C0-Programm c, Prädikat P, dann ist
  - $\mathrm{sp}(P, c)$ die **stärkste Nachbedingung** Q so dass $\models \{P\}\, c\, \{Q\}$
  - Prädikat Q **stärker** als Q' wenn $Q \Longrightarrow Q'$.
- Semantische Charakterisierung:

Stärkste Nachbedingung

Gegeben Zusicherung $P \in$ **Assn** und Programm $c \in$ **Stmt**, dann

$$\models \{P\}\, c\, \{Q\} \Longleftrightarrow \mathrm{sp}(P, c) \Longrightarrow Q$$

- Wie können wir $\mathrm{sp}(P, c)$ berechnen?

Berechnung von Nachbedingungen

- Wir berechnen die **approximative** stärkste Nachbedingung.
- Viele Klauseln sind ähnlich der schwächsten Vorbedingung.
- Ausnahmen:
  - While-Schleife: andere Verifikationsbedingungen
  - If-Anweisung: Weakening eingebaut
  - **Zuweisung**: Vorwärtsregel
- Nach jeder Zuweisung Nachbedingung **vereinfachen**

Überblick: Approximative stärkste Nachbedingung

$$\begin{aligned}
\mathrm{asp}(P, \{\,\}) &\stackrel{def}{=} P \\
\mathrm{asp}(P, x = e) &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathrm{asp}(P, c_1; c_2) &\stackrel{def}{=} \mathrm{asp}(\mathrm{asp}(P, c_1), c_2) \\
\mathrm{asp}(P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} \mathrm{asp}(b \wedge P, c_0) \vee \mathrm{asp}(\neg b \wedge P, c_1) \\
\mathrm{asp}(P, //{**}\ \{q\}\ {*}/) &\stackrel{def}{=} q \\
\mathrm{asp}(P, \textbf{while}\ (b)\ //{**}\ \textbf{inv}\ i\ {*}/\ c) &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathrm{svc}(P, \{\,\}) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, x = e) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, c_1; c_2) &\stackrel{def}{=} \mathrm{svc}(P, c_1) \cup \mathrm{svc}(\mathrm{asp}(P, c_1), c_2) \\
\mathrm{svc}(P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} \mathrm{svc}(P \wedge b, c_0) \cup \mathrm{svc}(P \wedge \neg b, c_1) \\
\mathrm{svc}(P, //{**}\ \{q\}\ {*}/) &\stackrel{def}{=} \{P \longrightarrow q\} \\
\mathrm{svc}(P, \textbf{while}\ (b)\ //{**}\ \textbf{inv}\ i\ {*}/\ c) &\stackrel{def}{=} \mathrm{svc}(i \wedge b, c) \cup \{P \longrightarrow i\} \\
&\qquad \cup \{\mathrm{asp}(i \wedge b, c) \longrightarrow i\} \\
\mathit{svc}(\{P\}\, c\, \{Q\}) &\stackrel{def}{=} \{\mathrm{asp}(P, c) \longrightarrow Q\} \cup \mathrm{svc}(P, c)
\end{aligned}$$

Beispiel: Fakultät

```
1  // {0 ≤ n}
2  p= 1;
3  c= 1;
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */
5    p = p * c;
6    c = c + 1;
7    }
8  // {p = n!}
```


Beispiel: Fakultät, stärkste Nachbedingung

Notation: asp_x = Stärkste Nachbedingung **nach** Zeile x.

```
1  // {0 ≤ n}
2  p= 1;
   // asp_2 = {∃V. 0 ≤ n[V/p] ∧ p = (1[V/p])}
   // ⇝ asp_2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp_3 = {∃V. (0 ≤ n ∧ p = 1)[V/c] ∧ c = (1[V/c])}
   // ⇝ asp_3 = {0 ≤ n ∧ p = 1 ∧ c = 1}
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */
5    p = p * c;
     //
6    c = c + 1;
     //
7    }
   // asp_4 = {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1  // {0 ≤ n}
2  p= 1;
   // svc2 = ∅
   c= 1;
   // svc3 = ∅
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */
5    p = p * c;
     //
7  SVCat5 = ∅
6    c = c + 1;
     // svc6 = ∅
7    }
     // svc4 = {asp3 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n), asp5 ⟹ (p = (c − 1)! ∧
   c − 1 ≤ n)}
8  // {p = n!}
```


