Übersicht

5.2 Zusicherungen

- Ausnahmen sind dazu da, Fehler, die während der Programm-ausführung auftreten können, abzufangen, z.B. einen Zugriff auf ein Arrayfeld, das außerhalb der definierten Grenzen liegt (siehe später).
- Zusätzlich gibt es in Java Möglichkeiten, unerwünschte Ergebnisse aufgrund von unzulässigen Eingabewerten durch Testen von Eigenschaften (*Zusicherungen*, *Assertions*) in Form von Ausdrücken vom Typ **`boolean`** an beliebigen Stellen im Programm abzuprüfen.
- Assertions sind also Anweisungen, die Annahmen über den Zustand des Programmes zur Laufzeit verifizieren (z.B. der Wert einer Variablen).
- **Mit Testen kann man NICHT die Korrektheit von Programmen vollständig beweisen!**
- **Man kann lediglich gewisse Eigenschaften des Programmes für bestimmte Eingabewerte testen.**

5.2 Zusicherungen

- Die Assert-Anweisung hat in Java folgende Form:

```
assert <ausdruck1> : <ausdruck2>;
```

- Wobei der Teil "`: <ausdruck2>`" optional ist.
- `<ausdruck1>` muss vom Typ **`boolean`** sein.
- `<ausdruck2>` darf von beliebigem Typ sein. Er dient als Text für eine Fehlermeldung.
- Genaugenommen wird, falls `<ausdruck1>` den Wert **`false`** hat, eine *Ausnahme* (*Exception*) ausgelöst und `<ausdruck2>` dieser Exception übergeben. Fehlt `<ausdruck2>`, wird der Exception eine "leere Fehlermeldung" übergeben. Was genau eine Exception ist, lernen wir später kennen.
- Ist der Wert von `<ausdruck1>` **`true`**, wird das Programm normal fortgeführt.

5.2 Zusicherungen

- Beispiel:

```
assert x >= 0;
```

  überprüft, ob die Variable `x` an dieser Stelle des Programms nicht-negativ ist.
- Unterschied zur bedingten Anweisung:
  - Kürzerer Programmcode.
  - Auf den ersten Blick ist zu erkennen, dass es sich um einen Korrektheits-Test handelt und nicht um eine Verzweigung zur Steuerung des Programmablaufs.
  - Assertions lassen sich zur Laufzeit wahlweise an- oder abschalten. Sind sie deaktiviert, verursachen sie (im Gegensatz zu einer einfachen **`if`**-Anweisung) praktisch keine Verschlechterung des Laufzeitverhaltens.

5.2 Zusicherungen

- An- und Abschalten von Assertions beim Ausführen eines Programms durch Kommandozeilenargument der JVM:
- Anschalten: Parameter -ea, z.B.

```
java -ea MyProgramm
```

- Abschalten (**default**!): Parameter -da, z.B.

```
java -da MyProgramm
```

5.2 Zusicherungen

- Mittels Assertions könnte man z.B. überprüfen, ob der Definitionsbereich einer Methode eingehalten wird:

```
public static int quadrat(int a)
{

  // Precondition als Zusicherung
  assert a >= 0;

  int y;
  int z;

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

  return z;
}
```

5.2 Zusicherungen

- Ist dies sinnvoll?
- Die Überprüfung, ob der Definitionsbereich einer Methode oder sogar eines gesamten Programms eingehalten wird, sollte nicht einfach zur Laufzeit abschaltbar sein!
- Vielmehr sollte die Überprüfung von Werten, die einem Programm übergeben werden, immer aktiv sein.
- Für die Überprüfung von Preconditions eignen sich daher die Ausnahmen, da diese nicht abschaltbar sind. Wie dies geht, lernen wir noch kennen
- Assertions sind eher geeignet, Programmfehler zu entdecken und nicht fehlerhafte Eingabedaten.
- Assertions sollten daher ausschließlich in der Debugging-Phase eingesetzt werden.
- **Folgerung:** Ein Assert-Statement wird vom Programmierer als "immer wahr" angenommen.

