Technische Universität
Kaiserslautern

o1o1seda o1o1oo

software engineering dependability

Software Entwicklung 2

Formale Verifikation

Inhalt

- Verifikation
  - Intuitive Einführung: Die induktive Zusicherungsmethode
  - Zusicherungen
  - Spezifizieren mit Anfangs- und Endebedingung
  - Der Hoare-Kalkül
  - Totale Korrektheit: Termination von Schleifen
  - Entwickeln von Schleifen
  - Umgang mit strukturierten Datentypen: Quantoren
- Algebraische Spezifikationen

Lernziele

- Zusicherungen schreiben können
- Einfache Programme mit einer Anfangs- und Endebedingung (auch mit Quantoren) spezifizieren können
- Verifikationsregeln für die Zuweisung, die Auswahl, die abweisende Wiederholung, sowie die Kompositions- und Inferenzregeln kennen und anwenden können
- Programme, die als PAP oder Struktogramm beschrieben sind, formal verifizieren können
- Den Unterschied zwischen partieller und totaler Korrektheit kennen
- Terminationsbeweise führen können
- Eine Schleife aus einer gegebenen Invariante und Terminationsfunktion entwickeln können
- Methoden zur Entwicklung einer Schleifeninvariante kennen und auf einfache Programme anwenden können
- Wissen, wie algebraische Spezifikationen aufgebaut sind

Verifikation

- Zur Historie
  - Prof. Charles A. R. Hoare

    *11.1.1934 in Colombo, Sri Lanka
  - Professor für Informatik
    Universität Oxford
  - Wichtige Arbeiten zur formalen Semantikdefinition von Programmen
  - Wesentliche Beiträge zur nebenläufigen Programmierung
  - 1980: ACM Turing Award

Fragen zu Algorithmen

- Korrektheit
  - Löst der Algorithmus das gewünschte Problem richtig?
  - Anders ausgedrückt: Wie kann man sicher sein, dass für alle Eingaben die gewünschten Ergebnisse erzielt werden?
- Termination
  - Endet der Algorithmus für alle zulässigen Eingaben, und wie beweist man die Termination?
- Aufwand, Effizienz
  - Wie viele Ressourcen (Zeit, Speicher, …) benötigt der Algorithmus zur Problemlösung in Abhängigkeit des Problemumfangs?

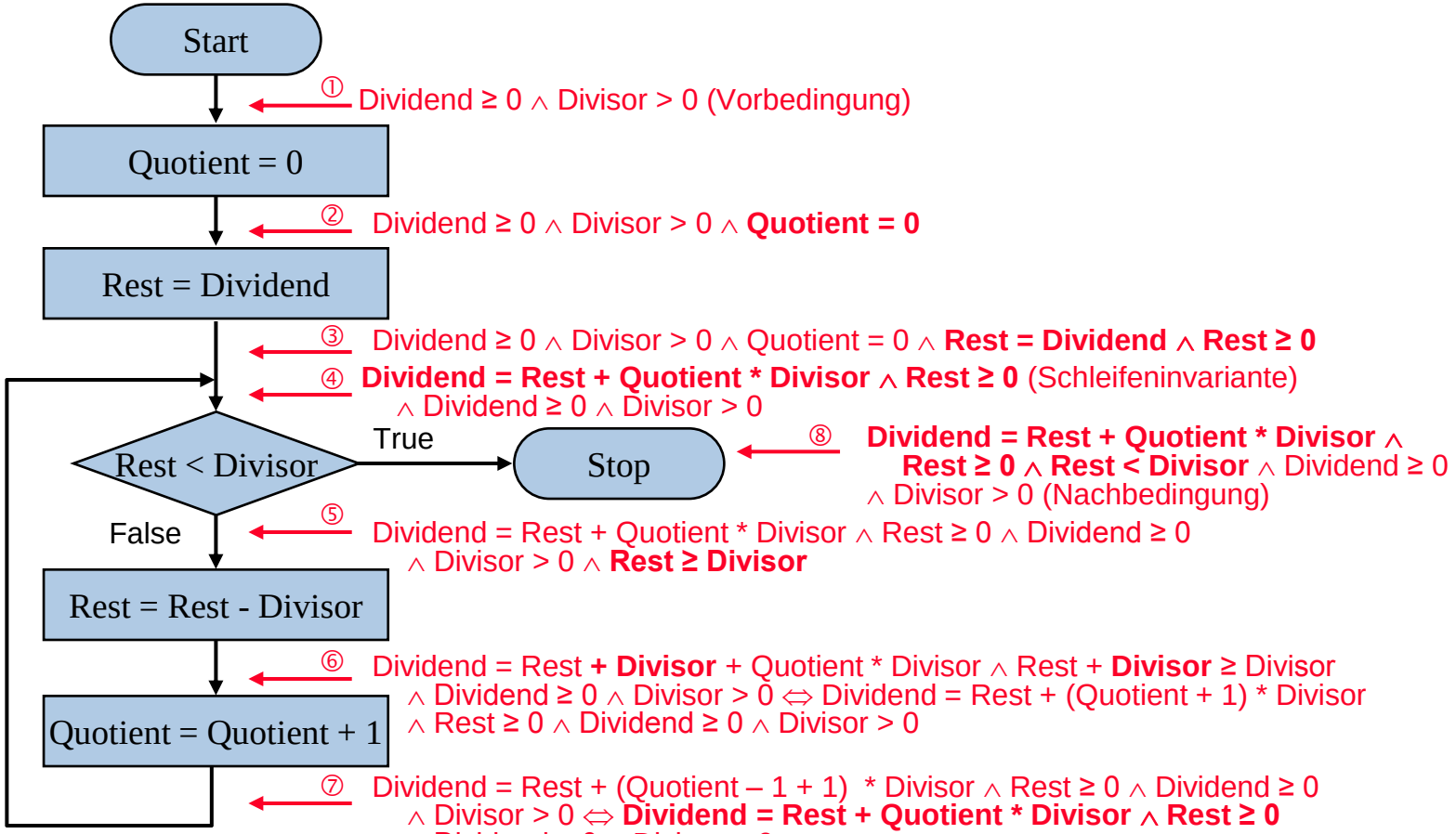
Andere Ansätze

- Testen
  - Für **einige** möglichst gut ausgewählte Testdaten wird das Programm ausgeführt und beobachtet, ob das gewünschte Ergebnis ermittelt wird.
  - Nicht alle möglichen Kombinationen von Eingabedaten können getestet werden.
- Statische Analysen, Inspektionen
  - Das Programm wird werkzeuggestützt oder manuell nach Fehlern und anderen Schwachpunkten durchsucht.
- Diese Ansätze besitzen einerseits eine **hohe praktische Bedeutung** und einige **charakteristische Vorteile**. Andererseits wird **keine Gewissheit** über die Korrektheit des Programms für alle **möglichen** Eingaben erreicht.

Verifikation: Eigenschaften

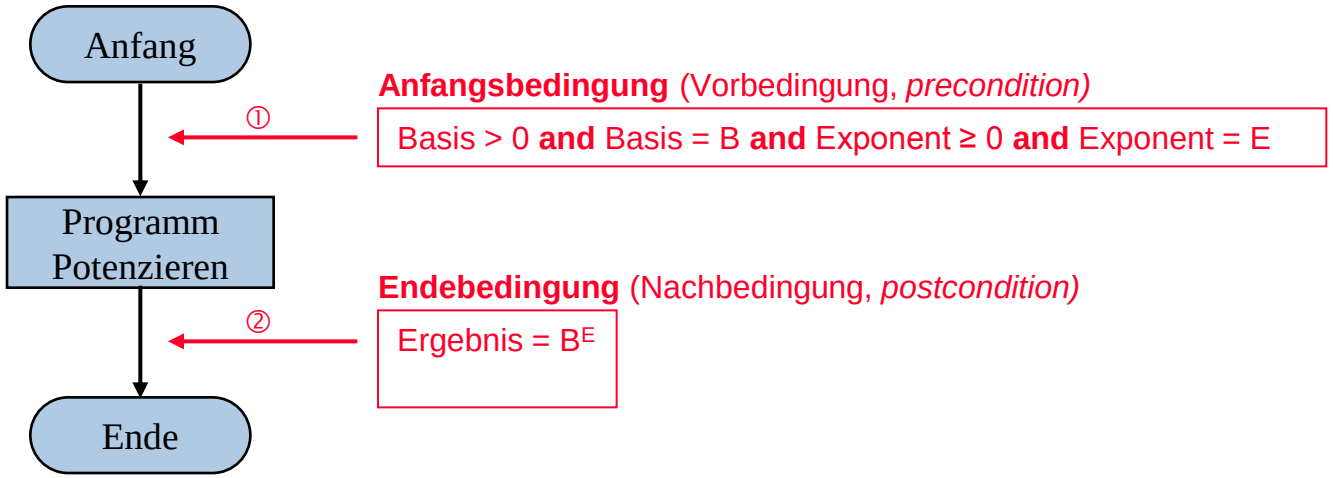
- Die Verifikation demonstriert mit mathematischen Mitteln die Konsistenz zwischen Spezifikation und Implementation (formal exakte Methode).
- Eine formale Spezifikation ist notwendig.
- Die Verifikation kann die Korrektheit beweisen: Größeres Vertrauen in die Fehlerfreiheit von Programmen

- Verifikation eines Programm-Ablauf-Plans mit der induktiven Zusicherungsmethode nach Floyd



