

Korrekte Software: Grundlagen und Methoden

Vorlesung 10 vom 21.06.22

Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2022

17:53:38 2022-07-04

1 [35]



Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül I
- Der Floyd-Hoare-Kalkül II: Invarianten
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- **Vorwärts mit Floyd und Hoare**
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen und Speichermodelle
- Ausblick und Rückblick

Korrekte Software

2 [35]



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 [35]



Nachteile der Rückwärtsberechnung

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

Errechnete **Vorbedingung** (AWP)

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

- 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 [35]



I. Der Floyd-Hoare-Kalkül Vorwärts

Korrekte Software

5 [35]



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: ...

Korrekte Software

6 [35]



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)

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

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

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

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

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

Korrekte Software

7 [35]



Arbeitsblatt 10.1: If-Regel Vorwärts

- Wie kann die If-Regel vorwärts aussehen?

Korrekte Software

8 [35]



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** logischen Variablen in P .
- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden
- Ist keine abgeleitete Regel — muss als korrekt **bewiesen** werden

Arbeitsblatt 10.2: Das Leben mit dem 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)

$$\exists x. x < 7$$

$$\exists x. x < 3 \wedge x > 7$$

$$\exists x. x < 7 \vee x < 3$$

$$\exists y \exists x. x + 3 = y$$

$$\forall x \exists y. x \cdot y = 3$$

$$\exists x \forall y. x \cdot y = y$$

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:

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

Regeln der Vorwärtsverkettung

Eigenschaften des Existenzquantors:

$$P(V) \wedge V = t \longrightarrow P[t/V] \wedge V = t \quad V \notin FV(t) \quad (1)$$

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

$$(\exists V. P) \wedge Q \iff \exists V. P \wedge Q \quad V \notin FV(Q) \quad (3)$$

$$\exists V. P \longrightarrow P \quad V \notin FV(P) \quad (4)$$

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 weggelassen (4)

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:

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

Vorwärtsverkettung bei der Arbeit

Vereinfachung erst am Ende:

```
// {i · i ≤ a ∧ t = 2 · i + 1 ∧ s = i · i + t ∧ s ≤ a}
t = t + 2;
// {∃ T. (i · i ≤ a ∧ t = 2 · i + 1 ∧ s = i · i + t ∧ s ≤ a) [T/t] ∧ t = (t + 2) [T/t]}
// {∃ T. i · i ≤ a ∧ T = 2 · i + 1 ∧ s = i · i + T ∧ s ≤ a ∧ t = T + 2}
s = s + t;
// {∃ S. (∃ T. i · i ≤ a ∧ T = 2 · i + 1 ∧ s = i · i + T ∧ s ≤ a ∧ t = T + 2) [S/s] ∧ s = (s + t) [S/s]}
// {∃ S. ∃ T. i · i ≤ a ∧ T = 2 · i + 1 ∧ S = i · i + T ∧ s ≤ a ∧ t = T + 2 ∧ s = S + t}
i = i + 1;
// {∃ I. (∃ S. ∃ T. i · i ≤ a ∧ T = 2 · i + 1 ∧ S = i · i + T ∧ s ≤ a ∧ t = T + 2 ∧ s = S + t) [I/i] ∧ i = (i + 1) [I/i]}
// {∃ I. ∃ S. ∃ T. I · I ≤ a ∧ T = 2 · I + 1 ∧ S = I · I + T ∧ s ≤ a ∧ t = T + 2 ∧ s = S + t ∧ i = I + 1}
// {∃ I. ∃ S. ∃ T. I · I ≤ a ∧ S = I · I + T ∧ s ≤ a ∧ t = T + 2 ∧ s = S + t ∧ i = I + 1}
// {∃ I. ∃ S. I · I ≤ a ∧ S = I · I + 2 · I + 1 ∧ s ≤ a ∧ t = 2 · I + 1 + 2 ∧ s = S + t ∧ i = I + 1}
// {∃ I. ∃ S. I · I ≤ a ∧ S ≤ a ∧ t = 2 · I + 1 + 2 ∧ s = S + t ∧ i = I + 1 ∧ s = (I + 1) · (I + 1)}
// {∃ I. I · I ≤ a ∧ (I + 1) · (I + 1) ≤ a ∧ t = 2 · I + 1 + 2 ∧ s = (I + 1) · (I + 1) + t ∧ i = I + 1}
// {∃ I. I · I ≤ a ∧ (I + 1) · (I + 1) ≤ a ∧ t = 2 · I + 3 ∧ s = (I + 1) · (I + 1) + t ∧ i = I + 1}
// {(i - 1) · (i - 1) ≤ a ∧ ((i - 1) + 1) · ((i - 1) + 1) ≤ a ∧ t = 2 · (i - 1) + 3 ∧ s = ((i - 1) + 1) · ((i - 1) + 1) + t}
// {i · i ≤ a ∧ t = 2 · i + 1 ∧ s = i · i + t}
```