5.2 Zusicherungen

- Beispiele, bei denen der Einsatz von Assertions sinnvoll ist:
  - Überprüfung von Postconditions in der Debugging-Phase um die Korrektheit einer Methode (oder des gesamten Programms) für einige Testfälle (Eingabebeispiele) zu evaluieren.
    Achtung: Testen ist kein formaler Korrektheitsbeweis!
  - Überprüfung von *Schleifeninvarianten*. Dies sind Bedingungen, die am Anfang oder am Ende einer Schleife *bei jedem Durchlauf* erfüllt sein müssen. Dies ist besonders für sehr komplexe Schleifen sinnvoll.
  - Die Markierung von *toten Zweigen* in `if`- oder `case`-Anweisungen, die nicht erreicht werden sollten. Anstatt hier einen Kommentar der Art "kann niemals erreicht werden" zu plazieren, könnte auch eine Assertion `assert false` gesetzt werden. Wird dieser Zweig bei einem Test während der Debugging-Phase wegen eines Programmfehlers durchlaufen, wird dies wirklich erkannt.
- Die Allgemeingültigkeit von Zusicherungen kann wiederum nur mit einem speziellen Logik-Kalkül (z.B. Hoare-Kalkül) bewiesen werden.

Übersicht

5.3 Hoare-Kalkül

- Wie bereits diskutiert, benötigt man zum formalen Beweis der Korrektheit imperativer Programme (oder auch der Allgemeingültigkeit von Zusicherungen) entsprechende Logik-Kalküle.
- Wir betrachten im Folgenden den *Hoare-Kalkül*.
- Der Hoare-Kalkül ist ein allgemeines Konzept, das an jede Programmiersprache (und sogar an Pseudo-Code) angepasst werden kann.
- Um das Grundprinzip zu verstehen, beschränken wir uns hier auf eine Anpassung an ein Fragment von Java, welches lediglich aus folgenden drei Arten von Anweisungen besteht:
  - Wertzuweisungen
  - `if (<bedingung>) <anweisung1> else <anweisung2>`
  - `while (<bedingung>) <anweisung>`
- Da die Terminierung nicht vom Hoare-Kalkül unterstützt wird, beschränken wir uns auf den Nachweis der *partiellen Korrektheit*.

5.3 Hoare-Kalkül

- Im Allgemeinen haben wir folgende Situation:

  `(PRE) {p_1; … ; p_n;} (POST)`

  wobei
  - `(PRE)` die Precondition darstellt,
  - `p_1, …, p_n` die einzelnen Anweisungen des Programms sind,
  - `(POST)` die Postcondition darstellt.
- Es ist also für das Programm, bestehend aus der Anweisungsfolge `p_1; ... ; p_n;`, zu zeigen, dass,
  - falls das Programm auf Eingabewerte, die der Precondition `(PRE)` genügen, angewendet wird, und
  - falls es terminiert,

  anschließend die Postcondition `(POST)` gilt (partielle Korrektheit).

5.3 Hoare-Kalkül

- Intuitiv: jede Anweisung `p_i` führt zu einer Zustandsänderung
- Die Precondition `(PRE)` formalisiert eine Eigenschaft, die im Zustand vor Ausführung der Anweisungen (*Initialzustand*) gelten muss
- Analog formalisiert die Postcondition `(POST)` eine Eigenschaft, die im Zustand nach Ausführung der Anweisungen (*Endzustand*) gelten muss, d.h. in dem Zustand, der durch Abarbeiten der einzelnen `p_i` aus dem Initialzustand erreicht wird.
- Die erste Anweisung `p_1` führt also den Initialzustand in einen „Zwischenzustand“ über, die zweite Anweisung `p_2` dann in einen zweiten „Zwischenzustand“, etc.
- Kernidee des Korrekheitsbeweises ist, geeignete Eigenschaften für diese Zwischenzustände zu finden

5.3 Hoare-Kalkül

- Dieses Grundprinzip des Korrektheitsbeweises mit dem Hoare-Kalkül ist wie folgt umgesetzt:
- Schritt 1
  - Finde eine *Zwischenbedingung* `Z_1` für `p_n` und *spalte* den Beweis in
    1. `(PRE) {p_1; ... ; p_(n-1);} (Z_1)`

    und

    2. `(Z_1) {p_n;} (POST)`
       (Der Fall `(PRE) {p_1; ... ; p_(n-1);} (Z_1)` wird in Schritt 2 behandelt).

