# Korrekte Software: Grundlagen und Methoden

## Vorlesung 9 vom 08.06.21

## Verifikationsbedingungen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

# Fahrplan

# Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;
//
y = x;
//
x = z;
// {X = y ∧ Y = x}
```

# Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;
//
y = x;
// {X = y ∧ Y = z}
x = z;
// {X = y ∧ Y = x}
```

# 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}
```

# 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 sehen:

1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
2. Die Verifikation kann **berechnet** werden.

# 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 sehen:

1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
2. Die Verifikation kann **berechnet** werden.

- Geht das immer?

# Berechnung von Vorbedingungen

- Die Rückwärtsrechnung von einer gegebenen Nachbedingung entspricht der Berechnung einer Vorbedingung.
- Gegeben C0-Programm $c$, Prädikat $Q$, dann ist
  - $\mathrm{wp}(c, Q)$ die **schwächste Vorbedingung** $P$ so dass $\models \{P\}\, c\, \{Q\}$;
  - Prädikat $P$ **schwächer** als $P'$ wenn $P' \Longrightarrow P$
- Semantische Charakterisierung:

**Schwächste Vorbedingung**

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

$$\models \{P\}\, c\, \{Q\} \Longleftrightarrow P \Longrightarrow \mathrm{wp}(c, Q)$$

- Wie können wir $\mathrm{wp}(c, Q)$ berechnen?

# Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — $P$ passt auf jede beliebige Nachbedingung

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

# Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — $P$ passt auf jede beliebige Nachbedingung

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

- Was ist mit den anderen Regeln?

$$\frac{}{\vdash \{A\}\, \{\,\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\, \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

# Arbeitsblatt 9.1: Eine Kleine Fallunterscheidung

Berechnet die Vorbedingung für folgendes Programm:

```
// ?
if (y == 7) {
  //
  x= 3;
  //
  }
else {
  y= 0;
  x= 10;
  //
  }
// x+ y == 10
```

# Rückwärtsanwendung: if

```
//
if (b) {
  //
  ...
  // {Q}
  }
else {
  //
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \land b\}\, c_0\, \{B\} \qquad \vdash \{A \land \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

# Rückwärtsanwendung: if

```
//
if (b) {
  //
  ...
  // {Q}
  }
else {
  // {P_2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

# Rückwärtsanwendung: if

```
//
if (b) {
  // {P_1}
  ...
  // {Q}
  }
else {
  // {P_2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \land b\}\, c_0\, \{B\} \qquad \vdash \{A \land \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \mathbf{if}\ (b)\ c_0\ \ \mathbf{else}\ \ c_1\, \{B\}}$$

# Rückwärtsanwendung: if

```
// ?
if (b) {
  // {P_1}
  ...
  // {Q}
  }
else {
  // {P_2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Regel in der Form nicht geeignet. Besser:

$$A \stackrel{def}{=} (P_1 \wedge b) \vee (P_2 \wedge \neg b)$$

$$(P_1 \wedge b) \vee (P_2 \wedge \neg b) \wedge b \Longleftrightarrow (P_1 \wedge b) \vee \mathit{false} \Longleftrightarrow P_1 \wedge b$$

$$(P_1 \wedge b) \vee (P_2 \wedge \neg b) \wedge \neg b \Longleftrightarrow \mathit{false} \vee (P_2 \wedge \neg b) \Longleftrightarrow P_2 \wedge \neg b$$

ergibt neue Regel:

$$\frac{\dfrac{}{\vdash \{P_1 \wedge b\}\, c_0\, \{B\}} \qquad \dfrac{}{\vdash \{P_2 \wedge \neg b\}\, c_1\, \{B\}}}{\vdash \{(P_1 \wedge b) \vee (P_2 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

# Rückwärtsanwendung: if

```
//
if (b) {
  // {P_1}
  ...
  // {Q}
  }
else {
  // {P_2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Regel in der Form nicht geeignet. Besser:

$$A \stackrel{def}{=} (P_1 \wedge b) \vee (P_2 \wedge \neg b)$$

$$(P_1 \wedge b) \vee (P_2 \wedge \neg b) \wedge b \Longleftrightarrow (P_1 \wedge b) \vee \mathit{false} \Longleftrightarrow P_1 \wedge b$$

$$(P_1 \wedge b) \vee (P_2 \wedge \neg b) \wedge \neg b \Longleftrightarrow \mathit{false} \vee (P_2 \wedge \neg b) \Longleftrightarrow P_2 \wedge \neg b$$

Kombiniert mit Weakening ergibt neue Regel:

$$\frac{\dfrac{P_1 \wedge b \Longrightarrow P_1 \quad \vdash \{P_1\}\, c_0\, \{B\}}{\vdash \{P_1 \wedge b\}\, c_0\, \{B\}} \qquad \dfrac{P_2 \wedge \neg b \Longrightarrow P_2 \quad \vdash \{P_2\}\, c_0\, \{B\}}{\vdash \{P_2 \wedge \neg b\}\, c_1\, \{B\}}}{\vdash \{(P_1 \wedge b) \vee (P_2 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

# Neue Regeln

- Wir können aus dem Hoare-Kalkül **neue Regeln** ableiten, in dem wir
  1. Existierende Regeln **instantiieren**, oder
  2. existierende Regeln **verknüpfen**.
- Wir benötigen das hier, um die Regeln des Hoare-Kalkül in eine Form zu bringen, welche die Rückwärtsrechnung ermöglicht.

