Vorlesungsnotiz 6
Die `while`-Sprache als Kern imperativer Sprachen

Ein wesentlicher **Vorteil funktionaler Sprachen** ist ihre **„referentielle Transparenz“**: Der Wert eines Funktionsaufrufs ist eindeutig durch die aufgerufene Funktion und die Werte der aktuellen Parameter bestimmt. Da Konstanten nullstellige und Operatoren zweistellige Funktionen sind, überträgt sich diese Eigenschaft auf alle Ausdrücke. Wegen der referentiellen Transparenz lassen sich Programmeigenschaften verhältnismäßig leicht mit mathematischen Beweisen (Induktion) nachweisen.

Im Unterschied zu funktionalen Sprachen verfügen **imperative Sprachen** über ein „wiederbeschreibbares Gedächtnis“: Neben Konstanten und Parametern gibt es benannte **Variablen**, in denen man Werte eines bestimmten Typs zwischenspeichern kann. Ein imperatives Programm wird in einer **Umgebung** ausgeführt, die aus einer Menge solcher Variablen besteht. Die Möglichkeit, sich auf diese Weise einen komplexen „Zustand“ zu merken, führt oft zu einer natürlicheren Programmstruktur, wenn die Programme größer werden. Außerdem kann man große Datenstrukturen (z.B. Suchbäume) effizient komponentenweise aktualisieren. Andererseits zerstört die Verwendung von Variablen die referentiellen Transparenz von Programmen.

Im Vergleich: *Die Vorteile imperativer Sprachen sind die Nachteile funktionaler Sprachen und umgekehrt die Vorteile funktionaler Sprachen die Nachteile imperativer Sprachen .*

Bei der schon in Vorlesungsnotiz `1` skizzierten *von-Neumann-Architektur* heutiger Computer besteht der **Arbeitsspeicher** aus einer linearen Folge gleich großer **Speicherzellen**, die (von `0` an) durchnummeriert sind. Diese Nummern heißen **Adressen**, weil man über sie die zugehörigen Speicherzellen erreicht. Der Arbeitsspeicher enthält zu jedem Zeitpunkt die gerade ausgeführten Programme und ihre Daten. Bei der Übersetzung eines Programms werden nicht nur die Befehle, sondern auch die Variablen eines Programms in geeigneter Weise auf Speicherzellen abgebildet. Wie diese **Speicherverwaltung** im Detail funktioniert, wird in der Vorlesung „Einführung in die Informatik III“ beschrieben.

Im Folgenden definieren wir die wesentlichen Begriffe und Grundoperationen.

Eine **Programmvariable** (kurz: „Variable“) ist ein Tripel `(<name>,<typ>,<wert>)`. Wenn – wie im Beispiel unten – alle Variablen den gleichen Typ (Integer) enthalten, begnügen wir uns mit der Angabe des Paars `(<name>,<wert>)`. Eine **Umgebung** ist eine Menge von Programmvariablen, in der je zwei verschiedene Tupel auch verschiedene Namen haben: Variablen sind innerhalb einer Umgebung durch ihren Namen eindeutig bestimmt. Der **Wert** einer Variablen ist entweder ein Wert des **Typs** der Variablen (bei einer Integer-Variablen ein Integer-Wert, bei einer Suchbaum-Variablen ein Suchbaum, etc.) oder er ist undefiniert. Wenn eine Variable noch keinen Wert erhalten hat, ist ihr Wert **„undefiniert“**, kurz bezeichnet durch „⊥“.

Ist „`sp`“ eine Umgebung und „`x`“ der Name einer darin enthaltenen Variablen, dann bezeichnet „**`sp(x)`**“ den (aktuellen) Wert der Variablen „`x`“ aus „`sp`“:

`sp(x) = w,` falls `(x,w) ∈ sp` oder `(x,t,w) ∈ sp` für irgendein `t`

`sp(x) = ⊥` sonst

Grundlegende **Operationen** auf einer Umgebung und ihren Variablen sind:

`sp ∪ {(x, ...)}`: Hinzufügen einer neuen Variablen `x` zur Umgebung `sp` durch **Deklaration**. Dabei sind mindestens der Name und der Typ festzulegen, bei einer Deklaration mit **Initialisierung** auch der Wert der Variablen.

`[[<ausdruck>]]sp`: Beschreibt den **Wert**, der sich ergibt, wenn man alle Namen im Ausdruck durch ihre aktuellen Werte in `sp` ersetzt und dann den Ausdruck „ausrechnet".

`sp[x->w]`: Ergibt die Umgebung `sp'`, die sich von `sp` genau dadurch unterscheidet, daß in dem Variablentupel mit Namen `x` der Wert der Variablen durch `[[w]]sp` ersetzt wird. Diese Operation heißt **„update"**.

Die `while`-Sprache

Mit geringfügigen, syntaktischen Abweichungen kommt die nun betrachtete Sprache als Teilsprache in praktisch allen imperativen Programmiersprachen vor. Ein **Beispiel** aus dieser Sprache ist die Anweisungsfolge **DR** („Division mit Rest"):

```
d := 0;
while x >= y do
   d := (d+1);
   x := (x-y);
od
```

Anweisungen der Form „`n := a`" heißen **Zuweisungen** und beschreiben das Update der Variablen `n` mit dem aktuellen Wert des Ausdrucks `a`, formal: `sp[n->[[a]]sp]`.