Vorwärtsverkettung bei der Arbeit II

Mit Vereinfachung on-the-fly:

```
// {i · i ≤ a ∧ t = 2 · i + 1 ∧ s = i · i + t ∧ s ≤ a}
t = t + 2;
// {∃ T. (i · i ≤ a ∧ t = 2 · i + 1 ∧ s = i · i + t ∧ s ≤ a) [T/t] ∧ t = (t + 2) [T/t]}
// {∃ T. i · i ≤ a ∧ s = i · i + T ∧ s ≤ a ∧ t = T + 2 ∧ T = 2 · i + 1}
// {i · i ≤ a ∧ s = i · i + 2 · i + 1 ∧ s ≤ a ∧ t = (2 · i + 1) + 2}
// {i · i ≤ a ∧ s = (i + 1) · (i + 1) ∧ s ≤ a ∧ t = 2 · i + 3}
s = s + t;
// {∃ S. (i · i ≤ a ∧ s = (i + 1) · (i + 1) ∧ s ≤ a ∧ t = 2 · i + 3) [S/s] ∧ s = (s + t) [S/s]}
// {∃ S. i · i ≤ a ∧ S = (i + 1) · (i + 1) ∧ s ≤ a ∧ t = 2 · i + 3 ∧ s = S + t}
// {i · i ≤ a ∧ (i + 1) · (i + 1) ≤ a ∧ t = 2 · i + 3 ∧ s = (i + 1) · (i + 1) + t}
i = i + 1;
// {∃ I. (i · i ≤ a ∧ (i + 1) · (i + 1) ≤ a ∧ t = 2 · i + 3 ∧ s = (i + 1) · (i + 1) + t) [I/i] ∧ i = (i + 1) [I/i]}
// {∃ I. I · I ≤ a ∧ (I + 1) · (I + 1) ≤ a ∧ t = 2 · I + 3 ∧ s = (I + 1) · (I + 1) + t ∧ i = I + 1}
// {∃ I. I · I ≤ a ∧ (I + 1) · (I + 1) ≤ a ∧ t = 2 · I + 3 ∧ s = (I + 1) · (I + 1) + t ∧ i = I + 1}
// {(i - 1) · (i - 1) ≤ a ∧ ((i - 1) + 1) · ((i - 1) + 1) ≤ a ∧ t = 2 · (i - 1) + 3 ∧ s = ((i - 1) + 1) · ((i - 1) + 1) + t}
// {i · i ≤ a ∧ t = 2 · i + 1 ∧ s = i · i + t}
```

Arbeitsblatt 10.3: Vorwärtsverkettung

Gegeben folgendes Programm. Berechnet die Vorwärtsverkettung der Vorbedingung mit Vereinfachung:

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

Was bewirkt das Programm?

