Programmverifikation mit dem Hoare-Kalkül
Eine informelle Einführung für Info II

H. J. Ohlbach und N. Eisinger, mit Ergänzungen von M. Hammer

5. Mai 2006

1 Motivation

Algorithmen und Programme sollten das tun, wofür sie entwickelt wurden, dem stimmt sicherlich jeder zu. Das Problem ist, wie kann man nachweisen, dass ein Algorithmus oder Programm auch wirklich das tut, wofür es gedacht ist? Schon bei relativ kleinen Beispielen sieht man, dass das meist gar nicht so einfach ist.

Folgendes Programm soll das Quadrat einer Zahl a berechnen:

```
class Quadrat {
    public static void main(String[] args) {
        final int a = Integer.parseInt(args[0]);
        int y, z;

     // Anfang der Berechnung
        y = 0;
        z = 0;
        while (y != a) {
            z = z + 2*y + 1;
            y = y + 1;
        }
     // Ende der Berechnung

        System.out.printf("quadrat(%d) == %d%n", a,z);
    }
}
```

(Als Java-Programm macht das nicht viel Sinn. Als Programm für einen Mikroprozessor, für den Multiplikation viel aufwendiger ist als Addition und Multiplikation mit 2, kann es aber sehr wohl Sinn machen.)

Es ist gar nicht offensichtlich, dass für alle Zahlen a das Programm wirklich den Wert a^2 berechnet, und in der Tat stimmt das auch nur für alle $a \geq 0$. Man kann das für einzelne Zahlen a *testen*, aber nicht für *alle* Zahlen a. Die übliche Vorgehensweise, Programme mit einzelnen Eingabewerten zu testen, funktioniert daher zuverlässig nur in einfachen Fällen. In den allermeisten Fällen müsste man unendlich viele Fälle testen, und das geht nicht.

Ein Ausweg ist *Verifikation*, d.h. die Korrektheit des Programms *mathematisch* zu beweisen. Dazu gibt es verschiedene Verfahren. Hier soll ein Verfahren vorgestellt werden, das auf den britischen Computer-Wissenschaftler Sir Charles Antony Richard Hoare zurückgeht, daher der Name Hoare-Kalkül. Das Verfahren muss für jede Programmiersprache angepasst werden. Um das Grundprinzip zu verstehen und kleine Beispiele rechnen zu können, genügt es jedoch, das Verfahren für das Fragment von Java vorzustellen, das aus folgenden Arten von Anweisungen besteht: Wertzuweisungen, **if**(...)...**else**... und **while**(...)... . Dabei werden die Variablentypen von Java ignoriert (die überprüft sowieso der Compiler).

2 Partielle und totale Korrektheit

Zunächst muss man festlegen, wie man die Aussagen, die man für ein Programm machen will, formuliert. Dazu nehmen wir eine mathematische Notation, z.B. $0 \leq a$ oder $z = a^2$ usw. (Man kann hier eine formale Sprache definieren, aber das ist für die Zwecke dieser Einführung nicht nötig).

Welche Art von Aussagen will man nun für Programme überhaupt beweisen? Typischerweise sind die Aussagen von der Art:

Wenn für die Eingabewerte des Programms $\mathtt{p}$ die Bedingung φ gilt, dann gilt nach Ausführung des Programms für die Ausgabevariablen die Bedingung ψ. Man schreibt das meist

$$\{\varphi\}\,\mathtt{p}\,\{\psi\}$$

Man sagt, φ ist die *Vorbedingung* für das Programm $\mathtt{p}$ und ψ ist die *Nachbedingung* für das Programm $\mathtt{p}$.

Im obigen „Quadrat"-Beispiel wäre das z.B.

$$\{0 \leq a\}\,\mathtt{p}\,\{z = a^2\}$$

wobei $\mathtt{p}$ der Teil des Programms ist, der zwischen den beiden Kommentaren „Anfang der Berechnung" und „Ende der Berechnung" steht. $0 \leq a$ ist die *Vorbedingung* und $z = a^2$ ist die *Nachbedingung*.

Es gibt allerdings noch ein Problem: wenn man in dem „Quadrat"-Beispiel für a den Wert -1 eingibt, dann terminiert das Programm nicht. Die **`while`**-Schleife läuft unendlich weiter. Daher unterscheidet man zwischen *partieller Korrektheit* und *totaler Korrektheit* eines Programms.

Definition 2.1 (Partielle Korrektheit) *Partielle Korrektheit eines Programms* $\mathtt{p}$ *heißt: falls* $\mathtt{p}$ *auf Eingabewerte, die der Vorbedingung* φ *genügen, angewandt wird, und falls es terminiert, dann muss anschließend die Nachbedingung* ψ *gelten.*

Definition 2.2 (Totale Korrektheit) *Totale Korrektheit eines Programms* $\mathtt{p}$ *heißt: falls* $\mathtt{p}$ *auf Eingabewerte, die der Vorbedingung* φ *genügen, angewandt wird, dann terminiert es und danach muss die Nachbedingung* ψ *gelten.*

Der Beweis der Terminierung eines Programms ist meist nicht einfach und wird von dem Hoare-Kalkül nicht unterstützt. Der Hoare-Kalkül beschränkt sich auf den Nachweis der partiellen Korrektheit. In Abschnitt 8 gehen wir auf die Techniken ein, mit denen Terminierungsbeweise geführt werden.

3 Das Hoare-Kalkül

Das Hoare-Kalkül stellt eine Reihe von Schlussregeln zur Verfügung, die, basierend auf der Syntax des Programms, den Nachweis partieller oder totaler Korrektheit ermöglichen. Die Regeln sind von der Form

$$\frac{A_1, \ldots, A_n}{B}$$

was bedeutet, daß, wenn A_1 bis A_n bewiesen sind, auch B bewiesen ist. Dabei sind A_1 bis A_n und B Hoare-Tripel. Manche Regeln haben zusätzliche Bedingungen, die festlegen, wann sie angewendet werden dürfen.