DR beginnt mit einer Zuweisung an `d`. Die restlichen vier Zeilen enthalten eine **`while`-Schleife**, deren **Rumpf** (die beiden Zuweisungen zwischen `do` und `od`) solange wiederholt ausgeführt wird, bis einmal die **Schleifenbedingung** „`x >= y`" verletzt ist. Dann endet die Schleife und DR insgesamt.

Führt man ausgehend von der Umgebung `sp = {(x,8),(y,3),(d,⊥)}` eine **Handsimulation** des Ablaufs von DR durch, dann verändert sich die Umgebung durch Zuweisungen wie folgt (erste Spalte Ausgangswerte, danach nur Veränderungen fett):

x	**8**	8	8	**5**	5	**2**
y	**3**	3	3	3	3	3
d	⊥	**0**	**1**	1	**2**	2

In der letzten Spalte ist `x = 2` und `y = 3`, damit die Schleifenbedingung „`x >= y`" nicht mehr erfüllt: DR terminiert.

Was passiert bei Handsimulation ausgehend von `sp = {(x,8),(y,-3),(d,⊥)}`?

Im nächsten Abschnitt beschreiben wir die **Syntax** der `while`-Sprache formal durch eine Grammatik. Der Abschnitt **operationale Semantik** definiert (in Form eines Ersetzungssystems) eine hypothetische Maschine, die `while`-Programme ausführt. Für **Beweise** ist diese Semantik wenig geeignet, weil zu detailbehaftet. Der Rest der Notiz handelt daher von einer zur Programmverifikation besser geeigneten **axiomatischen Semantik** in Form des **Hoare-Kalküls**.

Syntax der `while`-Sprache

Sätze der **while**-Sprache sind **Anweisungsfolgen** mit vier Arten von Anweisungen:

```
<Anweisungsfolge> ::=
      <Anweisung> | <Anweisungsfolge> ; <Anweisung>

<Anweisung> ::=
      <leereAnweisung>    | <Zuweisung> |
      <bedingteAnweisung> | <Schleife>

<leereAnweisung> ::= skip

<Zuweisung> ::=
      <Name> := <Ausdruck>

<Ausdruck> ::=
      <Name> | <Zahl> | (<Ausdruck> <Op> <Ausdruck>)

<Name> ::= ... (nicht ausgeführt)

<Zahl> ::= ... (nicht ausgeführt)

<Op> ::= + | - | div | mod

<bedingteAnweisung> ::=
      if <Bedingung>
      then <Anweisungsfolge>
      else <Anweisungsfolge>
      fi

<Bedingung> ::=
      true | false | <Ausdruck> <Rop> <Ausdruck>

<Rop> ::= <= | < | = | /= | >= | >

<Schleife> ::=
      while <Bedingung> do <Anweisungsfolge> od
```

Als Übung wird empfohlen, den kompletten Syntaxbaum zu DR auf ein großes Blatt Papier zu zeichnen!

Operationale Semantik

Wir definieren die **abstrakte Maschine** $M = (Z, \Delta, z_0)$ als Ersetzungssystem mit

- Z für die Menge der **Zustände**, die M annehmen kann
- z_0 für den **Anfangszustand** von M
- $\Delta : Z \to Z$ für die im Anhang definierte **Zustandsübergangsfunktion**

Jeder **Zustand** $z \in Z$ hat drei Komponenten (sp, w, p)

- **sp** ist die aktuelle Umgebung
- **w** ist der Wertekeller (oberes Ende rechts)
- **p** ist der Programmkeller (oberes Ende links)

Im **Anfangszustand** z_0 ist definitionsgemäß der Wertekeller leer und der Programmkeller enthält das auszuführende Programm; in der ersten Komponente befindet sich die Anfangsumgebung sp_{anf}, d.h. die Menge der Variablen des Programms mit den Werten zu Beginn des Programmlaufs.

Ausgehend von diesem Anfangszustand wird das **Programm** von M **ausgeführt**, indem die Ersetzungsregeln Δ aus dem Anhang angewendet werden, so oft es geht. Man beachte, daß stets höchstens eine der Regeln anwendbar ist: Die linken Regelseiten unterscheiden sich meist im obersten Element des Programmkellers (also des als nächsten auszuführendem Befehls); wo das nicht der Fall ist (Regelpaare 13, 14 und 16, 17), unterscheiden sich die obersten Elemente des Wertekellers. Damit ist Δ (wie oben behauptet) eine Funktion!