    Die Bedingung `Z_1` modelliert den „letzten Zwischenzustand“, der durch Anwendung von `p_n` in den Endzustand überführt wird
  - Der Beweis von

    `(Z_1) {p_n;} (POST)`

    hängt von der Anweisung `p_n` ab.

    Für jede Anweisungsart (hier: Wertzuweisung, **`if`**-Schleife oder **`while`**-Schleife) benötigt man eine extra Regel (siehe später).
  - Durch die Aufspaltung haben wir also jeweils potentiell mehrere (z.B. zwei) Beweise zu führen (wieviel genau hängt, wie wir gleich sehen werden, von `p_n` ab)!

5.3 Hoare-Kalkül

- Schritt 2
  - Nach dem gleichen Schema werden die Anweisungen `p_1; ... ; p_(n-1);` behandelt (es wird also weiter aufgespaltet), bis man schließlich die Situation

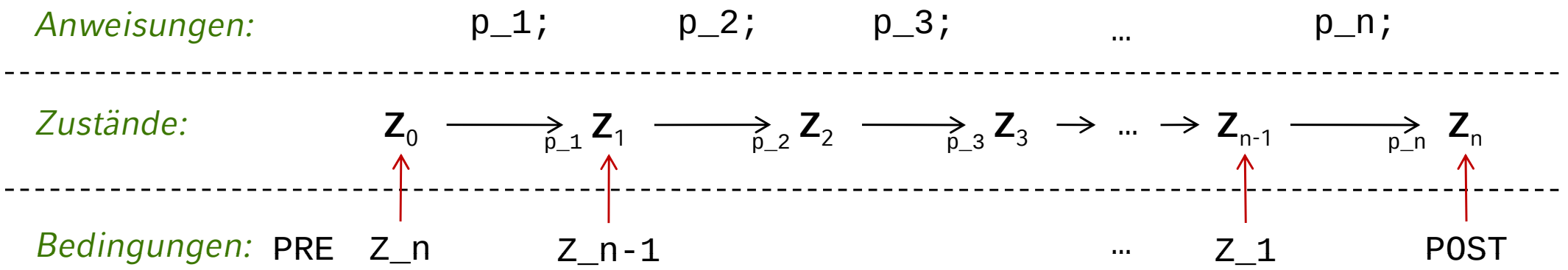
    `(PRE) {} (Z_n)`

    erreicht.

  - Partielle Korrektheit ist bewiesen, wenn in dieser Situation der Ausdruck

    `PRE => Z_n`

    den Wert **`true`** hat, also eine wahre Aussage darstellt und alle anderen Beweise geführt wurden
- Schematisch:



5.3 Hoare-Kalkül

- Beispiel (für Schritt 2):

  ```
  (x>=0 && y>=0) {} (x*y>=0)
  ```

  führt zu

  ```
  (x>=0 && y>=0) => (x*y>=0)
  ```

  und dieser Ausdruck hat den Wert **`true`**.

- Die Hoare'sche Methode reduziert also die Verifikation auf rein mathematische Beweisprobleme (sog. *Beweisverpflichtungen*).
- Wenn *alle* anfallenden Beweisverpflichtungen auch tatsächlich bewiesen werden können, dann ist die partielle Korrektheit gezeigt.

- Nun betrachten wir einzelne Verifikationsregeln für drei Arten von Anweisungen (Zuweisung, **`if`**-Schleife, **`while`**-Schleife) und deren Beweisverpflichtungen genauer.

