

Korrekte Software: Grundlagen und Methoden  
Vorlesung 6 vom 24.05.22  
Floyd-Hoare-Logik II: Invarianten

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2022

10:21:00 2022-06-28

1 [16]



## 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 [16]



## Die Floyd-Hoare-Logik bis hierher

- **Hoare-Tripel**  $\{P\} c \{Q\}$  spezifizieren was  $c$  berechnet (**Korrektheit**)
  - Semantische **Gültigkeit** von Hoare-Tripeln:  $\models \{P\} c \{Q\}$ .
  - Syntaktische **Herleitbarkeit** von Hoare-Tripeln:  $\vdash \{P\} c \{Q\}$
- **Zuweisungen** werden durch **Substitution** modelliert, d.h. die Menge der gültigen Aussagen ändert sich.
- Für Iterationen wird eine **Invariante** benötigt (die **nicht** hergeleitet werden kann).

Korrekte Software

3 [16]



## Überblick: die Regeln des Floyd-Hoare-Kalküls

$$\frac{}{\vdash \{P[e/x]\} x = e \{P\}}$$
$$\frac{\vdash \{A \wedge b\} c_0 \{B\} \quad \vdash \{A \wedge \neg b\} c_1 \{B\}}{\vdash \{A\} \text{if } (b) \text{ c}_0 \text{ else } c_1 \{B\}}$$
$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \text{while}(b) c \{A \wedge \neg b\}}$$
$$\frac{}{\vdash \{A\} \{ \} \{A\}} \quad \frac{\vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\}}{\vdash \{A\} c_1; c_2 \{C\}}$$
$$\frac{A' \Rightarrow A \quad \vdash \{A\} c \{B\} \quad B \Rightarrow B'}{\vdash \{A'\} c \{B'\}}$$

Korrekte Software

4 [16]



## Invarianten finden: die Fakultät

Invariante:

$$p = (c-1)! \wedge c-1 \leq n \wedge c > 0$$

- Kern der Invariante: Fakultät bis  $c-1$  berechnet.
- Invariante impliziert Nachbedingung  $p = n! = (c-1)!$ 
  - $\neg(c \leq n) \Leftrightarrow c-1 \geq n$  — was fehlt?
- Nebenbedingung für Weakening innerhalb der Schleife.
  - $c! = c * (c-1)!$  gilt nur für  $c > 0$ .

Korrekte Software

5 [16]



## Invarianten finden

1. Initiale Invariante: momentaner Zustand der Berechnung
2. Invariante und negierte Schleifenbedingung muss Nachbedingung implizieren; ggf. Invariante verstärken.
3. Beweise innerhalb der Schleife benötigen ggf. weitere Nebenbedingungen; Invariante verstärken.

Korrekte Software

6 [16]



## Zählende Schleifen

- Fakultät ist Beispiel für zählende Schleife (**for**).
- Für Nachbedingung  $\psi[n]$  ist Invariante:

$$\psi[i-1/n] \wedge i-1 \leq n$$

- Ggf. weitere Nebenbedingungen erforderlich
- Variante:  $i = 0, \dots, n-1$

```
for (i = 1; i <= n; i++) {  
    ...  
}  
  
ist syntaktischer Zucker für  
  
i = 1;  
while (i <= n) {  
    ...  
    i = i+1;  
}
```

Korrekte Software

7 [16]



## Arbeitsblatt 6.1: Summe I

```
1 // {0 ≤ n}  
2 x = 0;  
3 c = 1;  
4 while (c <= n) {  
5     x = x+c;  
6     c = c+1;  
7 }  
8 // {x = sum(0, n)}
```

1. Was ist die initiale Invariante?
  2. Was fehlt, um aus der initialen Invariante die Nachbedingung zu schließen?
  3. Was fehlt, damit der Schleifenrumpf die Invariante erhält?
- Annotiert das Programm mit den Korrektheitszusicherungen!

Hierbei ist  $\text{sum}(a, b)$  die Summe der Zahlen von  $a$  bis  $b$ , mit folgenden Eigenschaften:

$$\begin{aligned} a > b &\Rightarrow \text{sum}(a, b) = 0 \\ a \leq b &\Rightarrow \text{sum}(a, b) = a + \text{sum}(a+1, b) \\ a \leq b &\Rightarrow \text{sum}(a, b) = \text{sum}(a, b-1) + b \end{aligned}$$

Korrekte Software

8 [16]



## Variante der zählenden Schleife

```
// {0 ≤ y}
// {0 = sum(0, 0) ∧ 0 ≤ y ∧ 0 ≤ 0}
x = 0;
// {x = sum(0, 0) ∧ 0 ≤ y ∧ 0 ≤ 0}
c = 0;
// {x = sum(0, c) ∧ c ≤ y ∧ 0 ≤ c}
while (c < y) {
  // {x = sum(0, c) ∧ c ≤ y ∧ 0 ≤ c ∧ c < y}
  // {x + (c + 1) = sum(0, c) + (c + 1) ∧ c < y ∧ 0 ≤ c}
  // {x + c + 1 = sum(0, c + 1) ∧ c + 1 ≤ y ∧ 0 ≤ c + 1}
  c = c + 1;
  // {x + c = sum(0, c) ∧ c ≤ y ∧ 0 ≤ c}
  x = x + c;
  // {x = sum(0, c) ∧ c ≤ y ∧ 0 ≤ c}
}
// {x = sum(0, c) ∧ c ≤ y ∧ 0 ≤ c ∧ ¬(c < y)}
// {x = sum(0, c) ∧ c ≤ y ∧ c ≥ y}
// {x = sum(0, y)}
```

- Was ist hier die Invariante?

$$x = \text{sum}(0, c) \wedge c \leq y \wedge 0 \leq c$$

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

## Arbeitsblatt 6.2: Summe II

```
// {n = N ∧ 0 ≤ n}
x = 0;
while (n != 0) {
  x = x + n;
  n = n - 1;
}
// {x = sum(0, N)}
```

- Was ist der erste Teil der Invariante?
- Der Rest ist wie vorher?
- Annotiert das Programm mit dem Korrektheitszusicherungen.