Für $sp_{anf} = \{(x,8),(y,3),(d,\perp)\}$ ergeben sich bei der Bearbeitung der ersten Zuweisung `d := 0` des Programms DR die Schritte:

$(sp_{anf}, \varepsilon, DR)$
$= \quad (sp_{anf}, \varepsilon, d:=0;\text{while } x\geq y \text{ do } AF_2 \text{ od})$
$=11= \quad (sp_{anf}, \varepsilon, d:=0\bullet\text{while } x\geq y \text{ do } AF_2 \text{ od})$
$=9= \quad (sp_{anf}, \varepsilon, 0\bullet:=\bullet d\bullet\text{while } x\geq y \text{ do } AF_2 \text{ od})$
$=1= \quad (sp_{anf}, 0, :=\bullet d\bullet\text{while } x\geq y \text{ do } AF_2 \text{ od})$
$=10= \quad (sp_{anf}[d\text{->}0], \varepsilon, \text{while } x\geq y \text{ do } AF_2 \text{ od})$

Man sieht, daß selbst eine einfache Zuweisung eine Reihe von Abarbeitungsschritten erfordert. Andererseits sind die Bearbeitungsschritte so einfach, daß man sie offenbar von einer Maschine ausführen lassen kann – wir sprechen daher von einer „abstrakten Maschine".

Wir notieren die wiederholte Ausführung von Δ als Δ^*. Die vom Anfangszustand aus erreichbaren Zustände bilden die Menge $\Delta^*(sp_{anf}, \varepsilon, P)$. Weil Δ eine Funktion ist, bilden die Elemente dieser Menge eine Kette, die mit dem Anfangszustand beginnt und in der jedes Element höchstens einen Nachfolger besitzt. Wenn die abstrakte Maschine einen Zustand erreicht, in dem das Programm P „abgearbeitet" (d.h. leer) ist, dann terminiert sie, weil kein Übergang mehr möglich ist. Wesentlich an dem dann erreichten Endzustand $(sp_{end}, k, \varepsilon)$ ist die entstandene Umgebung $\mathbf{sp_{end}}$; der Wertekellerinhalt k dagegen wird als unwesentlich erachtet. Nicht jede Ausführung eines Programms terminiert. Schuld daran ist die Ersetzungsregel 16), bei der das Restprogramm im Programmkeller wächst (inhaltlich klar: hier wird die Schleife erneut durchlaufen). Alle anderen Regeln verkleinern entweder den Programmkeller oder sie lösen das oberste Programmkellerelement in seine Bestandteile auf.

Wir können nun formal beschreiben, was wir unter der „Bedeutung" eines `while`-Programms P verstehen wollen. Die **Bedeutung `[[P]] | SP -> SP`** von P ist eine Funktion, die einer Anfangsumgebung sp_{anf} eine Endumgebung sp_{end} zuordnet.

Es gilt:

$$[[P]]sp_{anf} =_{def} sp_{end}\ ,\ \text{falls}\ \ (sp_{end}, \ldots, \varepsilon) \in \Delta^*(sp_{anf}, \varepsilon, P)$$

Wenn die Ausführung eines Programms P, ausgehend von sp_{anf} nicht terminiert, ist $[[P]]sp_{anf}$ nicht definiert, die Funktion [[P]] also **partiell**. Wir machen die Funktion **total**, indem wir den Wert $\mathbf{sp_\perp}$ (**„terminiert nicht"**) einführen und festlegen:

$$[[P]]sp_{anf} =_{def} sp_\perp\ ,\ \text{falls P ausgehend von } sp_{anf} \text{ nicht terminiert}$$

[[P]] ist eine wohldefinierte Funktion, weil Δ Funktion ist und weil es zu Endzuständen $(sp_{end}, \ldots, \varepsilon)$ keine Nachfolgerzustände gibt.

Induktionsbeweis zu DR

Ziel: Es soll gezeigt werden, daß DR ausgehend von $sp_{anf}=\{(x,m),(y,n),(d,\perp)\}$ mit $m \geq 0$ und $n > 0$ zu m und n die ganzzahlige Division mit Rest durchführt.

Genauer gilt **(*)**:

1. Ausgehend von sp_{anf} terminiert DR mit einer Endumgebung sp_{end}.
2. $m = sp_{end}(d)*n + sp_{end}(x)$ und $0 \leq sp_{end}(x) < n$

Sei im Folgenden stets s_i die Umgebung unmittelbar nach dem i-ten Schleifendurchlauf bzw. unmittelbar vor dem (i+1)-ten Schleifendurchlauf.

Hilfreich ist die **Beobachtung**, daß DR den mit n vorbesetzten Wert von y niemals ändert, daß also $n = s_i(y)$ für alle i. Wir unterscheiden daher nicht zwischen den Bezeichnungen $s_i(y)$ und n.

Induktionsbehauptung (**): Nach dem i-ten (tatsächlich ausgeführten) Schleifendurchlauf gilt für den erreichten Zustand$(s_i, \varepsilon, \text{while } x \geq y \text{ do } AF_2 \text{ od})$, daß

$$m = s_i(d)*n + s_i(x) \text{ und } 0 \leq s_i(x).$$

Bevor wir die Induktionsbehauptung (**) nachweisen, zeigen wir, daß daraus das Ziel (*) unmittelbar folgt:

- Die Terminierung von DR ergibt sich aus der Beobachtung, daß das mit $m \geq 0$ initialisierte x in jedem Schleifendurchlauf um $n>0$ verringert, wegen (**) aber nie negativ wird.
- Sei j die Gesamtzahl der Schleifendurchläufe. Nach dem j-ten Schleifendurchlauf entsteht der Zustand $(s_j, \varepsilon, \text{while } x \geq y \text{ do } AF_2 \text{ od})$, der vermöge der Regeln 15), 6), 2), 2), 7), 17) übergeht in $(s_j, \varepsilon, \varepsilon)$. Also $s_j = sp_{end}$. Der Übergang mit der Regel 7) bedeutet, daß $s_j(x) < s_j(y)$, d.h. $sp_{end}(x) < sp_{end}(y) = n$. Die übrigen Aussagen in (*).2 folgen direkt aus der Induktionsbehauptung (**).