Ein Beweis im Hoare-Kalkül hat die Form eines Baumes, bei dem die Wurzel (die unten steht) das zu beweisende Tripel ist, und die Blätter aus Regeln bestehen, die keine Voraussetzung haben (weiter unten werden wir sehen, daß dies die Skip- und die Zuweisungsregel sind). Die inneren Knoten entsprechen Regelanwendungen. Es ist dabei üblich, neben die Striche die verwendete Regel zu schreiben.

3.1 Skip

$$\frac{}{\{\psi\}\ \{\}\ \{\psi\}}$$

Die Skip-Regel besagt, daß für eine Anweisung, die nichts tut (wie sie z.B. in einem leeren `else`-Zweig vorkommt), Vor- und Nachbedingung gleich sind. (Eine Anwendung dieser Regel finden Sie in dem Beispiel der Block-Regel im Abschnitt 3.6.)

3.2 Zuweisung

$$\frac{}{\{\psi[\mathtt{x}/\mathtt{t}]\}\ \mathtt{x} = \mathtt{t};\ \{\psi\}}$$

Dies ist neben der Skip-Regel die einzige Regel, die ohne vorher zu beweisende Elemente auskommt. Für Korrektheitsbeweise bildet sie den Basisfall, auf dem die anderen Regeln aufbauen können.

Die Regel ist so zu lesen: Die Vorbedingung einer Zuweisung `x = t` ist genau die Formel, die entsteht, wenn man in der Nachbedingung alle Vorkommen von `x` durch `t` ersetzt.

3.3 Abschwächung

$$\frac{\{\varphi\}\,\mathtt{p}\,\{\psi\}}{\{\varphi'\}\,\mathtt{p}\,\{\psi'\}}, \text{wenn } \varphi' \implies \varphi \text{ und } \psi \implies \psi'$$

Abschwächung wird benötigt, um aus speziellen Vorbedingungen, wie sie z.B. durch die Zuweisungsregel entstehen, auf allgemeinere Bedingungen schliessen zu können.

Beispiel:
Es wäre denkbar, das Tripel $\{y > 0\}\,\mathtt{x} = 15;\ \{x > 0\}$ beweisen zu wollen. Mit der Zuweisungsregel wird aus der Zuweisung und der Nachbedingung $\{15 > 0\}\,\mathtt{x} = 15;\ \{x > 0\}$. Da diese Vorbedingung stets erfüllt ist, funktioniert das Tripel laut Abschwächungsregel auch mit jeder anderen Vorbedingung (da $\varphi' \implies \mathit{true}$ allgemeingültig ist, also für beliebige φ' gilt).

Damit erlaubt uns die Abschwächungsregel, auf das ursprünglich zu beweisende Tripel zu schliessen:

$$\dfrac{\dfrac{}{\{15 > 0\}\,\mathtt{x} = 15;\ \{\mathtt{x} > 0\}}\ \text{(Zuweisung)}}{\{\mathtt{y} > 0\}\,\mathtt{x} = 15;\ \{\mathtt{x} > 0\}}\ \text{(Abschwächung)}$$

3.4 Fallunterscheidung

$$\frac{\{\varphi \wedge \mathtt{B}\}\,\mathtt{p}\,\{\psi\} \qquad \{\varphi \wedge \neg\mathtt{B}\}\,\mathtt{q}\,\{\psi\}}{\{\varphi\}\ \mathtt{if}(\mathtt{B})\ \mathtt{p}\ \mathtt{else}\ \mathtt{q}\ \{\psi\}}, \text{ wenn } \mathtt{B} \text{ seiteneffektfrei}$$

Die Fallunterscheidungsregel besagt, daß sowohl der `then`- als auch der `else`-Zweig, wenn sie mit einer gültigen Vorbedingung abgearbeitet werden, die gleiche Nachbedingung erfüllen. (Ein Beispiel für die Anwendung dieser Regel ist bei der der Block-Regel im Abschnitt 3.6 zu finden.)

`B` muß seiteneffektfrei sein, d.h., der Zustand des Programms darf bei der Auswertung von `B` nicht verändert werden. Dies ist in Java allerdings nicht gefordert, und es ist völlig legitim, folgenden Code zu verwenden:

```
if ((x += 15) > y) { ...
```

Hier wird nicht nur geprüft, ob $\mathtt{x}+15$ größer als $\mathtt{y}$ ist, sondern auch der Wert von $\mathtt{x}$ um 15 erhöht. Dies lässt sich jedoch immer vermeiden, indem man den booleschen Ausdruck vorher auswertet:

```
boolean b = (x += 15) > y;
if (b) { ...
```

3.5 Sequentielle Komposition

$$\frac{\{\varphi\}\,\mathtt{p}_1\,\{\alpha\} \qquad \{\alpha\}\,\mathtt{p}_2\,\{\psi\}}{\{\varphi\}\,\mathtt{p}_1\mathtt{p}_2\,\{\psi\}}$$

Diese Regel erlaubt es uns, mehrere Teilschritte im Beweis zusammenzufassen.

Beispiel: Das Tripel $\{true\}\,\mathtt{y}=0;\mathtt{x}=\mathtt{y};\,\{x\geq 0\}$ muß mit zweimaliger Anwendung der Zuweisungsregel bewiesen werden. Für die zweite Zuweisung entsteht das Tripel $\{y\geq 0\}\,\mathtt{x}=\mathtt{y};\,\{x\geq 0\}$. Wählen wir nun $\alpha\equiv y\geq 0$, dann können wir für die erste Zuweisung das Tripel $\{0\geq 0\}\,\mathtt{y}=0;\,\{y\geq 0\}$ bekommen und nun die Regel der sequentiellen Komposition anwenden:

$$\dfrac{\dfrac{}{\{0\geq 0\}\,\mathtt{y}=0;\,\{\mathtt{y}\geq 0\}}\,\text{(Zuweisung)} \qquad \dfrac{}{\{\mathtt{y}\geq 0\}\,\mathtt{x}=\mathtt{y};\,\{\mathtt{x}\geq 0\}}\,\text{(Zuweisung)}}{\{true\}\,\mathtt{y}=0;\mathtt{x}=\mathtt{y};\,\{\mathtt{x}\geq 0\}}\,\text{(seq. Komposition)}$$

3.6 Block

$$\frac{\{\varphi\}\,\mathtt{p}\,\{\psi\}}{\{\varphi\}\,\{\mathtt{p}\}\,\{\psi\}}$$

Die Blockregel erlaubt uns, beliebigen bewiesenen Code als Block in anderen Regeln (z.B. der Fallunterscheidungsregel) einzusetzen.

