

Beweis:

Sei $\pi = (u_0, \sigma_0) s_1 (u_1, \sigma_1) \dots s_m (u_m, \sigma_m)$

Gelte: $\sigma_0 \models A$.

Wir müssen zeigen: $\sigma_m \models B$.

Idee:

Induktion nach der Länge m der Programmausführung :-)

$m = 0$:

Der Endpunkt der Ausführung ist gleich dem Startpunkt

$\implies \sigma_0 = \sigma_m$ und $A \equiv B$

\implies Behauptung gilt :-)

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Behandeln wir erst einmal Tests $s_m \equiv b$.

Dann ist insbesondere $\sigma_{m-1} = \sigma_m$:-)

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Behandeln wir erst einmal Tests $S_m \equiv b$.

Dann ist insbesondere $\sigma_{m-1} = \sigma_m$:-)

1. Möglichkeit: $\sigma_m \models b$

$$\implies B' \Rightarrow \mathbf{WP}[b](C, B) \quad \text{wobei}$$
$$\mathbf{WP}[b](C, B) \equiv (\neg b \Rightarrow C) \wedge (b \Rightarrow B)$$

$$\implies \sigma_m \models b \wedge (b \Rightarrow B)$$

$$\implies \sigma_m \models B \quad \text{:-)}$$

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Behandeln wir erst einmal Tests $S_m \equiv b$.

Dann ist insbesondere $\sigma_{m-1} = \sigma_m$:-)

2. Möglichkeit: $\sigma_m \models \neg b$

$$\implies B' \Rightarrow \mathbf{WP}[b](B, C) \quad \text{wobei}$$
$$\mathbf{WP}[b](B, C) \equiv (\neg b \Rightarrow B) \wedge (b \Rightarrow C)$$

$$\implies \sigma_m \models \neg b \wedge (\neg b \Rightarrow B)$$

$$\implies \sigma_m \models B \quad \text{:-)}$$

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Nun behandeln wir Anweisungen.

1. Möglichkeit: $S_m \equiv ;$

Dann gilt:

- $\sigma_{m-1} = \sigma_m$
- $\mathbf{WP}[[;]](B) \equiv B$

$$\implies B' \Rightarrow B$$

$$\implies \sigma_{m-1} = \sigma_m \models B \quad :-)$$

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Nun behandeln wir Anweisungen.

2. Möglichkeit: $s_m \equiv \text{write}(e);$

Dann gilt:

- $\sigma_{m-1} = \sigma_m$
- $\text{WP}[\text{;write}(e)](B) \equiv B$

$$\implies B' \Rightarrow B$$

$$\implies \sigma_{m-1} = \sigma_m \models B \quad \text{: -)}$$

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Nun behandeln wir Anweisungen.

3. Möglichkeit: $s_m \equiv x = \text{read}()$;

Dann gilt: • $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto c\}$ für ein $c \in \mathbb{Z}$

• $\mathbf{WP}[\![x = \text{read}();]\!] (B) \equiv \forall x. B$

$$\implies B' \Rightarrow \forall x. B \Rightarrow B[c/x]$$

$$\implies \sigma_m \models B \quad :-)$$

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$$\implies \sigma_{m-1} \models B'$$

Nun behandeln wir Anweisungen.

4. Möglichkeit: $s_m \equiv x = e$; Dann gilt:

• $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto c\}$ für $c = \text{Wert von } e \text{ in } \sigma_{m-1}$

• $\mathbf{WP}[\![x = e;\!] (B) \equiv B[e/x]$

$$\implies B' \Rightarrow B[e/x] \implies \sigma_{m-1} \models B[c/x]$$

$$\implies \sigma_m \models B \quad (:-))$$

Fazit:

- Das Verfahren nach Floyd ermöglicht uns zu beweisen, dass eine Zusicherung B bei Erreichen eines Programmpunkts stets (bzw. unter geeigneten Zusatzannahmen $:-)$ gilt ...
- Zur Durchführung benötigen wir:
 - Zusicherungen an jedem Programmpunkt $:-)$
 - Nachweis, dass die Zusicherungen lokal konsistent sind
 \implies Logik, automatisches Beweisen

3.3 Optimierung

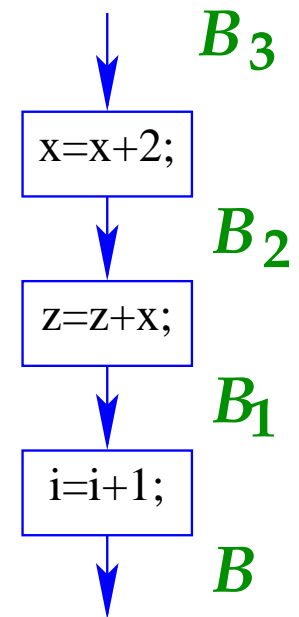
Ziel: Verringerung der benötigten Zusicherungen

Beobachtung:

Hat das Programm **keine Schleifen**, können wir für jeden Programmpunkt eine hinreichende Vorbedingung **ausrechnen !!!**

Beispiel:

```
x = x+2;  
z = z+x;  
i = i+1;
```



Beispiel (Fort.):

Sei $B \equiv z = i^2 \wedge x = 2i - 1$

Dann rechnen wir:

$$\begin{aligned} B_1 &\equiv \mathbf{WP}[[i = i+1;]](B) &&\equiv z = (i+1)^2 \wedge x = 2(i+1) - 1 \\ &&&\equiv z = (i+1)^2 \wedge x = 2i + 1 \end{aligned}$$

Beispiel (Fort.):

Sei $B \equiv z = i^2 \wedge x = 2i - 1$

Dann rechnen wir:

$$B_1 \equiv \mathbf{WP}[[i = i+1;]](B) \equiv z = (i+1)^2 \wedge x = 2(i+1) - 1$$

$$\equiv z = (i+1)^2 \wedge x = 2i + 1$$

$$B_2 \equiv \mathbf{WP}[[z = z+x;]](B_1) \equiv z + x = (i+1)^2 \wedge x = 2i + 1$$

$$\equiv z = i^2 \wedge x = 2i + 1$$

Beispiel (Fort.):

Sei $B \equiv z = i^2 \wedge x = 2i - 1$

Dann rechnen wir:

$$\begin{aligned} B_1 &\equiv \mathbf{WP}[[i = i+1;]](B) &&\equiv z = (i+1)^2 \wedge x = 2(i+1) - 1 \\ & &&\equiv z = (i+1)^2 \wedge x = 2i+1 \\ B_2 &\equiv \mathbf{WP}[[z = z+x;]](B_1) &&\equiv z + x = (i+1)^2 \wedge x = 2i+1 \\ & &&\equiv z = i^2 \wedge x = 2i+1 \\ B_3 &\equiv \mathbf{WP}[[x = x+2;]](B_2) &&\equiv z = i^2 \wedge x + 2 = 2i+1 \\ & &&\equiv z = i^2 \wedge x = 2i-1 \\ & &&\equiv B \quad ;-)) \end{aligned}$$