Damit ist die Behauptung (*) vollständig auf (**) zurückgeführt.

□

Nun zum Beweis von (**)!

Induktionsanfang (i = 0):

Wie eingangs ausführlich dargestellt, gelangt man vom Startzustand $(sp_{anf}, \varepsilon, DR)$ vermöge der Anwendung der Regeln 11), 9), 1) und 10) in den Zustand

$(s_0, \varepsilon, \text{while } x \geq y \text{ do } AF_2 \text{ od})$

mit $s_0 = sp_{anf}[d->0] = \{(x,m),(y,n),(d,0)\}$

Daraus folgt, daß wie behauptet:

$$m = 0*n + m = s_0(d)*n + s_0(x) \text{ und } 0 \leq m = s_0(x)$$

Induktionsvoraussetzung(IV):

Nach dem i-ten Schleifendurchlauf ist Zustand $(s_i, \varepsilon, \texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$ erreicht und es gilt $m = s_i(d)*n + s_i(x)$ und $0 \leq s_i(x)$.

Induktionsschritt (i → i+1):

Im (i+1)-ten Schleifendurchlauf werden folgende Zwischenzustände durchlaufen:

$(s_i, \varepsilon, \texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$

$\Rightarrow$ 15),2),2),7),16) $\Rightarrow$ *(Bedingung auswerten)*

$(s_i, \varepsilon, d:=(d+1);x:=(x-y)\bullet\texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$

$\Rightarrow$ 11) $\Rightarrow$

$(s_i, \varepsilon, d:=(d+1)\bullet x:=(x-y)\bullet\texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$

$\Rightarrow$ 9),3),2),1),4),10) $\Rightarrow$ *(Zuweisung* d:=(d+1) *auswerten)*

$(s_i[d->s_i(d)+1], \varepsilon, x:=(x-y)\bullet\texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$

$\Rightarrow$ 9),3),2),2),4),10) $\Rightarrow$ *(Zuweisung* x:=(x-y) *auswerten)*

$(s_i[d->s_i(d)+1][x-> s_i(x)-s_i(y)], \varepsilon, \texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$

$= (s_{i+1}, \varepsilon, \texttt{while } x \geq y \texttt{ do } AF_2 \texttt{ od})$

mit $s_{i+1} =_{def} s_i[d->s_i(d)+1][x-> s_i(x)-s_i(y)]$

$= \{(x, s_i(x)-s_i(y)), (y,n), (d, s_i(d)+1)\}$

Daraus folgt mit der Induktionsvoraussetzung:

$\mathbf{s_{i+1}(d)*n + s_{i+1}(x)}$

$= (s_i(d)+1)*n + s_i(x)-s_i(y)$

$= (s_i(d)+1)*n + s_i(x)-n$

$= s_i(d)*n + n + s_i(x)-n$

$\mathbf{= m}$ *(wegen (IV))*

Da der Schleifentest $s_i(x)>=s_i(y)$ zu true ausgewertet wurde, folgt auch

$\mathbf{s_{i+1}(x)} = s_i(x) - s_i(y) \geq \mathbf{0}$

□

Die letzten zwei Seiten belegen, daß Programmbeweise auf der Basis einer operationalen Semantik zwar möglich sind, aber schon bei extrem einfachen Programmen umfangreiche Argumentationsketten erfordern. Das liegt an der niedrigen Abstraktionsstufe: Einzelne Ausführungsschritte sind zwar genau beschrieben, leisten aber wenig. Der Beweis leidet unter der Detailfülle. Weil zu viele triviale Schritte behandelt werden müssen, nennt sie der Beweis nur und führt sie nicht aus. Das ist ein fehleranfälliges Vorgehen!

Die im Folgenden vorgestellte **axiomatische Semantik** ist besser geeignet für den **Beweis von Programmeigenschaften**, weil von unwesentlichen Details abstrahiert wird.

Axiomatische Semantik

Als „irrelevant“ für die Bedeutung eines Programms empfinden wir u.a. die Details der Auswertung arithmetischer Ausdrücke. Die abstrakte Maschine legt fest, daß Ausdrücke mit Hilfe eines Wertekellers auszuwerten sind und wie dabei genau vorzugehen ist.

Was ist im Unterschied dazu als „relevant“ anzusehen?

