

**5.1 Fehlerquellen**

**5.2 Verifikation**

**5.3 wp-Kalkül**

**5.4 Schleifeninvariante**



## Innensicht

### **Imperatives Programmieren**

#### (Konstrukte aus Java)

- Praxis: Basistypen, Anweisungen, Verbunde, Felder, Schleifen, Rekursion
- Theorie: Syntaxbeschreibung, Induktion, Schleifeninvarianten
- Ausnahmebehandlung

### **Objektorientiertes Programmieren**

#### (Java)

- OO-Klassen
- Objekt und Zustände
- Methoden und Vererbung in Java
- Java: Einführung aller restlicher OO-Sprachkonstrukte

## Außensicht

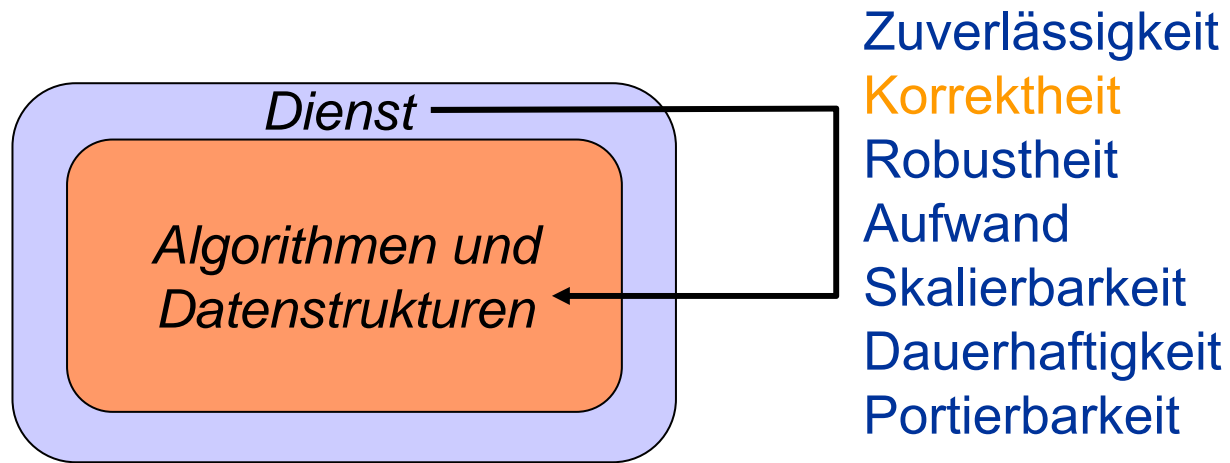
### **Dienste**

- Funktionalität/Schnittstellen
- Qualitätsparameter: Aufwand, Zuverlässigkeit, Skalierbarkeit, Persistenz
- Algorithmenwahl

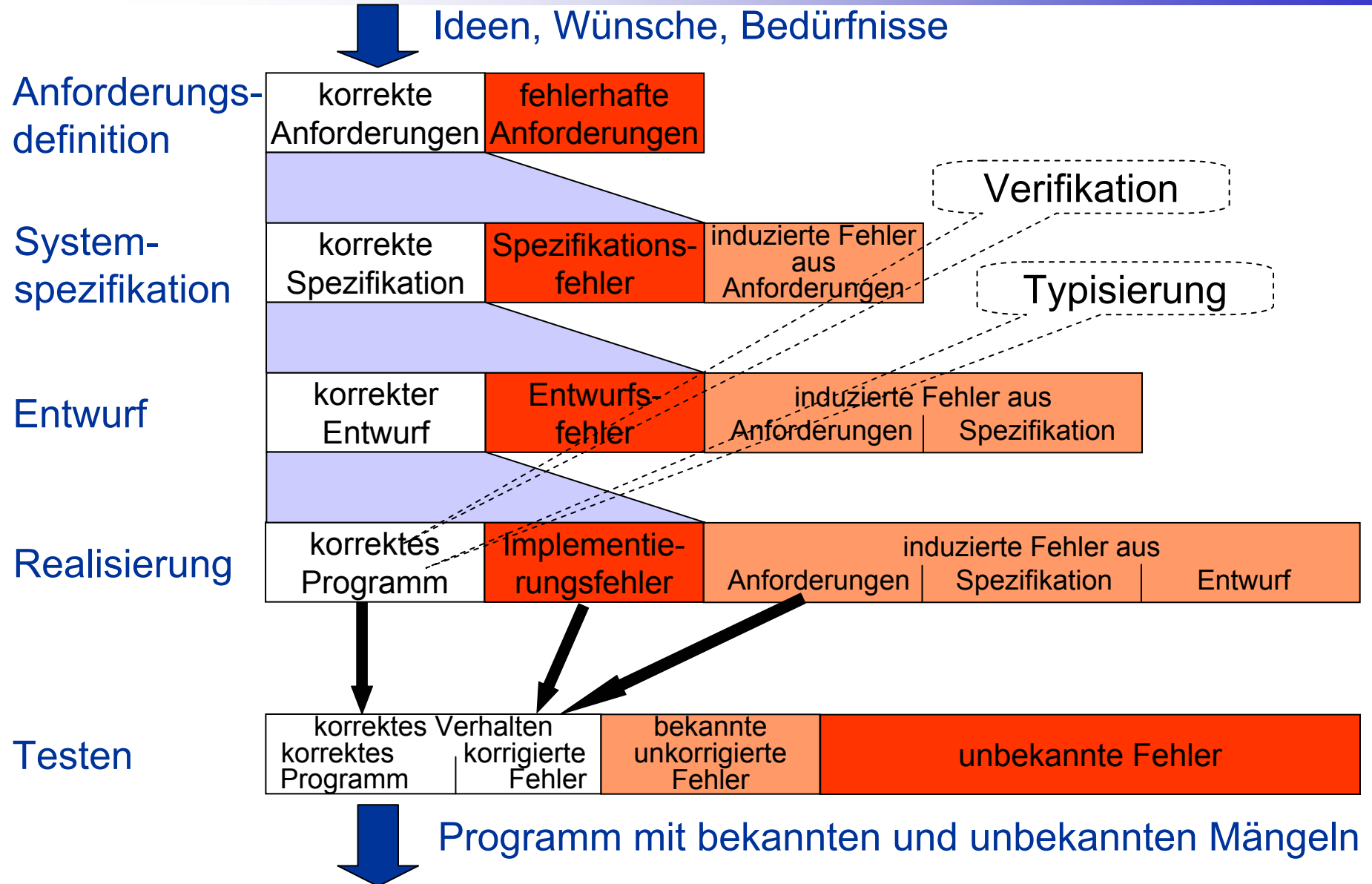
### **Objektorientierung**

- Objekte, Zustände, Methoden
- Entwurf mittels UML
- Vererbung





# 5.1 Fehlerquellen



- Regel: Je später ein Fehler in der Programmentwicklung aufgedeckt wird, umso (vielfach) teurer wird seine Behebung.
- Daher Ziel: Feststellung der Richtigkeit eines Programms schon während des Aufschreibens

## Ansatz Verifikation:

- Mathematischer Beweis, dass (formale) Spezifikation erfüllt wird.
- Beachte: Entdeckung von Anforderungsfehlern im Allgemeinen nicht möglich.

## Wirtschaftlichkeitskonflikt:

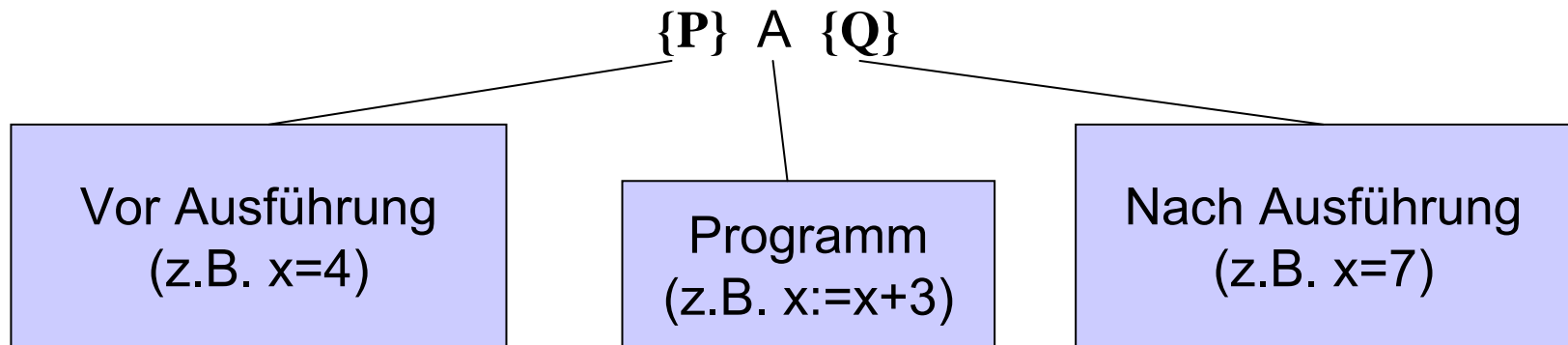
Formale Fehlerüberprüfung während des Aufschreibens ist meist sehr aufwändig (wenn nicht z.T. sogar unentscheidbar), daher muss man Aufwand und späteren Gewinn gegeneinander abwägen.



