Abschnitt 6: Korrektheit von imperativen Programmen

6. Korrektheit von imperativen Programmen
6.1 Einführung
6.2 Testen der Korrektheit in Java
6.3 Beweis der Korrektheit von (imperativen) Java-Programmen

Überblick

6. Korrektheit von imperativen Programmen
6.1 Einführung
6.2 Testen der Korrektheit in Java
6.3 Beweis der Korrektheit von (imperativen) Java-Programmen

Korrektheit von Algorithmen

- Algorithmen und Programme sollten das tun, wofür sie entwickelt wurden, d.h. sie sollten *korrekt* sein.
- Problem: wie kann man nachweisen, dass ein Algorithmus bzw. ein Programm korrekt ist?
- Schon bei relativ kleinen Beispielen kann man oft nicht mehr "mit dem Auge" entscheiden, ob ein Algorithmus/Programm korrekt ist.
- Im Folgenden werden wir uns auf Java-Programme konzentrieren.
- Wir werden aber allgemeine Konzepte besprechen, die auch für andere Programmiersprachen und auch ganz abstrakt für die Algorithmenentwicklung (beispielsweise mittels Pseudo-Code) gelten.

Korrektheit von Algorithmen

Beispiel: folgendes Java-Programm soll das Quadrat einer Zahl $a \in \mathbb{N}$ berechnen.

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

Korrektheit von Algorithmen

- Bemerkung: Als Java-Programm macht der Algorithmus wenig Sinn, aber als Programm für einen Mikroprozessor, für den die Multiplikation viel aufwendiger ist als die Addition und Multiplikation mit 2, sehr wohl!
- Es ist nicht offensichtlich, dass für alle `int`-Werte a das Programm wirklich den Wert a^2 berechnet.
- Tatsächlich stimmt dies nur für alle $a \geq 0$.
- Wie können wir die Korrektheit dieses Java-Programms nachweisen?

Korrektheit von Algorithmen

- Zur Erinnerung: in der funktionalen Programmierung sind Algorithmen Funktionen, die, auf die Eingabewerte angewendet, spezielle Werte zurückliefern.
- Der Beweis, dass eine Funktion eine gewisse Eigenschaft hat, ist meist (relativ) einfach zu führen (z.B. durch vollständige Induktion, wenn die Funktion rekursiv definiert ist).
- Bei imperativen Programmen ist dies i.d.R. deutlich schwieriger.
  - Wir haben keinen direkten funktionalen Zusammenhang zwischen Eingabewerten und Ausgabewerten.
  - Das Resultat kommt vielmehr durch Abarbeiten verschiedener Berechnungsschritte zustande.
  - Diese Berechnungsschritte können Seiteneffekte haben.
- Der Beweis der Korrektheit imperativer Programme ist daher nicht ganz so einfach.

Korrektheit von Algorithmen

- Zunächst sollte festgelegt werden, wie man die Aussagen, die man für ein Programm machen will, formuliert: wie formuliert man z.B., dass die Methode `quadrat` das Quadrat des Eingabewerts berechnet?
- Dazu kann man eine eigene formale Sprache definieren (ist in diesem Rahmen aber nicht nötig).
- Wir verwenden Boole'sche Java Ausdrücke, um Aussagen zu machen, z.B. `0 <= a` oder `z == a*a`.
- Zusätzlich verwenden wir als gängige Abkürzung für `!a | b` die Notation `a => b`, wobei `a` und `b` beliebige Boole'sche Ausdrücke sein können.
- `a => b` drückt die logische Implikation "wenn `a`, dann `b`" aus.
- Damit hat der Ausdruck `a => b` den Wert **`true`** gdw. `b` den Wert **`true`** oder `a` den Wert **`false`** hat.

Korrektheit von Algorithmen

- Welche Art von Aussagen über ein Programm `p` will man nun beweisen?
- Typischerweise Aussagen der Art:
  "Wenn für die Eingabewerte der Programms `p` die Bedingung `PRE` gilt, dann gilt nach Ausführung von `p` für die Ausgabewerte die Bedingung `POST`".
- Dies schreibt man meist:

  `(PRE) p (POST)`
- `PRE` heißt *Vorbedingung* (*Precondition*) und `POST` heißt *Nachbedingung* (*Postcondition*).
- Beispiel: Für das Programm `p = quadrat(int a)` wären entsprechende Vor- und Nachbedingungen

  `(0 <= a) p (z == a*a)`

Korrektheit von Algorithmen

Nun unterscheiden wir zwei Arten von Korrektheit von Programmen:

- **Partielle Korrektheit:**
  Partielle Korrektheit eines Programms `p` heißt: falls `p` auf Eingabewerte, die der Vorbedingung `PRE` genügen, angewendet wird, und falls es terminiert, dann muss anschließend die Nachbedingung `POST` gelten.
- **Totale Korrektheit:**
  Totale Korrektheit eines Programms `p` heißt: falls `p` auf Eingabewerte, die der Vorbedingung `PRE` genügen, angewendet wird, dann terminiert es und danach muss die Nachbedingung `POST` gelten.
- Beobachtung: Totale Korrektheit = Partielle Korrektheit + Terminierung

Korrektheit von Algorithmen

- Wie kann man nun die Korrektheit imperativer Programme überprüfen?
- Testen: Man kann z.B. für alle möglichen Eingabewerte testen, ob das Programm die richtigen Ergebniswerte liefert.
- Offensichtlich geht dies im Allgemeinen nicht für alle möglichen Eingabewerte (primitive Datentypen haben zwar keinen unendlichen Wertebereich, die vollständige Aufzählung z.B. aller **`int`**-Werte ist allerdings wenig praktikabel).
- Testen kann dennoch sehr wichtig sein: insbesondere in der Entwicklungsphase verwendet man Tests zum Debugging.
- Zum formalen Beweis der Korrektheit imperativer Programme benötigt man stattdessen spezielle Logik-Kalküle (z.B. den Hoare-Kalkül).