Beispiel: Betrachte folgende **`if`**-Regel:

$$\{true\}\,\mathtt{if}(\mathtt{x}\,!=0)\{\mathtt{y}=0;\mathtt{x}=\mathtt{y};\}\,\mathtt{else}\,\{\}\,\{\mathtt{x}=0\}$$

Mit der sequentiellen Kompositionsregel kann der für den Fall $\mathtt{x}\neq 0$ das Tripel $\{true\}\,\mathtt{y}=0;\mathtt{x}=\mathtt{y};\,\{\mathtt{x}\geq 0\}$ hergeleitet werden. Mit der Blockregel dürfen wir dieses nun verwenden, um die Fallunterscheidung zu beweisen:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{true\}\,\mathtt{y}=0;\mathtt{x}=\mathtt{y};\,\{x=0\}}\,\text{(analog zu 3.5)}}{\{\mathtt{x}\neq 0\}\,\mathtt{y}=0;\mathtt{x}=\mathtt{y};\,\{\mathtt{x}=0\}}\,\text{(Abschwächung)}}{\{x\neq 0\}\,\{\mathtt{y}=0;\mathtt{x}=\mathtt{y};\}\,\{\mathtt{x}=0\}}\,\text{(Block)} \qquad \dfrac{}{\{\mathtt{x}=0\}\,\{\}\,\{\mathtt{x}=0\}}\,\text{(Skip)}}{\{true\}\,\mathtt{if}(\mathtt{x}!=0)\,\{\mathtt{y}=0;\mathtt{x}=\mathtt{y};\}\,\mathtt{else}\,\{\}\;\{\mathtt{x}=0\}}\,\text{(Falluntersch.)}$$

3.7 Iteration

Iteration ist der einzige richtig schwierige Fall beim Hoare-Kalkül, sowie der einzigen Fall, wo zwischen partieller und totaler Korrektheit unterschieden werden muß. Es gibt daher zwei Regeln, die beide eine Invariante I benötigen. Das Auffinden dieser Invariante wird in Abschnitt 5.3 besprochen.

Regel Iteration$_{\text{partiell}}$:

$$\frac{\{I \wedge B\}\,\mathtt{p}\,\{I\}}{\{I\}\,\mathtt{while(B)\ p}\ \{I \wedge !B\}}\text{, wenn B seiteneffektfrei}$$

Regel Iteration$_{\text{total}}$:

$$\frac{\begin{array}{c}\{I \wedge \mathtt{B}\}\,\mathtt{p}\,\{I\} \\ \{I \wedge \mathtt{B} \wedge t = z\}\,\mathtt{p}\,\{t < z\}\end{array}}{\{I\}\,\mathtt{while(B)\ p}\ \{I \wedge !B\}}\text{, wenn B seiteneffektfrei, } I \implies t \geq 0 \text{ und } z \text{ ist freie logische Variable}$$

Wie bei der Fallunterscheidungsregel fordern wir, daß B seiteneffektfrei ist. Für die Iterationsregel für totale Korrektheit benötigen wir zusätzlich eine „logische“ Variable z, die frei ist. Das bedeutet, daß sie in $I, \mathtt{B}, \mathtt{p}$ und t nicht vorkommen darf (sie dient nur dazu, den alten Wert von t zwischenzuspeichern).

Offensichtlich ist die Regel für die totale Korrektheit eine Erweiterung der Regel für partielle Korrektheit. Es wird hierbei eine fundierte Ordnung verwendet (siehe Abschnitt 8.1), wie Sie sie bei Terminierungsbeweisen funktionaler Programme kennengelernt haben sollten. Genau das gleiche Schema wird hier verwendet: p muß einen Integer-Ausdruck t echt kleiner werden lassen, während die Invariante garantiert, daß der Ausdruck einen kleinsten Wert nicht unterschreitet.

Die Regeln des Hoare-Kalkül stellen einen Rahmen dar, in dem Programmverifikation möglich ist. Diese benötigt jedoch viel kreative Arbeit, insbesondere beim Auffinden der Invarianten. Einige der Regeln, wie Abschwächung oder Block, werden zudem eher benötigt, mathematisch exakte Beweise hinschreiben zu können. Im nächsten Abschnitt soll daher eine Anleitung gegeben werden, wie die Regeln auf den Nachweis von Hoare-Tripeln mit konkreten Programmen anwendbar sind.

4 Das allgemeine Schema des partiellen Korrektheitsbeweises

Da die Vor- und Nachbedingungen i.a. gegeben sind, liegt folgende Situation vor:

$$\{\varphi\}\,\mathtt{p}_1;\ldots;\mathtt{p}_n\,\{\psi\}$$

wobei $\mathtt{p}_1, \ldots, \mathtt{p}_n$ die einzelnen Anweisungen des Programms sind.

Der Korrektheitsbeweis geht von hinten nach vorne vor, und zwar nach folgendem Schema:

1. Finde eine Zwischenbedingung α_1 für p_n und spalte den Beweis in

   $$\{\varphi\}\,\mathtt{p}_1;\ldots;\mathtt{p}_{n-1}\,\{\alpha_1\} \qquad \text{und} \qquad \{\alpha_1\}\,\mathtt{p}_n\,\{\psi\}$$

   (Für $n = 1$ hat der erste Teil die Gestalt $\{\varphi\}\ \{\alpha_1\}$, was im nächsten Fall behandelt wird). (Dieses Vorgehen entspricht der sequentiellen Kompositionsregel.)

   Die Weiterverarbeitung von $\{\alpha_1\}\,\mathtt{p}_n\,\{\psi\}$ hängt von der Art der Anweisung $\mathtt{p}_n$ ab (Wertzuweisung, **if**(...)...**else**... oder **while**(...)...). Für jede Anweisungsart benötigt man eine extra Regel (siehe unten). Diese Regeln geben auch z.T. Hinweise, was die richtige Zwischenbedingung α_1 ist.