Beweis der Zuweisungsregel Vorwärts

Erinnert Euch an das **Substitutionslemma**:

$$\sigma \models^I B[e/x] \iff \sigma[x \mapsto \llbracket e \rrbracket_A(\sigma)] \models^I B$$

Zu zeigen:

$$\begin{aligned} & \models \{P\} x = e \{ \exists V. P[V/x] \wedge x = (e[V/x]) \} \\ \iff & \forall I. \forall \sigma. \sigma \models^I P \wedge \exists \sigma'. (\sigma, \sigma') \in \llbracket x = e \rrbracket_c \longrightarrow \sigma' \models^I \exists V. P[V/x] \wedge x = (e[V/x]) \\ \iff & \forall I. \forall \sigma. \sigma \models^I P \longrightarrow \sigma[x \mapsto \llbracket e \rrbracket_A(\sigma)] \models^I \exists V. P[V/x] \wedge x = (e[V/x]) \\ \iff & \forall I. \forall \sigma. \sigma \models^I P \longrightarrow \sigma \models^I (\exists V. P[V/x] \wedge x = (e[V/x]))[e/x] \\ \iff & \forall I. \forall \sigma. \sigma \models^I P \longrightarrow \sigma \models^I (\exists V. P[V/x] \wedge e = (e[V/x])) \\ \iff & \forall I. \forall \sigma. \sigma \models^I P \longrightarrow \sigma \models^I (P[x/x] \wedge e = (e[x/x])) \\ \iff & \forall I. \forall \sigma. \sigma \models^I P \longrightarrow \sigma \models^I P \quad \square \end{aligned}$$

Vorwärtsverkettung

- ▶ Vorwärtsregelaxiom äquivalent zum Rückwärtsregelaxiom.
- ▶ Vorteil: Vorbedingung bleibt kleiner
- ▶ Nachteil: in der Anwendung **umständlicher**
- ▶ Die entstehende Nachbedingung beschreibt die **symbolische Auswertung**
- ▶ 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**.

II. Vorwärtsberechnung von Verifikationsbedingungen

Stärkste Nachbedingung

- ▶ Vorwärtsberechnung von Verifikationsbedingungen: Nachbedingung
- ▶ Gegeben C0-Programm c , Prädikat P , dann ist
 - ▶ $\text{sp}(P, c)$ 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 $P \in \mathbf{Assn}$ und Programm $c \in \mathbf{Stmt}$, dann

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

- ▶ Wie können wir $\text{sp}(P, c)$ 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

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

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;
  // {∃ V. 0 ≤ n ∧ p = 1[V/p]}
  // {0 ≤ n ∧ p = 1}
3 c = 1;
  // {∃ V. (0 ≤ n ∧ p = 1)[V/c] ∧ c = 1[V/c]}
  // {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 }
  // {¬(c ≤ n) ∧ p = (c-1)! ∧ c-1 ≤ n}
8 // {p = n!}
```

$$VC_1 = \{\text{asp}_3 \longrightarrow p = (c-1)! \wedge c-1 \leq n\}$$

Beispiel: Fakultät, stärkste Nachbedingung (Schleifenrumpf)

```

1  // {0 ≤ n}
2  p = 1;
3  // {0 ≤ n ∧ p = 1}
4  c = 1;
5  // {0 ≤ n ∧ p = 1 ∧ c = 1}
6  while (c <= n) /** inv p = (c-1)! ∧ c-1 ≤ n; */ {
7    p = p * c;
8    // {∃ V1. (p = (c-1)! ∧ (c-1) ≤ n ∧ c ≤ n) [V1/p] ∧ p = (p · c) [V1/p]}
9    // {∃ V1. (V1 = (c-1)! ∧ (c-1) ≤ n ∧ c ≤ n) ∧ p = (V1 · c)}
10   // {c-1 ≤ n ∧ c ≤ n ∧ p = (c-1)! · c}
11   c = c + 1;
12   // {∃ V2. (c-1 ≤ n ∧ c ≤ n ∧ p = (c-1)! · c) [V2/c] ∧ c = (c+1) [V2/c]}
13   // {∃ V2. (V2-1 ≤ n ∧ V2 ≤ n ∧ p = (V2-1)! · V2) ∧ c = (V2+1)}
14   // {c-2 ≤ n ∧ c-1 ≤ n ∧ p = (c-2)! · (c-1)}
15 }
16 // {¬(c ≤ n) ∧ p = (c-1)! ∧ c-1 ≤ n}
17 // {p = n!}

```

Korrekte Software

25 [35]



Beispiel: Fakultät, Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```

1  // {0 ≤ n}
2  p = 1;
3  // svc2 = ∅
4  c = 1;
5  // svc3 = ∅
6  while (c <= n) /** inv {p = (c-1)! ∧ c-1 ≤ n; */ {
7    p = p * c;
8    // svc5 = ∅
9    c = c + 1;
10   // svc6 = ∅
11   // svc4 = {asp3 → (p = (c-1)! ∧ c-1 ≤ n),
12   //          asp6 → (p = (c-1)! ∧ c-1 ≤ n)}
13   // svc4 = {(0 ≤ n ∧ p = 1 ∧ c = 1) → (p = (c-1)! ∧ c-1 ≤ n),
14   //          (c-2 ≤ n ∧ c-1 ≤ n ∧ p = (c-2)! · (c-1))
15   //          → (p = (c-1)! ∧ c-1 ≤ n)}
16 }
17 // {p = n!}