Beispiel: Fakultät, stärkste Nachbedingung

Notation: asp_x = Stärkste Nachbedingung **nach** Zeile x.

```
1  // {0 ≤ n}
2  p= 1;
   // asp2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp3 = {0 ≤ n ∧ p = 1 ∧ c = 1}
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */
5    p = p * c;
     // asp5 = {∃V1. (p = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n)[V1/p] ∧ p = (p · c)[V1/p]}
     // ⇝ asp5 = {∃V1. (V1 = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n) ∧ p = (V1 · c)}
     // ⇝ asp5 = {c − 1 ≤ n ∧ c ≤ n ∧ p = (c − 1)! · c}
6    c = c + 1;
     // asp6 = {∃V2. (c − 1 ≤ n ∧ c ≤ n ∧ p = (c − 1)! · c)[V2/c] ∧ c = (c + 1)[V2/c]}
     // ⇝ asp6 = {∃V2. (V2 − 1 ≤ n ∧ V2 ≤ n ∧ p = (V2 − 1)! · V2) ∧ c = (V2 + 1)}
     // ⇝ asp6 = {c − 2 ≤ n ∧ c − 1 ≤ n ∧ p = (c − 2)! · (c − 1)}
7    }
   // asp4 = {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```


Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1  // {0 ≤ n}
2  p= 1;
   // svc2 = ∅
   c= 1;
   // svc3 = ∅
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */
5    p = p * c;
     // svc5 = ∅
6    c = c + 1;
     // svc6 = ∅
7    }
   // svc4 = {asp3 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n),
   //                asp5 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n)}
   // ⇝ svc4 = {(0 ≤ n ∧ p = 1 ∧ c = 1) ⟹ (p = (c − 1)! ∧ c − 1 ≤ n),
   //                (c − 2 ≤ n ∧ c − 1 ≤ n ∧ p = (c − 2)! · (c − 1))
   //                      ⟹ (p = (c − 1)! ∧ c − 1 ≤ n)}
8  // {p = n!}
```


Schließlich zu zeigen

$$
\begin{aligned}
svc_8 &= \{\{asp_8 \Longrightarrow p = n!\} \cup svc_4 \\
&= \{(p = (c-1)! \wedge c - 1 \leq n \;\&\&\; \neg(c <= n)) \Longrightarrow p = n!\}, \\
&\quad (0 \leq n \wedge p = 1 \wedge c = 1) \Longrightarrow (p = (c-1)! \wedge c - 1 \leq n), \\
&\quad (c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1)) \\
&\qquad\qquad \Longrightarrow (p = (c-1)! \wedge c - 1 \leq n)\} \\
&\rightsquigarrow \{true\}
\end{aligned}
$$

Beispiel: Suche nach dem Maximalen Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i!= n)//** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n} */
5    if (a[r] < a[i]) {
6      r= i;
7      }
8    else {
9      }
10   i= i+1;
11   }
12 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

- Problem: wir müssen u.a. zeigen

$$
\begin{aligned}
&(\exists V_1.\, (\forall j.\, 0 \leq j < i - 1 \longrightarrow a[j] \leq a[V_1]) \wedge \\
&\quad i - 1 \neq n \wedge a[V_1] < a[i-1] \wedge r = i - 1) \longrightarrow 0 \leq r < n
\end{aligned}
$$

Deshalb: Invariante **verstärken**!

Beispiel: Suche nach dem Maximalen Element

Verstärkte Invariante (und Schleifenbedingung):

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i< n) //** inv  (∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r])
                          ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n  */ {
5    if (a[r] < a[i]) {
6      r= i;
7      }
8    else {
9      }
10   i= i+1;
11   }
12 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$
\begin{aligned}
&(\exists V_1.\, (\forall j.\, 0 \leq j < i - 1 \longrightarrow a[j] \leq a[V_1]) \wedge \\
&\quad 0 \leq i - 1 < n \wedge a[V_1] < a[i-1] \wedge r = i - 1) \longrightarrow 0 \leq r < n
\end{aligned}
$$

Läuft!

Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls sind **symmetrisch**: die Zuweisungsregel gibt es"rückwärts" und "vorwärts".
- Dual zu Beweis und Verifikationsbedingung rückwärts gibt es Regel und Verifkationsbedingungen vorwärts.
- Bis auf die Invarianten an Schleifen können wir Korrektheit automatisch prüfen.
- Kern der Vorwärtsberechnung ist die Zuweisungsregel nach Floyd.
- Vorwärtsberechnung erzeugt kleinere Terme, ist aber umständlicher zu handhaben.
- Rückwärtsberechnung ist einfacher zu handhaben, erzeugt aber (tendenziell sehr) große Terme.