Im folgenden wird die Korrektheit *imperativer Programme* untersucht. Aber was heißt Korrektheit?

## Zunächst

- Spezifikation gewünschter Eigenschaften eines Programms A. Angabe einer zustandsbezogenen Vorbedingung **P** und einer zustandsbezogenen Nachbedingung **Q** zum Programm A.



- Bedeutung
  - Wenn **P** vor der Ausführung von A gilt, so gilt **Q** nach der Ausführung von A. (Exakter: wenn dies beweisbar ist)

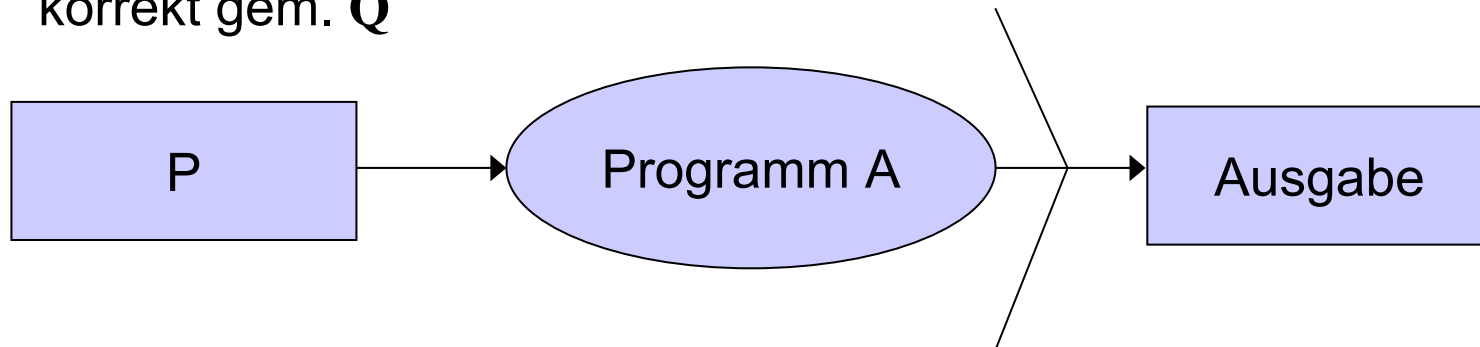


Ein Programm  $A$  heißt *partiell korrekt* bezüglich  $P$  und  $Q$  gdw.  
 $\{P\} A \{Q\}$  formal beweisbar ist (d.h. „wahr“ ist).

- Diese Definition verlangt nicht, dass  $A$  terminiert.

Ein Programm  $A$  heißt *total korrekt* bezüglich  $P$  und  $Q$  gdw.  
 $A$  partiell korrekt ist und immer dann terminiert, wenn die  
 Vorbedingung  $P$  gilt.

- Partielle Korrektheit: falls dieser Punkt erreicht wird, dann ist dies korrekt gem.  $Q$



- Totale Korrektheit: dieser Punkt wird erreicht und dann ist dies korrekt gem.  $Q$

## Testen

- Testeingabe und erwartete Ergebnisse zusammenstellen
- Programm laufen lassen und Ergebnis mit Erwartung vergleichen
- Korrektheit kann nicht garantiert werden
- Anforderungsfehler und Entwurfsfehler können entdeckt werden

## Verfeinerung

- Programm wird Schritt für Schritt gemäß Spezifikation entwickelt, so dass es zu jedem Zeitpunkt in jeder Version korrekt ist.

## Verifikation

- Mathematischer Beweis, dass (formale) Spezifikation erfüllt ist.
- Entdeckung von Anforderungsfehlern nicht möglich.



# Edsger W. Dijkstra



„Testing can be used to show the Presence of bugs, never their absence.“

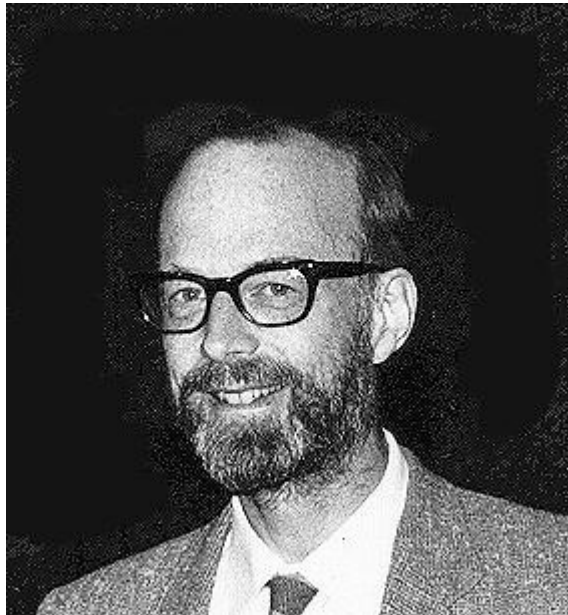
- \* 11. Mai 1930 in Rotterdam
- † 6. August 2002 in Neunen (Niederlande)
- Programmierer am „Mathematisch Centrum“, Amsterdam
- Mathematikprofessor an der T.H. Eindhoven,
- Schlumberger Centennial Chair in Computer Sciences an der Universität von Texas in Austin
- Wichtige Beiträge zur Informatik
  - Berechnung des kürzesten Weges in einem Graphen
  - Abhandlungen über den `goto`-Befehl und warum er nicht benutzt werden soll.
- Im Jahre 1972 erhielt Dijkstra den Turing-Preis.



<http://www.net-lexikon.de/Edsger-Dijkstra.html>



# Sir Tony Hoare



- Studium: Philosophie, Latein, Griechisch, Mathematik
- Programmierer bei Elliot Brothers
- Professor für Informatik Queens University
- Professor für Informatik Oxford University
- Arbeitet heute bei Microsoft Research in Cambridge
- Forschung: Beweisen von Programmen
- Wichtige Beiträge zur Informatik
  - Quicksort
  - Entwicklung des ersten Compilers für „Algol 60“
  - Entwicklung der Spezifizierungs-Sprache „Z“
  - Programmiermodell „CSP“
- Im Jahre 2000 von der Queen zum Ritter geschlagen, „for services to Computing Science“.