5.3 Hoare-Kalkül

- Wertzuweisung (*Zuweisungsregel*):
  - Ausgangssituation:

    `(PRE) {p_1; ...; p_(n-1); x = t;}(POST)`

    (`x = t;` = p_n)

    wobei `x` eine Variable und `t` ein Ausdruck ist.
  - Die *Zuweisungsregel* transformiert dieses Problem in eine neue Beweisverpflichtung

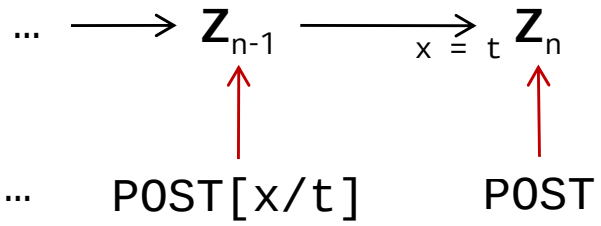
    `(PRE) {p_1; ...; p_(n-1);} (POST[x/t])`

    wobei `POST[x/t]` bedeutet, dass alle Vorkommen der Variablen `x` in `POST` durch den Ausdruck `t` zu ersetzen sind (entspricht der Substitution von `t` für `x` im Ausdruck `POST`).
  - Die Zuweisungsregel erlaubt, die Zwischenbedingung `Z_1` explizit zu berechnen, nämlich

    `Z_1 = POST[x/t].`

5.3 Hoare-Kalkül

- Schematisch



- Beispiel

  Aus

```
(PRE) {p_1; ...; p_(n-1); y = x*x;} (y>=0 && y<=10)
```

  wird

```
(PRE) {p_1; ...; p_(n-1)} (x*x>=0 && x*x<=10)
```

- Beobachtung: eine Wertzuweisung führt bei der Aufspaltung *keine* zusätzliche Beweisverpflichtung ein

5.3 Hoare-Kalkül

- Bedingte Anweisung (**`if`**-*Regel*)
  - Ausgangssituation:

    `(PRE) {p_1; ...; p_(n-1); if(B) p else q}(POST),`

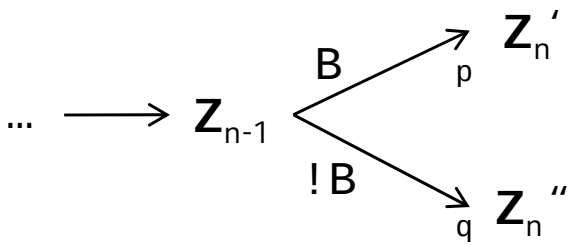
    (`if(B) p else q` = p'_n)

    wobei `B` die Testbedingung ist und `p` und `q` Programmstücke sind.
  - Die **`if`**-Regel transformiert dieses Problem in vier einzelne Probleme (davon 3 Beweisverpflichtungen):
    - Finde eine geeignete Zwischenbedingung `Z_1` als neue Vorbedingung für die **`if`**-Anweisungen.
    - Beweise den **`true`**-Zweig: `(Z_1 && B) {p} (POST).`
    - Beweise den **`false`**-Zweig: `(Z_1 && !B) {q} (POST).`
    - Mache weiter mit den restlichen Anweisungen vor der **`if`**-Anweisung für die Nachbedingung `Z_1`

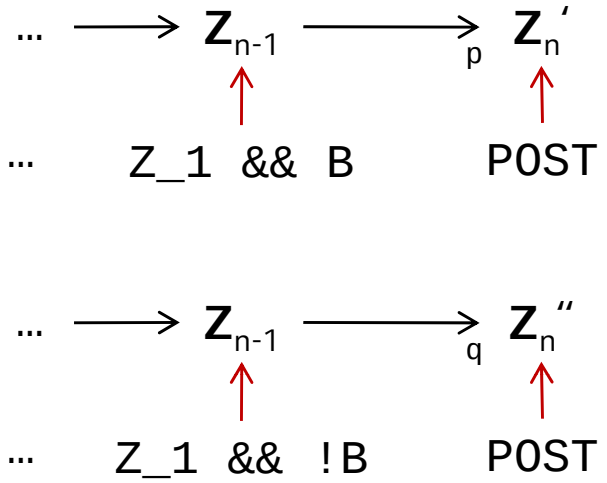
      `(PRE) {p_1; ...; p_(n-1);} (Z_1).`

5.3 Hoare-Kalkül

- Schematisch



bzw.