```

Korrekte Software

26 [35]



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

27 [35]



Arbeitsblatt 10.4: 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

28 [35]



Beispiel: Suche nach dem Maximalen Element

```

1  // {0 < n}
2  i = 0;
3  // {∃ b0. (0 < n) [b0/i] ∧ i = 0 [b0/i]}
4  // {0 < n ∧ i = 0}
5  r = 0;
6  // {0 < n ∧ i = 0 ∧ r = 0}
7  while (i != n)
8  /** inv (∀ j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n */ {
9    if (a[r] < a[i]) {
10     r = i;
11   }
12   else {
13     }
14   i = i + 1;
15 }
16 // {∀ j. 0 ≤ j < i → a[j] ≤ a[r] ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ ¬(i ≠ n)}
17 // {(∀ j. 0 ≤ j < n → a[j] ≤ a[r]) ∧ 0 ≤ r < n}

```

Korrekte Software

29 [35]



Beispiel: Suche nach dem Maximalen Element (Schleifenrumpf)

```

1  while (i != n)
2  /** inv (∀ j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n */ {
3    if (a[r] < a[i]) {
4      // {(∀ j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
5      r = i;
6      // {∃ R0. ((∀ j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]) [R0/r] ∧ r = i [R0/r]}
7      // {∃ R0. (∀ j. 0 ≤ j < i → a[j] ≤ a[R0]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ R0 < n ∧ a[R0] < a[i] ∧ r = i}
8    }
9    else {
10     // {(∀ j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
11     // {(∀ j. 0 ≤ j < i → a[j] ≤ a[R0]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ R0 < n ∧ a[R0] < a[i] ∧ r = i}
12     // {(∀ j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ a[r] ≤ a[i]}
13     i = i + 1;
14     // {∃ b0. ((∃ R0. (∀ j. 0 ≤ j < b0 → a[j] ≤ a[R0]) ∧ 0 ≤ b0 < n ∧ 0 ≤ R0 < n ∧ a[R0] < a[b0] ∧ r = b0)
15     //          ∨ ((∀ j. 0 ≤ j < b0 → a[j] ≤ a[r]) ∧ 0 ≤ b0 < n ∧ 0 ≤ r < n ∧ a[b0] ≤ a[r])) ∧ i = b0 + 1}
16   }

```

Korrekte Software

30 [35]



Verifikationsbedingungen

- $0 < n \wedge i = 0 \wedge r = 0 \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n$
- $(\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge \neg(i \neq n) \rightarrow (\forall j. 0 \leq j < n \rightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$
- $(\exists b_0. ((\exists R_0. (\forall j. 0 \leq j < b_0 \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq b_0 < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[b_0] \wedge r = b_0) \vee ((\forall j. 0 \leq j < b_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq b_0 < n \wedge 0 \leq r < n \wedge a[b_0] \leq a[r]))) \wedge i = b_0 + 1 \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n$

Korrekte Software

31 [35]



Weitere Vereinfachungsregeln

Existenzquantoren und Disjunktionen können mit folgenden Regeln vereinfacht werden:

