

Korrekte Software: Grundlagen und Methoden
Vorlesung 9 vom 16.06.20
Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020



Feedback Online-Lehre

- ▶ Was kann besser werden?
 - ▶ Aufgezeichnete Vorlesungen?
 - ▶ Lesematerial/"Flipped Classroom"?
 - ▶ Andere Formen der Gruppenarbeit?
- ▶ Was ist gut/schlecht an Zoom?
 - ▶ Technische Probleme?
 - ▶ Funktionalität?
 - ▶ Break-Out Rooms?
- ▶ Was wollen wir ändern?



Fahrplan

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



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



Regelanwendung rückwärts

- ▶ Um Regel **rückwärts** anwenden zu können:

- 1 **Nachbedingung** der Konklusion muss offene Variable sein
- 2 Alle **Vorbedingungen** der Prämissen müssen disjunkte, offene Variablen sein
- 3 Gegenbeispiele: while-Regel, if-Regel

- ▶ Um Regeln **vorwärts** anwenden zu können:

- 1 **Vorbedingung** der Konklusion muss offene Variable sein
- 2 Alle **Nachbedingungen** der Prämissen müssen disjunkte, offene Variablen sein.
- 3 Gegenbeispiele: ...



Vorwärtsanwendung der Regeln

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

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

- ▶ Andere Regeln passen bis auf if-Regel (keine **disjunkten** Variablen)

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

$$\frac{\vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\}}{\vdash \{A\} c_1; c_2 \{C\}} \quad \frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \text{while } (b) c \{A \wedge \neg b\}}$$

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



If-Regel Vorwärts

- ▶ Abgeleitete If-Regel:

$$\frac{\vdash \{A \wedge b\} c_0 \{B_1\} \quad \vdash \{A \wedge \neg b\} c_1 \{B_2\}}{\vdash \{A\} \text{if}(b) c_0 \text{else } c_1 \{B_1 \vee B_2\}}$$

- ▶ Durch Verkettung der If-Regel mit Weakening: $B_1 \implies B_1 \vee B_2$



Zuweisungsregel Vorwärts

- ▶ Alternative Zuweisungsregel (nach Floyd):

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

- ▶ $FV(P)$ sind die **freien** Variablen in P .
- ▶ Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden
- ▶ Ist keine abgeleitete Regel — muss als korrekt **bewiesen** werden



Arbeitsblatt 9.1: Das Leben mit Quantor

- ▶ Was bedeutet $\exists V.P?$
 - ▶ Die Formel ist wahr, wenn es **irgendeinen** Wert t für V gibt, so dass $P[t/V]$ wahr ist.
- ▶ Was bedeutet $\forall V.P?$
 - ▶ Die Formel ist wahr, wenn für **alle** Werte t für V $P[t/V]$ wahr ist.
- ▶ Sind folgende Formeln wahr (für $x, y \in \mathbb{Z}$)? (Finde Gegenbeispiele oder Zeugen)

$$\begin{array}{lll} \exists x. x < 7 & \exists x. x < 3 \wedge x > 7 & \exists x. x < 7 \vee x < 3 \\ \exists y \exists x. x + 3 = y & \forall x \exists y. x * y = 3 & \exists x \forall y. x * y > 1 \end{array}$$



Vorwärtsverkettung

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

```
// {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:

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



Regeln der Vorwärtsverkettung

Eigenschaften des Existenzquantors:

$$P[V] \wedge V = t \implies P[t/V] \wedge V = t \quad (1)$$

$$\exists V. P[V] \wedge V = t \implies P[t/V] \quad (2)$$

$$\text{wenn } V \notin FV(Q) \text{ dann } (\exists V. P) \wedge Q \iff \exists V. P \wedge Q \quad (3)$$

$$\text{wenn } V \notin FV(P) \text{ dann } \exists V. P \implies P \quad (4)$$

Damit gelten folgende Regeln bei 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 (4)



Beispiel Vorwärtsverkettung