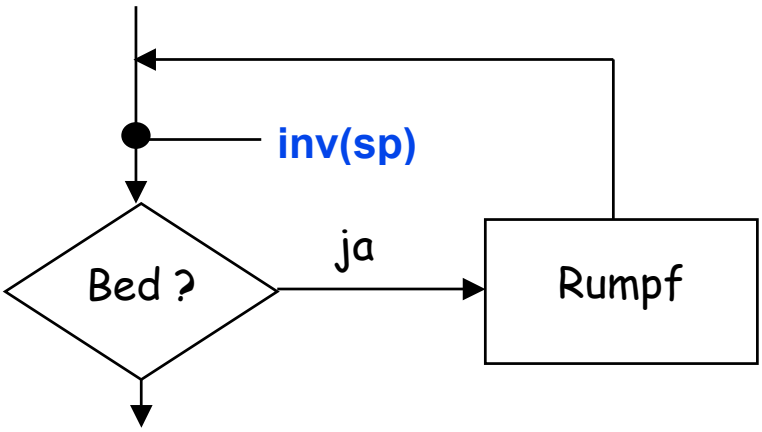
Beim Beweis von DR sind die Voraussetzung, die Behauptung (*), die Beobachtung zum Wert von `y` und die Induktionsbehauptung (**) i.w. Aussagen über Umgebungen, d.h. über Variablenwerte und die Beziehungen zwischen diesen Werten. Solche Aussagen nennt man bei Programmbeweisen **Zusicherungen**.

Zusicherungen sind mit bestimmten Stellen der Programmausführung verbunden:

- Am Anfang, also vor Ausführung eines Programms P steht die **Vorbedingung** `pre(sp)`, die Voraussetzungen für die „bestimmungsgemäße“ Verwendung von P beschreibt.
- Als die **Nachbedingung** eines Programms P bezeichnet man die Zusicherung `post(sp)`, die den Zustand nach Ausführung von P beschreibt.

Von Tony Hoare stammt die Idee, Zusicherungen mit **Stellen im Programmtext** zu verbinden, wobei eine solche Zusicherung sich auf die Stellen der Programmausführung bezieht, an denen die Textstelle erreicht wird. Insbesondere Zusicherungen innerhalb von Schleifen werden mehrfach erreicht und müssen dann immer erfüllt sein; man spricht von „Invarianten“.

- Mit einer Schleife verbunden ist die **Schleifeninvariante** `inv(sp)`, die vor Eintritt in die Schleife und am Ende jedes Schleifendurchlaufs erfüllt sein muß. Wie der nachstehende Ausschnitt aus einem Kontrollflußdiagramm zeigt, handelt es sich in Wirklichkeit um ein und dieselbe Stelle.



Man kann Programme mit Zusicherungen **„annotieren“**, um auszudrücken, was an welcher Stelle des Programms gelten soll. Damit ist zunächst nichts bewiesen, sondern nur formal beschrieben, was man über das Verhalten des Programms zu wissen glaubt.

Wenn man das Programm DR in diesem Sinn mit den aus dem Induktionsbeweis bekannten Zusicherungen annotiert, dann ergibt sich:

```
{pre: x = m  ∧  y = n  ∧  m ≥ 0  ∧  n > 0}
d := 0 ;
{inv: m = d*n + x  ∧  x ≥ 0  ∧  y = n  ∧  n > 0}
while x >= y do
   {m = d*n + x  ∧  x ≥ 0  ∧  y = n  ∧  n > 0  ∧  x ≥ y}
   d := (d+1);
   { ??? }
   x := (x-y)
   {inv : m = d*n + x  ∧  x ≥ 0  ∧  y = n  ∧  n > 0}
od
{post: m = d*n + x  ∧  x ≥ 0  ∧  x < n}
```

Man beachte, daß die Schleifeninvariante im Text an zwei Stellen identisch auftaucht: vor der Schleife und am Ende des Schleifenrumpfs. Außerdem erscheint sie noch am Anfang des Schleifenrumpfs, angereichert um die Schleifenbedingung. Was zwischen den beiden Zuweisungen des Schleifenrumpfs gilt, ist zunächst nicht klar.

Um tatsächlich Programmbeweise durchzuführen, bedienen wir uns des nach Tony Hoare benannten Hoare-Kalküls. Bausteine solcher Beweise sind sogenannte Hoare-Tripel oder **Hoare-Formeln** der Form:

$$\{Q(sp)\}\ \text{Anw}\ \{P(sp)\}$$

Die Formel besagt genau: „Wenn vor Ausführung der Anweisung Anw die Zusicherung $\{Q(sp)\}$ erfüllt ist *und* wenn Anw zusätzlich terminiert, dann ist nach Ausführung von Anw die Zusicherung $\{P(sp)\}$ erfüllt.

Eine solche Formel ist eine Behauptung, die wahr oder falsch sein kann. Wahre Aussagen über Programme lassen sich im Hoare-Kalkül herleiten. Die Gültigkeit der dort beschriebenen Axiome und Schlußregeln ist in der Literatur durch Rückführung auf andere Semantikbeschreibungen (wie obige operationale Semantik) nachgewiesen worden.