2. Nach dem gleichen Schema werden die Anweisungen $\mathtt{p}_{n-1}, \ldots, \mathtt{p}_1$ behandelt, bis man schließlich eine Situation erreicht

   $$\{\varphi\}\ \{\alpha_n\}$$

   wo kein Programmstück zwischen den Bedingungen mehr vorhanden ist. Partielle Korrektheit ist bewiesen, wenn man in dieser Situation

   $$\varphi \Rightarrow \alpha_n,$$

   d.h. aus φ folgt α_n, mit rein mathematischen Mitteln beweisen kann (dies entspricht einer Anwendung der Abschwächungsregel).

   Ein Beispiel ist $\{x \geq 0 \ \wedge \ y \geq 0\}\ \{x * y \geq 0\}$, was zu $(x \geq 0 \ \wedge \ y \geq 0) \ \Rightarrow \ (x * y \geq 0)$ führt, und das ist eine wahre Aussage.

Das Hoare'sche Verfahren reduziert das Verifikationsproblem also auf rein mathematische Beweisprobleme (auch Beweisverpflichtungen genannt). Wenn alle anfallenden Beweisverpflichtungen auch tatsächlich bewiesen werden können, dann ist die partielle Korrektheit gezeigt.

5 Die Anwendung der einzelnen Verifikationsregeln

Wir sehen uns nun die Verifikationsregeln für die einzelnen Anweisungsarten an.

5.1 Zuweisungen

Die Ausgangssituation ist:

$$\{\varphi\}\, \mathtt{p}_1; \ldots; \mathtt{p}_{n-1};\ \mathtt{x} = \mathtt{t};\ \{\psi\}$$

wobei `x` eine Programmvariable ist und `t` ein Ausdruck.

Die *Zuweisungsregel* transformiert dieses Problem in

$$\{\varphi\}\, \mathtt{p}_1; \ldots; \mathtt{p}_{n-1};\ \{\psi[\mathtt{x}/\mathtt{t}]\}$$

wobei $\psi[\mathtt{x}/\mathtt{t}]$ bedeutet, dass alle Vorkommen der Variablen `x` in ψ durch den Ausdruck `t` zu ersetzen sind. Das spiegelt genau die Bedeutung der Zuweisungsoperation wieder: nach ihrer Ausführung sind `x` und `t` identisch.

Beispiel:
Aus

$$\{\varphi\}\, \mathtt{p}_1; \ldots; \mathtt{p}_{n-1};\ \mathtt{y} = \ \mathtt{x} * \mathtt{x};\ \{y \geq 0 \ \wedge \ \mathtt{y} \leq 10\}$$

wird

$$\{\varphi\}\, p_1; \ldots; p_{n-1};\ \{\mathtt{x} * \mathtt{x} \geq 0 \ \wedge \ \mathtt{x} * \mathtt{x} \leq 10\}$$

Die Zuweisungsregel erlaubt, die Zwischenbedingung α_1 explizit zu berechnen, nämlich $\alpha_1 = \psi[\mathtt{x}/\mathtt{t}]$. Alle anderen Regeln erfordern an dieser Stelle etwas Kreativität.

5.2 `if(...)...else...`

Die Ausgangssituation ist:

$$\{\varphi\}\, \mathtt{p}_1; \ldots; \mathtt{p}_{n-1};\ \mathbf{if}(\mathtt{B})\ \mathtt{p}\ \mathbf{else}\ \mathtt{q};\ \{\psi\}$$

wobei B die Testbedingung ist und `p` und `q` Programmstücke.

Die ***if*-Regel** transformiert dieses Problem in vier einzelne Probleme:

1. finde eine geeignete Zwischenbedingung α_1 als neue Vorbedingung für die **`if`**-Anweisung;
2. beweise den „true"-Zweig:

   $$\{\alpha_1 \ \wedge \ \mathtt{B}\}\, \mathtt{p}\, \{\psi\},$$

3. beweise den „false“-Zweig:
$$\{\alpha_1 \ \wedge \ \neg \mathtt{B}\}\, \mathtt{q}\, \{\psi\},$$

4. mache weiter mit den restlichen Anweisungen vor der **`if`**-Anweisung für die Nachbedingung α_1
$$\{\varphi\}\, \mathtt{p}_1; \ldots; \mathtt{p}_{n-1};\, \{\alpha_1\}.$$

Der Kern dieser Regel ist also der Nachweis, dass nach der **`if`**-Anweisung in jedem Fall ψ gilt, egal ob die Testbedingung B wahr oder falsch ist. Wie jede Verzweigung hat die **`if`**-Anweisung den Zweck, bei unterschiedlichen Voraussetzungen den gleichen Zustand herzustellen (und nicht etwa umgekehrt, wie man auf den ersten Blick meinen könnte).

Beispiel:
$\{true\}$

```
if (x > 0) y = x;
else       y = -x;
```

$\{\mathtt{y} = |\mathtt{x}|\}$
Dieses Programmstück berechnet den Absolutbetrag von x.
Die vier Schritte der **`if`**-Regel sind jetzt folgendermaßen:

1. Zwischenbedingung wird keine benötigt, d.h., $\alpha_1 = \varphi = true$.

2. Der „true“-Zweig:
   $\{\mathtt{x} > 0\}\, \mathtt{y} \ = \ \mathtt{x}\, \{\mathtt{y} = |\mathtt{x}|\}$
   Daraus wird mit der Zuweisungsregel
   $\{\mathtt{x} > 0\}\ \{\mathtt{x} = |\mathtt{x}|\}$
   und daraus wiederum
   $(\mathtt{x} > 0) \ \Rightarrow \ (\mathtt{x} = |\mathtt{x}|)$
   und das stimmt.

3. Der „false“-Zweig:
   $\{!(\mathtt{x} > 0)\}\, \mathtt{y} \ = \ -\mathtt{x}\, \{\mathtt{y} = |\mathtt{x}|\}$
   Daraus wird mit der Zuweisungsregel
   $\{!(\mathtt{x} > 0)\}\ \{-\mathtt{x} = |\mathtt{x}|\}$
   und daraus wiederum
   $(\mathtt{x} \leq 0) \ \Rightarrow \ (-\mathtt{x} = |\mathtt{x}|)$
   und das stimmt ebenfalls.