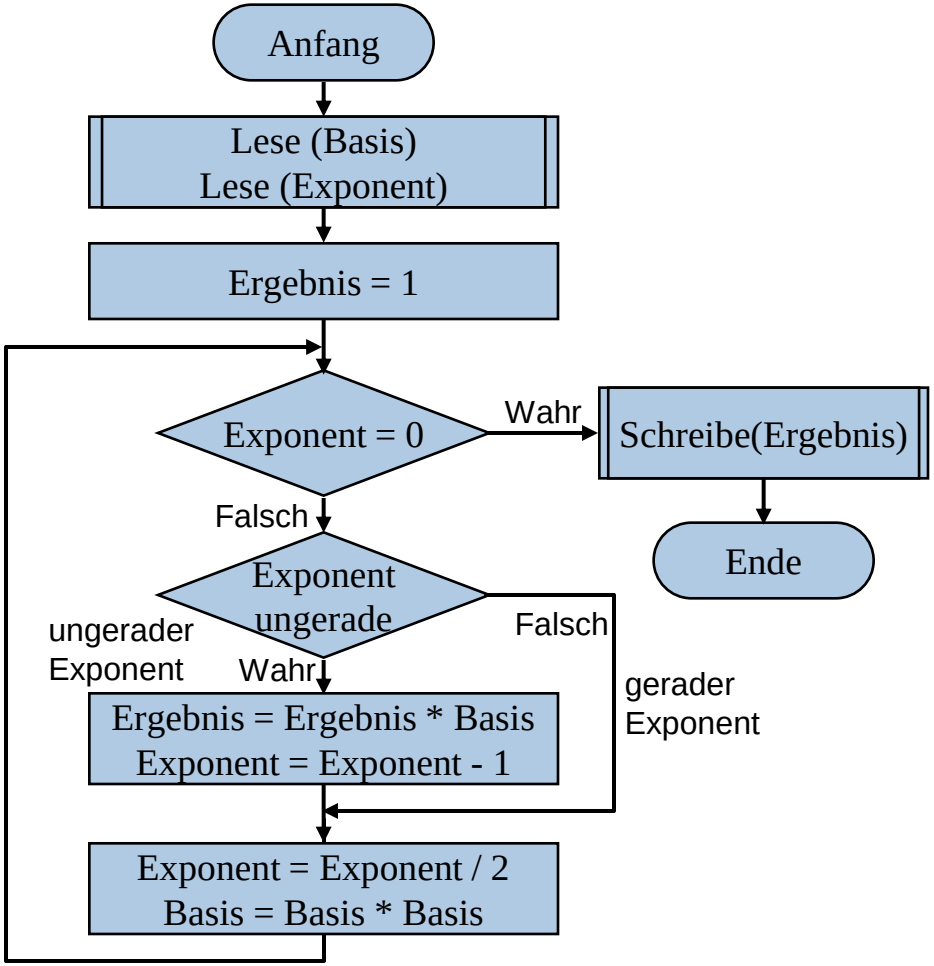
Intuitive Einführung II

- Beispiel
  - Programm, das aus einer positiven ganzzahligen Basis B und einem nicht-negativen ganzzahligem Exponenten E die Potenz B^E ermittelt
  - Das Programm ist dann korrekt, wenn folgende Beziehungen gelten



Intuitive Einführung II

- Beim Test würden konkrete Werte gewählt, wie: Basis = 5, Exponent = 4



Beispiel

Basis = 5
Exponent = 4

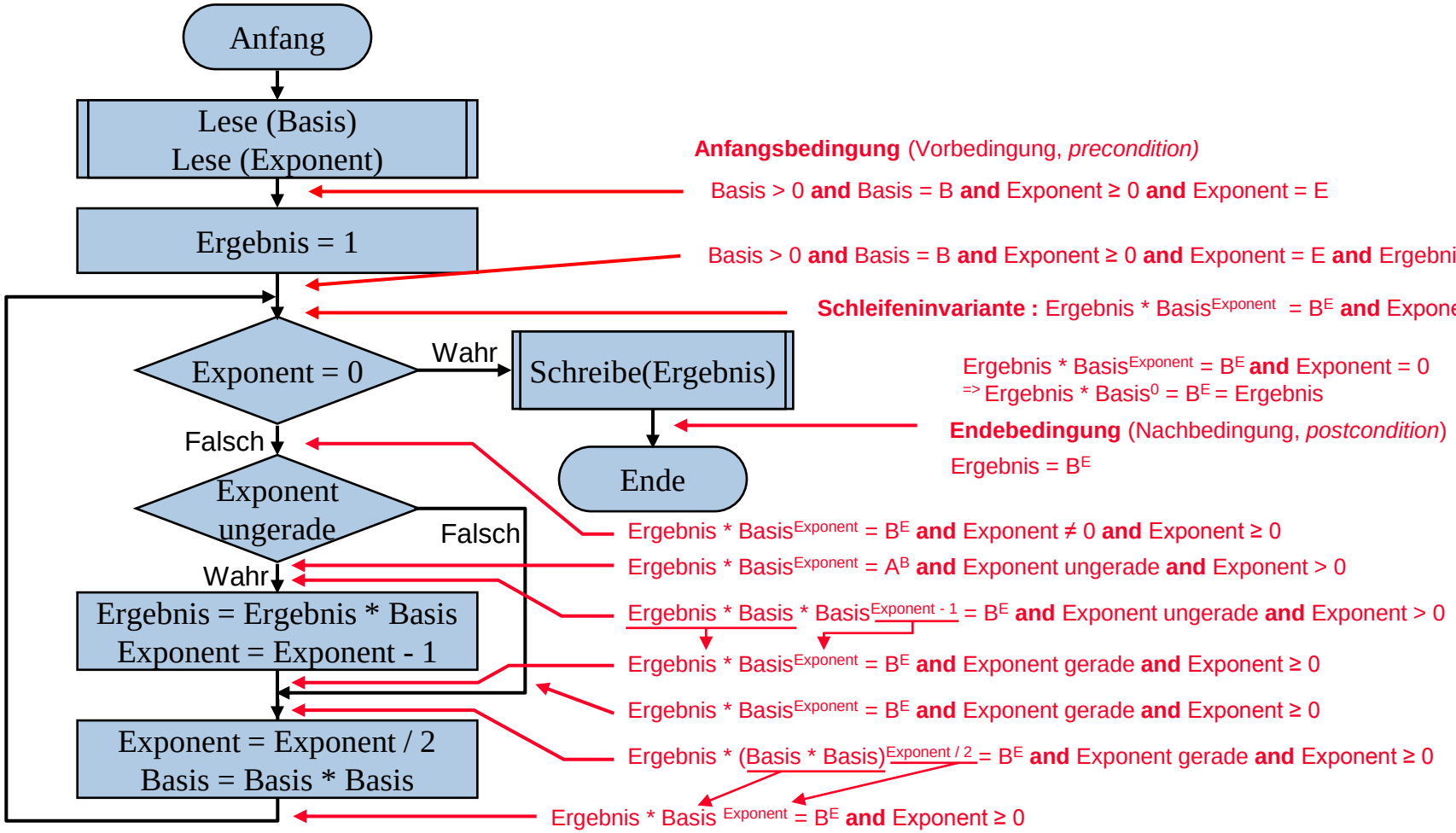
Ergebnis = 1

1. Durchlauf	2. Durchlauf	3. Durchlauf	4. Durchlauf
4 = 0? Falsch	2 = 0? Falsch	1 = 0? Falsch	0 = 0? Wahr **Ergebnis = 625**
4 = ungerade? Falsch	2 = ungerade? Falsch	1 = ungerade? Wahr	
		Ergebnis = 625 Exponent 0	
Exponent = 2 Basis = 25	Exponent = 1 Basis = 625	Exponent = 0 Basis = 390625	

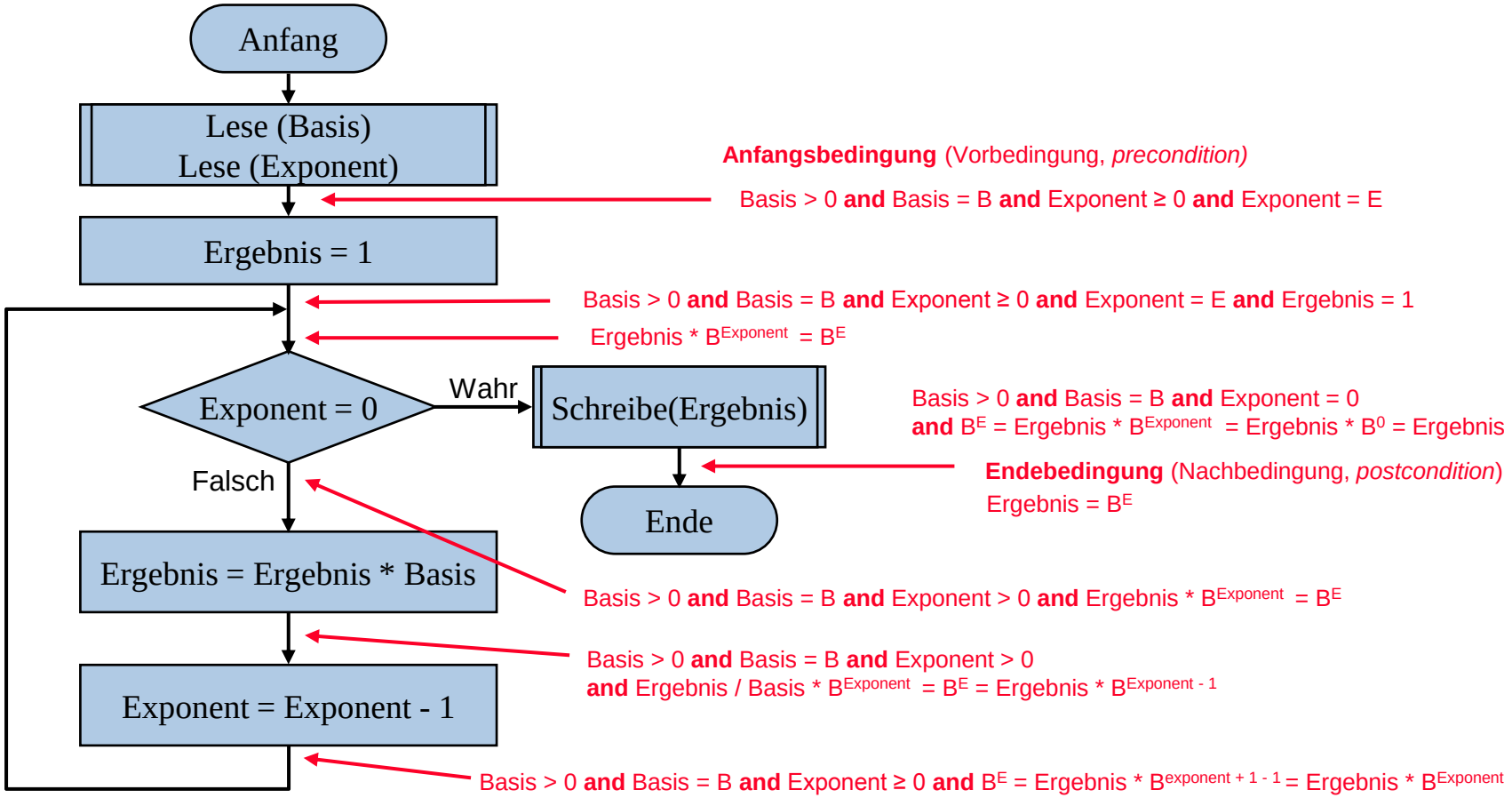
Intuitive Einführung II

- Das Programm nutzt offensichtlich die folgenden mathematischen Beziehungen aus
  - Für geraden Exponenten: $\mathbf{Basis^{Exponent} = (Basis * Basis)^{Exponent/2}}$
  - Für ungeraden Exponenten: $\mathbf{Basis^{Exponent} = Basis * Basis^{Exponent-1}}$

- PAP mit Zusicherungen



- Alternative Realisierung der Potenzierung



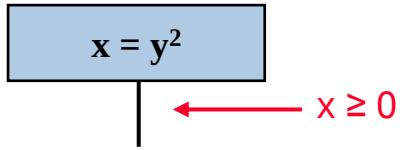
Intuitive Einführung II

- Dass das Programm nach endlich vielen Wiederholungen endet, d.h. terminiert, wurde nicht bewiesen und muss gesondert gezeigt werden.