Das Hinzufügen abgeleiteter Regeln ist eine **konservative Erweiterung** — es lassen sich damit nicht mehr oder weniger Hoare-Tripel $\vdash \{P\}\, c\, \{Q\}$ herleiten.

# Regeln für die Rückwärtsrechnung

1. **Nachbedingung** der **Konklusion** ist von der Form $\{Q\}$ (**offene** Meta-Variable)
2. Alle **Vorbedingungen** der **Prämissen** ist von der Form $\{P_i\}$ (**unterschiedliche** $P_i$)
3. Alle Variablen in den Vorbedingungen der Konklusion, den Weakenings und Nachbedingungen der Prämisse sind **determiniert**[1].

Welche Regeln passen noch nicht?

[1] **Entweder** in der Nachbedingung oder dem Programmausdruck der Konklusion, **oder** den Vorbedingungen der Prämisse enthalten.

# Regeln für die Rückwärtsrechnung

1. **Nachbedingung** der **Konklusion** ist von der Form $\{Q\}$ (**offene** Meta-Variable)
2. Alle **Vorbedingungen** der **Prämissen** ist von der Form $\{P_i\}$ (**unterschiedliche** $P_i$)
3. Alle Variablen in den Vorbedingungen der Konklusion, den Weakenings und Nachbedingungen der Prämisse sind **determiniert**[1].

Welche Regeln passen noch nicht? **while**-Regel passt noch nicht ...

[1] **Entweder** in der Nachbedingung oder dem Programmausdruck der Konklusion, **oder** den Vorbedingungen der Prämisse enthalten.

# Regeln für die Rückwärtsrechnung: **while**

- **while**-Regel (1) wird mit Weakening zu (2):

$$\frac{\vdash \{I \wedge b\}\, c\, \{I\}}{\vdash \{I\}\ \textbf{while}\ (b)\ c\, \{I \wedge \neg b\}} \tag{1}$$

$$\frac{I \wedge b \Longrightarrow R \qquad \vdash \{R\}\, c\, \{I\} \qquad I \wedge \neg b \Longrightarrow Q}{\vdash \{I\}\ \textbf{while}\ (b)\ c\, \{Q\}} \tag{2}$$

- Implikationen $I \wedge b \Longrightarrow R$, $I \wedge \neg b \Longrightarrow Q$ werden zu **Beweisverpflichtungen**
- Bedingung $I$ (**Invariante**) muss **vorgegeben** werden.

# Übersicht: Regeln für den Hoare-Kalkül Rückwärts

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}} \qquad \frac{}{\vdash \{A\}\, \{\,\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{\vdash \{A_0\}\, c_0\, \{B\} \quad \vdash \{A_1\}\, c_1\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{I \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{I\} \quad I \wedge \neg b \Longrightarrow C}{\vdash \{I\}\ \textbf{while}\ (b)\ c\, \{C\}}$$

# Annotierte Programme

- Wir helfen dem Rechner weiter und **annotieren** die Schleifeninvariante $I$ am Programm.
- Damit berechnen wir:
  - die **approximative** schwächste Vorbedingung $\mathsf{awp}(c, Q)$
  - zusammen mit einer Menge von **Verifikationsbedingungen** $\mathsf{wvc}(c, Q)$
- Die Verifikationsbedingungen treten dort auf, wo die Weakening-Regel angewandt wird.
- Es gilt:

$$\bigwedge \mathsf{wvc}(c, Q) \Longrightarrow \models \{\mathsf{awp}(c, Q)\}\, c\, \{Q\}$$

# Überblick: Approximative schwächste Vorbedingung

$$\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(l = e, P) &\stackrel{def}{=} P[e/l] \qquad\qquad (\textbf{Genauer}\text{: Folie 24 letzte VL}) \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv}\ \ i\ {*}/\ \ c, P) &\stackrel{def}{=} i \\
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(l = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv}\ \ i\ {*}/\ \ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\} \\
\mathrm{wvc}(\{P\}\, c\, \{Q\}) &\stackrel{def}{=} \{P \longrightarrow \mathrm{awp}(c, Q)\} \cup \mathrm{wvc}(c, Q)
\end{aligned}$$

# Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(l = e, P) &\stackrel{def}{=} P[e/l] \qquad\qquad (\textbf{Genauer}\text{: Folie 24 letzte VL}) \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \land \mathrm{awp}(c_0, P)) \lor (\neg b \land \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(l = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \land b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \land \neg b \longrightarrow P\} \\
\mathrm{wvc}(\{P\}\ c\ \{Q\}) &\stackrel{def}{=} \{P \longrightarrow \mathrm{awp}(c, Q)\} \cup \mathrm{wvc}(c, Q)
\end{aligned}
$$

# Berechnung der Verifikationsbedingungen

**Programmkorrektheit**

- Gegeben: Annotiertes Programm $c$ mit Vorbedingung $P$ und Nachbedingung $Q$.
- Gesucht: $\mathrm{wvc}(\{P\}\, c\, \{Q\})$

1. Rekursiv von der Nachbedigung ausgehend berechnen wir für jede Zeile des Programmes die gültige approximative Vorbedingung $\mathrm{awp}(c, -)$.