4. der Rest vor der **`if`**-Anweisung ist leer, und damit wäre man fertig.

5.3 `while(...)...`

Wir betrachten die **`while`**-Schleife ohne **`break`**- und **`continue`**-Anweisungen.

Die Ausgangssituation ist:
$$\{\varphi\}\, \mathtt{p}_1; \ldots; \mathtt{p}_{n-1};\ \mathbf{while}(\mathtt{B})\, \mathtt{p}\, \{\psi\}$$

wobei B die Testbedingung ist und $\mathtt{p}$ ein Programmstück.

Die ***`while`*-Regel** transformiert dieses Problem in folgende einzelne Probleme:

1. Finde eine geeignete *Schleifeninvariante* I. Dies ist ein mathematischer Ausdruck, der bei jedem Durchgang durch das Programmstück $\mathtt{p}$ gültig (invariant) bleibt. Das Auffinden der Schleifeninvariante ist oft ein kreativer Vorgang, bei dem man genau verstehen muss, was in der Schleife passiert.

2. Finde geeignete Zwischenbedingung α_1 als neue Vorbedingung für die while-Anweisung, so dass

$$\alpha_1 \Rightarrow I$$

gilt. D.h. die Zwischenbedingung α_1 ist die spezielle Form der Schleifeninvariante, die vor Eintritt in die Schleife gilt.

3. Verifiziere den Erhalt der Schleifeninvariante:

$$\{I \ \wedge \ \mathtt{B}\}\,\mathtt{p}\,\{I\}$$

Dies bestätigt, dass die Schleifeninvariante so lange gültig bleibt, wie die Bedingung B gilt.

4. Weise nach, dass die Schleifeninvariante stark genug ist, dass sie die Nachbedingung ψ erzwingt

$$(I \ \wedge \ \neg \mathtt{B} \ \Rightarrow \ \psi)$$

d.h. nachdem die Bedingung B falsch geworden ist und die Schleife verlassen wurde, muss ψ folgen.

5. Mache weiter mit den restlichen Anweisungen vor der Schleife für die Nachbedingung α_1

$$\{\varphi\}\,\mathtt{p}_1;\ldots;\mathtt{p}_{n-1};\,\{\alpha_1\}.$$

6 Beispiele

6.1 Das „Quadrat“-Beispiel

Wir wollen das „Quadrat“-Programm aus Abschnitt 1 verifizieren.
$\{0 \leq \mathtt{a}\}$

```
{
    y = 0;
    z = 0;
    while (y != a) {
        z = z + 2*y + 1;
        y = y + 1;
    }
}
```

$\{\mathtt{z} = \mathtt{a}^2\}$

Die letzte Anweisung ist eine **while**-Schleife. Daher wird die **while**-Regel angewandt.

1. Schleifeninvariante: $\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2$
(durch genaues Hinsehen, was die Schleife bewirkt).

2. Vorbedingung α_1: $0 \leq \mathtt{a} \ \wedge \ \mathtt{y} = 0 \ \wedge \ \mathtt{z} = 0$.
Dies impliziert offensichtlich $\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2$.

3. Erhalt der Schleifeninvariante:
$\{\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2 \ \wedge \ \mathtt{y} \neq \mathtt{a}\}\,\mathtt{z = z + 2*y+ 1; \ y= y+ 1}\,\{\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2\}$
Das zeigt man durch zweimalige Anwendung der Regel für Zuweisungen:

   (a) $\{\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2 \ \wedge \ \mathtt{y} \neq \mathtt{a}\}\,\mathtt{z = z + 2*y+ 1;}\,\{\mathtt{y} + 1 \leq \mathtt{a} \ \wedge \ \mathtt{z} = (\mathtt{y}+1)^2\}$

   (b) $\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2 \ \wedge \ \mathtt{y} \neq \mathtt{a} \mathtt{y} + 1 \leq \mathtt{a} \ \wedge \ \mathtt{z} + 2 * \mathtt{y} + 1 = (\mathtt{y}+1)^2$

Daraus wird $(\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2 \ \wedge \ \mathtt{y} \neq \mathtt{a}) \ \Rightarrow \ (\mathtt{y} + 1 \leq \mathtt{a} \ \wedge \ \mathtt{z} + 2 * y + 1 = (y+1)^2)$.
Und dies gilt, denn:

   (a) Aus $\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{y} \neq \mathtt{a}$ folgt $\mathtt{y} < \mathtt{a}$ und damit $\mathtt{y} + 1 \leq \mathtt{a}$.

(b) Aus $\mathtt{z} = \mathtt{y}^2$ folgt $\mathtt{z} + 2 * \mathtt{y} + 1 = \mathtt{y}^2 + 2 * \mathtt{y} + 1 = (\mathtt{y} + 1)^2$.

4. Nachweis der Nachbedingung:
   $(\mathtt{y} \leq \mathtt{a} \ \wedge \ \mathtt{z} = \mathtt{y}^2 \wedge !(\mathtt{y} \neq \mathtt{a})) \ \Rightarrow \ \mathtt{z} = \mathtt{a}^2$ und dies stimmt wegen $\mathtt{y} = \mathtt{a}$ ebenfalls.

5. Weiter mit den restlichen Anweisungen vor der Schleife und der Vorbedingung α_1 der Schleife als Nachbedingung dieser Anweisungen:
   $\{0 \leq \mathtt{a}\}\,\mathtt{y = 0; \ z = 0}\,\{0 \leq \mathtt{a} \ \wedge \ \mathtt{y} = 0 \ \wedge \ \mathtt{z} = 0\}$,
   was durch zweimalige Anwendung der Regel für Zuweisungen erledigt wird, so dass nur noch die Implikation
   $(0 \leq \mathtt{a}) \ \Rightarrow \ (0 \leq \mathtt{a} \ \wedge \ 0 = 0 \ \wedge \ 0 = 0)$
   übrig bleibt, die offensichtlich stimmt.