- Totaler Korrektheitsbeweis besteht aus 2 Teilen
  - Beweis, dass das korrekte Ergebnis bei Termination geliefert wird: Partielle Korrektheit
  - Beweis der Termination: Totale Korrektheit

Zusicherungen

- Zusicherungen *(assertions)*
  - Garantieren an bestimmten Stellen im Programm bestimmte Eigenschaften oder Zustände
  - Sind logische Aussagen über die Werte der Programmvariablen an den Stellen im Programm, an denen die jeweiligen Zusicherungen stehen
- Beispiele
  - Im Programm Potenzieren gilt am Ende immer die Zusicherung Ergebnis = B^E
  - Nach dem Quadrieren einer reellen Zahl kann z.B. zugesichert werden, dass diese Zahl nicht negativ ist

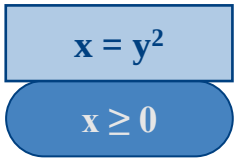


Zusicherungen – Formulierung

- umgangssprachlich
  - Beispiel: x ist nicht negativ
- formal
  - Beispiel: $x \geq 0$
  - Boolesche Ausdrücke mit Konstanten und Variablen
    - mit Vergleichsoperatoren
      - $<, \leq, =, \neq, \geq, >$
    - logischen Operatoren
      - and, or, not, $\Leftrightarrow$, $\Rightarrow$.
  - Zusätzlich Quantoren: „Für Alle gilt …“, „Es gibt ein …“

Zusicherungen

- Drei Notationen
  - Annotation durch Linien oder Pfeile an einem Programmablaufplan oder Kontrollflussgraph
  - Ergänzung von Struktogrammen durch Annotationssymbole

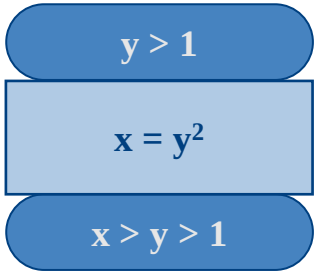


  - Spezielle Kommentare in Programmiersprachen, z.B.

```
assert (x >= 0)          // Zusicherung
// ist ungültig, wenn x negativ ist
```

Zusicherungen

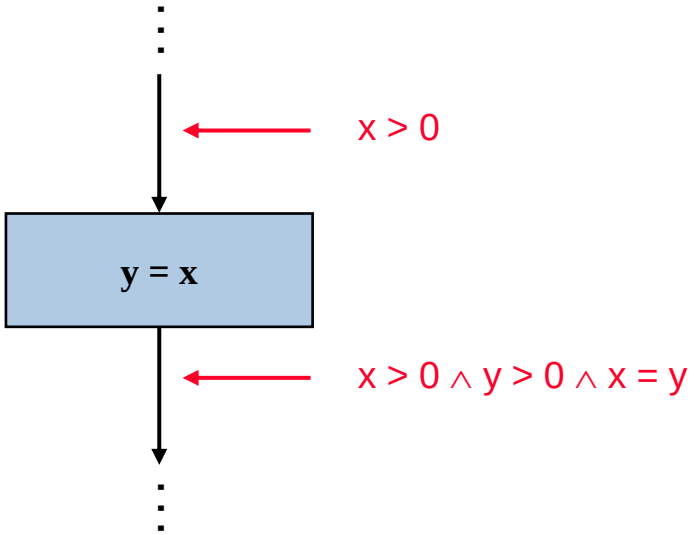
- Beispiel
  - Ist vor der Zuweisung $x := y^2$ sichergestellt, dass y größer als 1 ist, dann kann man nach der Zuweisung zusichern, dass x größer als y ist
  - Nachher gilt auch weiterhin $y > 1$ sowie zusätzlich $x = y^2$



- Zusicherungen beschreiben die Wirkung von Programmen (Semantik), nicht deren Form (Syntax)

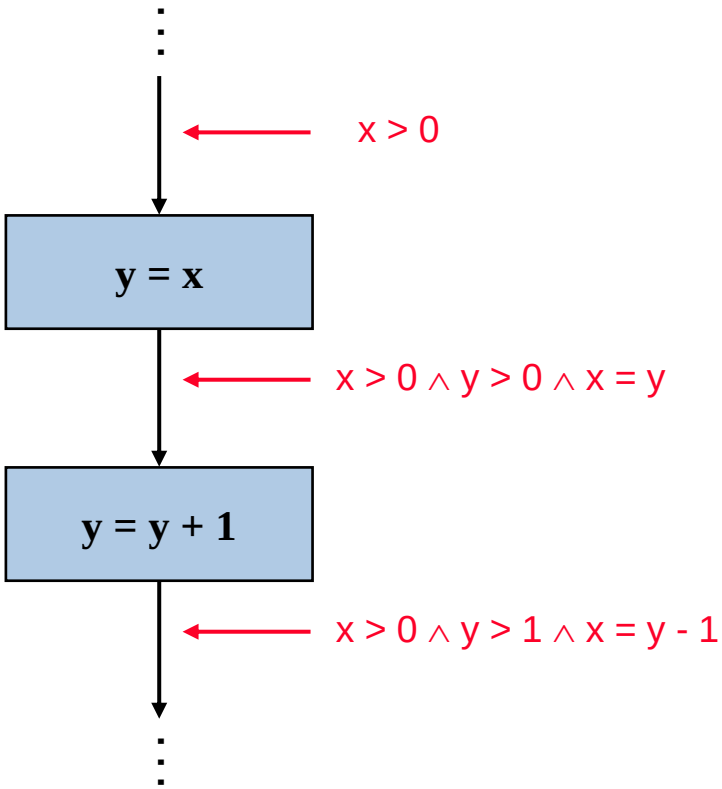
Zusicherungen

- Beispiel



Zusicherungen

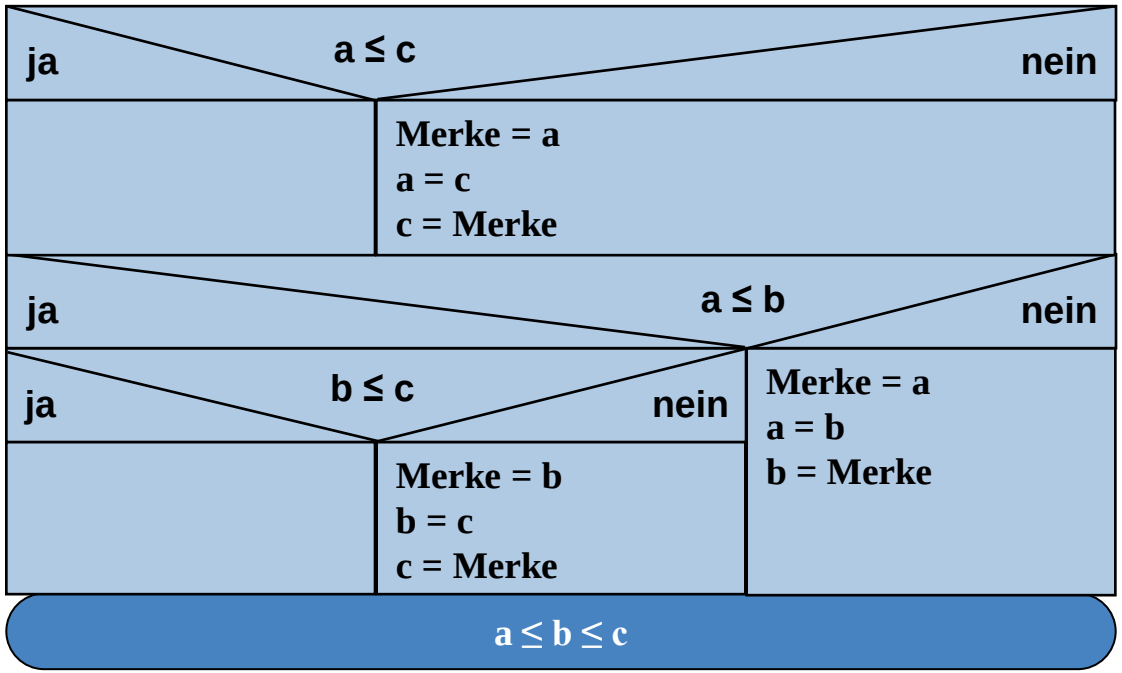
- Beispiel



- Beispiel: Programm Vertausche
  - Die Werte der Variablen a, b und c durch Vertauschungen so umordnen, dass am Schluss $a \leq b \leq c$ gilt
  - Es ist schwierig zu erkennen, ob in allen Zweigen des Programms das richtige Ergebnis erzielt wird.
  - Mit Hilfe von eingefügten Zusicherungen ist die Wirkung besser zu verstehen.
  - Die einzelnen Zusicherungen gelten an den jeweiligen Stellen im Programm, unabhängig von den konkreten Anfangswerten von a, b und c.