Der **Hoare-Kalkül** enthält zu jeder elementaren Anweisung ein **Axiomenschema** (manchmal als „Axiom“ bezeichnet)

(A1) $\{Q\}\ \textbf{skip}\ \{Q\}$

(A2) $\{[t/x]Q\}\ x:=t\ \{Q\}$

und zu jeder zusammengesetzten Anweisung eine **Schlußregel**:

(K)
$$\frac{Q \Rightarrow R,\ \{R\}\ \text{Anw}\ \{S\},\ S \Rightarrow T}{\{Q\}\ \text{Anw}\ \{T\}}$$
Logische Konsequenz

(S1)
$$\frac{\{Q\}\ \text{Anw}_1\ \{R\},\ \{R\}\ \text{Anw}_2\ \{S\}}{\{Q\}\ \text{Anw}_1\ ;\ \text{Anw}_2\ \{S\}}$$
Nacheinanderausführung

$$\textbf{(S2)}\quad \frac{\{Q \wedge b\}\ Anw_1\ \{R\},\ \{Q \wedge \neg b\}\ Anw_2\ \{R\}}{\{Q\}\ \texttt{if b then } Anw_1 \texttt{ else } Anw_2 \texttt{ fi}\ \{R\}}$$

Fallunterscheidung

$$\textbf{(S3)}\quad \frac{\{Inv \wedge b\}\ Anw\ \{Inv\}}{\{Inv\}\ \texttt{while b do } Anw \texttt{ od}\ \{Inv \wedge \neg b\}}$$

Schleife

Die Schlußregeln gestatten, auf die Gültigkeit **Konklusio** (unterhalb des Strichs) zu schließen, sobald man alle **Prämissen** (oberhalb des Strichs) als gültig nachgewiesen hat. **Axiomenschemata** gelten stets; das Schema **(A2)** muß man durch „matchen" von **x** mit der Variablen auf der linken Seite der Zuweisung und von **t** mit dem Ausdruck auf der rechten Seite an die vorliegende Zuweisung anpassen.

In einer logischen Implikation der Form **Q => R** folgt **R** aus **Q**. Daher gilt **Q** als „stärker" und **R** als „schwächer". Die **Konsequenzregel (K)** besagt, daß man stets eine Vorbedingung durch eine stärkere Zusicherung ersetzen darf und umgekehrt eine Nachbedingung durch eine schwächere Zusicherung. Die **Nacheinanderausführungsregel (S1)** besagt, daß man zwei Tripel (zu zwei Anweisungen) wie gezeigt zu einem Tripel (für die Folge der beiden Anweisungen) verschmelzen darf, wenn die Nachbedingung des ersten Tripels mit der Vorbedingung des zweiten Tripels übereinstimmt. Wenn Nachbedingung des ersten Tripels stärker ist als die Vorbedingung des zweiten Tripels, dann stellt man diese Übereinstimmung formal mit Hilfe der Konsequenzregel her.

Die **Fallunterscheidungsregel (S2)** zeigt, was für die Anweisungen des `then`- bzw. `else`-Zweigs gelten muß, wenn man das gewünschte Tripel für die gesamte `if`-Anweisung herleiten möchte. Die Prämisse der **Schleifenregel (S3)** besagt, daß die Invariante **Inv** bei Ausführung des Schleifenrumpfs erhalten bleiben muß. Zur Vorbedingung kommt dabei die Schleifenbedingung **b** hinzu, weil der Schleifenrumpf nur durchlaufen wird, wenn **b** wahr ist. In der Nachbedingung der Konklusio wird festgehalten, daß beim Verlassen der Schleife die Schleifenbedingung **b** *nicht mehr* erfüllt ist.

Ungewohnt ist die Anwendung des Axiomenschemas **(A2)**, bei der man **rückwärts** geht: Aus der Nachbedingung (also dem, was nach der Zuweisung gelten soll) schließt man formal auf die Vorbedingung (also was vor der Zuweisung gelten muß, damit nachher die gewünschte Nachbedingung erfüllt ist). Man spricht von **„Rückwärts-Propagation"**. Auf diese Weise finden wir heraus, was im Programm DR an der Stelle { ??? } gelten muß, indem wir auf die gewünschte Nachbedingung (die Invariante **Inv**) die durch die Zuweisung `x:=(x-y)` definierte Substitution anwenden:

$$\{[x-y/x]Inv\}$$
$$\equiv \{[x-y/x](m = d*n + x \ \wedge\ x \geq 0 \ \wedge\ y = n \ \wedge\ n > 0)\}$$
$$\equiv \{m = d*n + x-y \ \wedge\ x-y \geq 0 \ \wedge\ y = n \ \wedge\ n > 0\}$$

Damit haben wir die Gültigkeit der folgenden Hoare-Formel gezeigt:

$$\{m = d*n + x-y \ \wedge\ x-y \geq 0 \ \wedge\ y = n \ \wedge\ n > 0\}$$
$$x:=(x-y)$$
$$\{Inv\}$$

Erneute Rückwärts-Propagation der gerade berechneten Zusicherung über die Zuweisung `d:=(d+1)` ergibt die Zusicherung `Inv'` (Benennung wird später plausibel):

```
{[d+1/d](m = d*n + x-y  ∧  x-y ≥ 0  ∧  y = n  ∧  n > 0)}
≡ {Inv': m =(d+1)*n + x-y  ∧  x-y ≥ 0  ∧  y = n  ∧  n > 0}
```