2. Dabei notieren wir alle auftretenden Verifikationsbedingungen $\mathrm{wvc}(c, -)$

3. Dabei werden **keine** Vereinfachungen vorgenommen.

# Beispiel: das Fakultätsprogramm

- Sei $F$ das annotierte Fakultätsprogramm:

```
// {0 ≤ n}
p= 1;
c= 1;
while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n} */
{ p = p * c;
  c = c + 1;
  }
// {p = n!}
```

- Berechnung der Verifikationsbedingungen zur Nachbedingung $\mathrm{wvc}(\{0 \leq n\}\, F\, \{p = n!\})$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
//
while (c <= n)
  /** inv p= (c−1)! ∧ c−1 ≤ n */ {
  //
  p = p * c;
  //
  c = c + 1;
  //
  }
// {p = n!}
```

WVC wird daneben notiert:

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
//
while (c <= n)
  /** inv p= (c−1)! ∧ c−1 ≤ n */ {
  //
  p = p * c;
  //
  c = c + 1;
  //
  }
// {p = n!}
```

WVC wird daneben notiert:

$$1 \quad \left| \begin{array}{l} p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \\ \quad \longrightarrow p = n! \end{array} \right.$$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
//
while (c <= n)
  /** inv p= (c−1)! ∧ c−1 ≤ n */ {
  //
  p = p * c;
  //
  c = c + 1;
  // {p = (c − 1)! ∧ c − 1 ≤ n}
  }
// {p = n!}
```

WVC wird daneben notiert:

$$1 \;\Big|\; p = (c-1)! \wedge c - 1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
//
while (c <= n)
   /** inv p= (c−1)! ∧ c−1 ≤ n */ {
   //
   p = p * c;
   // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   c = c + 1;
   // {p = (c − 1)! ∧ c − 1 ≤ n}
   }
// {p = n!}
```

WVC wird daneben notiert:

$$1 \;\Big|\; p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
//
while (c <= n)
  /** inv p= (c−1)! ∧ c−1 ≤ n */ {
  // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  p = p * c;
  // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  c = c + 1;
  // {p = (c − 1)! ∧ c − 1 ≤ n}
  }
// {p = n!}
```

WVC wird daneben notiert:

$$1 \;\Big|\; p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \longrightarrow p = n!$$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
//
while (c <= n)
   /** inv p= (c−1)! ∧ c−1 ≤ n */ {
   // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   p = p * c;
   // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   c = c + 1;
   // {p = (c − 1)! ∧ c − 1 ≤ n}
   }
// {p = n!}
```

WVC wird daneben notiert:

$$
\begin{array}{r|l}
1 & p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \\
 & \quad \longrightarrow p = n! \\
2 & p = (c-1)! \land c - 1 \leq n \land c \leq n \\
 & \quad \longrightarrow p \cdot c = ((c+1)-1)! \land \\
 & \qquad\quad (c+1) - 1 \leq n
\end{array}
$$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
//
c= 1;
// {p = (c − 1)! ∧ c − 1 ≤ n}
while (c <= n)
  /** inv p= (c−1)! ∧ c−1 ≤ n */ {
  // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  p = p * c;
  // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  c = c + 1;
  // {p = (c − 1)! ∧ c − 1 ≤ n}
  }
// {p = n!}
```

WVC wird daneben notiert:

1. $p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n)$
   $\longrightarrow p = n!$
2. $p = (c-1)! \land c - 1 \leq n \land c \leq n$
   $\longrightarrow p \cdot c = ((c+1)-1)! \land$
   $(c+1) - 1 \leq n$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
//
p= 1;
// {p = (1 − 1)! ∧ 1 − 1 ≤ n}
c= 1;
// {p = (c − 1)! ∧ c − 1 ≤ n}
while (c <= n)
  /** inv p= (c−1)! ∧ c−1 ≤ n */ {
  // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  p = p * c;
  // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  c = c + 1;
  // {p = (c − 1)! ∧ c − 1 ≤ n}
  }
// {p = n!}
```

WVC wird daneben notiert:

1. $$p = (c-1)! \land c-1 \leq n \land \neg(c \leq n) \longrightarrow p = n!$$
2. $$p = (c-1)! \land c-1 \leq n \land c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \land (c+1)-1 \leq n$$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
// {1 = (1 − 1)! ∧ 1 − 1 ≤ n}
p= 1;
// {p = (1 − 1)! ∧ 1 − 1 ≤ n}
c= 1;
// {p = (c − 1)! ∧ c − 1 ≤ n}
while (c <= n)
   /** inv p= (c−1)! ∧ c−1 ≤ n */ {
   // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   p = p * c;
   // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   c = c + 1;
   // {p = (c − 1)! ∧ c − 1 ≤ n}
   }
// {p = n!}
```

WVC wird daneben notiert:

1. $p = (c-1)! \wedge c - 1 \leq n \wedge \neg(c \leq n)$
   $\longrightarrow p = n!$
2. $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n$
   $\longrightarrow p \cdot c = ((c+1)-1)! \wedge$
   $(c+1) - 1 \leq n$

# Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
// {0 ≤ n}
// {1 = (1 − 1)! ∧ 1 − 1 ≤ n}
p= 1;
// {p = (1 − 1)! ∧ 1 − 1 ≤ n}
c= 1;
// {p = (c − 1)! ∧ c − 1 ≤ n}
while (c <= n)
   /** inv p= (c−1)! ∧ c−1 ≤ n */ {
   // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   p = p * c;
   // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
   c = c + 1;
   // {p = (c − 1)! ∧ c − 1 ≤ n}
   }
// {p = n!}
```

