

Korrekte Software: Grundlagen und Methoden  
Vorlesung 10 vom 12.06.18: Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

00:46:32 2018-06-19

1 [20]



## Fahrplan

- ▶ Einführung
- ▶ Operationale Semantik
- ▶ Denotationale Semantik
- ▶ Äquivalenz der Operationalen und Denotationalen Semantik
- ▶ Die Floyd-Hoare-Logik
- ▶ Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- ▶ Strukturierte Datentypen
- ▶ Modellierung und Spezifikation
- ▶ Verifikationsbedingungen
- ▶ **Vorwärts mit Floyd und Hoare**
- ▶ Funktionen und Prozeduren
- ▶ Referenzen
- ▶ Ausblick und Rückblick

Korrekte Software

2 [20]



## 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?

Korrekte Software

3 [20]



## 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.

Korrekte Software

4 [20]



# Der Floyd-Hoare-Kalkül Vorwärts

Korrekte Software

5 [20]



## 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:

$$\frac{}{\vdash \{A\} \{ \{A\} \}}$$

$$\frac{\vdash \{A \wedge b\} c_0 \{B\} \quad \vdash \{A \wedge \neg b\} c_1 \{B\}}{\vdash \{A\} \text{if } (b) \text{ c}_0 \text{ else } c_1 \{B\}}$$

$$\frac{\vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\}}{\vdash \{A\} c_1; c_2 \{C\}}$$

$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \text{while } (b) \text{ c } \{A \wedge \neg b\}}$$

$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$

Korrekte Software

6 [20]



## Zuweisungsregel 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.

Korrekte Software

7 [20]



## 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:

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

Korrekte Software

8 [20]



## 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:  

$$V \notin FV(P) \implies \exists V. P \equiv P$$
- 4 Wenn  $x$  vorher zugewiesen wurde, Vereinfachung mit  

$$\exists V. P[V] \wedge V = t \implies P[t/V]$$



## Vorwärtsverkettung

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

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
  - ▶  $sp(P, c)$  die **stärkste Nachbedingung**  $Q$  so dass  $\models \{P\} c \{Q\}$
  - ▶ Prädikat  $Q$  **stärker** als  $Q'$  wenn  $Q \implies Q'$ .
- ▶ Semantische Charakterisierung:

**Stärkste Nachbedingung**

Gegeben Zusicherung  $P \in \mathbf{Assn}$  und Programm  $c \in \mathbf{Stmt}$ , dann

$$\models \{P\} c \{Q\} \iff sp(P, c) \implies Q$$

- ▶ Wie können wir  $sp(P, c)$  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

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



## 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!}
    
```



## Fakultät: Stärkste Vorbedingung

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

$$\begin{aligned} asp_2 &= \exists V. 0 \leq n[V/p] \wedge p = (1[V/p]) \\ &\rightsquigarrow 0 \leq n \wedge p = 1 \\ asp_3 &= \exists V. (0 \leq n \wedge p = 1)[V/c] \wedge c = (1[V/c]) \\ &\rightsquigarrow 0 \leq n \wedge p = 1 \wedge c = 1 \\ asp_4 &= \neg(c \leq n) \wedge p = (c-1)! \wedge c-1 \leq n \\ asp_5 &= \exists V_1. (p = (c-1)! \wedge (c-1) \leq n \wedge c \leq n)[V_1/p] \\ &\quad \wedge p = (p \cdot c)[V_1/p] \\ &\rightsquigarrow \exists V_1. (V_1 = (c-1)! \wedge (c-1) \leq n \wedge c \leq n) \wedge p = (V_1 \cdot c) \\ &\rightsquigarrow c-1 \leq n \wedge c \leq n \wedge p = (c-1)! \cdot c \\ asp_6 &= \exists V_2. (c-1 \leq n \wedge c \leq n \wedge p = (c-1)! \cdot c)[V_2/c] \\ &\quad \wedge c = (c+1)[V_2/c] \\ &\rightsquigarrow \exists V_2. (V_2-1 \leq n \wedge V_2 \leq n \wedge p = (V_2-1)! \cdot V_2) \wedge c = (V_2+1) \\ &\rightsquigarrow c-2 \leq n \wedge c-1 \leq n \wedge p = (c-2)! \cdot (c-1) \end{aligned}$$



## Fakultät: Verifikationsbedingungen

Notation:  $vc_x$  = in Zeile  $x$  generierte Verifikationsbedingung

$$vc_4 = \{asp_3 \rightarrow p = (c-1)! \wedge c-1 \leq n, \\ asp_6 \rightarrow p = (c-1)! \wedge c-1 \leq n\}$$
$$vc_8 = asp_4 \rightarrow p = n!$$

Vereinfachung:  $vc_4 \iff vc_{4.1} \wedge vc_{4.2} \wedge vc_{4.3} \wedge vc_{4.4}$

$$vc_{4.1} = 0 \leq n \wedge p = 1 \wedge c = 1 \rightarrow p = (c-1)! \\ = 0 \leq n \rightarrow 1 = (1-1)!$$
$$vc_{4.2} = 0 \leq n \wedge p = 1 \wedge c = 1 \rightarrow c-1 \leq n \\ = 0 \leq n \rightarrow 0 \leq n$$
$$vc_{4.3} = c-2 \leq n \wedge c-1 \leq n \wedge p = (c-2)! \cdot (c-1) \rightarrow p = (c-1)! \\ = c-1 \leq n \wedge p = (c-2)! \cdot (c-1) \rightarrow p = (c-1)!$$
$$vc_{4.4} = c-2 \leq n \wedge c-1 \leq n \wedge p = (c-2)! \cdot (c-1) \rightarrow c-1 \leq n \\ = c-1 \leq n \rightarrow c-1 \leq n$$
$$vc_8 = n \leq c-1 \wedge c-1 \leq n \wedge p = (c-1)! \rightarrow p = n!$$

Korrekte Software

17 [20]



## 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

$$(\exists V_1. (\forall j. 0 \leq j < i-1 \rightarrow a[j] \leq a[V_1]) \wedge \\ i-1 \neq n \wedge a[V_1] < a[i-1] \wedge r = i-1) \rightarrow 0 \leq r < n$$

Deshalb: Invariante **verstärken!**

Korrekte Software

18 [20]



## 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 {
6     if (a[r] < a[i]) {
7         r = i;
8     }
9     else {
10    }
11    i = i + 1;
12 }
13 // {(∀j. 0 ≤ j < n → a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$(\exists V_1. (\forall j. 0 \leq j < i-1 \rightarrow a[j] \leq a[V_1]) \wedge \\ 0 \leq i-1 < n \wedge a[V_1] < a[i-1] \wedge r = i-1) \rightarrow 0 \leq r < n$$

**Läuft!**

Korrekte Software

19 [20]



## 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 Verifikationsbedingungen vorwärts. 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.

Korrekte Software

20 [20]