<http://www.research.microsoft.com/%7Ethoare/>



### Beschreibung von Daten:

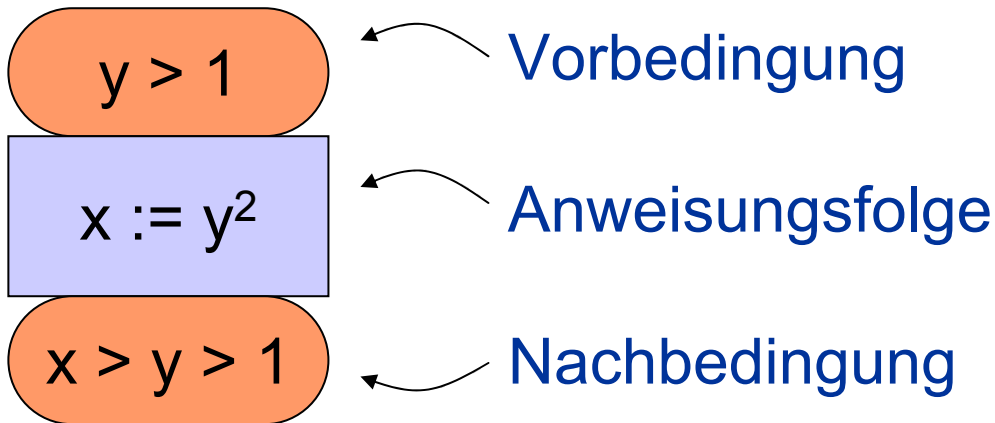
- Bei einer formalen Vorgehensweise zum Beweis der Korrektheit müssen alle möglichen Eingabedaten betrachtet werden, und es muss überprüft werden, ob die gewünschte Ausgabe immer erreicht wird.
- Statt einer Aufzählung dieser Eingabewerte bzw. Ausgabewerte wird eine Beschreibung derselben durch Prädikate verwendet.  
z.B.  $P(v) ::= \text{„Variable } v \text{ liegt im Intervall } [a, b]\text{“}$   
z.B.  $P(v) ::= a \leq v \leq b$

Allgemein: Prädikat  $P(z)$  mit  $z ::= \text{Zustand} = \text{Belegung aller interessierenden Variablen.}$



- Die Wirkung eines Programms oder Codestücks kann durch die Angabe von Prädikaten zu Beginn und am Ende des Codes spezifiziert werden.
- Zusicherungen sind prädikatenlogische Aussagen über die Werte der Programmvariablen an den Stellen im Programm, an denen die jeweiligen Zusicherungen stehen.
- Eine Zusicherung vor einer bestimmten Anweisungsfolge wird in Bezug auf diese Vorbedingung (precondition) genannt.
- Entsprechend ist eine Nachbedingung (postcondition) eine Zusicherung nach einer Anweisungsfolge.
- Vorbedingung und Nachbedingung des gesamten Programms bilden eine prädikatenlogische Spezifikation.





**Aussage:**  
Wenn vor dem Code  $y > 1$  wahr ist, dann ist nach Ausführung des Codes  $x > y > 1$  wahr.



Eine Verifikation prüft, ob ein Programm bestimmte Zusicherungen erfüllt, d.h. ob die Nachbedingung **Q** einer Anweisungsfolge **A** nach Abarbeitung der Anweisungen aus der Vorbedingung **P** ableitbar ist.

Partielle Korrektheit: Wenn die Ausführung von **A** in einem Zustand, der **P** genügt, beginnt, und falls die Ausführung von **A** nach endlich vielen Schritten terminiert, dann erfüllt der erreichte Zustand die Nachbedingung **Q**.

- Schreibweise:  $P \{A\} Q$

Totale Korrektheit verlangt außerdem den Beweis der Terminierung.

- Schreibweise:  $\{P\} A \{Q\}$



Beweis der Terminierung kann schwierig sein.

Terminiert das folgende Programm oder nicht?

$x = 2$

```
while (x == Summe von 2 Primzahlen) x += 2;
```

$x > 2$

Auf das Halteproblem wird in Informatik III eingegangen.



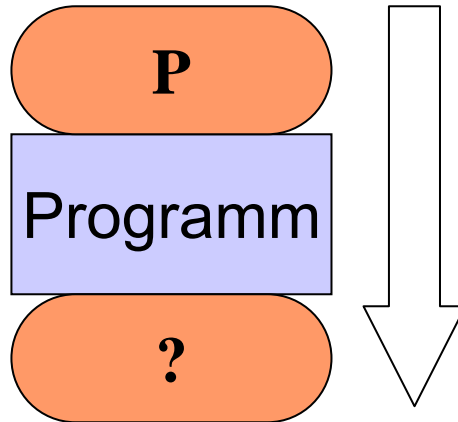
- Wir haben es nicht mit einem Zustand zu tun, über den wir Aussagen machen wollen, sondern mit einer Folge von Zuständen  $z_0, z_1, \dots, z_n$  während des Ablaufs eines Programms.
- Da imperative Programme lineare Kontrollstrukturen haben, folgt aus der Korrektheit von Teilen die Korrektheit der Sequenz.

## Verifikationskalküle

- verwenden (rein syntaktische) Regelwerke zur Ableitung zu verifizierender Aussagen: Regeln sind rein mechanisch anwendbar;
- können korrekt sein: alles, was abgeleitet werden kann, ist wahr;
- können vollständig sein: alles, was wahr ist, kann abgeleitet werden.

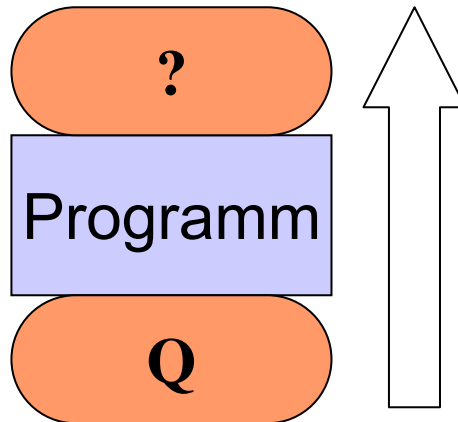


Vorwärts:

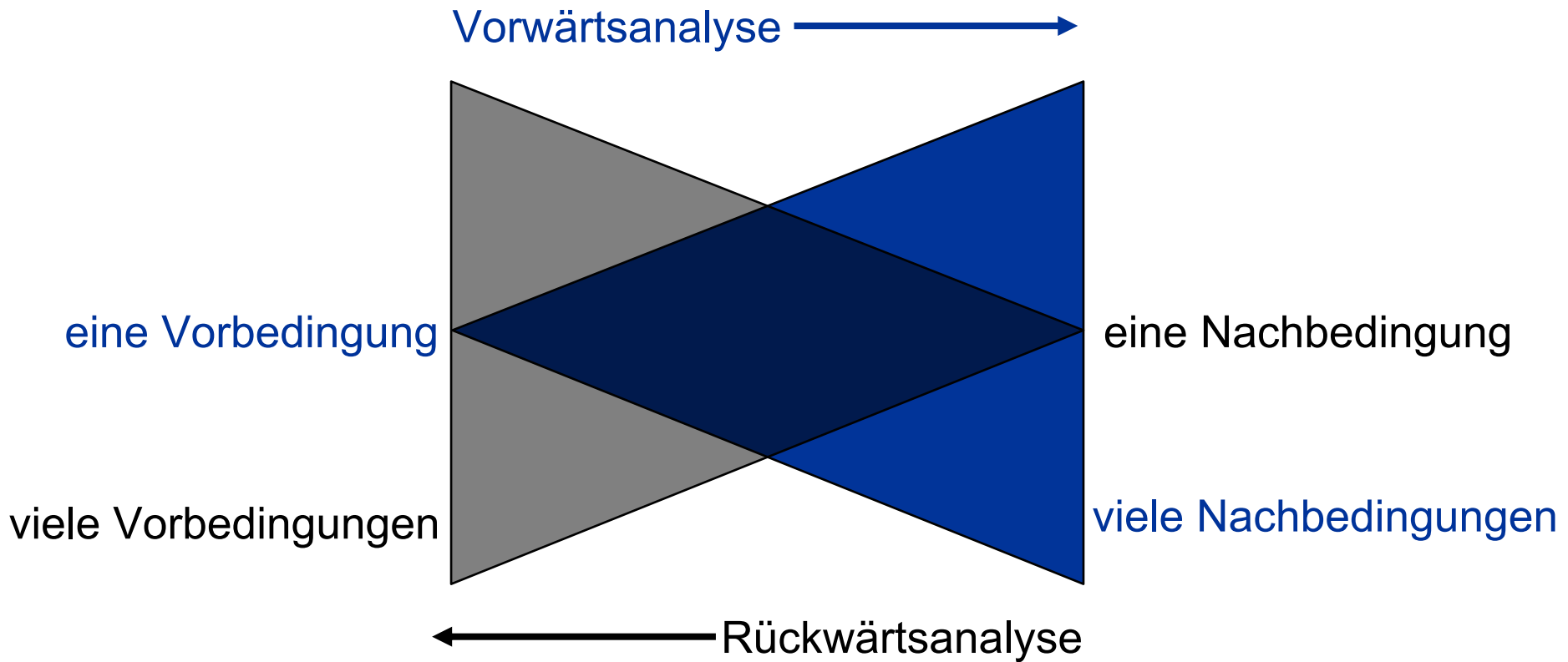


- gegeben **P** und Programm
- bestimme „gute“ Nachbedingung

Rückwärts:

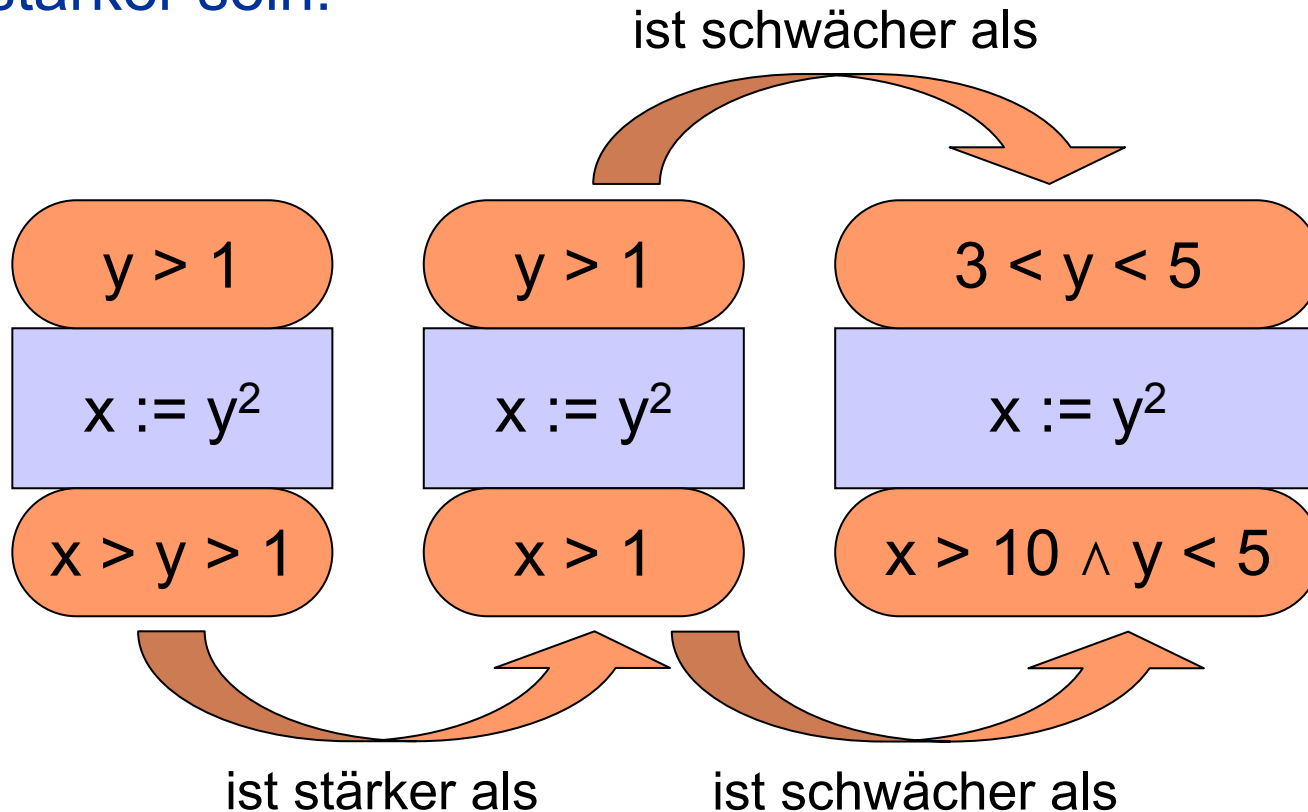


- gegeben **Q** und Programm
- bestimme „gute“ Vorbedingung



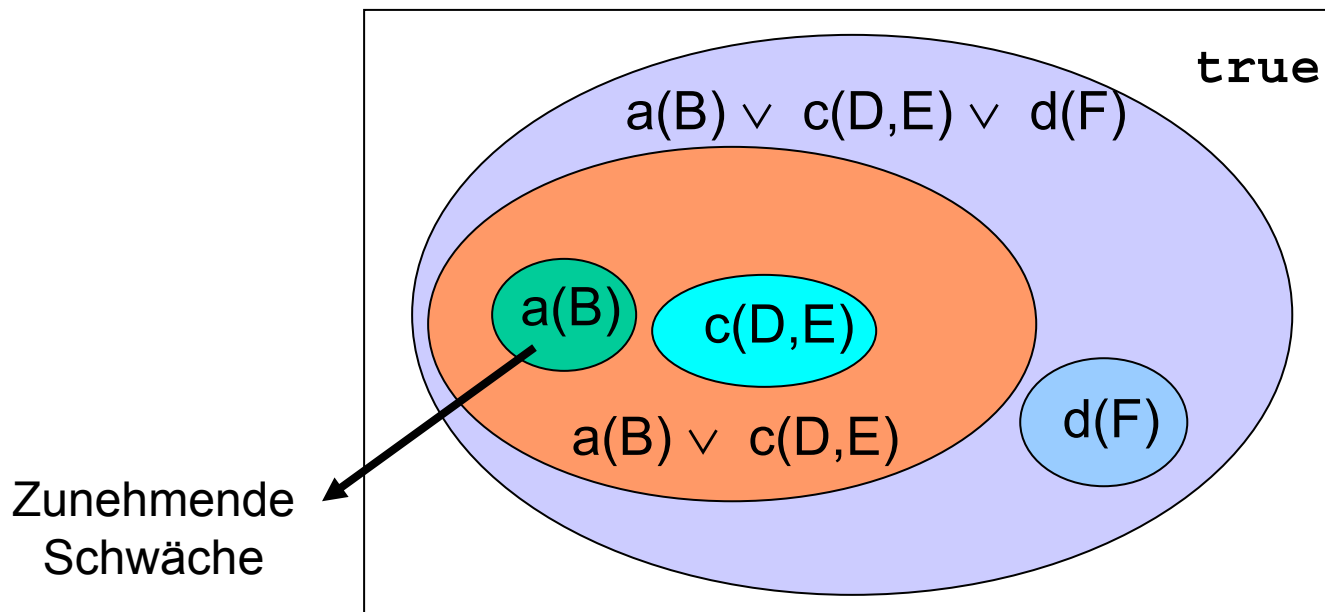
$P$  ist schwächer als  $P' ::= P' \Rightarrow P$

Zusicherungen sind nicht eindeutig, sondern können schwächer oder stärker sein:



Zusicherungen bilden eine Halbordnung mit der Relation „ $\Rightarrow$ “.

- **false** entspricht der leeren Zustandsmenge
- Disjunktion (oder,  $\vee$ ) von Prädikaten entspricht der Vereinigung ( $\cup$ ) der durch sie beschriebenen Zustandsmengen.
- Konjunktion (und,  $\wedge$ ) von Prädikaten entspricht der Schnittmenge ( $\cap$ ) der durch sie beschriebenen Zustandsmengen.



Idee: Untersuche statt vieler Vorbedingungen nur die schwächste Vorbedingung (Weakest Precondition):

- Diese beweist die Nachbedingung „gerade noch“ und wird zur Nachbedingung der vorangehenden Anweisung.
- Es ergibt sich die schwächste Vorbedingung für das ganze Programm und damit ist die Verifikation für den allgemeinsten Anwendungskontext gezeigt.

Definition: Wenn  $Q$  ein Prädikat ist und  $A$  eine Anweisungsfolge, dann ist die schwächste Vorbedingung von  $A$  in Bezug auf  $Q$ ,  $wp(A, Q)$ , ein Prädikat, das die Menge aller Startzustände beschreibt, so dass wenn die Ausführung von  $A$  in einem der Zustände beginnt, die Ausführung in einem Zustand, der  $Q$  genügt, terminiert.