WVC wird daneben notiert:

1. $p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \longrightarrow p = n!$
2. $p = (c-1)! \land c - 1 \leq n \land c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \land (c+1) - 1 \leq n$
3. $0 \leq n \longrightarrow 1 = (1-1)! \land (1-1) \leq n$

# Vereinfachung von Verifikationsbedingungen

Wir nehmen folgende **strukturelle Vereinfachungen** vor:

1. Konjunktionen in der Konklusion werden zu einzelnen Verifikationsbedingungen
   - Bsp: $A_1 \wedge A_2 \wedge A_3 \longrightarrow P \wedge Q \rightsquigarrow A_1 \wedge A_2 \wedge A_3 \longrightarrow P, A_1 \wedge A_2 \wedge A_3 \longrightarrow Q$
2. Auswertung konstanter arithmetischer Ausdrücke, einfache arithmetische Gesetze
   - Bsp. $(x + 1) - 1 \rightsquigarrow x$, $1 - 1 \rightsquigarrow 0$
3. Normalisierung der Relationen (zu $<$, $\leq$, $=$, $\neq$) und Vereinfachung
   - Bsp: $\neg(x \leq y) \rightsquigarrow x > y \rightsquigarrow y < x$, $x \leq x \rightsquigarrow \mathit{true}$, $4 \leq 5 \rightsquigarrow \mathit{true}$
4. Alle Bedingungen mit einer Prämisse *false* oder einer Konklusion *true* sind trivial erfüllt.

# Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\leadsto p = (c-1)! \wedge c - 1 \leq n \wedge n < c \longrightarrow p = n!$

Es bleibt zu zeigen:

# Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge n < c \longrightarrow p = n!$
- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1) - 1 \leq n$
  $\rightsquigarrow \quad p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$

Es bleibt zu zeigen:

# Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c-1 \leq n \wedge n < c \longrightarrow p = n!$
- 2: $p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1)-1 \leq n$
  $\rightsquigarrow \quad p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  $p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$
- 3: $0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n$
  $\rightsquigarrow \quad 0 \leq n \longrightarrow 1 = 0!$
  $0 \leq n \longrightarrow 0 \leq n \rightsquigarrow \mathit{true}$

Es bleibt zu zeigen:

# Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge n < c \longrightarrow p = n!$
- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1) - 1 \leq n$
  $\rightsquigarrow \quad p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  $\quad p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$
- 3: $0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n$
  $\rightsquigarrow \quad 0 \leq n \longrightarrow 1 = 0!$
  $\quad 0 \leq n \longrightarrow 0 \leq n \rightsquigarrow \mathit{true}$

Es bleibt zu zeigen:

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge n < c \longrightarrow p = n!$
  Aus $n > c$ folgt $n \geq c - 1$, also $c - 1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.

# Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c-1 \leq n \wedge n < c \longrightarrow p = n!$
- 2: $p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1)-1 \leq n$
  $\rightsquigarrow \quad p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  $\quad p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$
- 3: $0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n$
  $\rightsquigarrow \quad 0 \leq n \longrightarrow 1 = 0!$
  $\quad 0 \leq n \longrightarrow 0 \leq n \rightsquigarrow \mathit{true}$

Es bleibt zu zeigen:

- 1: $p = (c-1)! \wedge c-1 \leq n \wedge n < c \longrightarrow p = n!$
  Aus $n > c$ folgt $n \geq c-1$, also $c-1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.
- 2: $p = (c-1)! \wedge c-1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  Aus $p = (c-1)!$ folgt $p \cdot c = c \cdot (c-1)!$, und mit $c \cdot (c-1)! = c!$ folgt die Behauptung.

# Vereinfachung am Beispiel

- 1: $p = (c-1)! \land c-1 \leq n \land \neg(c \leq n) \longrightarrow p = n!$
  $\leadsto p = (c-1)! \land c-1 \leq n \land n < c \longrightarrow p = n!$
- 2: $p = (c-1)! \land c-1 \leq n \land c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \land (c+1)-1 \leq n$
  $\leadsto \quad p = (c-1)! \land c-1 \leq n \land c \leq n \longrightarrow p \cdot c = c!$
  $\quad p = (c-1)! \land c-1 \leq n \land c \leq n \longrightarrow c \leq n \leadsto \mathit{true}$
- 3: $0 \leq n \longrightarrow 1 = (1-1)! \land (1-1) \leq n$
  $\leadsto \quad 0 \leq n \longrightarrow 1 = 0!$
  $\quad 0 \leq n \longrightarrow 0 \leq n \leadsto \mathit{true}$

Es bleibt zu zeigen:

- 1: $p = (c-1)! \land c-1 \leq n \land n < c \longrightarrow p = n!$
  Aus $n > c$ folgt $n \geq c-1$, also $c-1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.
- 2: $p = (c-1)! \land c-1 \leq n \land c \leq n \longrightarrow p \cdot c = c!$
  Aus $p = (c-1)!$ folgt $p \cdot c = c \cdot (c-1)!$, und mit $c \cdot (c-1)! = c!$ folgt die Behauptung.
- 3: $1 = 0!$ folgt direkt aus der Definition der Fakultät.