Damit haben wir die Gültigkeit der folgenden Hoare-Formel gezeigt:

```
{Inv'}
d:=(d+1)
{m = d*n + x-y  ∧  x-y ≥ 0  ∧  y = n  ∧  n > 0}
```

Mit der Regel **(S1)** ergibt sich aus den beiden abgeleiteten Hoare-Tripeln ein weiteres:

```
{Inv'}
d:=(d+1);x:=(x-y)
{Inv}
```

Auf der Vorlesungsfolie K8-35 wird ausführlich die Gültigkeit der Implikation

```
(Inv ∧ x≤y) => Inv'
```

nachgewiesen (dort heißen die Zusicherungen `I` und `I'` statt `Inv` und `Inv'`).

Mit der Konsequenzregel **(K)** folgt nun, daß `Inv` tatsächlich Schleifeninvariante ist:

```
{Inv ∧ x≤y}
d:=(d+1);x:=(x-y)
{Inv}
```

Damit die Schleifenregel **(S3)** anwendbar und ergibt:

```
{Inv}
while x <= y do d:=(d+1) ; x:=(x-y) od
{Inv ∧ ¬x≤y}
```

Im Zusammenhang mit Schleifen sind häufig **drei Beweisschritte** erforderlich:

1. Die Schleifeninvariante ist beim Schleifeneintritt erfüllt.
2. Die Schleifeninvariante bleibt durch Ausführung des Schleifenrumpfs erhalten.
3. Aus der Schleifeninvarianten und dem Negat der Schleifenbedingung folgt die Zusicherung, die nach der Schleife gelten soll.

Für die Schleife des DR-Programms haben wir oben Beweisschritt 2. erledigt. Die komplette Behandlung der DR-Schleife findet man auf den Vorlesungsfolien K8-34 bis K8-36.

Mit Hilfe des Hoare-Kalküls kann man die **partielle Korrektheit** eines Programms P beweisen, d.h. daß P sich so verhält wie durch `{pre}P{post}` beschrieben, *wenn* P terminiert.

Für den Beweis der **(totalen) Korrektheit** von P ist zusätzlich der Nachweis zu erbringen, *daß* P terminiert. Da ein `while`-Programm stets terminiert, wenn alle seine Schleifen terminieren, kann man sich beim Beweis der Terminierung auf die Schleifen beschränken.

Die gängige Technik, die **Terminierung einer Schleife** zu beweisen, besteht darin, eine Funktion `f | SP -> Integer` zu finden mit folgenden Eigenschaften:

(i) Der Wert von `f` nimmt bei jedem Schleifendurchlauf echt ab.

(ii) Der Wert von `f` wird niemals negativ.

Offenbar bricht eine Schleife, zu der es eine **Terminierungsfunktion** mit diesen Eigenschaften gibt, nach endlich vielen Durchläufen ab, da man eine ganze Zahl nur endlich oft verkleinern kann, bevor das Ergebnis negativ wird.

Für die Schleife in DR ist $f(sp) =_{def} x$ eine geeignete Terminierungsfunktion:

- Da x in jedem Schleifendurchlauf um y (mit $y = n$ und $n > 0$) verringert wird, nimmt der Wert von f bei jedem Schleifendurchlauf echt ab.
- Da anfangs gemäß der Vorbedingung gilt $x = m$ und $m \geq 0$ und da der nächste Schleifendurchlauf jeweils nur dann begonnen wird, wenn $x >= y$, wird der Wert von f niemals negativ.

Allgemeine **Hinweise zur Verifikation** im Hoare-Kalkül:

- Zuerst muß man folgende **Zusicherungen finden**:
  - Vor- und Nachbedingung des gesamten Programms
  - Für jede Schleife eine Schleifeninvariante

  In Übungs- und Klausuraufgaben sind diese Zusicherungen häufig vorgegeben. Wenn nicht, dann hilft es, das Programm zu verstehen, z.B. indem man es für geeignete Testdaten „von Hand" ausführt. Oder man kann die Zusicherungen hinschreiben, weil sie das ausdrücken, was man sich beim Aufstellen des Programms überlegt hat. Eine gegebene Nachbedingung läßt sich über Anweisungsfolgen rückwärts propagieren (wie oben gezeigt). Eine gesuchte Schleifeninvariante kann man eventuell aus der Nachbedingung und der Schleifenbedingung erschließen. Viele weitere Heuristiken zum Aufstellen von Schleifeninvarianten findet man in der Literatur.
- Im Zusammenhang mit **Zuweisungen** (bzw. Folgen von Zuweisungen) arbeitet man grundsätzlich mit Rückwärts-Propagation.
- In den beiden Zweigen einer **`if`-Anweisung** propagiert man ebenfalls die Nachbedingung bis an den Anfang des Zweigs zurück und zeigt, daß die zurückpropagierte Zusicherung aus der Vorbedingung und der `if`-Bedingung (im `then`-Zweig) bzw. aus der Vorbedingung und dem Negat der `if`-Bedingung (im `else`-Zweig) logisch folgt. Formal ist das eine Anwendung der Konsequenzregel.
- Auch bei der **Schleife** geht man (wie oben gezeigt) ähnlich vor: Ausgangspunkt ist hier die Schleifeninvariante, die am Ende des Schleifenrumpfs gelten soll. Wie bei anderen Anweisungen propagiert man die Zusicherung rückwärts bis an den Anfang des Schleifenrumpfs. Dort ist zu zeigen, daß die rückgerechnete Invariante `Inv'` eine logische Konsequenz der Schleifeninvariante und der Schleifenbedingung ist (formal wieder eine Anwendung der Konsequenzregel). Abschließend kann man die Schleifenregel anwenden (und weiter in Richtung Nachbedingung argumentieren). Da man Schleifen als Mechanismus kennenlernt, mit dem man den Programmzustand im Speicher systematisch *verändert*, ist es zunächst ungewohnt, sich auf das zu konzentrieren, was sich *nicht ändert*, die **Schleifeninvariante**. Nur durch Konzentration auf das Gleichbleibende kann man die Veränderung beherrschen!