$wp(\text{"Programm"}, \text{Nachbedingung}) = \text{schwächste Vorbedingung}$



## Formal:

- Wenn  $P$  die Terminierung von  $A$  garantiert und  $\{P\} A \{Q\}$ , dann gilt  $P \Rightarrow wp(A, Q)$  für eine Vorbedingung  $P$ .
- Somit:  $\{wp(A, Q)\} A \{Q\}$

andere Schreibweise für die Zuweisung

## Beispiel:

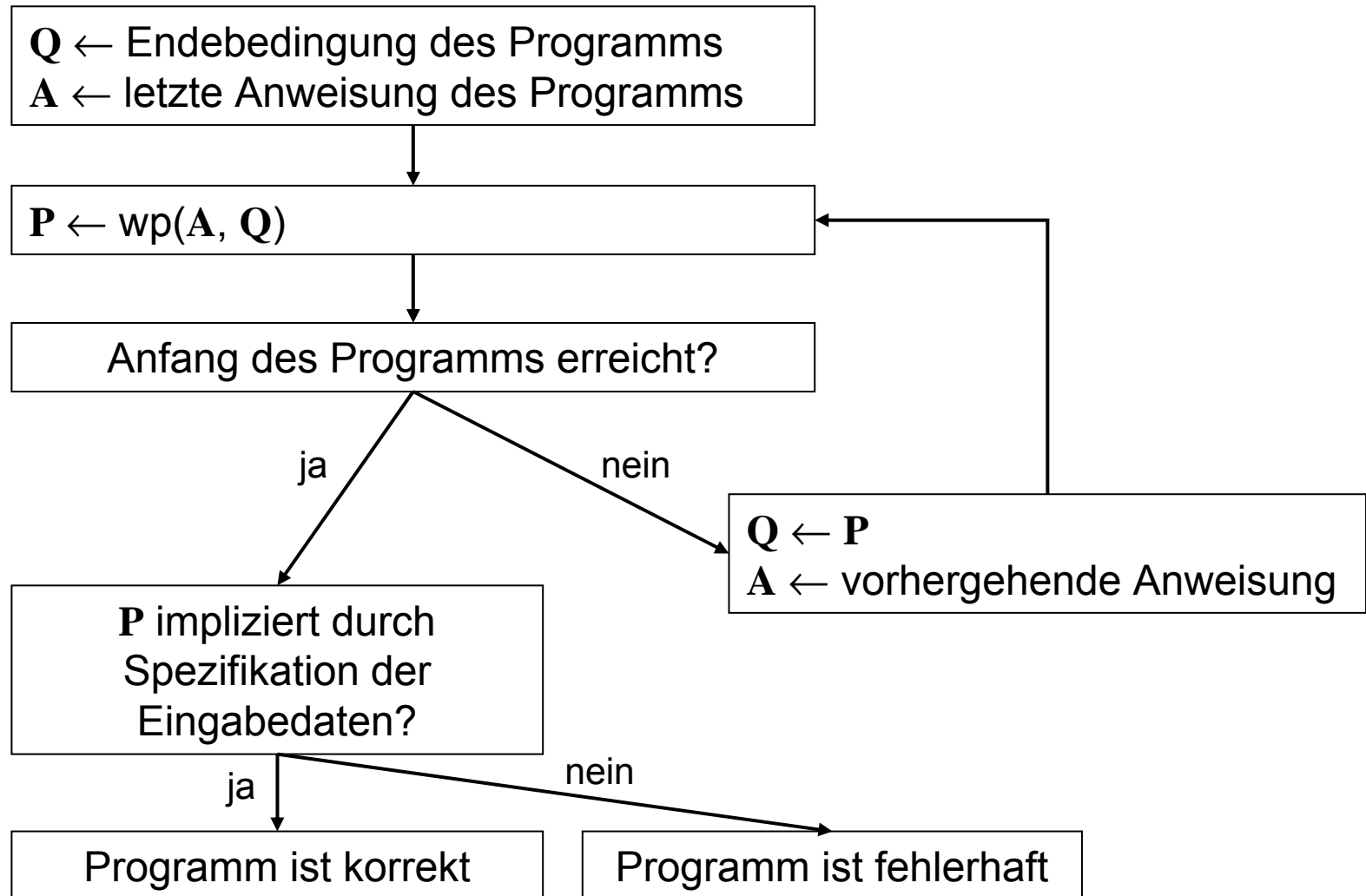
$$\{P: y = 1\} x := y \{Q: x = 1\}$$

$$wp("x := y", x=1) = (y=1)$$

bedeutet: wenn nach Ausführung von  $x := y$  gelten soll:  $x=1$ , dann muss vorher mindestens  $y=1$  gegolten haben.

- Anders ausgedrückt:  $\{wp("x := y", x=1)\} x := y \{x = 1\}$   
Es gilt wegen  $P \Rightarrow wp(A, Q)$ :  $(y = 1) \Rightarrow (y = 1)$





$\text{wp}(\text{leer}, Q) = Q$

Wenn nichts passiert, ändern sich die Bedingungen nicht.

$\text{wp}(\text{fehler}, Q) = \text{false}$

Fehler dürfen nicht passieren.

$\text{wp}(\text{abbruch}, Q) = \text{false}$

Nach Abbruch gilt nichts mehr.



## Ausgeschlossene Wunder

$$\text{wp} ( A, \text{false} ) = \text{false}$$

## Distributivität der Konjunktion

$$\text{wp} ( A, Q ) \wedge \text{wp} ( A, R ) = \text{wp} ( A, Q \wedge R )$$

## Distributivität der Disjunktion

$$\text{wp} ( A, Q ) \vee \text{wp} ( A, R ) \Rightarrow \text{wp} ( A, Q \vee R )$$

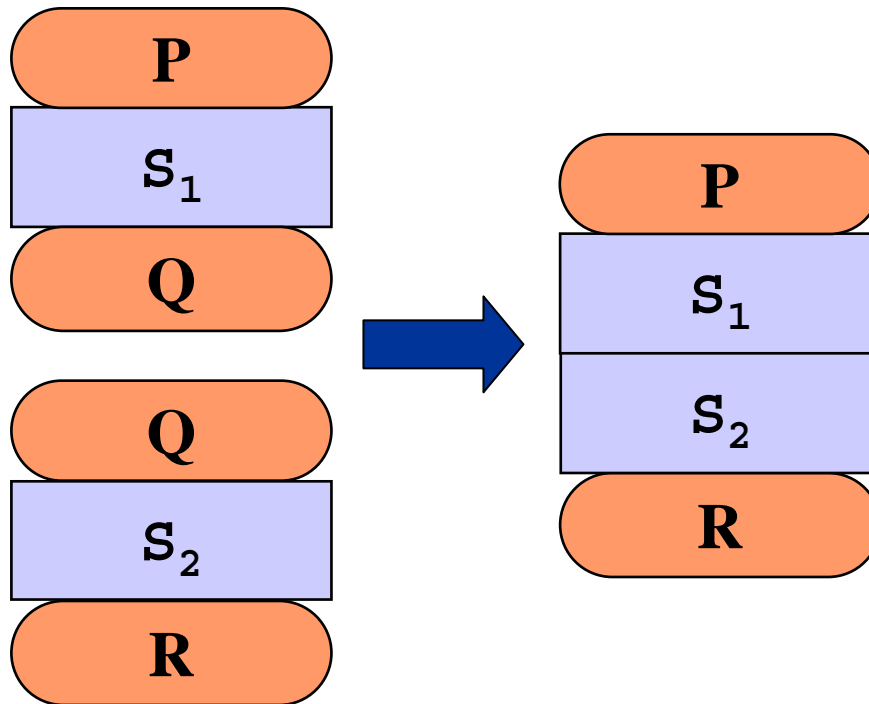
## Monotonie

|      |   |
|------|---|
| Wenn | $Q \Rightarrow R$                                   |
| dann | $\text{wp} ( A, Q ) \Rightarrow \text{wp} ( A, R )$ |