# Arbeitsblatt 9.2: Da summt was. . .

```
// {0 ≤ n ∧ n = N}
p= 0;
while (n>0) //** inv p= sum(n+1,N); */
{ p = p + n;
  n = n - 1;
  }
// {p = sum(1, N)}
```

1. Berechnet zuerst die **unvereinfachten** VCs (für sind die AWPs nötig)
2. Danach vereinfacht die VCs **schematisch** wie oben beschrieben.
3. Welche VCs sind beweisbar?

Dabei gilt: $sum(i,j) = \begin{cases} 0 & i > j \\ i + sum(i+1,j) & i \leq j \end{cases}$

# Weiteres Beispiel: Maximales Element

```
// {0 < n}
i= 0;
r= 0;
```

$$\texttt{while (i!= n) //** inv } \{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}^{\varphi(i,r)}\} \texttt{ */}$$

```
{ if (a[r] < a[i]) {
    r= i;
  }
  else {
  }
  i= i+1;
  }
```

$$//\ \{\underbrace{(\forall j.\, 0 \leq j < \boxed{n} \longrightarrow a[j] \leq a[\boxed{r}]) \wedge 0 \leq r < \boxed{n}}_{\varphi(\boxed{n},\,\boxed{r})}\}$$

# Maximales Element (Schleifenrumpf)

```
while (i != n)
```

$$\overbrace{(\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \land 0 \le r < i}^{\varphi(i,r)}$$

```
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    //
  }
  //
  i= i+1;
  //
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[ r ]) ∧ 0 ≤ r < n }
```

$$\underbrace{(\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \land 0 \le r < n}_{\varphi(n,\, r)}$$

VC:

# Maximales Element (Schleifenrumpf)

```
while (i!= n)
```

$\overbrace{\{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r < i\}}^{\varphi(i,r)}$

```
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    //
  }
  //
  i= i+1;
  // {φ(i, r)}
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$\underbrace{\{(\forall j.\, 0 \leq j < \boxed{n} \longrightarrow a[j] \leq a[\boxed{r}]) \land 0 \leq r < \boxed{n}\}}_{\varphi(\boxed{n},\boxed{r})}$

VC:

1 | $\varphi(i, r) \land \neg(i \neq n) \longrightarrow \varphi(n, r)$

# Maximales Element (Schleifenrumpf)

```
while (i != n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    //
  }
  // {φ(i+1, r)}
  i= i+1;
  // {φ(i, r)}
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$\underbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}_{\varphi(i,r)}$$

$$\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n}_{\varphi(n,r)}$$

VC:

1 $\varphi(i, r) \wedge \neg(i \neq n) \longrightarrow \varphi(n, r)$

# Maximales Element (Schleifenrumpf)

```
while (i!= n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    // {φ(i+1, r)}
  }
  // {φ(i+1, r)}
  i= i+1;
  // {φ(i, r)}
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$\overbrace{\{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i\}}^{\varphi(i,r)}$$

$$\underbrace{\{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n\}}_{\varphi(n,r)}$$

VC:

1 | $\varphi(i, r) \wedge \neg(i \neq n) \longrightarrow \varphi(n, r)$

# Maximales Element (Schleifenrumpf)

```
while (i!= n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    // {φ(i+1, r)}
    }
  else {
    // {φ(i+1, r)}
  }
  // {φ(i+1, r)}
  i= i+1;
  // {φ(i, r)}
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}^{\varphi(i,r)}$

$\underbrace{(\forall j.\, 0 \leq j < \boxed{n} \longrightarrow a[j] \leq a[\boxed{r}]) \wedge 0 \leq r < \boxed{n}}_{\varphi(\boxed{n},\boxed{r})}$

VC:

1 | $\varphi(i,r) \wedge \neg(i \neq n) \longrightarrow \varphi(n,r)$

# Maximales Element (Schleifenrumpf)

```
while (i!= n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    // {φ(i+1, i)}
    r= i;
    // {φ(i+1, r)}
    }
  else {
    // {φ(i+1, r)}
  }
  // {φ(i+1, r)}
  i= i+1;
  // {φ(i, r)}
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Invariant: $\overbrace{(\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i}^{\varphi(i,r)}$

Postcondition: $\underbrace{(\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n}_{\varphi(n,r)}$

VC:

1 $\varphi(i, r) \wedge \neg(i \neq n) \longrightarrow \varphi(n, r)$

# Maximales Element (Schleifenrumpf)

```
while (i!= n)
```

$$/**\ \ \textbf{inv}\ \ \overbrace{\{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r < i\}}^{\varphi(i,r)}\ \ */$$