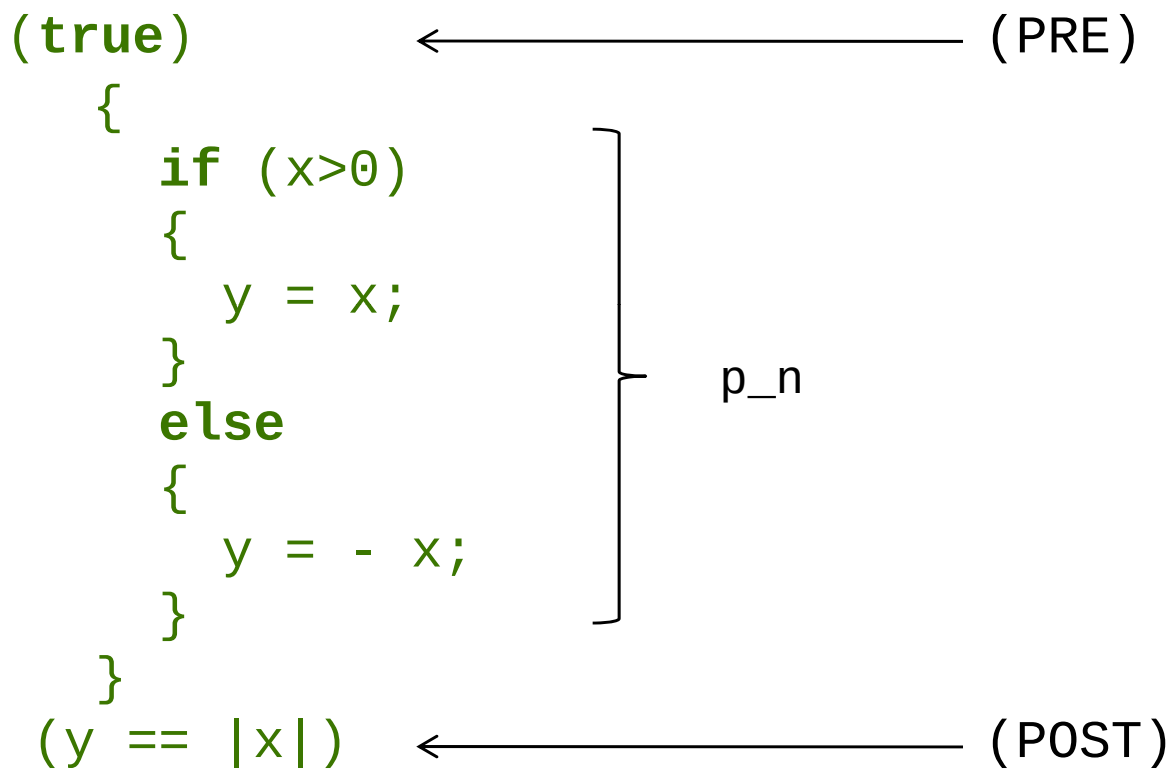


d.h. für beide Nachfolgezustände muss POST gelten!!

- Beobachtung: eine **`if`**-Anweisung führt bei der Aufspaltung *zwei* zusätzliche Beweisverpflichtungen (für jeden Zweig) ein

5.3 Hoare-Kalkül

- Beispiel



dabei soll |x| den Absolutbetrag von x bezeichnen, d.h. das Programmstück soll den Absolutbetrag von x berechnen.

5.3 Hoare-Kalkül

- Die vier Schritte der **`if`**-Regel:
  - Zwischenbedingung wird keine benötigt, d.h. `Z_1 = PRE = true`.
  - Der **`true`**-Zweig:

```
(true && x>0) {y = x;} (y == |x|)
(true && x>0) {y = x;} (y == |x|)[y/x]
(true && x>0) {} (x == |x|)
(true && x>0) => (x == |x|)
```

Durch Zuweisungsregel:
daraus wird:
und daraus wird wiederum:
und das stimmt (hat den Wert **`true`***).*

  - Der **`false`**-Zweig:

```
(true && !(x>0)) {y = -x;} (y == |x|)
(true && !(x>0)) {y = -x;} (y == |x|)[y/-x]
(true && !(x>0)) {} (-x == |x|)
(true && x<=0) => (-x == |x|)
```

Durch Zuweisungsregel:
daraus wird:
und daraus wird wiederum:
und das stimmt auch.