```
// {a < b}
a = b + a ;
// {∃ a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃ a1. a1 < b ∧ a = b + a1}
b = 3 * a + b ;
// {∃ b1. (∃ a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃ b1 ∃ a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a = b - 2 * a ;
// {∃ a2. (∃ b1 ∃ a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b - 2a)[a2/a]}
// {∃ a2 ∃ b1 ∃ a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b - 2a2}
// {∃ a2 ∃ b1 ∃ a1. a1 < b1 ∧ b = 3a2 + b1 ∧ a = b - 2a2 ∧ a2 = b1 + a1}
// {∃ b1 ∃ a1. a1 < b1 ∧ b = 3(b1 + a1) + b1 ∧ a = b - 2(b1 + a1)}
// {∃ b1 ∃ a1. a1 < b1 ∧ b = 3b1 + 3a1 + b1 ∧ a = b - 2b1 - 2a1}
// {∃ b1 ∃ a1. a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = b - 2b1 - 2a1}
// {∃ b1 ∃ a1. a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = (4b1 + 3a1) - 2b1 - 2a1}
// {∃ b1 ∃ a1. a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = 2b1 + a1}
```



Arbeitsblatt 9.2: Vorwärtsverkettung

Gegeben folgendes Programm. Berechnet die Vorwärtsverkettung der Vorbedingung

```
// {x = X ∧ y = Y}
x = x + y ;
// {???}
y = x - y ;
// {???}
x = x - y ;
// {???}
```

Was bewirkt das Programm?



Vorwärtsverkettung

- ▶ Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- ▶ Vorteil: Vorbedingung bleibt kleiner
- ▶ Nachteil: in der Anwendung **umständlicher**
- ▶ Vereinfachung benötigt Rechnung mit Existenzquantor

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 $P \in \text{Assn}$ und Programm $c \in \text{Stmt}$, 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

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



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!}
    
```



Beispiel: Fakultät, stärkste Nachbedingung

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

```

1 // {0 ≤ n}
2 p = 1;
  // asp2 = {∃V. 0 ≤ n[V/p] ∧ p = (1[V/p])}
  // asp2 = {0 ≤ n ∧ p = 1}
3 c = 1;
  // asp3 = {∃V. (0 ≤ n ∧ p = 1)[V/c] ∧ c = (1[V/c])}
  // asp3 = {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 }
  // 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 = ∅
3 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), asp6 ⇒ (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!}
    
```



Beispiel: Fakultät, Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```

1 // {0 ≤ n}
2 p = 1;
  // svc2 = 0
  c = 1;
  // svc3 = 0
4 while (c ≤ n) /** inv {p = (c-1)! ∧ c-1 ≤ n}; */ {
5   p = p * c;
  // svc5 = 0
6   c = c + 1;
  // svc6 = 0
7 }
  // svc4 = {asp3 ⇒ (p = (c-1)! ∧ c-1 ≤ n),
  //         asp6 ⇒ (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!}

```

Korrekte Software

25 [30]



Schließlich zu zeigen

$$\begin{aligned}
 \text{svc}_8 &= \{\{ \text{asp}_8 \Rightarrow p = n! \} \cup \text{svc}_4 \\
 &= \{(p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n)) \Rightarrow p = n!\}, \\
 &\quad (0 \leq n \wedge p = 1 \wedge c = 1) \Rightarrow (p = (c-1)! \wedge c-1 \leq n), \\
 &\quad (c-2 \leq n \wedge c-1 \leq n \wedge p = (c-2)! \cdot (c-1)) \\
 &\quad \Rightarrow (p = (c-1)! \wedge c-1 \leq n)\} \\
 &\rightsquigarrow \{\text{true}\}
 \end{aligned}$$

Korrekte Software

26 [30]



Arbeitsblatt 9.3: Jetzt seid ihr dran!

Berechnet die stärkste Nachbedingung und Verifikationsbedingungen für die ganzzahlige Division:

```

1 /** {0 ≤ a} */
2 r = a;
3 q = 0;
4 while (b ≤ r) /** inv { a = b*q+r ∧ 0 ≤ r } */ {
5   r = r-b;
6   q = q+1;
7 }
8 /** { a = b*q+r ∧ 0 ≤ r ∧ r < b } */

```

Korrekte Software

27 [30]



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

28 [30]



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}

```

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

Läuft!

Korrekte Software

29 [30]



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.
- 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.

Korrekte Software

30 [30]