## Fakultät Revisited

Dieses Programm berechnet die Fakultät von  $n$ :

```
// {n = N ∧ 0 ≤ n}
// {n! = N! ∧ n = N ∧ 0 ≤ n}
// {n! · 1 = N! ∧ n ≤ N ∧ 0 ≤ n}
p = 1;
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
while (0 < n) {
  // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ 0 < n}
  // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
  // {(n-1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
  p = n * p;
  // {(n-1)! · p = N! ∧ n ≤ N ∧ 0 < n}
  // {(n-1)! · p = N! ∧ n-1 ≤ N ∧ 0 ≤ n-1}
  n = n - 1;
  // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$\begin{aligned} n! \cdot p &= N! \\ \wedge 0 &\leq n \\ \wedge n &\leq n \end{aligned}$$

## Arbeitsblatt 6.3: Nicht-zählende Schleife

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

- Was ist hier die Invariante?
- Hinweis: es ist ganz einfach.

## Beispiel 5: Jetzt wird's kompliziert. . .

```
1 // {0 ≤ a}
2 t = 1;
3 s = 1;
4 i = 0;
5 while (s <= a) {
6   t = t + 2;
7   s = s + t;
8   i = i + 1;
9 }
10 // ? {i² ≤ a ∧ a < (i + 1)²}
```

- Was berechnet das? Ganzzahlige Wurzel von  $a$ .
- Invariante:
$$s - t \leq a \wedge t = 2 \cdot i + 1 \wedge s = i^2 + t$$
- Nachbedingung 1:
  - $s - t \leq a, s = i^2 + t \implies i^2 \leq a$ .
- Nachbedingung 2:
  - $s = i^2 + t, t = 2 \cdot i + 1 \implies s = (i + 1)^2$
  - $a < s, s = (i + 1)^2 \implies a < (i + 1)^2$

## Beispiel 5: Jetzt wird's kompliziert. . .

```
// {0 ≤ a}
// {1 - 1 ≤ a ∧ 1 = 2 · 0 + 1 ∧ 1 = 0² + 1}
t = 1;
// {1 - t ≤ a ∧ t = 2 · 0 + 1 ∧ 1 = 0² + t}
s = 1;
// {s - t ≤ a ∧ t = 2 · 0 + 1 ∧ s = 0² + t}
i = 0;
// {s - t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t}
while (s <= a) {
  // {s - t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t ∧ s ≤ a}
  // {t = 2 · i + 1 ∧ s = i² + t ∧ s ≤ a}
  // {s ≤ a ∧ t + 2 = 2 · i + 3 ∧ s = i² + 2 · i + 1}
  // {s + (t + 2) - (t + 2) ≤ a ∧ t + 2 = 2 · (i + 1) + 1 ∧ s + (t + 2) = (i + 1)² + (t + 2)}
  t = t + 2;
  // {s + t - t ≤ a ∧ t = 2 · (i + 1) + 1 ∧ s + t = (i + 1)² + t}
  s = s + t;
  // {s - t ≤ a ∧ t = 2 · (i + 1) + 1 ∧ s = (i + 1)² + t}
  i = i + 1;
  // {s - t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t}
}
// {s - t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t ∧ ¬(s ≤ a)}
// ? {i² ≤ a ∧ a < (i + 1)²}
```

## Zum Abschluss etwas Magie

**Fast Inverse Square Root**  
(Quake III, John Carmack)

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y = number;
  i = * (long *) &y;
  i = 0x5f3759df - ( i >> 1 );
  y = * (float *) &i;
  y = y * (threehalfs - (x2 * y * y));
  // y = y * (threehalfs - (x2 * y * y));
  return y;
}
```

- Berechnet effizient Approximation von  $\frac{1}{\sqrt{y}}$
- Verkürztes **Newton-Verfahren**
- „Evil floating-point bit-level hacking“
- $0x5f3759df = 1.597.463.007 \approx \sqrt{2^{127}}$
- Nicht zu verifizieren (nicht standard-konform)

## Zusammenfassung

- Der schwierigste Teil bei Korrektheitsbeweisen mit dem Floyd-Hoare-Kalkül sind die while-Schleifen.
- Die Regel für die while-Schleife braucht eine **Invariante**, die nicht aus der Anwendung erschlossen werden kann.
- Wir können die Invariante in drei Stufen konstruieren:
  - 1 Algorithmischer Kern: was wird bis hier berechnet?
  - 2 Ist die Invariante **stark** genug, um die Nachbedingung zu implizieren?
  - 3 Wird die Invariante durch die Schleife erhalten? Werden noch Nebenbedingungen benötigt?
- Vereinfachender Sonderfall: **zählende** Schleifen (for-Schleifen)