Die obige Darstellung betont den wichtigen Teil, nämlich die Reihenfolge, in der man vorgeht, um den Verifikationsbeweis zu finden. Wie immer in der Mathematik ist es dann einfacher, den einmal gefundenen Beweis an einem Stück aufzuschreiben, so dass man sich überzeugen kann, dass die einzelnen Schritte stimmen. Beweise im Hoare-Kalkül kann man so aufschreiben, dass die Vor- und Nachbedingungen einfach als Kommentare zwischen den Anweisungen stehen, zusammen mit der Angabe, welche Regel jeweils angewandt wurde. Auf diese Weise bleibt das Programm übersetzbar und enthält trotzdem die gesamte für die Verifikation erforderliche Information.

```
class Quadrat {
    public static void main(String[] args) {
        final int a = Integer.parseInt(args[0]);

        int y, z;


        // Vorbedingung:                    0<=a
        // logische Folgerung:              0<=a && 0==0 && 0==0
        y = 0;
        // Zuweisungsregel:                 0<=a && y==0 && 0==0
        z = 0;
        // Zuweisungsregel:                 0<=a && y==0 && z==0
        // arithm. Folgerung:               y<=a && z==y*y
        while (y != a) {
            // while-Regel:                 y<=a && z==y*y && y!=a
            // logische Folgerung:          y<a && z==y*y
            // arithm. Folgerung:           (y+1)<=a && (z+2*y+1)==(y+1)*(y+1)
            z = z + 2*y + 1;
            // Zuweisungsregel:             (y+1)<=a && z==(y+1)*(y+1)
            y = y + 1;
            // Zuweisungsregel:             y<=a && z==y*y
        }
        // while-Regel:                     y<=a && z==y*y && y==a
        // logische Folgerung:              z==a*a


        System.out.printf("quadrat(%d) == %d%\n", a, z);
    }
}
```

6.2 Das „ggT“-Beispiel

Es gibt einen sehr einfachen Algorithmus, um den größten gemeinsamen Teiler $ggT(a, b)$ von zwei positiven natürlichen Zahlen a und b zu berechnen: man zieht so lange von der größeren der beiden Zahlen die kleinere ab, bis die beiden Zahlen gleich sind. Folgendes ist die Verifikationsaufgabe:
$(0 < \mathtt{a} \ \wedge \ 0 < \mathtt{b})$

```
{
    x = a;
    y = b;
    while (x != y) {
        if (x < y)    { y = y - x; }
        else          { x = x - y; }
    }
}
```

$(ggT(\mathtt{a},\mathtt{b}) = \mathtt{x})$

Die letzte Anweisung ist eine **while**-Schleife. Daher wird die **while**-Regel angewandt.

1. Schleifeninvariante: $0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y})$
   (dies ist überhaupt nicht offensichtlich, stimmt aber).

2. Vorbedingung α_1: $0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y})$
   Dies ist identisch mit der Schleifeninvariante.

3. Erhalt der Schleifeninvariante:
   $\{0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y}\}$

   ```
   {   if (x < y)    {y = y - x;}
       else          {x = x - y;}
   }
   ```

   $\{0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y})\}$

   Um das zu zeigen, müssen wir die if-Regel anwenden.

   (a) „true“-Fall:
   $\{0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} < \mathtt{y}\}$

   ```
   {y = y - x;}
   ```

   $\{0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y})\}$
   Durch Anwendung der Zuweisungsregel reduziert sich das auf
   $(0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} < \mathtt{y})$
   $\Rightarrow \ (0 < \mathtt{x} \wedge 0 < \mathtt{y} - \mathtt{x} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y} - \mathtt{x}))$
   wovon der schwierige Teil ist:
   $(\mathtt{x} < \mathtt{y}) \ \Rightarrow \ (ggT(\mathtt{x},\mathtt{y}) = ggT(\mathtt{x},\mathtt{y} - \mathtt{x}))$
   Dies kann man mit etwas Zahlentheorie beweisen, womit der „true“-Fall beendet wäre.

   (b) „false“-Fall:
   $\{0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y}) \wedge \mathtt{x} \neq \mathtt{y} \wedge \mathtt{x} \geq \mathtt{y}\}$

   ```
   {x = x - y;}
   ```

   $\{0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y})\}$
   und das geht genau analog zum „true“-Fall.

4. Nachweis der Nachbedingung:
   $(0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y}) \wedge !(\mathtt{x} \neq \mathtt{y})) \ \Rightarrow \ (ggT(\mathtt{a},\mathtt{b}) = \mathtt{x})$
   Das stimmt, weil aus $\mathtt{x} = \mathtt{y}$ folgt dass $ggT(\mathtt{x},\mathtt{y}) = \mathtt{x}$ ist.

5. Weiter mit den restlichen Anweisungen vor der Schleife für die Nachbedingung α_1
   $(0 < \mathtt{a} \wedge 0 < \mathtt{b})\{\mathtt{x = a; \ y = b;}\}(0 < \mathtt{x} \wedge 0 < \mathtt{y} \wedge ggT(\mathtt{a},\mathtt{b}) = ggT(\mathtt{x},\mathtt{y}))$
   was durch zweimalige Anwendung der Zuweisungsregel trivial gelöst wird.