Invarianten sind auch sonst ein nützliches Hilfsmittel beim Problemlösen. Das deuten die Beispiele auf den letzten Vorlesungsfolien an. Auch die Frage, ob man ein Schachbrett, aus dem 2 gegenüberliegende Ecken gebrochen wurden, mit 2-Feld-Dominosteinen ohne Überschneidungen parkettieren kann, läßt sich mit einer geeigneten Invariante lösen.

Anhang: Operationale Semantik

Sei stets sp eine **Umgebung**, w ein **Wertekellerinhalt** (bzw. ein Teil davon) und r ein **Restprogramm** im Programmkeller. Weiter sei jeweils

$L(\langle Op\rangle) = \{+,-,div,mod\}$, $op \in L(\langle Op\rangle)$,

$L(\langle Rop\rangle) = \{<,<=,=,/=,>,>=\}$, $rop \in L(\langle Rop\rangle)$, $tf \in Bool$,

$n, n_1, n_2 \in L(\langle Zahl\rangle)$, $x \in L(\langle Name\rangle)$, $a, a_1, a_2 \in L(\langle Ausdruck\rangle)$,

$anw_1, anw_2 \in L(\langle Anweisung\rangle)$, $af, af_1, af_2 \in L(\langle Anweisungsfolge\rangle)$

Δ für Zahlen und Ausdrücke

1) $\Delta(sp, w, n \bullet r) =_{def} (sp, w \bullet n, r)$

2) $\Delta(sp, w, x \bullet r) =_{def} (sp, w \bullet sp(x), r)$ für $sp(x) \neq \perp$

3) $\Delta(sp, w, (a_1\ op\ a_2) \bullet r) =_{def} (sp, w, a_1 \bullet a_2 \bullet op \bullet r)$

4) $\Delta(sp, w \bullet n_1 \bullet n_2, op \bullet r) =_{def} (sp, w \bullet \langle n_1\ op\ n_2\rangle, r)$

Δ für Wahrheitswerte und Vergleiche

5) $\Delta(sp, w, tf \bullet r) =_{def} (sp, w \bullet tf, r)$

6) $\Delta(sp, w, (a_1\ rop\ a_2) \bullet r) =_{def} (sp, w, a_1 \bullet a_2 \bullet rop \bullet r)$

7) $\Delta(sp, w \bullet n_1 \bullet n_2, rop \bullet r) =_{def} (sp, w \bullet \langle n_1\ rop\ n_2\rangle, r)$

Δ für Anweisungen

8) $\Delta(sp, w, skip \bullet r) =_{def} (sp, w, r)$

9) $\Delta(sp, w, x{:=}a \bullet r) =_{def} (sp, w, a \bullet {:=} \bullet x \bullet r)$

10) $\Delta(sp, w \bullet n, {:=} \bullet x \bullet r) =_{def} (sp[x \to n], w, r)$

11) $\Delta(sp, w, anw_1; anw_2 \bullet r) =_{def} (sp, w, anw_1 \bullet anw_2 \bullet r)$

12) $\Delta(sp, w, \text{if } b \text{ then } af_1 \text{ else } af_2 \text{ fi} \bullet r) =_{def} (sp, w, b \bullet if \bullet af_1 \bullet af_2 \bullet r)$

13) $\Delta(sp, w \bullet true, if \bullet af_1 \bullet af_2 \bullet r) =_{def} (sp, w, af_1 \bullet r)$

14) $\Delta(sp, w \bullet false, if \bullet af_1 \bullet af_2 \bullet r) =_{def} (sp, w, af_2 \bullet r)$

15) $\Delta(sp, w, \text{while } b \text{ do } af \text{ od} \bullet r) =_{def} (sp, w, b \bullet do \bullet b \bullet af \bullet r)$

16) $\Delta(sp, w \bullet true, do \bullet b \bullet af \bullet r) =_{def} (sp, w, af \bullet \text{while } b \text{ do } af \text{ od} \bullet r)$

17) $\Delta(sp, w \bullet false, do \bullet b \bullet af \bullet r) =_{def} (sp, w, r)$