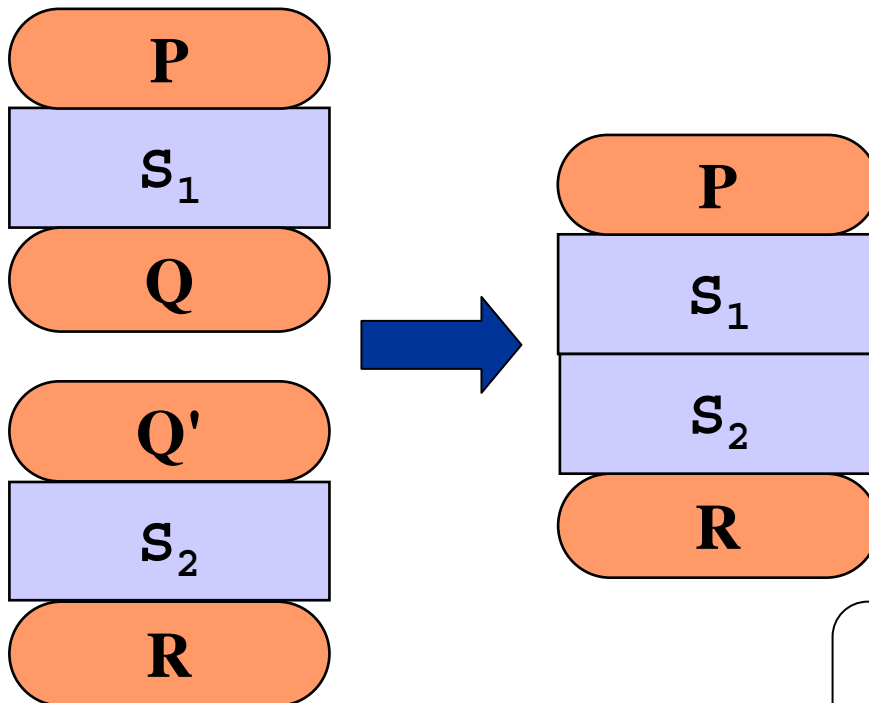
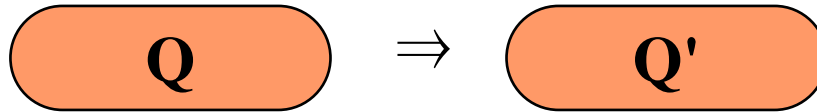
"=" nur bei  
deterministischen  
Programmen



Zwei Programmstücke  $s_1$  und  $s_2$  können zu einem Programmstück  $s_1 ; s_2$  zusammengesetzt werden, wenn die Nachbedingung von  $s_1$  mit der Vorbedingung von  $s_2$  identisch ist.



$$\begin{aligned} & \text{wp} ( "s_1 ; s_2" , R ) \\ & = \\ & \text{wp} ( s_1 , \text{wp} ( s_2 , R ) ) \end{aligned}$$



Wenn  $Q'$  eine Konsequenz von  $Q$  ist, gilt auch:

aus

$\{P\} S_1 \{Q\}$  und  $\{Q'\} S_2 \{R\}$

folgt

$\{P\} S_1 ; S_2 \{R\}$

Klar: Wenn  $Q'$  schon schwächste Vorbedingung, dann kann  $Q$  nicht noch schwächer sein!

$$\text{wp} ("x := e", Q) = Q [x/e]$$

wobei  $x$  eine Variable und  $e$  ein Ausdruck ist

In der Nachbedingung werden alle Vorkommen der Variablen durch die Ausdrücke substituiert; man nimmt also sozusagen den Effekt der Anweisung zurück.

$$\begin{aligned} \text{Beispiel: } \text{wp} ("v := v+1", v = m) &= (v = m) [v/(v+1)] \\ &= (v+1 = m) \\ &= (v = m-1) \end{aligned}$$

Formal korrekt muss bei der ermittelten Vorbedingung  $\text{zulässig}(e)$  ergänzt werden. Dies wird aber oft weggelassen.



Zu verifizieren:

$$\{P: y=X \wedge x=Y\} \quad h:=x; \quad x:=y; \quad y:=h \quad \{Q: x=X \wedge y=Y\}$$

$$\text{wp} ( "h:=x; \quad x:=y; \quad y:=h", x=X \wedge y=Y )$$

$$= \quad \text{wp} ( "h:=x; \quad x:=y", \text{wp}( "y:=h", x=X \wedge y=Y ) )$$

$$= \quad \text{wp} ( "h:=x; \quad x:=y", (x=X \wedge h=Y) )$$

$$= \quad \text{wp} ( "h:=x", \text{wp}( "x:=y", (x=X \wedge h=Y) ) )$$

$$= \quad \text{wp} ( "h:=x", (y=X \wedge h=Y) )$$

$$= \quad (y=X \wedge x=Y)$$

Stimmt mit **P** überein, daher korrekt!



**if-then-else-Regel:**

$$\text{wp}(\text{"if } b \text{ then } s_1 \text{ else } s_2", Q) = \\ (b \Rightarrow \text{wp}(s_1, Q)) \wedge (\neg b \Rightarrow \text{wp}(s_2, Q))$$

Wenn  $b$  gilt, dann muss als Vorbedingung mindestens  $\text{wp}(s_1, Q)$  gelten, damit  $Q$  als Nachbedingung gilt.

Wenn  $b$  nicht gilt, dann muss als Vorbedingung mindestens  $\text{wp}(s_2, Q)$  gelten, damit  $Q$  als Nachbedingung gilt.

- Zusätzlich muss  $\text{zulässig}(b)$  gelten.
- $b$  muss seiteneffektfrei sein.



## If-Regel:

$$\begin{aligned} \text{wp ("if b then A", Q)} &= \\ &(\text{b} \Rightarrow \text{wp (A, Q)}) \wedge (\neg \text{b} \Rightarrow \text{Q}) \end{aligned}$$

Fehlt der **else**-Teil, ist dies identisch mit:

$$\begin{aligned} &\text{wp ("if b then A else leer", Q)} \\ &= (\text{b} \Rightarrow \text{wp (A, Q)}) \wedge (\neg \text{b} \Rightarrow \text{wp (leer, Q)}) \\ &= (\text{b} \Rightarrow \text{wp (A, Q)}) \wedge (\neg \text{b} \Rightarrow \text{Q}) \end{aligned}$$



Wegen  $(A \Rightarrow B) \wedge (\neg A \Rightarrow C) \Leftrightarrow (A \wedge B) \vee (\neg A \wedge C)$  kann die **if-then-else-Regel** auch umgeformt werden zu:

$$\begin{aligned} & \text{wp}(\text{"if } b \text{ then } s_1 \text{ else } s_2", Q) \\ &= (b \Rightarrow \text{wp}(s_1, Q)) \wedge (\neg b \Rightarrow \text{wp}(s_2, Q)) \\ &= (b \wedge \text{wp}(s_1, Q)) \vee (\neg b \wedge \text{wp}(s_2, Q)) \end{aligned}$$

Es muss  $b$  und mindestens die Vorbedingung  $\text{wp}(s_1, Q)$  gelten...

...oder aber nicht  $b$  und mindestens die Vorbedingung  $\text{wp}(s_2, Q)$  gelten.



$\text{wp}(\text{"if } (x > y) \text{ max} := x; \text{ else max} := y; ", \text{max} = x)$

$= [ x > y \wedge \text{wp}(\text{"max} := x", \text{max} = x) ]$

$\vee [ x \leq y \wedge \text{wp}(\text{"max} := y", \text{max} = x) ]$

$= [ x > y \wedge x = x ]$

$\vee [ x \leq y \wedge y = x ]$

$= ( x > y \vee x = y )$

$= (x \geq y)$



**switch**-Anweisung gemäß Semantik von Java:

```
wp ( "switch b
    case  $b_1$  :  $S_1$ ;
    case  $b_2$  :  $S_2$ ;
    ...
    case  $b_n$  :  $S_n$ ;
    else  $S_{n+1}$ ", Q )
```

```
= wp ( "if b ==  $b_1$  then  $S_1$ ;  $S_2$ ; ...;  $S_n$ 
    else if b ==  $b_2$  then  $S_2$ ; ...;  $S_n$ 
    ...
    else  $S_{n+1}$ ", Q )
```



$$\begin{aligned} & \text{wp} ( \text{"if } b == b_1 \text{ then } S_1; S_2; \dots; S_n \\ & \quad \text{else if } b == b_2 \text{ then } S_2; \dots; S_n \\ & \quad \dots \\ & \quad \text{else } S_{n+1} \text{"}, Q ) = \end{aligned}$$

$$\begin{aligned} & (\forall i: (b_i \Rightarrow \text{wp} (S_i; \dots; S_n, Q))) \\ & \quad \wedge ((\neg \exists i: b_i) \Rightarrow \text{wp} (S_{n+1}, Q)) \end{aligned}$$

Wenn irgendein  $b_i$  gilt, dann gilt als schwächste Vorbedingung:


$\text{wp} (S_i; \dots; S_n, Q)$ , wobei gilt:

$$\text{wp} (S_i; \dots; S_k; \dots; S_n, Q) = \text{wp} (S_i; \dots; S_{k-1}, Q)$$

wenn  $S_k = \text{break}$



Die Korrektheit von Schleifen lässt sich auf die Betrachtung von **while**-Schleifen beschränkt, da sich alle Schleifenkonstrukte in **while**-Schleifen umwandeln lassen:

do A while b            A  
while b do A

`do A until b`      →      `A`

`while ¬b do A`

```
for (A,b,C) D            A
                    while b do { D ; C }
```