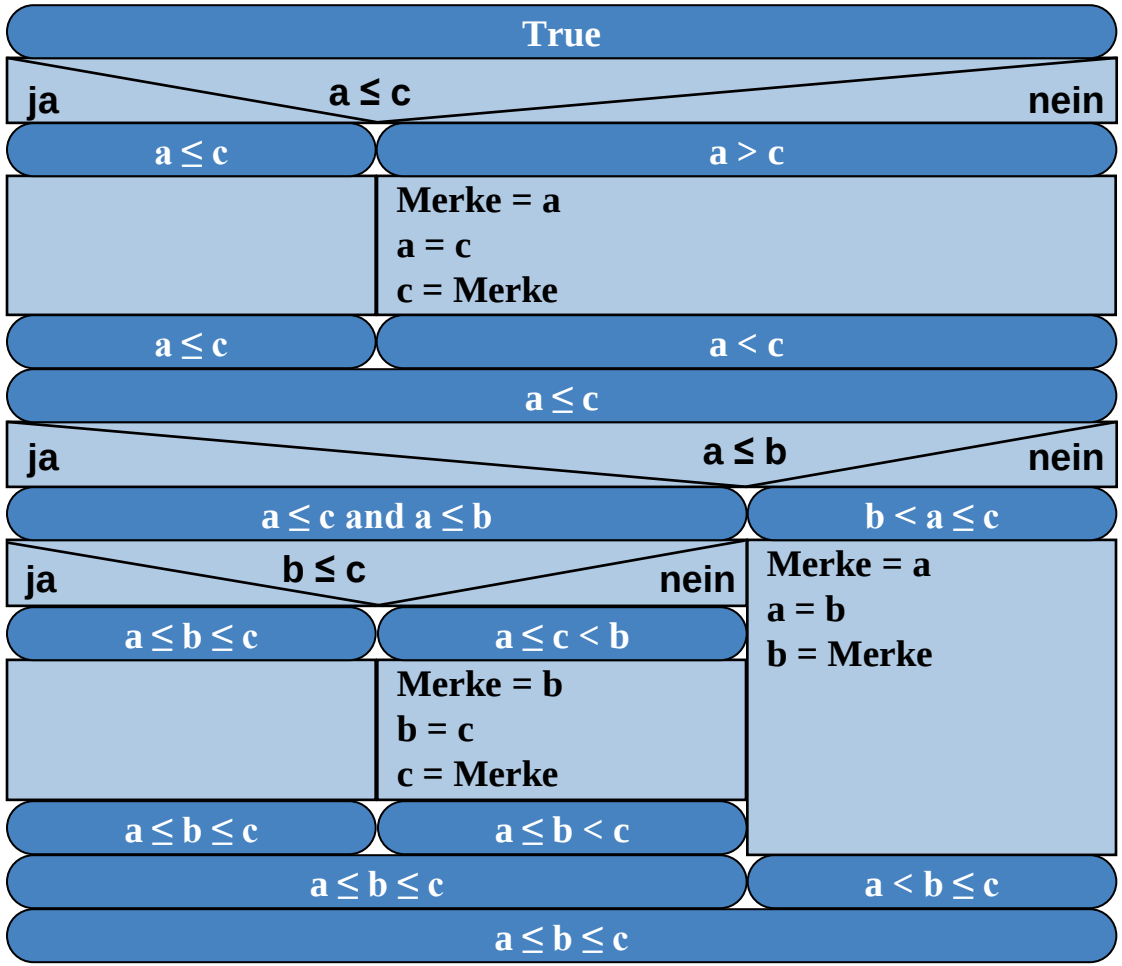
Zusicherungen

- Struktogramm



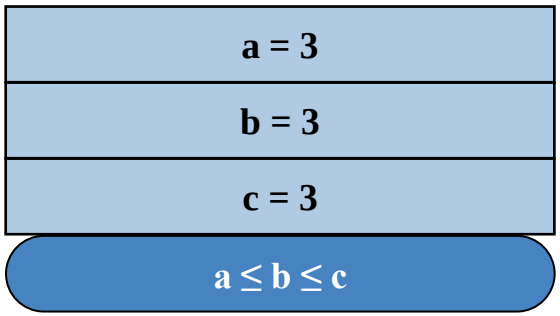
Zusicherungen

- *Vertausche* mit Zusicherungen



Zusicherungen

- Behauptung: Auch das hier angegebene Programm erfüllt die Spezifikation, ist aber sicher keine sinnvolle Implementierung von Vertausche. Wo ist der Fehler?

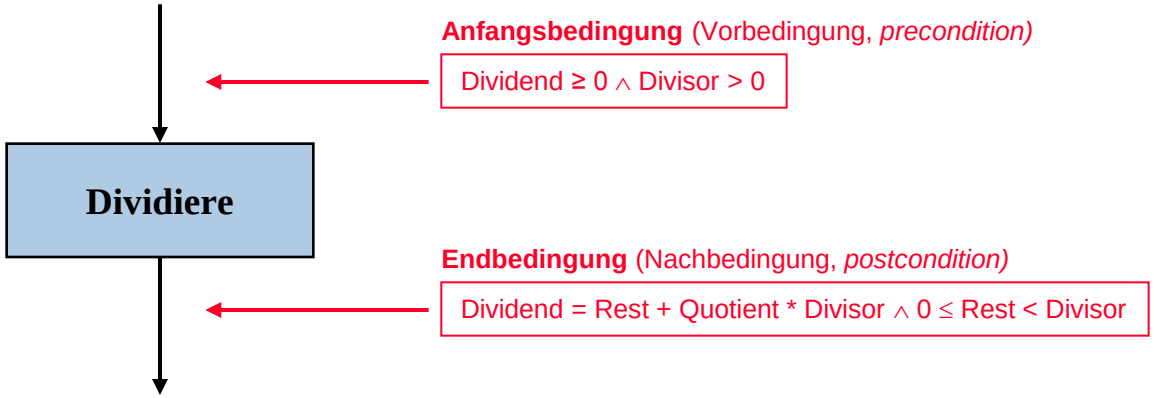


Spezifizieren mit Anfangs- / Endebedingung

- Wirkung eines Programms
  - Kann durch die beiden Zusicherungen Anfangsbedingung und Endebedingung spezifiziert werden
  - Anfangsbedingung (Vorbedingung, precondition) gilt vor dem spezifizierten Programm und legt die zulässigen Werte der Variablen vor dem Ablauf des Programms fest
  - Zusicherung `true` ist immer erfüllt
    - Schränkt den Wertebereich der Variablen in keinerlei Weise ein
    - Wird als erste Zusicherung (Anfangsbedingung) in einem Programm verwendet, wenn für die Werte der Input-Variablen keinerlei Einschränkungen existieren
  - Endebedingung (Nachbedingung, postcondition) gilt nach dem spezifizierten Programm und legt die gewünschten Werte der Variablen und Beziehungen zwischen den Variablen nach dem Ablauf des Programms fest

Spezifizieren mit Anfangs- / Endebedingung

- Für das Programm *Dividiere* gilt



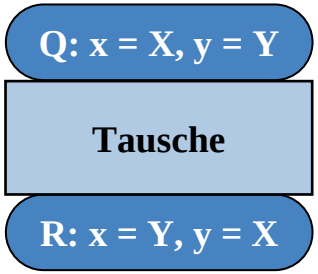
Spezifizieren mit Anfangs-/Endebedingung
Der Hoare-Kalkül

- Notation nach Hoare
  - {Q} S {R}
  - Ohne Bezug auf ein konkretes Programm S
    {Q} . {R}
  - Es ist Aufgabe des Programmierers, ein Programm S zu schreiben, so dass jedesmal, wenn vor dem Programm S die Vorbedingung Q erfüllt ist, das Programm terminiert und nach der Termination die Nachbedingung R erfüllt ist.

Spezifizieren mit Anfangs-/Endebedingung
Der Hoare-Kalkül

- Basis
  - Ein Programm enthält als formale Beschreibungsform alle zur Ermittlung der Programmeigenschaften und der Folgen einer Programmausführung notwendigen Informationen
  - Es ist möglich, das Verhalten eines Programms durch Anwendung von Inferenzregeln auf eine Menge von gültigen Axiomen zu ermitteln
  - Formale Beschreibung der Semantik
  - Der wesentliche Teil des Hoare-Kalküls ist die Idee, die Wirkung von Anweisungen über die beteiligten Variablen darzustellen
- Ist S ein Programm oder ein Teil eines Programms und ist Q die Vorbedingung (precondition) vor Ausführung von S und gilt nach Ausführung von S die Nachbedingung (postcondition) R unter der Voraussetzung, dass S terminiert, so schreibt man
  - {Q} S {R}

- Beispiel *Tausche*
  - Programm, das die Werte der zwei Variablen x und y vertauscht



  - Für alle Werte von x und y gilt
    - »Jedesmal, wenn vor dem Aufruf von Tausche
      x den Wert X und y den Wert Y hat, dann terminiert Tausche, und danach hat x den Wert Y und y den Wert X«
  - X und Y: stellvertretend für beliebige Eingabewerte
  - Da bei der Verifikation das Programm für alle Werte überprüft wird, werden so genannte externe Variable verwendet, die alle Eingabewerte repräsentieren.

Der Hoare-Kalkül
Verifikationsregeln

- Programme
  - Korrektheit: Ergibt sich aufgrund der Korrektheit der Teilstrukturen
  - Ein komplexes Programm kann schrittweise durch korrektes Zusammensetzen aus einfacheren Strukturen verifiziert werden
- Verifikationsregeln
  - A0. Zuweisungsaxiom
  - A1/2. Inferenzregeln
  - A3. Kompositionsregel
  - A4. `if`-Regel
  - A5. `while`-Regel

Der Hoare-Kalkül
Verifikationsregeln: Zuweisungsaxiom

- Man sagt, Q ist Vorbedingung von S bezüglich R
  - Falls S ein vollständiges Programm ist, so wird Q auch als Eingangszusicherung (*entry assertion*) und R als Ausgangszusicherung (*exit assertion*) bezeichnet
  - Falls keine Vorbedingung existiert, so schreibt man {TRUE} S {R}
- Stellt man die Wirkung einer Zuweisung x: = f auf diese Weise dar, so erhält man
  - $\{P_f^x\}$ x := f {P}
  - P_f^x entsteht aus P durch Ersetzen aller Vorkommen von x mit f. Die Zusicherung P, die nach der Ausführung der Zuweisung wahr sein soll, muss vor der Ausführung für die Variable auf der rechten Seite der Zuweisung erfüllt gewesen sein

Der Hoare-Kalkül
Verifikationsregeln: Zuweisungsaxiom

$\{ P_f^x \}\ x: = f\ \{ x > 0 \}$

- Wenn nach der Ausführung der Zuweisung x > 0 gelten soll, so muss vor der Ausführung der Zuweisung f > 0 erfüllt gewesen sein. Dies ist die Vorbedingung P_f^x, die durch einfaches Ersetzen aller Variablen x der Nachbedingung durch Variablen f erzeugt wird.
  - $\{ f > 0 \}\ x: = f\ \{ x > 0 \}$
- In allgemeiner Form kann die Semantik der Zuweisung als Axiom durch das Zuweisungsaxiom
  - A0. $\{ P_f^x \}\ x: = f\ \{ P \}$

  dargestellt werden.