Auch für dieses Beispiel kann man den Beweis am Stück in Form von Kommentaren im Programm angeben, so dass das Programm angereichert mit der Verifikations-Information übersetzbar bleibt.

```
class GGT {
    public static void main(String[] args) {
        final int a = Integer.parseInt(args[0]);
        final int b = Integer.parseInt(args[1]);

        int x, y;


        // Vorbedingung:                    0<a && 0<b
        // logische Folgerung:              0<a && 0<b && ggT(a,b)==ggT(a,b)
        x = a;
        // Zuweisungsregel:                 0<x && 0<b && ggT(a,b)==ggT(x,b)
        y = b;
        // Zuweisungsregel:                 0<x && 0<y && ggT(a,b)==ggT(x,y)
        while (x != y) {
            // while-Regel:                 0<x && 0<y && ggT(a,b)==ggT(x,y)
            //                              && x!=y
            if (x < y) {
                    // if-Regel:            0<x && 0<y && ggT(a,b)==ggT(x,y)
                    //                      && x!=y && x<y
                    // logische Folgerung:  0<x && x<y && ggT(a,b)==ggT(x,y)
                    // arithm. Folgerung:   0<x && 0<y-x && ggT(a,b)==ggT(x,y-x)
                    y = y - x;
                    // Zuweisungsregel:     0<x && 0<y && ggT(a,b)==ggT(x,y)
            } else {
                    // if-Regel:            0<x && 0<y && ggT(a,b)==ggT(x,y)
                    //                      && x!=y && !(x<y)
                    // logische Folgerung:  y<x && 0<y && ggT(a,b)==ggT(x,y)
                    // arithm. Folgerung:   0<x-y && 0<y && ggT(a,b)==ggT(x-y,x)
                    x = x - y;
                    // Zuweisungsregel:     0<x && 0<y && ggT(a,b)==ggT(x,y)
            }
            // if-Regel:                    0<x && 0<y && ggT(a,b)==ggT(x,y)
        }
        // while-Regel:                     0<x && 0<y && ggT(a,b)==ggT(x,y)
        //                                  && x==y
        // logische Folgerung:              ggT(a,b)==ggT(x,x)
        // arithm. Folgerung:               ggT(a,b)==x


        System.out.printf("ggt(%d,%d) == %d%\n", a, b, x);
    }
}
```

7 Kombination mit der Java-Anweisung `assert`

Java kennt eine Anweisung `assert` *BoolescherAusdruck*`;` mit der zur Laufzeit sichergestellt werden kann, dass Annahmen über den Programmzustand eingehalten werden. Die Anweisung bewirkt, dass der Boolesche Ausdruck ausgewertet wird. Wenn er den Wert **`true`** hat, passiert nichts weiter, wenn er den Wert **`false`** hat, wird ein Fehler ausgelöst.

Man könnte die gleiche Wirkung auch mit **`if`** erzielen. Zum Beispiel könnte man statt `assert 0<=a;` schreiben **`if`** `(!(0<=a))` **`throw new`** `AssertionError();` Abgesehen davon, dass die Schreibweise mit `assert` kompakter und daher übersichtlicher ist, besteht ein pragmatischer Unterschied zwischen den beiden Schreibweisen: man kann beim Ausführen des Programms angeben, ob die `assert`-Anweisungen aktiviert sein sollen oder deaktiviert und damit völlig wirkungslos. Dies geschieht mit den Optionen:

```
java -enableassertions KlassenName
```

```
java -disableassertions KlassenName
```

Wenn man keine der beiden Optionen angibt, wirkt das wie `-disableassertions`, standardmäßig sind die `assert`-Anweisungen also deaktiviert.

Wenn das Programm in Ordnung ist, kann man es also so laufen lassen, dass die darin enthaltenen `assert`-Anweisungen keinen Aufwand verursachen. Wenn man das Programm überprüfen will, lässt man es mit aktivierten `assert`-Anweisungen laufen und nimmt den zusätzlichen Aufwand in Kauf, um möglichst genaue Hinweise auf fehlerhafte Stellen zu bekommen.

Für manche Programme ist es möglich, mit den `assert`-Anweisungen den gesamten Beweis der partiellen Korrektheit sozusagen zur Laufzeit nachrechnen zu lassen:

```
class Quadrat {
    public static void main(String[] args) {
        final int a = Integer.parseInt(args[0]);
        int y, z;

        /* Vorbedingung:            */ assert 0<=a;
        /* logische Folgerung:      */ assert 0<=a && 0==0 && 0==0;
        y = 0;
        /* Zuweisungsregel:         */ assert 0<=a && y==0 && 0==0;
        z = 0;
        /* Zuweisungsregel:         */ assert 0<=a && y==0 && z==0;
        /* arithm. Folgerung:       */ assert y<=a && z==y*y;
        while (y != a) {
            /* while-Regel:         */ assert y<=a && z==y*y && y!=a;
            /* logische Folgerung:  */ assert y<a && z==y*y;
            /* arithm. Folgerung:   */ assert (y+1)<=a && (z+2*y+1)==(y+1)*(y+1);
            z = z + 2*y + 1;
            /* Zuweisungsregel:     */ assert (y+1)<=a && z==(y+1)*(y+1);
            y = y + 1;
            /* Zuweisungsregel:     */ assert y<=a && z==y*y;
        }
        /* while-Regel:             */ assert y<=a && z==y*y && y==a;
        /* logische Folgerung:      */ assert z==a*a;

        System.out.printf("quadrat(%d) == %d%\n", a, z);
    }
}
```

Normalerweise wird man aber nicht jeden Zwischenschritt des Beweises überprüfen lassen, sondern nur die besonders wichtigen Bedingungen. Dazu zählen insbesondere Schleifeninvarianten sowie Vor- und Nachbedingungen. Wenn man noch ausnutzt, dass die `assert`-Anweisung einen zusätzlichen String enthalten kann, der bei einer verletzten Bedingung als Fehlermeldung verwendet wird, schreibt man das obige Programm zum Beispiel so:

```
class Quadrat {
    public static void main(String[] args) {
        final int a = Integer.parseInt(args[0]);
        int y, z;

        assert 0<=a                : "Vorbedingung";
        y = 0;
        z = 0;
        while (y != a) {
            assert y<=a && z==y*y  : "Schleifeninvariante";
            z = z + 2*y + 1;
            y = y + 1;
        }
        assert z==a*a              : "Nachbedingung";
```

```
        System.out.printf("quadrat(%d) == %d%\n", a,z);
    }
}
```

Im allgemeinen kann man aber die Bedingungen des Hoare-Beweises gar nicht vollständig als `assert`-Anweisungen formulieren, wie man sich am „ggT“-Beispiel leicht klarmacht: in diesem Beipiel enthalten alle interessanten Bedingungen eine Aussage der Form $ggT(\mathtt{a},\mathtt{b}) = \ldots$ Darin ist ggT eine mathematische Funktion, deren Eigenschaften wir mit Mitteln der Zahlentheorie analysieren können, aber keine Methode, die unser Java-Programm aufrufen könnte. Solange die Bedingungen als Kommentare zwischen den Anweisungen stehen, ist das kein Problem. Aber in `assert`-Anweisungen können wir nur Boolesche Ausdrücke verwenden, und diese kann man mit Aufrufen implementierter Methoden bilden, aber nicht mit „Aufrufen“ mathematischer Funktionen.