```
{
  // {(a[r] < a[i] ∧ φ(i+1, i)) ∨ (¬(a[r] < a[i]) ∧ φ(i+1, r))}
  if (a[r] < a[i]) {
     // {φ(i+1, i)}
     r= i;
     // {φ(i+1, r)}
     }
  else {
     // {φ(i+1, r)}
  }
  // {φ(i+1, r)}
  i= i+1;
  // {φ(i, r)}
  }
```

$$//\ \underbrace{\{(\forall j.\, 0 \leq j < \boxed{n} \longrightarrow a[j] \leq a[\boxed{r}]) \land 0 \leq r < \boxed{n}\}}_{\varphi(\boxed{n},\boxed{r})}$$

VC:

$$1 \;\Big|\; \varphi(i,r) \land \neg(i \neq n) \longrightarrow \varphi(n,r)$$

# Maximales Element (Schleifenrumpf)

`while (i!= n)`<br>
`/** inv` $\overbrace{\{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i\}}^{\varphi(i,r)}$ `*/`<br>
`{`<br>
  `//` $\{(a[r] < a[i] \wedge \varphi(i+1, i)) \vee (\neg(a[r] < a[i]) \wedge \varphi(i+1, r))\}$<br>
  `if (a[r] < a[i]) {`<br>
    `//` $\{\varphi(i+1, i)\}$<br>
    `r= i;`<br>
    `//` $\{\varphi(i+1, r)\}$<br>
    `}`<br>
  `else {`<br>
    `//` $\{\varphi(i+1, r)\}$<br>
  `}`<br>
  `//` $\{\varphi(i+1, r)\}$<br>
  `i= i+1;`<br>
  `//` $\{\varphi(i, r)\}$<br>
  `}`<br>
`//` $\underbrace{\{(\forall j.\, 0 \leq j < \boxed{n} \longrightarrow a[j] \leq a[\boxed{r}]) \wedge 0 \leq r < \boxed{n}\}}_{\varphi(\boxed{n}, \boxed{r})}$

VC:

$$
\begin{array}{l|l}
1 & \varphi(i, r) \wedge \neg(i \neq n) \longrightarrow \varphi(n, r) \\
2 & \varphi(i, r) \wedge i \neq n \longrightarrow \\
 & \quad (a[r] < a[i] \wedge \varphi(i+1, i)) \\
 & \quad \vee \\
 & \quad (\neg(a[r] < a[i]) \wedge \varphi(i+1, r))
\end{array}
$$

# Maximales Element (Initialisierung)

```
// {0 < n}
//
i= 0;
//
r= 0;
// {φ(i,r)}
while (i!= n)
```

$$\text{/** inv } \underbrace{\{(\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i\}}^{\varphi(i,r)} \text{ */}$$

VC:

| | |
|---|---|
| 1 | $\varphi(i,r) \wedge \neg(i \neq n) \longrightarrow \varphi(n,r)$ |
| 2 | $\varphi(i,r) \wedge i \neq n \longrightarrow$ |
| | $(a[r] < a[i] \wedge \varphi(i+1,i))$ |
| | $\vee$ |
| | $(\neg(a[r] < a[i]) \wedge \varphi(i+1,r))$ |

# Maximales Element (Initialisierung)

```
// {0 < n}
//
i= 0;
// {φ(i,0)}
r= 0;
// {φ(i,r)}
while (i!= n)
```

$$/** \ \mathbf{inv} \ \{\underbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}_{\varphi(i,r)}\} \ */$$

VC:

1. $\varphi(i,r) \wedge \neg(i \neq n) \longrightarrow \varphi(n,r)$
2. $\varphi(i,r) \wedge i \neq n \longrightarrow$
   $(a[r] < a[i] \wedge \varphi(i+1,i))$
   $\vee$
   $(\neg(a[r] < a[i]) \wedge \varphi(i+1,r))$

# Maximales Element (Initialisierung)

```
// {0 < n}
// {φ(0,0)}
i= 0;
// {φ(i,0)}
r= 0;
// {φ(i,r)}
while (i!= n)
```

$$\texttt{/** inv } \{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r < i}^{\varphi(i,r)}\} \texttt{ */}$$

VC:

| | |
|---|---|
| 1 | $\varphi(i,r) \land \neg(i \neq n) \longrightarrow \varphi(n,r)$ |
| 2 | $\varphi(i,r) \land i \neq n \longrightarrow$ <br> $(a[r] < a[i] \land \varphi(i+1,i))$ <br> $\lor$ <br> $(\neg(a[r] < a[i]) \land \varphi(i+1,r))$ |

# Maximales Element (Initialisierung)