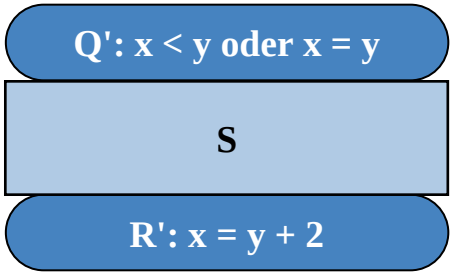
Der Hoare-Kalkül
Verifikationsregeln: Zuweisungsaxiom

- Beispiel
  - Vorbedingung Q: y = 10
  - Zuweisung x:= y
  - Nachbedingung R: x = 10
- Beispiel
  - {?} x:= x + 25 {x = 2y}
    - Vorbedingung ergibt sich dadurch, dass in der Nachbedingung x = 2y alle x durch den Ausdruck x + 25 ersetzt werden: x + 25 = 2y
    - Es ergibt sich die Vorbedingung {2y = x + 25}
  - {?} Ergebnis:= Ergebnis * Basis {Ergebnis*$Basis^{Exponent}$ = B^{E} and Exponent gerade}
    - Einsetzen des Ausdrucks Ergebnis * Basis in die Nachbedingung ergibt folgende Vorbedingung
      - {Ergebnis * Basis * $Basis^{Exponent}$ = B^{E} and Exponent gerade}
      - Vereinfacht: {Ergebnis * Basis $^{Exponent + 1}$ = B^{E} and Exponent gerade}

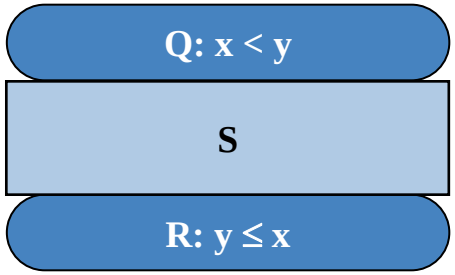
Der Hoare-Kalkül
Inferenzregeln

- Ist {Q'} S {R'} gegeben, dann kann jederzeit die Vorbedingung Q' durch eine »schärfere« Vorbedingung Q und die Nachbedingung R' durch eine »schwächere« Nachbedingung R ersetzt werden, so dass weiterhin {Q} S {R} gilt
- Gegeben sei ein Programm S, das die Spezifikation a erfüllt
- Frage: Erfüllt S auch die Spezifikation b?

Spezifikation a

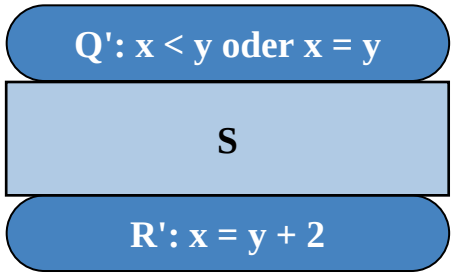


Spezifikation b

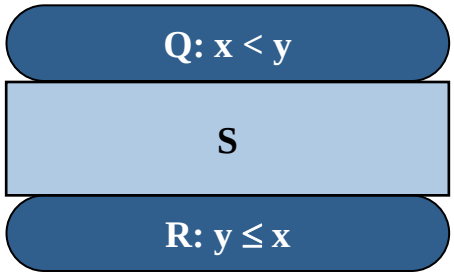


Der Hoare-Kalkül
Inferenzregeln

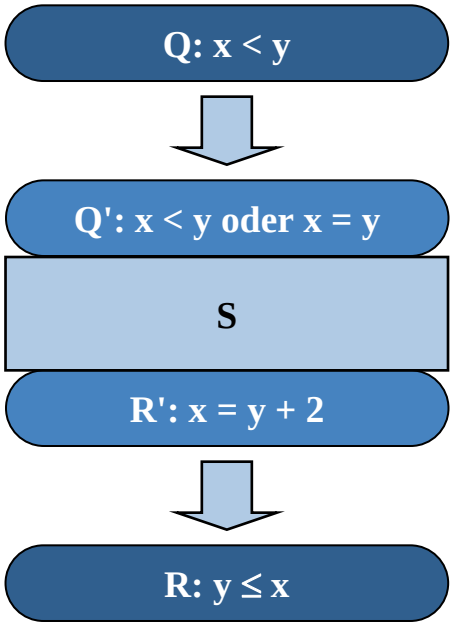
Spezifikation a



Spezifikation b



Anwendung der Konsequenz-Regel



»schärfere« Vorbedingung

»schwächere« Nachbedingung

- Die Antwort lautet ja, denn es gelten folgende Implikationen
  - $x < y \Rightarrow x < y$ oder $x = y$ (Q „ist schärfer“ als Q')
  - $x = y + 2 \Rightarrow y \leq x$ (R' „ist schärfer“ als R)

Der Hoare-Kalkül
Inferenzregeln

- Schwächen und Verschärfen von Bedingungen
  - Arbeitet man sich *vorwärts* durch ein Programm, dann darf man Bedingungen schwächen
    - Durch Hinzufügen eines beliebigen Terms mit *oder*-Verknüpfung
    - Durch Weglassen eines vorhandenen, *und*-verknüpften Terms
  - Arbeitet man sich *rückwärts* durch ein Programm, dann darf man Bedingungen verschärfen
    - Durch Hinzufügen eines beliebigen Term durch *und*-Verknüpfung
    - Durch Weglassen eines vorhandenen *oder*-verknüpften Terms

- Hoare notiert diese Regeln in der Form If H1 and H2 then H. Durchaus üblich ist aber auch die im Weiteren verwendete Schreibweise

$$\frac{H1,\ H2}{H}$$

- Die Inferenzregel A1. $\dfrac{\{P\}\ S\ \{Q\},\ Q \Rightarrow R}{\{P\}\ S\ \{R\}}$ bringt zum Ausdruck, dass
  - P die Vorbedingung von S bezüglich R ist,
  - falls P Vorbedingung von S bezüglich Q ist und
  - Q die Zusicherung R impliziert.
  - Jede Zusicherung R, die schwächer als eine Nachbedingung Q ist, kann ebenfalls als Nachbedingung verwendet werden.

$Q \Rightarrow R$ bedeutet, dass aus R aus Q folgt (alternativ: $Q \supset R$) , d.h. wenn Q wahr ist, so ist auch R wahr.

Beispiel: $(x>1) \Rightarrow (x>0)$; $(x>0)$ ist eine schwächere Bedingung als $(x>1)$

- Die dazu symmetrische Regel

$$\text{A2.}\ \frac{\{Q\}\ S\ \{R\},\ P \Rightarrow Q}{\{P\}\ S\ \{R\}}$$

lässt jede Zusicherung P, die stärker ist als eine Vorbedingung Q, als Vorbedingung zu

- Die Inferenzregeln A1 und A2 können zusammenfassend folgendermaßen dargestellt werden

$$\text{A1/2.}\ \frac{Q \Rightarrow Q',\ \{Q'\}\ S\ \{R'\},\ R' \Rightarrow R}{\{Q\}\ S\ \{R\}}$$

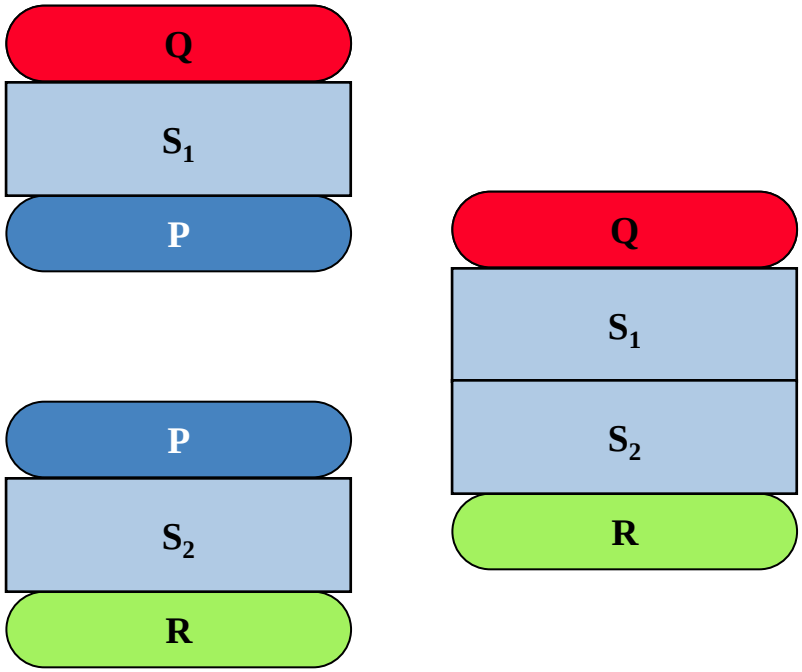
Der Hoare-Kalkül
Kompositonsregel

- Kompositionsregel
  - Zwei Programmstücke S1 und S2 können zu einem Programmstück S1 ; S2 zusammengesetzt werden, wenn die Nachbedingung von S1 mit der Vorbedingung von S2 identisch ist.

$$\text{A3.}\quad \frac{\{Q\}\ S1\ \{P\},\ \{P\}\ S2\ \{R\}}{\{Q\}\ S1\ ;\ S2\ \{R\}}$$

- Die Kompositionsregel gestattet die Behandlung von Anweisungssequenzen.

Der Hoare-Kalkül
Kompositonsregel



- Die Nachbedingung von S1 muss mit der Vorbedingung von S2 identisch sein.
- Dies kann ggf. durch Anwendung der Inferenzregeln A1 bzw. A2 erreicht werden.