Die `assert`-Anweisungen können also den mathematischen Beweis nicht ersetzen, aber sie können ihn trotzdem unterstützen.

8 Nachweis der Terminierung

Das einzige Programmkonstrukt unter den hier besprochenen Anweisungen, welches problematisch für die Terminierung ist, ist die **`while`**-Schleife. Eine **`while`**-Schleife kann u.U. eben beliebig lange laufen. Alle anderen Anweisungen terminieren jedoch garantiert. Zum Nachweis der Terminierung eines Programmstücks genügt daher der Nachweis, dass jede darin vorkommende **`while`**-Schleife terminiert.

Die Grundidee für den Terminierungsbeweis einer **`while`**-Schleife ist folgende:

1. man identifiziert einen Ausdruck $t(x_1, \ldots, x_n)$, der von den im Programm vorkommenden Variablen abhängt,
2. man zeigt, dass der Wert des Ausdrucks bei jedem Schleifendurchgang echt kleiner wird und
3. man zeigt, dass er nicht kleiner als eine bestimmte feste Grenze werden kann, bei der dann die **`while`**-Schleife abbricht.

Damit ist garantiert, dass die **`while`**-Schleife nach endlich vielen Schritten abbrechen muss.

Im obigen „Quadrat“-Beispiel ist der Ausdruck $a - y$. Da y am Anfang 0 ist und bei jedem Schleifendurchgang um genau 1 erhöht wird, ist nach dem a-ten Durchlauf $y = a$. In diesem Fall ist dann die **`while`**-Bedingung falsch geworden, und die Schleife bricht ab.

Im „ggT“-Beispiel ist der Ausdruck $x + y$. Da in jedem Schleifendurchgang entweder x oder y echt kleiner wird, wird auf jeden Fall die Summe immer kleiner. Die Summe kann aber nicht kleiner als 2 werden, da x und y immer > 0 bleiben, und aus $x + y = 2$ folgt, dass $x = y = 1$ ist. Dann bricht die Schleife ab.

In den Beispielen war der Wert des Ausdrucks $t(x_1, \ldots, x_n)$ immer eine nicht-negative ganze Zahl und die kleiner-Relation war die $<$-Relation auf Zahlen. Das kann man noch verallgemeinern zu sogenannten wohlfundierten Ordnungen.

8.1 Wohlfundierte Ordnungen

Eine wohlfundierte Ordnung ist eine Menge S mit einer geeigneten $<$-Relation und einem kleinsten Element a, sodass für alle Elemente $x \neq a$ der Menge S gilt: $a < x$.

Beispiele:

- Die natürlichen Zahlen oder die nicht-negativen reellen Zahlen mit 0 als kleinstem Element und der üblichen $<$-Relation bilden eine wohlfundierte Ordnung.
- Die Menge $\{(x, y) \mid x, y \in \mathbb{N}\}$ der Paare von natürlichen Zahlen mit folgender $<$-Relation bilden eine wohlfundierte Ordnung:

  $(x, y) < (u, v)$ falls $x < u$ ist oder $x = u$ und $y < v$ ist.

  Zum Beispiel gilt $(2, 5) < (3, 1)$ und $(2, 5) < (2, 6)$.

Das kleinste Element ist $(0, 0)$.

- Strings mit der lexikographischen Ordnung bilden eine wohlfundierte Ordnung. Das ist die Ordnung, nach der Lexikaeinträge sortiert sind. Es gilt dabei:

  $s_1 \dots s_k < t_1 \dots t_\ell$ falls entweder $k < \ell$, oder $s_i = t_i$ für $i = 1, \dots, j$ für ein j, und s_{j+1} kommt alphabetisch vor t_{j+1}.

  Beispiele sind `"ab"` $<$ `"cde"` und `"abc"` $<$ `"acc"`.

  Das kleinste Element ist der leere String.

Jede wohlfundierte Ordnung eignet sich zum Nachweis der Terminierung einer **`while`**-Schleife.

9 Ausblick

Die Grundidee des Hoare-Kalküls, dass man von der Nachbedingung aus rückwärts rechnet, und dabei Anweisung nach Anweisung eliminiert, bis man auf die Vorbedingung stößt, lässt sich leicht auf andere Programmiersprachenkonstrukte erweitern, z.B. **`for`**-Schleifen usw. Man muss sich genau ansehen, was dieses Konstrukt macht, was man als neue Vorbedingung braucht, und wie man für dieses Konstrukt zeigt, dass seine Vorbedingung die jeweilige Nachbedingung erzwingt.

Eine Übungsaufgabe, die sehr das Verständnis des Kalküls fördert, wäre, sich zu überlegen, wie man die Regeln für andere Programmiersprachenkonstrukte gestalten muss.

Ganz Fortgeschrittene sollten darüberhinaus zeigen, dass diese Regeln in der Tat korrekt sind.

Allerdings muss man sorgfältig darauf achten, bei der mathematischen Formulierung der Vor- und Nachbedingungen auch die Eigenschaften der verwendeten Datentypen der Programmiersprache zu berücksichtigen. Wenn man einen Zähler mit `0.0` initialisiert und in einer Schleife jeweils um `0.1` erhöht, gelten die Bedingungen der Gleitkomma-Arithmetik, so dass der Zähler den Wert `1.0` nie erreicht. Man darf den Zählerwert in den Bedingungen also nicht so behandeln, als sei er ein Element von $\mathbb{Q}$ oder von $\mathbb{R}$.

Die Ganzzahl-Arithmetik ist in dieser Hinsicht etwas „gutartiger“, aber sie hat auch ihre Anomalien: für alle $n \in \mathbb{Z}$ gilt $n < n + 1$, aber für für den Wert `Integer.MAX_VALUE` des Datentyps **`int`** gilt das nicht.