## Gesucht wird

$\text{wp}(\text{"while } b \text{ do } A", Q) = ?$

## Schreibe die Anweisung um zu:

$\text{if } b \{ A ; \text{if } b \{ A ; \text{if } b \{ A ; \text{if } b \{ \dots \} \} \} \}$

Zur Erinnerung:

$\text{wp}(\text{"if } b \text{ then } A", Q) = (\neg b \Rightarrow Q) \wedge (b \Rightarrow \text{wp}(A, Q))$

## Somit ergibt sich:

$\text{wp}(\text{"while } b \text{ do } A", Q) =$

$\neg b \Rightarrow Q \wedge$

$b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q \wedge$

$b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q \wedge$

$b \Rightarrow \text{wp}(A, \dots))$



$$\begin{aligned}
 \text{wp}(\text{"while } b \text{ do } A", Q) = & \\
 & \neg b \Rightarrow Q \wedge \\
 & b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q \wedge \\
 & \quad b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q \wedge \\
 & \quad \quad b \Rightarrow \text{wp}(A, \dots)))
 \end{aligned}$$

## Beispielschleife:

$$\begin{aligned}
 \text{wp}(\text{"while } (x > 0) \text{ do } x := x - 1", x = 0) = & \\
 & \neg(x > 0) \Rightarrow (x = 0) \wedge \\
 & (x > 0) \Rightarrow [ \neg(x - 1 > 0) \Rightarrow (x - 1 = 0) \wedge \\
 & \quad (x - 1 > 0) \Rightarrow [ \neg(x - 1 - 1 > 0) \Rightarrow (x - 1 - 1 = 0) \wedge \\
 & \quad \quad (x - 1 - 1 > 0) \Rightarrow [ \dots ] ] ]
 \end{aligned}$$



$$\begin{aligned}
 \text{wp ("while (x>0) do x:=x-1", (x=0) )} = \\
 \neg(x>0) \Rightarrow (x=0) \wedge \\
 (x>0) \Rightarrow [ \neg(x-1>0) \Rightarrow (x-1=0) \wedge \\
 (x-1>0) \Rightarrow [ \neg(x-1-1>0) \Rightarrow (x-1-1=0) \wedge \\
 (x-1-1>0) \Rightarrow [ \dots ] ] ] ]
 \end{aligned}$$

Wegen  $A \Rightarrow (B \wedge C) \Leftrightarrow (A \Rightarrow B) \wedge (A \Rightarrow C)$  lässt sich dies vereinfachen zu:

$$\begin{aligned}
 \text{wp ("while (x>0) do x:=x-1", (x=0) )} = \\
 (x \leq 0) \Rightarrow (x = 0) \wedge \\
 (x > 0) \Rightarrow [ (x \leq 1) \Rightarrow (x = 1) ] \wedge \\
 (x > 0) \Rightarrow [ (x > 1) \Rightarrow [ (x \leq 2) \Rightarrow (x = 2) ] ] \wedge \\
 (x > 0) \Rightarrow [ (x > 1) \Rightarrow [ (x > 2) \Rightarrow [ (x \leq 3) \Rightarrow (x = 3) ] ] ] \wedge \\
 \dots
 \end{aligned}$$

Noch einfacher:  $x \geq 0$



- Die obige Form ist als unbeschränkte Konjunktion praktisch nicht handhabbar.
- Die Terminierung einer Schleife ist oft recht einfach zu zeigen, etwa über Dekrementierung von Schleifenzählern, Durchlauf durch endliche Datenstruktur.

**Definition:**  $H(Q) ::= wp( \text{"while } b \text{ do } A", Q )$

**Definition:**  $H_k(Q)$  schwächste Vorbedingung, die garantiert, dass die Schleife höchstens  $k$ -mal ( $k \geq 0$ ) durchlaufen wird und dann  $Q$  gilt.



$$\text{wp}(\text{"while } b \text{ do } A", Q) =$$

$$\neg b \Rightarrow Q \wedge$$

$$b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q \wedge$$

$$b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q \wedge$$

$$b \Rightarrow \text{wp}(A, \dots))$$

Wegen Distributivität der Konjunktion umformen zu:

$$\text{wp}(\text{"while } b \text{ do } A", Q) =$$

|   |          |
|---|----------|
| $\neg b \Rightarrow Q \wedge$   | $H_0(Q)$ |
| $b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q) \wedge$                             | $H_1(Q)$ |
| $b \Rightarrow \text{wp}(A, b \Rightarrow \text{wp}(A, \neg b \Rightarrow Q)) \wedge$ | $H_2(Q)$ |

...

Somit gilt:

$$\text{wp}(\text{"while } b \text{ do } A", Q) = H(Q)$$

$$= H_0(Q) \wedge H_1(Q) \wedge H_2(Q) \wedge \dots = (\forall k \geq 0 : H_k(Q))$$



Offenkundig lässt sich die Schleifenzustandsfunktion  $H(Q)$  als Rekurrenz definieren:

$$H(Q) = (\forall k \geq 0 : H_k(Q))$$

$$H_0(Q) = (\neg b \Rightarrow Q)$$

$$H_{k+1}(Q) = (b \Rightarrow wp(A, H_k(Q)))$$

Zur Lösung der Rekurrenz suchen wir also ein  $H$ , für das gilt:

$$H \Rightarrow (\neg b \Rightarrow Q) \text{ und}$$

$$H \Rightarrow (b \Rightarrow wp(A, H))$$



$\text{wp} ("while (x > 0) \text{ do } x := x - 1", x = 0) =$

$$H_0(x=0) \wedge H_1(x=0) \wedge H_2(x=0) \wedge \dots$$

wobei

$$H_0(x=0) = \neg (x > 0) \Rightarrow (x=0) \quad \text{und}$$

$$H_{k+1}(x=0) = (x > 0) \Rightarrow \text{wp} ("x := x - 1", H_k(x=0))$$

Jetzt wird ein  $H$  gesucht, das die Rekurrenz erfüllt z.B.  $H = (x \geq 0)$

Überprüfen  $H \Rightarrow (\neg b \Rightarrow Q)$ :

$$(x \geq 0) \Rightarrow H_0(x=0)$$

und  $H \Rightarrow (b \Rightarrow \text{wp} (A, H))$ :

$$(x \geq 0) \Rightarrow [(x > 0) \Rightarrow \text{wp} ("x := x - 1", (x \geq 0))] =$$

$$(x \geq 0) \Rightarrow [(x > 0) \Rightarrow (x-1 \geq 0)] =$$

$$(x \geq 0) \Rightarrow [(x > 0) \Rightarrow (x \geq 1)]$$



Die schwächste Vorbedingung  $H$  einer Schleife kann aufgeteilt werden in:

- Eine Bedingung, welche die Schleife terminiert und
- eine Schleifeninvariante  $I$ : ein Prädikat, das vor und nach jeder Iteration einer Schleife immer wahr ist.

**Definition:** Eine Invariante der Schleife `while b do A` ist eine Bedingung  $I$ , für die gilt:  $\{ I \wedge b \} A \{ I \}$

Für die Schleife selbst:

$$\{ I \} \text{while } b \text{ do } A \{ I \wedge \neg b \}$$

Da eine Anweisungsfolge  $A$  terminiert falls  $\text{wp}(A, \text{true})$ , ergibt sich somit für eine Invariante einer terminierenden Schleife:

$$H ::= I \wedge \text{wp}(\text{"while } b \text{ do } A", \text{true}) \Rightarrow \\ \text{wp}(\text{"while } b \text{ do } A", I \wedge \neg b)$$



Wir wollen verifizieren:

```
{P:  $n \geq 0$ }  
s := 0;  
for (i=1; i<=n; i++)  
    s := s + i;  
{Q:  $s=(n+1)n/2$ }
```

Schleife in „Normalform“ umwandeln:

```
{P:  $n \geq 0$ }  
s := 0; i := 1;  
while (i<=n) {  
    s := s + i; i := i + 1;  
}  
{Q:  $s=(n+1)n/2$ }
```



Gesucht wird also:

```
wp("s:=0;i:=1;  
   while(i<=n){s:=s+i; i:=i+1;}",  
   s=(n+1)n/2 )
```

Mit der Sequenzregel ergibt sich:

```
wp("s:=0;i:=1;",  
wp("while(i<=n){s:=s+i; i:=i+1;}",  
   s=(n+1)n/2 ) )
```

Inneren Ausdruck mit der Schleife auflösen. Gesucht wird H:

```
H(s=(n+1)n/2) = wp("while(i<=n){s:=s+i; i:=i+1;}",  
                   s=(n+1)n/2 )
```



Somit ergeben sich

$$H_0(s = (n+1)n/2) = \neg (i \leq n) \Rightarrow (s = (n+1)n/2)$$

$$H_{k+1}(s = (n+1)n/2) = (i \leq n) \Rightarrow \text{wp} ("s := s + i; i := i + 1;", H_k(s = (n+1)n/2))$$

Welches H erfüllt die Rekurrenz?