Der Hoare-Kalkül
if-Regel

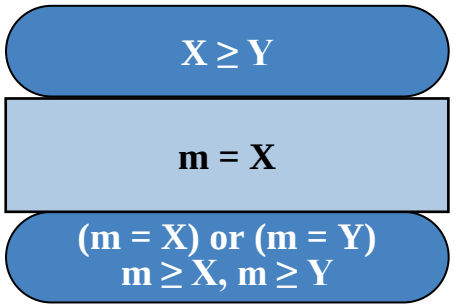
- Gibt an, unter welchen Voraussetzungen 2 Programmstücke S_1 und S_2 und eine Bedingung B zu einer zweiseitigen Auswahl mit der Vorbedingung Q und der Nachbedingung R zusammengesetzt werden können.

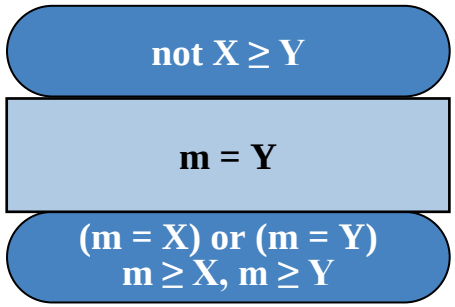
$$\text{A4.}\quad \frac{\{Q \text{ and } B\}\ S_1\ \{R\},\ \{Q \text{ and not } B\}\ S_2\ \{R\}}{\{Q\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{R\}}$$

- Gelten {Q and B} S_1 {R} und {Q and not B} S_2 {R}, dann können die Programme S_1 und S_2 zu einer if-Anweisung {Q} `if` B `then` S_1 `else` S_2 {R} zusammengesetzt werden

Der Hoare-Kalkül if-Regel

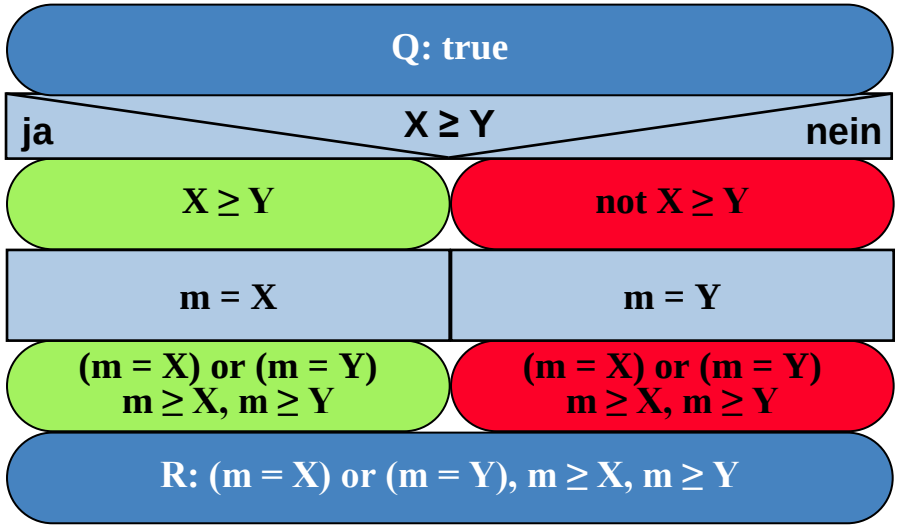
- Beispiel Maximum
  - Spezifikation
    - {Q : true} S {R : (m=X) or (m=Y), m ≥ X, m ≥ Y}
  - Das Maximum ist X oder Y
    - Das Maximum ist X, wenn X >= Y gilt
    - Das Maximum ist Y, wenn not X >= Y gilt





Der Hoare-Kalkül
if-Regel

- Wählt man als Bedingung B: X >= Y und als Vorbedingung Q : true, dann sind die Voraussetzungen der if-Regel erfüllt und die beiden Anweisungen können zu einer if-Anweisung zusammengesetzt werden.



X, Y fest

Der Hoare-Kalkül while-Regel

- Regel für abweisende Schleifen, also Schleifen der Form WHILE B DO S END
  - B ist die Schleifenbedingung, die vor Ausführung des Schleifenkörpers S und nach jeder Ausführung des Schleifenkörpers evaluiert wird
  - Besitzt B den Wert wahr, so wird ein Schleifendurchlauf ausgeführt
  - Andernfalls wird der Schleifenkörper nicht ausgeführt und mit der auf den Schleifenkörper folgenden Anweisung fortgefahren

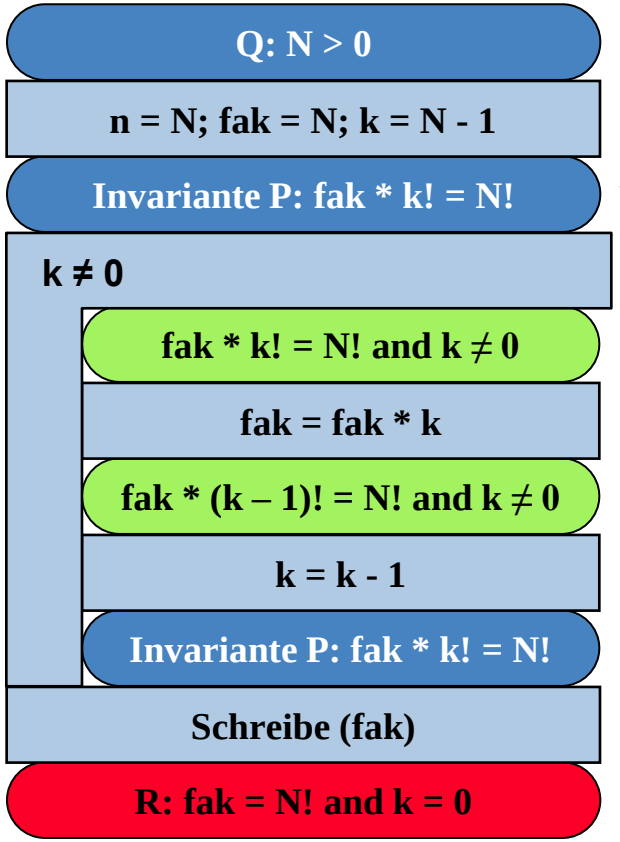
Der Hoare-Kalkül while-Regel

- Eine entscheidende Bedeutung bei der Verifikation von Schleifen besitzt die so genannte Schleifeninvariante.
  - Schleifeninvarianten sind Zusicherungen, die vor der Ausführung einer Schleife und nach jedem Schleifendurchlauf gültig sind, deren Wert also invariant wahr ist.
  - Invarianten gestatten eine geschlossene Beschreibung der Funktionalität einer Schleife und sind aus diesem Grund unverzichtbarer Bestandteil der Verifikation von Schleifen.
  - Man kann demnach schreiben

$$\text{A5.}\ \frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{WHILE B DO S END}\ \{P \wedge \neg B\}}$$

  - Ist $P \wedge B$ Vorbedingung von S bezüglich P, so gilt nach der Termination der Schleife weiterhin die Zusicherung P und B besitzt den Wahrheitswert falsch.

Der Hoare-Kalkül
while-Regel: Beispiel Fakultätsberechnung



N fest

Invariante gilt
vor der Schleife,
zu Beginn des Schleifenrumpfes,
am Ende des Schleifenrumpfes und
nach der Schleife

Für fak * k! kann auch fak * k *
(k – 1)! geschrieben werden,
da k > 0 ist

- Die linke Spalte nummeriert die Beweisschritte. Die rechte Spalte enthält die verwendete Regel und ggf. die Nummern der benutzten Beweisschritte.
- Die Spezifikation des Programms besteht aus einer Eingangszusicherung und einer Ausgangszusicherung.
- Da die Eingangswerte keiner besonderen Einschränkung *(War bei der Verifikation des gleichen Programms zu Beginn dieses Kapitels anders: Warum funktioniert der Beweis hier dennoch?)* unterliegen, ist die Eingangszusicherung die boolesche Konstante TRUE.
- Die Ausgangszusicherung ist

$$\text{Dividend} = \text{Rest} + \text{Quotient} * \text{Divisor} \wedge \neg (\text{Divisor} \leq \text{Rest})$$

Der Hoare-Kalkül
Beweis des Divisionsprogramms mit dem Hoare-Kalkül

1	*{TRUE ⇒ Dividend = Dividend + 0 * Divisor}*	lemma
2	*{Dividend = Dividend + 0 * Divisor}* **Rest = Dividend** *{Dividend = Rest + 0 * Divisor}*	A0. Zuweisungsaxiom
3	*{Dividend = Rest + 0 * Divisor}* **Quotient = 0** *{Dividend = Rest + Quotient * Divisor}*	A0. Zuweisungsaxiom
4	*{TRUE}* **Rest = Dividend** *{Dividend = Rest + 0 * Divisor}*	A2. (1, 2) Inferenzregel
5	*{TRUE}* **Rest = Dividend; Quotient = 0** *{Dividend = Rest + Quotient * Divisor}*	A3. (7, 8) Kompositionsregel
6	*{Dividend = Rest + Quotient * Divisor ∧ Divisor ≤ Rest ⇒ Dividend = (Rest - Divisor) + (Quotient + 1) * Divisor}*	lemma
7	*{Dividend = (Rest - Divisor) + (Quotient + 1) * Divisor}* **Rest = Rest – Divisor** *{Dividend = Rest + (Quotient + 1) * Divisor}*	A0. Zuweisungsaxiom
8	*{Dividend = Rest + (Quotient + 1) * Divisor}* **Quotient = Quotient + 1** *{Dividend = Rest + Quotient * Divisor}*	A0. Zuweisungsaxiom
9	*{Dividend = (Rest - Divisor) + (Quotient + 1) * Divisor}* **Rest = Rest - Divisor; Quotient = Quotient + 1** *{Dividend = Rest + Quotient * Divisor}*	A3. (7, 8) Kompositionsregel
10	*{Dividend = Rest + Quotient * Divisor ∧ Divisor ≤ Rest}* **Rest = Rest - Divisor; Quotient = Quotient + 1** *{Dividend = Rest + Quotient * Divisor}*	A2. (6, 9) Inferenzregel
11	*{Dividend = Rest + Quotient*Divisor}* **WHILE Divisor ≤ Rest DO Rest = Rest - Divisor; Quotient = Quotient + 1; END** *{¬ (Divisor ≤ Rest) ∧ Dividend = Rest + Quotient * Divisor}*	A5. (10) While-Regel
12	*{TRUE}* **Rest = Dividend; Quotient = 0; WHILE Divisor ≤ Rest Do Rest = Rest – Divisor; Quotient = Quotient + 1; END** *{¬ (Divisor ≤ Rest) ∧ Dividend = Rest + Quotient * Divisor}*	A3. (5, 11) Kompositionsregel

Totale Korrektheit
Termination von Schleifen

- Trotz des Fehlens einer allgemeinen algorithmischen Vorgehensweise für den Nachweis der Termination, kann für viele Programme die Termination gezeigt werden.
- Eine übliche Methode ist die Benutzung von wohlgeordneten Mengen. Jede nicht leere Teilmenge einer wohlgeordneten Menge besitzt ein kleinstes Element. Es sind demnach keine unendlich abnehmenden Sequenzen möglich.
- Man ordnet nun jedem Schleifendurchlauf eine so genannte Terminationsfunktion zu, deren Wertebereich in einer wohlgeordneten Menge W liegt.
- Kann gezeigt werden, dass die W-Funktion nach jedem Schleifendurchlauf einen geringeren Wert liefert, als vor der Abarbeitung, so bilden die Werte der W-Funktion während der Programmausführung eine streng monoton fallende Folge. Da in einer wohlgeordneten Menge ein kleinstes Element existiert, sind unter bestimmten Voraussetzungen keine unendlich fallenden Sequenzen möglich. Daraus folgt unmittelbar, dass das Programm terminiert.