```
// {0 < n}
// {φ(0,0)}
i= 0;
// {φ(i,0)}
r= 0;
// {φ(i,r)}
while (i!= n)
```

$$/** \ \textbf{inv} \ \{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}^{\varphi(i,r)}\} \ */$$

VC:

1 $\varphi(i,r) \wedge \neg(i \neq n) \longrightarrow \varphi(n,r)$

2 $\varphi(i,r) \wedge i \neq n \longrightarrow$
$(a[r] < a[i] \wedge \varphi(i+1, i))$
$\vee$
$(\neg(a[r] < a[i]) \wedge \varphi(i+1, r))$

3 $0 \leq n \longrightarrow \varphi(0,0)$

# Maximales Element (Verifikationsbedingungen)

Unvereinfacht:

1 $(\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge \neg(i \neq n) \longrightarrow$
$(\forall j. 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq n$

2 $(\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge i \neq n \longrightarrow$
$((a[r] < a[i] \wedge (\forall j. 0 \leq j < i + 1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i \leq i + 1) \vee$
$(\neg(a[r] < a[i]) \wedge (\forall j. 0 \leq j < i + 1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i + 1))$

3 $0 \leq n \longrightarrow (\forall j. 0 \leq j < 0 \longrightarrow a[j] \leq a[0]) \wedge 0 \leq 0 \leq 0$

- Sehr **lange** Verifikationsbedingungen (u.a. wegen Fallunterscheidung)
  - Insbesondere schwer zu **vereinfachen**
- Wie können wir das **beheben**?

# Maximales Element (Verifikationsbedingungen)

Vereinfacht:

$$
\begin{array}{ll}
1.1 & (\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r \leq i \land i = n \longrightarrow \forall j. 0 \leq j < n \longrightarrow a[j] \leq a[r] \\
1.2 & (\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r \leq i \land i = n \longrightarrow 0 \leq r \leq n \\
2 & (\forall j. 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r \leq i \land i \neq n \longrightarrow \\
 & \quad ((a[r] < a[i] \land (\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \land 0 \leq i \leq i+1) \lor \\
 & \quad\; (\neg(a[i] \leq a[r]) \land (\forall j. 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \land 0 \leq r \leq i+1)) \\
3.1 & 0 \leq n \longrightarrow \forall j. 0 \leq j < 0 \longrightarrow a[j] \leq a[0] \\
3.2 & 0 \leq n \longrightarrow 0 \leq 0 \leq 0
\end{array}
$$

- Sehr **lange** Verifikationsbedingungen (u.a. wegen Fallunterscheidung)
  - Insbesondere schwer zu **vereinfachen**
- Wie können wir das **beheben**?

# Explizite Vorbedingungen

Lange Vorbedingung:

```
// {(P₁ ∧ b) ∨ (P₂ ∧ ¬b)}
if (b) {
  // {P₁}
  ...
  // {Q}
  } else {
  // {P₂}
  ...
  // {Q}
  }
```

Kurze Vorbedingung:

```
// {A}
if (b) {
  // {A ∧ b}
  ...
  // {Q}
  } else {
  // {A ∧ ¬b}
  ...
  // {Q}
  }
```

Dazu VCs:

$$A \wedge b \longrightarrow P_1$$
$$A \wedge \neg b \longrightarrow P_2$$

# Spracherweiterung: Explizite Spezifikationen

- Erweiterung der Sprache C0 um Invarianten für Schleifen und **explizite Zusicherung**

$$
\begin{array}{lll}
\textbf{Assn} & a ::= \ldots & \text{— Zusicherungen} \\
\textbf{Stmt} & c ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
& \mid & \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ a\ {*}/\ c \\
& \mid & /{**}\ \{a\}\ {*}/
\end{array}
$$

- Zusicherungen haben **keine Semantik** (Kommentar!), sondern erzwingen eine neue Vorbedingung.
- Dazu vereinfachte Regel für Fallunterscheidung:

$$
\text{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) \stackrel{def}{=} (b \wedge \text{awp}(c_0, P)) \vee (\neg b \wedge \text{awp}(c_1, P))
$$

Wenn $\text{awp}(c_0, P) = b \wedge P_0$, $\text{awp}(c_1, P) = \neg b \wedge P_0$, dann gilt

$$
(b \wedge b \wedge P_0) \vee (\neg b \wedge \neg b \wedge P_0) = (b \wedge P_0) \vee (\neg b \wedge P_0) = (b \vee \neg b) \wedge P_0 = P_0
$$

# Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(l = e, P) &\stackrel{def}{=} P[e/l] \qquad \text{(\textbf{Genauer}: Folie 24 letzte VL)} \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0 \ \textbf{else } \ c_1, P) &\stackrel{def}{=} Q \quad \text{wenn } \ \mathrm{awp}(c_0, P) = b \wedge Q, \mathrm{awp}(c_1, P) = \neg b \wedge Q \\
\mathrm{awp}(\textbf{if } (b)\ c_0 \ \textbf{else } \ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \mathrm{q} \\
\mathrm{awp}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv } \ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(l = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if } (b)\ c_0 \ \textbf{else } \ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \{q \longrightarrow P\} \\
\mathrm{wvc}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv } \ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}
$$

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
//
while (i!= n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
     //
     r= i;
     //
     }
 else {
     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
     //
     }
  //
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
//
while (i!= n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
    //
    r= i;
    //
    }
 else {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
    //
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
//
while (i!= n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
    //
    r= i;
    //
    }
 else {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
//
while (i != n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
    //
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
 else {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
//
while (i!= n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  //
  if (a[r] < a[i]) {
     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < i + 1}
     r= i;
     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
     }
 else {
     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
     }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
//
while (i != n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i}
  if (a[r] < a[i]) {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < i + 1}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
 else {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
//
i= 0;
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i}
while (i != n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i}
  if (a[r] < a[i]) {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < i + 1}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
 else {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung

```
// {0 < n}
// {(∀j. 0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 < 0}
i= 0;
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i}
while (i != n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
{
  // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i}
  if (a[r] < a[i]) {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < i + 1}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
 else {
    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i + 1}
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Maximales Element mit Zusicherung: Beweisverpflichtungen

Unvereinfacht:

$$
\begin{array}{ll}
(1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge \neg(i \neq n) \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \\
(2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge \neg(a[r] < a[i]) \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i+1 \\
(3) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[r] < a[i] \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq r < i+1 \\
(4) & 0 < n \longrightarrow (\forall j.\, 0 \leq j < 0 \longrightarrow a[j] \leq a[0]) \wedge 0 \leq 0 < n
\end{array}
$$