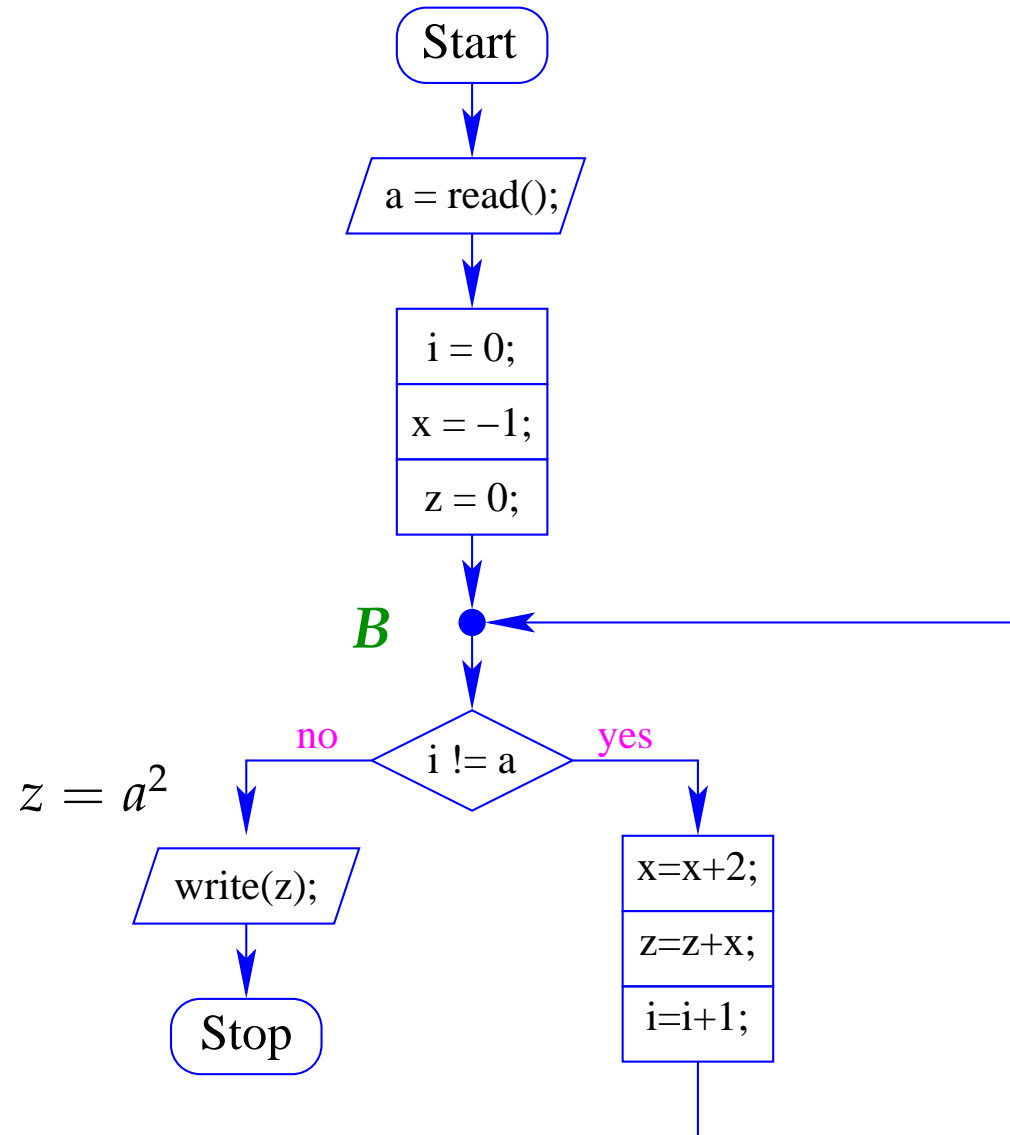
Idee:

- Für jede Schleife wähle **einen** Programmpunkt aus.
Sinnvolle Auswahlen:
 - Vor der Bedingung;
 - Am Beginn des Rumpfs;
 - Am Ende des Rumpfs ...
- Stelle für jeden gewählten Punkt eine Zusicherung bereit
 - ⇒ **Schleifen-Invariante**
- Für alle übrigen Programmpunkte bestimmen wir Zusicherungen mithilfe $\mathbf{WP}[\dots](\) \text{ :-}$

Beispiel:

```
int a, i, x, z;
a = read();
i = 0;
x = -1;
z = 0;
while (i != a) {
    x = x+2;
    z = z+x;
    i = i+1;
}
assert(z==a*a);
write(z);
```