Vorschlag:  $H ::= s = (i-1)i/2 \wedge (i \leq n + 1)$

Zu zeigen:

**I**

Terminierung

$$H \Rightarrow (\neg b \Rightarrow Q)$$

$$[s = (i-1)i/2 \wedge (i \leq n + 1)] \Rightarrow [\neg (i \leq n) \Rightarrow (s = (n+1)n/2)]$$

wenn  $i \leq n$ :  $[s = (i-1)i/2] \Rightarrow [\text{false} \Rightarrow (s = (n+1)n/2)] = \text{true}$  ✓

wenn  $i = n+1$ :  $[s = (n+1)n/2] \Rightarrow [\text{true} \Rightarrow (s = (n+1)n/2)] = \text{true}$  ✓

wenn  $i > n+1$ :  $[\text{false}] \Rightarrow [\text{true} \Rightarrow (s = (n+1)n/2)] = \text{true}$  ✓

q.e.d.



Zu zeigen:

$$H \Rightarrow (b \Rightarrow wp(A, H))$$

$$[s = (i-1)i/2 \wedge (i \leq n + 1)] \Rightarrow [(i \leq n) \Rightarrow wp("s := s + i; i := i + 1;", s = (i-1)i/2 \wedge (i \leq n + 1))]$$

Mit Sequenzregel:

$$[s = (i-1)i/2 \wedge (i \leq n + 1)] \Rightarrow [(i \leq n) \Rightarrow wp("s := s + i;", wp("i := i + 1;", s = (i-1)i/2 \wedge (i \leq n + 1)))]$$

$$= [s = (i-1)i/2 \wedge (i \leq n + 1)] \Rightarrow [(i \leq n) \Rightarrow [s + i = i(i+1)/2 \wedge (i \leq n)]]$$

$$\text{wenn } i \leq n: \quad [s = (i-1)i/2] \Rightarrow [true \Rightarrow s + i = i(i+1)/2] \quad = true$$

$$\text{wenn } i > n: \quad [false] \Rightarrow [false \Rightarrow false] \quad = true$$

q.e.d.



Gesucht wird immer noch

```
wp("s:=0;i:=1;",  
   wp("while (i<=n) {s:=s+i; i:=i+1;} ",  
      s=(n+1)n/2 ) )
```

Schleife auflösen, indem gefundenes H eingesetzt wird:

```
wp("s:=0;i:=1;", s=(i-1)i/2 ∧ ( i ≤ n + 1 ) )  
=  
wp("s:=0;", wp ("i:=1;", s=(i-1)i/2 ∧ ( i ≤ n + 1 ) ) )  
=  
wp("s:=0;", s=0 ∧ ( 1 ≤ n + 1 ) )  
=  
( 0 ≤ n )
```

Somit muss vor Beginn des Codestückes mindestens  $0 \leq n$  gelten. Dies entspricht genau der im Code angegebenen Vorbedingung **P**. Daher wurde die totale Korrektheit bewiesen.  
q.e.d.



Die Invariante **I** bestimmt sich genauer wie folgt:

- Als H wurde gewählt

$$H := s = (i-1)i/2 \wedge (i \leq n + 1)$$

- Mit **Q**:  $s = (n+1)n/2$  und  $b = (i \leq n)$  ist

$$I = s = (i-1)i/2 \wedge (i \leq n)$$

Vorgehen: Man suche von vorne herein nach der Invarianten einer beabsichtigten Schleife.



Bestimme **I** und konstruiere Rumpf **S**.

**{ I }**

**while b do**

**{ I ∧ b } S { I }**

**{ I ∧ ¬ b }**

**{ Q }**

- Beweise, dass **I** vor Eintritt in die Schleife gilt.
- Beweise, dass **{ I ∧ b } S { I }**, dass also **I** tatsächlich eine Invariante ist.
- Beweise, dass **( I ∧ ¬ b ) ⇒ Q**, so dass bei Terminierung das gewünschte Ergebnis erreicht wird.
- Beweise, dass die Schleife terminiert.



$\{ P: a = A \wedge b = B \wedge a > 0 \wedge b \geq 0 \}$

Zulässigkeit:  $\text{ganz}(a) \wedge \text{ganz}(b)$

```
while (b != 0) {
```

```
    h := a; a := b; b := h % b;
```

```
}
```

$\{ Q: a = ggT(A, B) \}$

## Schleifeninvariante:

■  $I = ( a > 0 \wedge b \geq 0 \wedge ggT(a,b) = ggT(A, B) )$

## Begründung:

■ I gilt zu Beginn

■ nach Durchlaufen des Schleifenrumpfes:

$(b \neq 0) \Rightarrow ( ggT(a, b) = ggT(b, a \% b) )$

■ und am Ende, wenn die Schleife anhält:

$(b = 0) \Rightarrow ( ggT(a, 0) = a = ggT(A, B) )$



```
{ P:  $a = A \wedge b = B \wedge c = C \wedge a > 0 \wedge b > 0 \wedge c > 0$  }
```

```
a0 = a; b0 = b; c0 = c;
```

```
while (a != b || b != c) {  
    if (a < b) a = a + a0;  
    else if (b < c) b = b + b0;  
    else c = c + c0;  
}
```

```
{ Q:  $a = \text{kgV}(A,B,C)$  }
```



## Denkbare Schleifeninvariante:

- $a, b, c$  sind Vielfache von  $A, B$  bzw.  $C$  und  $a, b, c \leq \text{kgV}(A, B, C)$ :

$$I_1 = (A, B, C > 0) \wedge$$

$$\exists p, q, r : ( (p, q, r > 0) \wedge (a = p * A) \wedge (b = q * B) \wedge (c = r * C) )$$

$$I_2 = (a \leq \text{kgV}(A, B, C)) \wedge (b \leq \text{kgV}(A, B, C)) \wedge (c \leq \text{kgV}(A, B, C))$$

$$I = I_1 \wedge I_2$$

## Dann gilt bei Schleifenabbruch:

- $p * A = q * B = r * C = a = b = c \leq \text{kgV}(A, B, C)$
- und somit notwendig  $a = b = c = \text{kgV}(A, B, C)$

## Terminierung:

- $a, b, c$  können nur wachsen.
- Da  $\text{kgV}(A, B, C) \leq A * B * C$ , fällt die Differenz  $3 * A * B * C - a - b - c$  streng monoton und ist durch 0 nach unten beschränkt.



## Vorteile:

- Beweis, dass ein Programm entsprechend seiner (formalen) Spezifikation, d.h. seiner Vor- und Nachbedingungen implementiert ist.
- Vollständiger Korrektheitsbeweis ist möglich

## Nachteile:

- Aufwändig und teilweise für umfangreiche, komplexe Programme nicht möglich
- Aufbereitung der Programme für Beweis erfordert hohe Qualifikation
- Programmiersprache muss eine formale Semantik haben
- Teile des Programms, die Sprachkonstrukte verwenden, die keine formale Semantik besitzen müssen getestet werden
  - Z.B. Gleitpunktarithmetik, externe Ein-/Ausgaben, Interrupts
- Maschineneigenschaften werden nicht berücksichtigt
- Spezifikationstechnik mit Vor- und Nachbedingungen erforderlich



Balzer: "Lehrbuch Grundlagen der Informatik",  
950 Seiten, Spektrum Akademischer Verlag, ISBN: 3827403588