# Maximales Element mit Zusicherung: Beweisverpflichtungen

Vereinfacht:

$$
\begin{array}{ll}
(1.1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge i = n \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \\
(1.2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge i = n \longrightarrow 0 \leq r < n \\
(2.1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[i] \leq a[r] \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \\
(2.2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[i] \leq a[r] \longrightarrow 0 \leq r < i+1 \\
(3.1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[r] < a[i] \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \\
(3.2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[r] < a[i] \longrightarrow 0 \leq r < i+1 \\
(4.1) & 0 < n \longrightarrow (\forall j.\, 0 \leq j < 0 \longrightarrow a[j] \leq a[0]) \\
(4.2) & 0 < n \longrightarrow 0 \leq 0 < n
\end{array}
$$

# Beweismethoden

- Um $P_1 \wedge \ldots \wedge P_n \longrightarrow Q$ zu zeigen, nehmen wir $P_1, \ldots, P_n$ an und zeigen $Q$.
- Dabei nutzen wir **u.a.** folgende Regeln:

Wenn $P$, dann $P$ (Trivial)
Wenn $P$ und $x = t$, dann $P[t/x]$ (Subst)
$x \leq x$ (Reflexität)
Wenn $x \leq y$ und $y \leq z$, dann $x \leq z$ (Transitivität)
Wenn $x \leq y$ und $y \leq x$, dann $x = y$ (Antisymmetrie)
Wenn $x < y$, dann $x \leq y + 1$ oder $x + 1 \leq y$ (Inc)
Wenn $\forall x.\, P$, dann $P[t/x]$ (Instantiierung)
Wenn *false*, dann $P$ (Ex falso)
Wenn $a \leq b$ und $x \leq y$, dann $a + x \leq b + y$ und Variation mit $x = 0$ etc.
Umformungen mit $(0, +)$ und $(1, \cdot)$
Domänenspezifische Regeln

# Arbeitsblatt 9.3: Beweisverpflichtungen Beweisen

Betrachtet die vereinfachten Verifikationsbedingungen:

$$
\begin{array}{ll}
(1.1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge i = n \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \\
(1.2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge i = n \longrightarrow 0 \leq r < n \\
(2.1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[i] \leq a[r] \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \\
(2.2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[i] \leq a[r] \longrightarrow 0 \leq r < i+1 \\
(3.1) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[r] < a[i] \\
 & \quad \longrightarrow (\forall j.\, 0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \\
(3.2) & (\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i \wedge a[r] < a[i] \longrightarrow 0 \leq r < i+1 \\
(4.1) & 0 < n \longrightarrow (\forall j.\, 0 \leq j < 0 \longrightarrow a[j] \leq a[0]) \\
(4.2) & 0 < n \longrightarrow 0 \leq 0 < n
\end{array}
$$

Wie würdet ihr sie beweisen? Was für Methoden verwendet ihr?

# Arbeitsblatt 9.4: Kopien

Dieses Programm kopiert ein Array:

```
i= 0;
while (i< m)
  /** inv ??? */ {
  b[m-1-i]= a[i];
  i= i+1;
  }
```

1. Spezifiziert die Funktionalität.
2. Findet die Invariante.
3. Berechnet die Verifikationsbedingungen (VCs) und schwächste Vorbedingung.
4. Beweist die VCs.

# Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls lassen sich, weitgehend schematisch, rückwärts (vom Ende her) anwenden — nur Schleifen machen Probleme.
- Wir **annotieren** daher die Invarianten an Schleifen, und können dann die schwächste Vorbedingung und Verifikationsbedingungen automatisch berechnen.
  - Dabei sind die **Verifikationsbedingungen** das Interessante.
- Um die Verifikationsbedingungen zu vereinfachen führen wir **explizite Zusicherungen** in C0 ein
- Die Generierung von Verifikationsbedingungen korrespondiert zur relativen Vollständigkeit der Floyd-Hoare-Logik.

# Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls lassen sich, weitgehend schematisch, rückwärts (vom Ende her) anwenden — nur Schleifen machen Probleme.
- Wir **annotieren** daher die Invarianten an Schleifen, und können dann die schwächste Vorbedingung und Verifikationsbedingungen automatisch berechnen.
  - Dabei sind die **Verifikationsbedingungen** das Interessante.
- Um die Verifikationsbedingungen zu vereinfachen führen wir **explizite Zusicherungen** in C0 ein
- Die Generierung von Verifikationsbedingungen korrespondiert zur relativen Vollständigkeit der Floyd-Hoare-Logik.
- Warum eigentlich immer **rückwärts**?

# Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls lassen sich, weitgehend schematisch, rückwärts (vom Ende her) anwenden — nur Schleifen machen Probleme.
- Wir **annotieren** daher die Invarianten an Schleifen, und können dann die schwächste Vorbedingung und Verifikationsbedingungen automatisch berechnen.
  - Dabei sind die **Verifikationsbedingungen** das Interessante.
- Um die Verifikationsbedingungen zu vereinfachen führen wir **explizite Zusicherungen** in C0 ein
- Die Generierung von Verifikationsbedingungen korrespondiert zur relativen Vollständigkeit der Floyd-Hoare-Logik.
- Warum eigentlich immer **rückwärts**?
  Jetzt gleich. . .