- Der Gültigkeitsbereich des Existenzquantors kann verkleinert werden:

$$\exists x. P \vee Q \rightsquigarrow (\exists x. P) \vee (\exists x. Q)$$

- Disjunktionen in der Prämisse ergeben eine Fallunterscheidung:

$$A_1 \vee A_2 \rightarrow B \rightsquigarrow A_1 \rightarrow B, A_2 \rightarrow B$$

- Konjunktion distributiert über Disjunktion:

$$(A_1 \vee A_2) \wedge B \rightsquigarrow (A_1 \wedge B) \vee (A_2 \wedge B)$$

- ... und andersherum:

$$(A_1 \wedge A_2) \vee B \rightsquigarrow (A_1 \vee B) \wedge (A_2 \vee B)$$

Korrekte Software

32 [35]



Vereinfachte Verifikationsbedingungen

- 1.1 $0 < n \wedge i = 0 \wedge r = 0 \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$
- 1.2 $0 < n \wedge i = 0 \wedge r = 0 \rightarrow 0 \leq r < n$
- 1.3 $0 < n \wedge i = 0 \wedge r = 0 \rightarrow 0 \leq i \leq n$
- 2.1 $(\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge \neg(i \neq n) \rightarrow (\forall j. 0 \leq j < n \rightarrow a[j] \leq a[r])$
- 2.2 $(\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge \neg(i \neq n) \rightarrow 0 \leq r < n$
- 3.1 $(\exists l_0. (\exists R_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq l_0 < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[l_0] \wedge r = l_0) \wedge i = l_0 + 1) \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$
- 3.1 $(\exists R_0. ((\forall j. 0 \leq j < r \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1) \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$
- 3.2 $(\exists l_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq l_0 < n \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1) \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$
- 3.3 $(\exists l_0. (\exists R_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq l_0 < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[l_0] \wedge r = l_0) \wedge i = l_0 + 1) \rightarrow 0 \leq i \leq n$
- 3.3 $(\exists R_0. (\forall j. 0 \leq j < r \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1 \rightarrow 0 \leq i \leq n$
- 3.4 $(\exists l_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq l_0 < n \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1) \rightarrow 0 \leq i \leq n$
- 3.5 $(\exists l_0. (\exists R_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq l_0 < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[l_0] \wedge r = l_0) \wedge i = l_0 + 1) \rightarrow 0 \leq r < n$
- 3.5 $(\exists R_0. (\forall j. 0 \leq j < r \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1 \rightarrow 0 \leq r < n$
- 3.6 $(\exists l_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq l_0 < n \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1) \rightarrow 0 \leq r < n$

Korrekte Software

33 [35]



Vereinfachte Verifikationsbedingungen

- 1.1 $0 < n \wedge i = 0 \wedge r = 0 \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$ ✓
- 1.2 $0 < n \wedge i = 0 \wedge r = 0 \rightarrow 0 \leq r < n$ ✓
- 1.3 $0 < n \wedge i = 0 \wedge r = 0 \rightarrow 0 \leq i \leq n$ ✓
- 2.1 $(\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge \neg(i \neq n) \rightarrow (\forall j. 0 \leq j < n \rightarrow a[j] \leq a[r])$ ✓
- 2.2 $(\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n \wedge \neg(i \neq n) \rightarrow 0 \leq r < n$ ✓
- 3.1 $(\exists R_0. ((\forall j. 0 \leq j < r \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1) \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$ ✓
- 3.2 $(\exists l_0. ((\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq l_0 < n \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1) \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$ ✓
- 3.3 $(\exists R_0. (\forall j. 0 \leq j < r \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1 \rightarrow 0 \leq i \leq n$ ✓
- 3.4 $(\exists l_0. ((\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq l_0 < n \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1) \rightarrow (\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r])$ ✓
- 3.5 $(\exists R_0. (\forall j. 0 \leq j < r \rightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1 \rightarrow 0 \leq r < n$ ✓
- 3.6 $(\exists l_0. (\forall j. 0 \leq j < l_0 \rightarrow a[j] \leq a[r]) \wedge 0 \leq l_0 < n \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1) \rightarrow 0 \leq r < n$ ✓

Korrekte Software

34 [35]



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

35 [35]