Totale Korrektheit
Termination von Schleifen

- Ein Spezialfall der geforderten W-Funktion ist die sogenannte Terminationsfunktion t, die die Werte der Programmvariablen in die Menge der nichtnegativen ganzen Zahlen abbildet und deren Wert sich bei jedem Schleifendurchlauf reduziert.
- Beispiel
  - In dem Divisionsprogramm sind alle beteiligten Variablen ganzzahlig.
  - Es gilt Divisor > 0, also Divisor ≥ 1
  - Es gilt Dividend ≥ 0 und daher auch Rest ≥ 0
  - In jedem Schleifendurchlauf wird Rest um Divisor reduziert, gleichzeitig gilt aber Rest ≥ 0; daraus folgt: Die Schleife terminiert; die Terminationsfunktion ist t = Rest.

- Intuitive Erklärung
  - Wiederholungsbedingung B darf nach einer endlichen Anzahl von Schleifendurchläufen nicht mehr erfüllt sein.
  - Prüfung der Termination
    - Terminationsfunktion t, die die Programmzustände auf ganze Zahlen abbildet
  - Der ganzzahlige Wert der Terminationsfunktion t muss bei jedem Schleifendurchlauf, z. B.

    1 um mindestens 1 kleiner werden

    2 stets positiv bleiben
  - Existiert eine solche Terminationsfunktion, dann muss die Schleife nach einer endlichen Anzahl von Durchläufen terminieren.

Totale Korrektheit
Termination von Schleifen

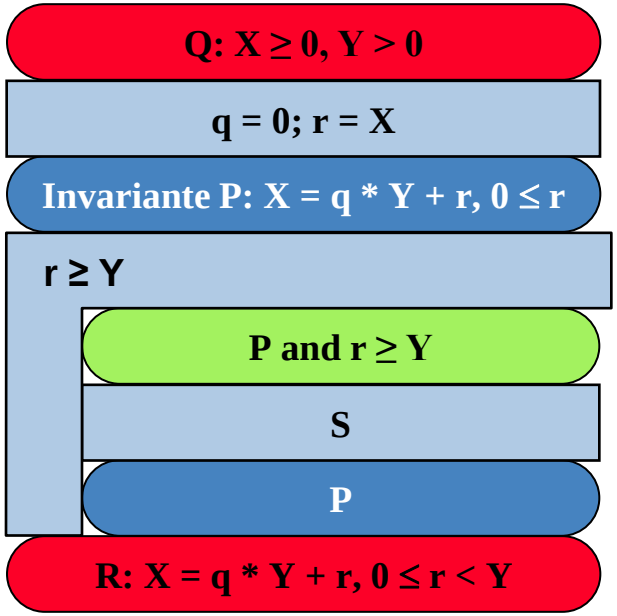
- Die Schleifenregel des Hoare-Kalküls kann entsprechend angepasst werden

$$\text{A5*.}\ \frac{\{P \wedge B\}\ S\ \{P\},\ \{P \wedge B \wedge t = z\}\ S\ \{t < z\},\ P \Rightarrow t \geq 0}{\{P\}\ \text{WHILE B DO S END}\ \{P \wedge \neg B\}}$$

- z muss für den betrachteten Programmausschnitt (die Schleife) konstant sein: Falls vor Ausführung des Schleifenrumpfs S gilt: t = z, so gilt anschließend t< z, d.h. der Wert der Terminationsfunktion wird geringer.
- Aus der Gültigkeit der Schleifeninvariante P muss folgen, dass auch $t \geq 0$ gilt.

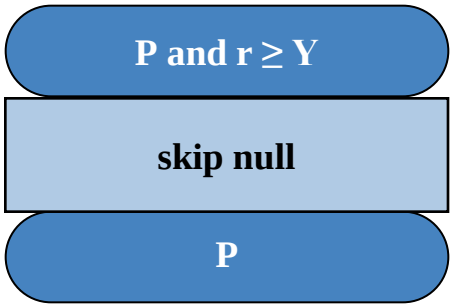
- Beispiel Divisionsroutine: z = Dividend, t = Rest

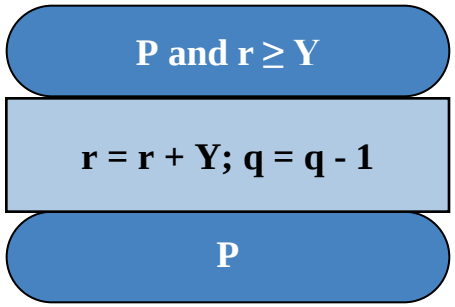
- Beispiel Divisionsprogramm
  - Es soll der ganzzahlige Quotient q = (X / Y) und der Rest r = (X mod Y) zweier ganzer Zahlen X und Y unter ausschließlicher Verwendung von Addition und Subtraktion berechnet werden.

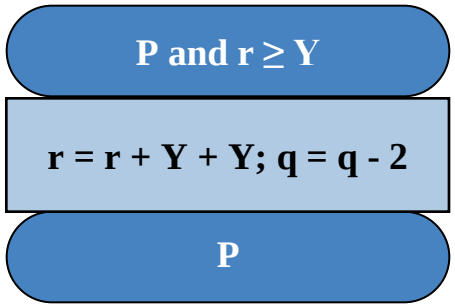


Totale Korrektheit
Termination von Schleifen

- In dem gezeigten Programmskelett fehlt nur noch der Schleifenrumpf S.
- Dieses Programm ist aufgrund der `while`-Regel für alle Schleifenrümpfe S mit der Vorbedingung *P and r >= Y* und der Nachbedingung P partiell korrekt.
- Es gibt verschiedene Programmstücke, die diese Bedingungen erfüllen und für den Schleifenrumpf S eingesetzt werden können.







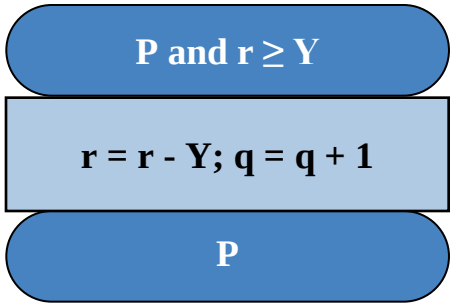
Totale Korrektheit
Termination von Schleifen

- Keiner dieser Schleifenrümpfe führt zur Termination des Gesamtprogramms.
  - Keines der Programme macht „einen Schritt näher zur Termination“, d.h. zur Erfüllung der Abbruchbedingung r < Y.
- Vor dem Schleifenrumpf gilt r >= Y und Y ist konstant.
  - In endlicher Anzahl von Schleifendurchläufen soll r < Y erreicht werden.
  - Daher muss der Wert von r im Schleifenrumpf kleiner werden.
- Aufgabe des Schleifenrumpfes
  - »Verkleinern von r unter Invarianz von P«

Totale Korrektheit
Termination von Schleifen

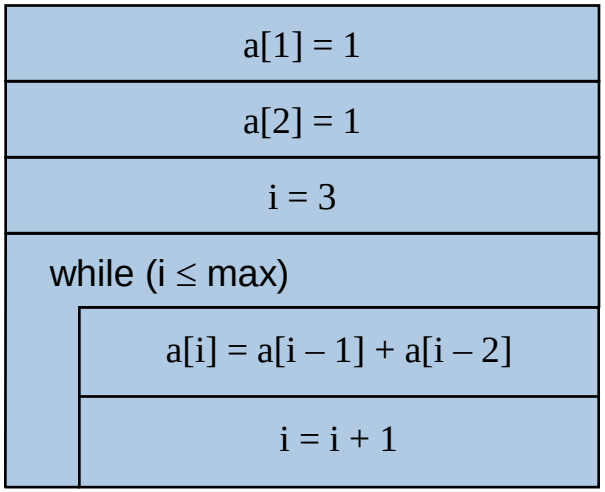
- Wird r um Y verkleinert, muss q um 1 erhöht werden, damit *P: X = q * Y + r and 0 <= r* invariant bleibt.
- Der nebenstehende Schleifenrumpf führt nach endlicher Anzahl von Schritten zu einem Wert r < Y und damit zur Termination des Programms.
- Der Wert von r wird in jedem Schritt um einen konstanten Betrag Y (laut Vorbedingung gilt Y > 0) kleiner, bleibt aber stets positiv (laut Invariante: 0 <= r).