  - Der Rest vor der **`if`**-Anweisung ist leer, es gilt:

```
(true) { if(x>0){y = x;} else {y = - x;} } (y == |x|)
```

5.3 Hoare-Kalkül

- Iteration (**`while`**-*Regel*)

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

  - Ausgangssituation:

    `(PRE) {p_1; ...; p_(n-1); while(B) p}(POST)`

    wobei `B` die Testbedingung ist und `p` ein Programmstück.
  - Die **`while`**-*Regel* transformiert dieses Problem in fünf einzelne Probleme (davon 3 Beweisverpflichtungen:

5.3 Hoare-Kalkül

Die einzelnen Probleme

1. Finde eine geeignete *Schleifeninvariante* `INV`, die bei jedem Durchgang durch das Programmstück `p` gültig (invariant) bleibt. Das Auffinden der Invariante ist oft ein kreativer Vorgang ☺.
2. Finde eine geeignete Zwischenbedingung `Z_1` als neue Vorbedingung für die **`while`**-Anweisung, so dass gilt:

   ```
   Z_1 => INV.
   ```

   `Z_1` ist die spezielle Form von `INV`, die vor dem Eintritt in die Schleife gilt.
3. Verifiziere den Erhalt der Schleifeninvariante

   ```
   (INV && B) {p} (INV).
   ```

   Dies bestätigt, dass `INV` solange gültig bleibt, wie `B` gilt.

5.3 Hoare-Kalkül

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

   ```
   (INV && !B) => POST
   ```

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

5. Mache weiter mit den restlichen Anweisungen vor der Schleife für die Nachbedingung Z_1

   ```
   (PRE) {p_1; ...; p_(n-1);} (Z_1).
   ```

5.3 Hoare-Kalkül

- Beispiel: Die Methode `quadrat` vom Beginn des Abschnitts.

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

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

5.3 Hoare-Kalkül

Die fünf Schritte der **`while`**-Regel sind:

- Schleifeninvariante `INV`:

  ```
  y<=a && z==y*y
  ```

- Zwischenbedingung `Z_1`:

  ```
  0 <= a && y==0 && z==0
  ```

  Dies impliziert offensichtlich

  ```
  y<=a && z==y*y
  ```

5.3 Hoare-Kalkül

- Erhalt der Schleifeninvariante :

```
(y<=a && z==y*y && y!=a)
{z = z + 2*y + 1; y = y + 1;}
(y<=a && z==y*y)
```

Dies zeigt man durch zweimaliges Anwenden der Zuweisungsregel:

1.
```
(y<=a && z==y*y && y!=a)
{z = z + 2*y + 1;}
((y+1)<=a && z==(y+1)*(y+1))
```
2.
```
(y<=a && z==y*y && y!=a)
{}
((y+1)<=a && (z + 2*y + 1)==(y+1)*(y+1))
```

5.3 Hoare-Kalkül

- Erhalt der Schleifeninvariante (cont.):
  Daraus wird

```
(y<=a && z==y*y && y!=a)
=>
((y+1)<=a && (z + 2*y + 1)==(y+1)*(y+1))
```

  Dies gilt, denn:
  - Aus `y<=a && y!=a` folgt `y<a` und damit `y+1<=a`
  - Aus `z==y*y` folgt

```
z+2*y+1 == y*y+2*y+1 == (y+1)*(y+1)
```

5.3 Hoare-Kalkül

- Nachweis der Nachbedingung:

  `(INV && !B) => POST` also
  `((y<=a && z==y*y) && !(y!=a)) => z == a*a`

  dieser Ausdruck ist immer wahr, denn wenn die linke Seite der Implikation wahr ist, muss es auch die rechte sein (`!(y!=a)` entspricht `y==a`).

- Die restlichen Anweisungen vor der Schleife mit neuer Nachbedingung `Z_1` ergeben:

  `(0<=a) {y=0; z=0;} (0<=a && y==0 && z==0)`

  Zweimalige Anwendung der Zuweisungsregel ergibt:

  `(0<=a) => (0<=a && 0==0 && 0==0)`

  was offensichtlich wahr ist.