Beispiel:



Wir überprüfen:

$\mathbf{WP}[[i \neq a]](z = a^2, B)$

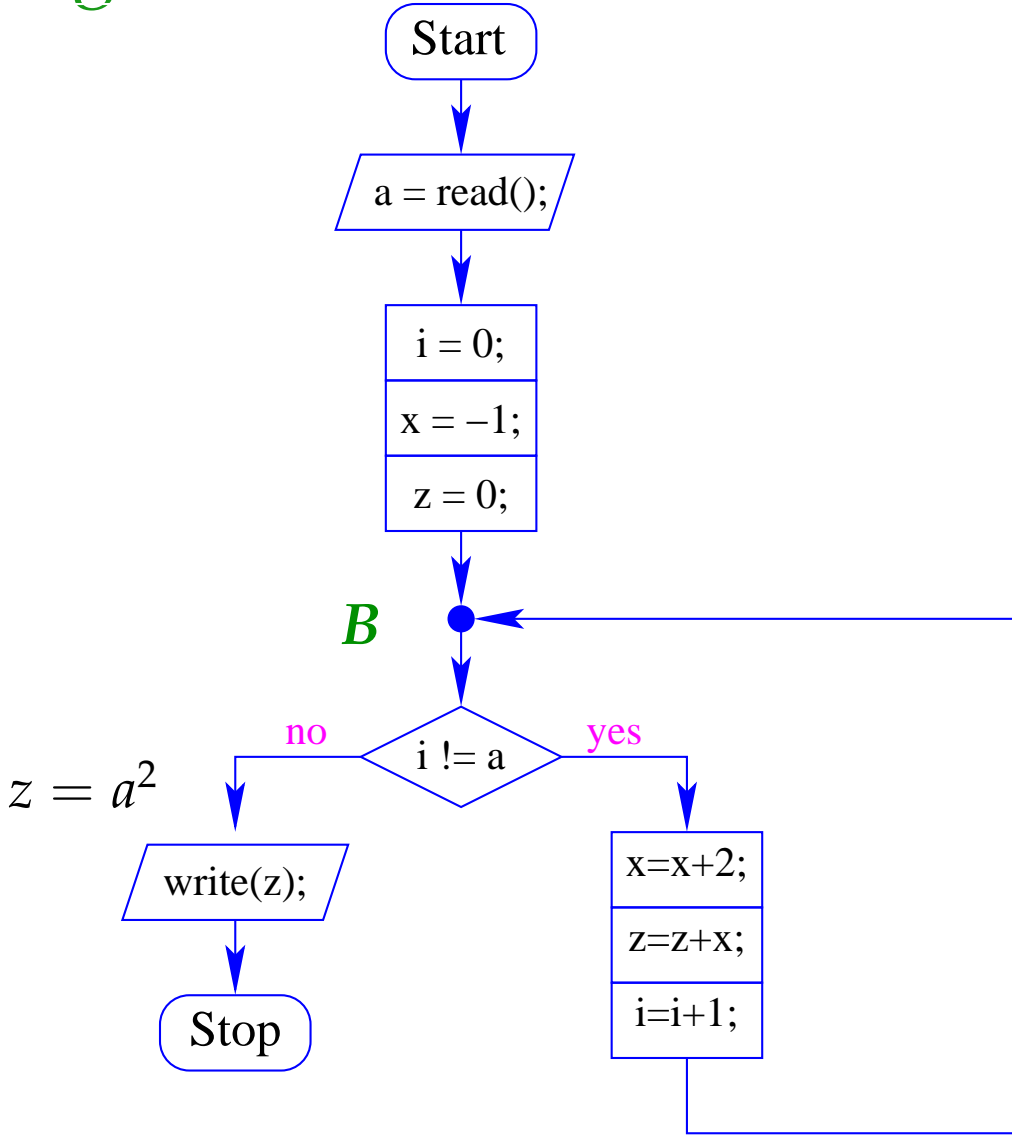
$$\equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge B) \vee (z = a^2 \wedge B)$$

$$\equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \vee \\ (z = a^2 \wedge z = i^2 \wedge x = 2i - 1)$$

$$\Leftarrow (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \vee (i = a \wedge z = i^2 \wedge x = 2i - 1)$$

$$\equiv z = i^2 \wedge x = 2i - 1 \quad \equiv \quad B \quad \text{:-)}$$

Orientierung:



Wir überprüfen:

$$\begin{aligned}\mathbf{WP}[\mathbf{z} = 0;](B) &\equiv 0 = i^2 \wedge x = 2i - 1 \\ &\equiv i = 0 \wedge x = -1 \\ \mathbf{WP}[\mathbf{x} = -1;](i = 0 \wedge x = -1) &\equiv i = 0 \\ \mathbf{WP}[\mathbf{i} = 0;](i = 0) &\equiv \mathbf{true} \\ \mathbf{WP}[\mathbf{a} = \text{read}();](\mathbf{true}) &\equiv \mathbf{true} \quad \text{: -))}\end{aligned}$$

3.4 Terminierung

Problem:

- Mit unserer Beweistechnik können wir nur beweisen, dass eine Eigenschaft gilt wann immer wir einen Programmpunkt erreichen !!!
- Wie können wir aber garantieren, dass das Programm immer terminiert ?
- Wie können wir eine Bedingung finden, unter der das Programm immer terminiert ??

Beispiele:

- Das ggT-Programm terminiert nur für Eingaben a, b mit: $a > 0$ und $b > 0$.
- Das Quadrier-Programm terminiert nur für Eingaben $a \geq 0$.
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer :-)

Beispiele:

- Das ggT-Programm terminiert nur für Eingaben a, b mit: $a > 0$ und $b > 0$.
- Das Quadrier-Programm terminiert nur für Eingaben $a \geq 0$.
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer :-)

Lässt sich dieses Beispiel verallgemeinern ??

Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
    j = read();
    while (j>0) { t = t+1; j = j-1; }
    i = i-1;
}
write(t);
```

- Die gelesene Zahl i (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-)) die Summe der positiven für j gelesenen Werte

Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
    j = read();
    while (j>0) { t = t+1; j = j-1; }
    i = i-1;
}
write(t);
```

- Die gelesene Zahl i (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-)) die Summe der positiven für j gelesenen Werte
 \implies das Programm terminiert immer !!!

Programme nur mit for-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
```

// im Rumpf wird i nicht modifiziert

... terminieren ebenfalls immer :-))

Programme nur mit for-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
```

// im Rumpf wird i nicht modifiziert

... terminieren ebenfalls immer :-))

Frage:

Wie können wir aus dieser Beobachtung eine Methode machen, die auf beliebige Schleifen anwendbar ist ?

Idee:

- Weise nach, dass jede Scheife nur endlich oft durchlaufen wird
...
- Finde für jede Schleife eine Kenngröße r , die zwei Eigenschaften hat:
 - (1) Wenn immer der Rumpf betreten wird, ist $r > 0$;
 - (2) Bei jedem Schleifen-Durchlauf wird r kleiner :-)
- Transformiere das Programm so, dass es neben der normalen Programmausführung zusätzlich die Kenngrößen r mitberechnet.
- Verifiziere, dass (1) und (2) gelten :-)

Beispiel: Sicheres ggT-Programm

```
int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
    else {
        while (x != y)
            if (y > x) y = y-x;
            else      x = x-y;
        write(x);
    }
```

Wir wählen: $r = x + y$

Transformation:

```
int a, b, x, y, r;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
    else { r = x+y;
        while (x != y) {
            if (y > x) y = y-x;
            else      x = x-y;
            r = x+y; }
        write(x);
    }
```