➔ **das Programm muss terminieren**



Umgang mit strukturierten Datentypen: Quantoren

- Bisher (Divisionsprogramm, usw.) war es möglich, alle Zusicherungen in Aussagenlogik zu schreiben. Die Ursache ist die ausschließliche Verwendung einfacher Datentypen
- Für das angegebene Beispiel funktioniert dies nicht mehr

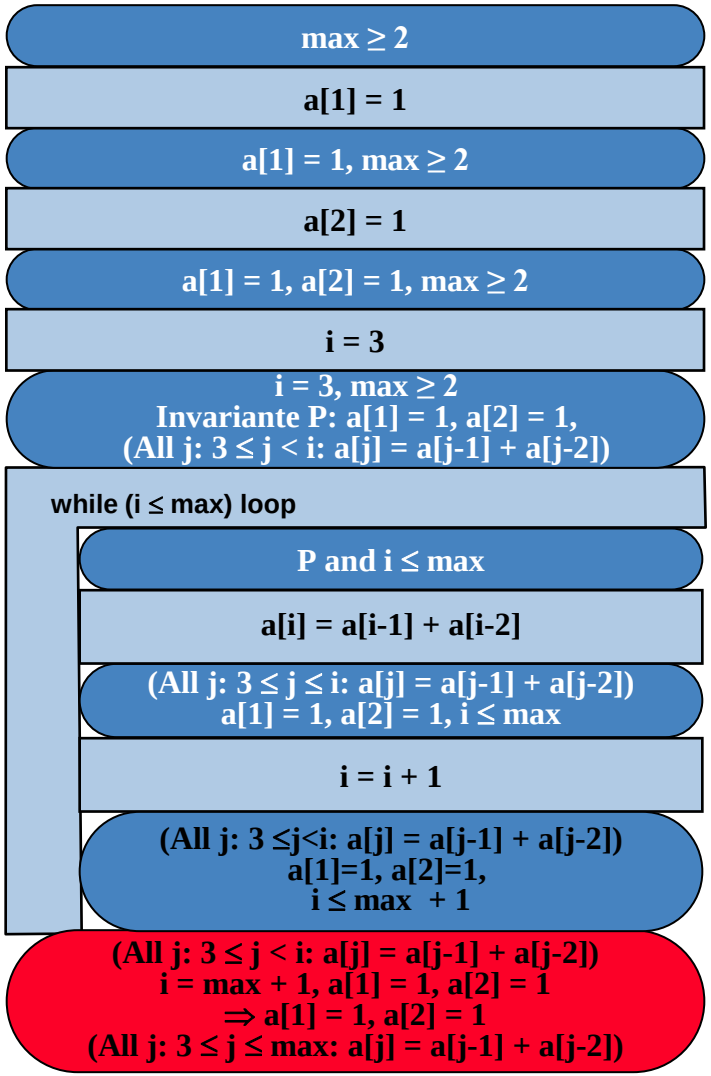


- Die Routine soll einem Feld (array) a mit dem Indexbereich 1 bis max, mit max >= 2 folgende Werte zuweisen.
- a(1) und a(2) erhalten den Wert 1. Alle Feldelemente, deren Index größer als zwei ist, erhalten die Summe der Werte der zwei vorausgehenden Feldelemente (d.h. a(3) = 2, a(4) = 3, a(5) = 5, usw., sog. Fibonacci-Zahlen)
- Um diesen Sachverhalt beschreiben zu können, benötigen wir Ausdrucksmöglichkeiten der Art:
  - Für alle Feldelemente gilt ... (sogenannter Allquantor)
  - Es gibt mindestens ein Feldelement, für das gilt ... (sogenannter Existenzquantor)
- Quantoren sind in Aussagenlogik nicht verfügbar.
- Eine um Quantoren erweiterte Aussagenlogik heißt Prädikatenlogik erster Ordnung.

- Ein Feld sei zwischen den Indizes 0 und n aufsteigend sortiert
  - All j: 0 ≤ j < n: a[j] ≤ a[j+1]
  - $\forall_{j|0\le j<n} a[j] \le a[j+1]$
  - $\bigwedge_{j|0\le j<n} a[j] \le a[j+1]$

- Ein Feld besitzt mindestens ein positives Element zwischen den Indizes 0 und n

  - Ex j: 0 ≤ j ≤ n: a[j] > 0
  - $\exists_{j|0\le j\le n} a[j] > 0$
  - $\bigvee_{j|0\le j\le n} a[j] < 0$



Formale Verifikation
Algebraische Spezifikationen

Eine geeignete Spezifikationsform für Datenabstraktionen ist die algebraische Spezifikation.

- Datenabstraktion
  - Datenstruktur (internes Gedächtnis)
  - Anzahl von Zugriffsoperationen auf dieses Gedächtnis
- Die einzige Zugriffsmöglichkeit zum Gedächtnis bilden die zur Datenabstraktion gehörenden Zugriffsoperationen.
- Algebra: Menge von Elementen, zu der eine Anzahl von Operationen, die auf die Elemente der Menge anwendbar sind, gehören
- Die Spezifikation beschreibt die Datenobjekte und die Auswirkungen der Operationen.

Algebraische Spezifikationen: Beispiel

- Q sei eine Variable vom Typ Warteschlange für Integer und i eine ganze Zahl
  - Anhängen (Q, i) fügt i am Warteschlangenende von Q an; Ergebnistyp Warteschlange
  - Entfernen (Q) entfernt ein Element vom Kopf der Warteschlange; Ergebnistyp Warteschlange
  - IstNeu (Q) stellt fest, ob die Warteschlange leer ist. Das Resultat ist ein boolescher Wert.
  - Length (Q) liefert die aktuelle Anzahl der Elemente in der Warteschlange Q; Ergebnistyp Integer, nicht negativ

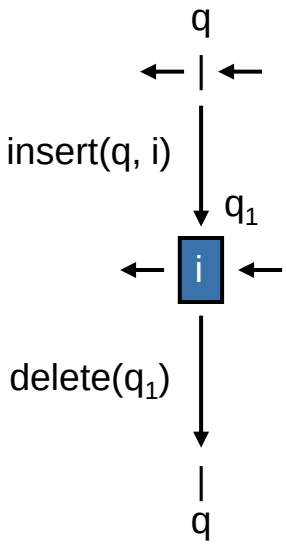
Algebraische Spezifikationen: Beispiel

- Mögliche Axiome in der Spezifikation sind
  - Entfernen (Anhängen (Q, i)) = IF IstNeu (Q) THEN Q ELSE Anhängen (Entfernen (Q), i) END
  - IF IstNeu(Q) THEN Length (Q) = 0 ELSE Length (Q) > 0
  - Length (Anhängen (Q, i)) = Length (Q) + 1
  - Length (Entfernen (Q)) = IF IstNeu(Q) THEN error ELSE Length (Q) – 1
  - IstNeu(Q) = (Length(Q) = 0)
  - ...

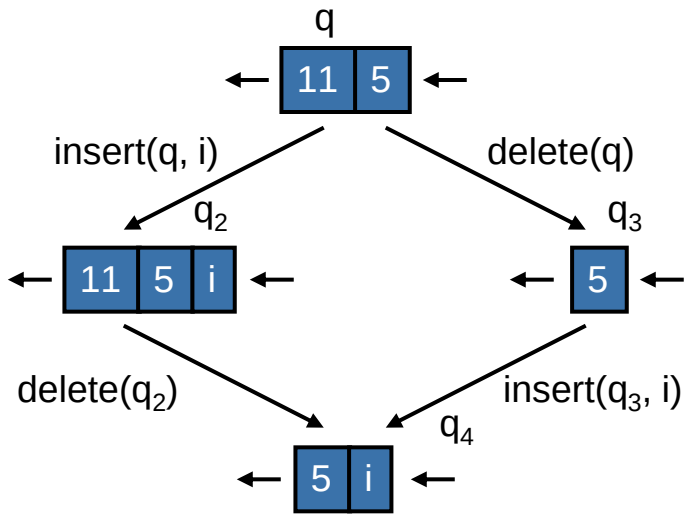
- Darstellung von
  - Entfernen (Anhängen (Q, i)) = IF IstNeu (Q) THEN Q ELSE Anhängen (Entfernen (Q), i) END

  Bemerkung: Entfernen = delete; Anhängen = insert

Fall 1: Leere Warteschlange



Fall 2: Nicht leere Warteschlange



1. Entfernen (Anhängen (Q, i)) = IF IstNeu (Q) THEN Q ELSE Anhängen (Entfernen (Q), i)
2. IF IstNeu(Q) THEN Length (Q) = 0 ELSE Length (Q) > 0
3. Length (Anhängen (Q, i)) = Length (Q) + 1
4. Length (Entfernen (Q)) = IF IstNeu(Q) THEN error ELSE Length (Q) – 1
5. IstNeu(Q) = (Length(Q) = 0)

Ersetzen von Q durch *Anhängen (Q,i)* in 4

- Length (Entfernen (Anhängen (Q, i))) = IF IstNeu(Anhängen (Q, i)) THEN error ELSE Length (Anhängen (Q, i)) – 1

Anwenden von 5

- Length (Entfernen (Anhängen (Q, i))) = IF (Length (Anhängen (Q, i)) = 0) THEN error ELSE Length (Anhängen (Q, i)) – 1

Anwenden von 3

- Length (Entfernen (Anhängen (Q, i))) = IF ((Length (Q) +1) = 0) THEN error ELSE Length (Anhängen (Q, i)) – 1 = Length (Anhängen(Q,i)) – 1

logisch Falsch

Anwenden von 3

- Length (Entfernen (Anhängen (Q, i))) = Length (Q) + 1 – 1 = Length (Q)

Anwenden von 1

- Length (Entfernen (Anhängen (Q, i))) = IF IstNeu (Q) THEN Length (Q) ELSE Length (Anhängen (Entfernen (Q), i)) = Length (Q)

Anwenden von 3

- IF IstNeu (Q) THEN Length (Q) ELSE Length (Entfernen (Q)) + 1 = Length (Q)

Anwenden von 4

logisch Falsch

- IF IstNeu (Q) THEN Length (Q) ELSE (IF IstNeu(Q) THEN error ELSE (Length (Q) – 1) + 1 = Length (Q)

⇒

- IF IstNeu (Q) THEN Length (Q) ELSE Length (Q) – 1 + 1 = Length (Q)

⇒

- IF IstNeu (Q) THEN Length (Q) ELSE Length (Q) = Length (Q)

⇒

- Length (Q) = Length (Q) (wahre Aussage)

Literatur

- Floyd R.W., Assigning meanings to Programs, in: Proceedings of the American Mathematical Society Symposium in Applied Mathematics, Vol. 19, 1967, pp. 19-32
- Hoare C.A.R., Proof of a Program: FIND, in: Communications of the ACM, Vol. 14, No. 1, January 1971, pp. 39-45
- Liggesmeyer P., Software-Qualität, 2. Aufl., Spektrum-Verlag Heidelberg, 2009
- Logrippo L., Melanchuk T., Du Wors R.J., The Algebraic Specification Language LOTOS: An Industrial Experience, in: Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development, Napa, May 1990, Software Engineering Notes, Vol. 15, No. 4, September 1990, pp. 59-66