Definitionsbereiche von Programmen

- Frage: Was passiert eigentlich, wenn Eingabewerte, die der Vorbedingung `PRE` nicht genügen, verarbeitet werden?
- Zunächst wiederum nur funktionale Algorithmen:
  - Der Algorithmus stellt eine Funktion dar.
  - Dies ist eine Abbildung $f : D \rightarrow B$ vom Definitionsbereich D auf/in den Bildbereich B.
  - Korrektheit bedeutet informell: für Eingaben aus D erhält man durch Auswertung von f die entsprechend gewünschten Werte aus B.
  - Für Eingabewerte, die nicht Element von D sind, erhält man durch Auswertung von f (je nach Definition von f) beliebige Werte aus B oder gar keine Werte oder f terminiert nicht, etc.

Definitionsbereiche von Programmen

- Bei imperativen Algorithmen ist die Situation ähnlich:
  - Eine Methode ist ebenfalls eine Abbildung $f : D \rightarrow B$ vom Definitionsbereich D auf/in den Bildbereich B mit möglichen Seiteneffekten.
  - Die "Vorschrift", wie diese Abbildung berechnet wird, ist nun als Folge von Anweisungen notiert, statt als mathematische Funktion.
  - Analog bedeutet Korrektheit damit informell: für Eingaben aus D erhält man durch Abarbeiten der Anweisungen die entsprechend gewünschten Werte aus B bzw. (z.B. wenn $B = \emptyset$) entsprechend gewünschte Seiteneffekte.
  - Für Eingabewerte, die nicht Element von D sind, erhält man durch Abarbeiten der Anweisungen (je nach Anweisungen) wiederum beliebige Werte oder gar keine Werte oder der Algorithmus terminiert nicht, etc.
  - Zusätzlich können bei imperativen Algorithmen für Eingabewerte, die nicht Element von D sind, (möglicherweise unerwünschte) Nebeneffekte auftreten.
- Offensichtlich sollte also die Eingabe von Werten, die nicht aus dem Definitionsbereich des Algorithmus stammen, vermieden werden.

Definitionsbereiche von Programmen

- Beispiel: Java-Methode `quadrat`
  - Definitionsbereich $D =$ **`int`**
  - Bildbereich $B =$ **`int`**
- Frage:
  Was passiert, wenn ich die Methode `quadrat` statt mit einem **`int`**-Wert mit einem **`double`**-Wert aufrufe?
- Antwort:
  - Falls dies bereits zur Kompilierzeit klar ist, wird das Programm gar nicht erst kompilieren.
  - Falls dies erst zur Laufzeit klar ist, wird die JRE eine Fehlermeldung in Form einer *Ausnahme* (*Exception*, siehe später) produzieren.
- Folgerung: Dann ist ja alles O.K., oder?

Definitionsbereiche von Programmen

- Nochmal eine Frage:
  Was passiert, wenn ich die Methode `quadrat` mit einem **`int`**-Wert `a < 0` oder einem **`char`**-Wert aufrufe?
- Antwort für **`int`**-Wert `a < 0`:
  Die Methode terminiert nicht.
- Antwort für **`char`**-Wert:
  Der **`char`**-Wert wird in einen positiven **`int`**-Wert konvertiert und damit ist der Ergebniswert wohldefiniert.
- Vermutlich ist weder diese Wohldefiniertheit, noch die Nicht-Terminierung beabsichtigt.
- In der Praxis sind solche Situationen häufig, daher ist ausführliches Kommentieren umso wichtiger!

Definitionsbereiche von Programmen

- Zusätzlich gibt es in Java Möglichkeiten, unerwünschte Ergebnisse aufgrund von unzulässigen Eingabewerten abzufangen.
- Zusicherungen (Assertions) können benutzt werden, um beliebige Bedingungen (in Form von Boole'schen Ausdrücken) an beliebigen Stellen im Programm zu plazieren (siehe nächster Unter-Abschnitt).
- Ausnahmen sind dazu da, Fehler, die während der Programmausführung auftreten können, abzufangen, z.B. einen Zugriff auf ein Arrayfeld, das außerhalb der definierten Grenzen liegt (siehe später).

Überblick

6. Korrektheit von imperativen Programmen
6.1 Einführung
6.2 Testen der Korrektheit in Java
6.3 Beweis der Korrektheit von (imperativen) Java-Programmen

Zusicherungen

- Mit Testen kann man NICHT die Korrektheit von Programmen vollständig beweisen!
- Man kann lediglich gewisse Eigenschaften des Programmes für bestimmte Eingabewerte testen.
- In Java kann man Tests durch sog. *Zusicherungen* (*Assertions*) durchführen.
- Assertions sind Anweisungen, die Annahmen über den Zustand des Programmes zur Laufzeit verifizieren (z.B. der Wert einer Variablen).

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, und wie man damit umgehen kann, lernen wir später kennen.
- Ist der Wert von `<ausdruck1>` **`true`**, wird das Programm normal fortgeführt.

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.
- 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
    ```


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;
}
```

Zusicherungen vs. Ausnahmen

- 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.
- 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.

Zusicherungen vs. Ausnahmen

- 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: Wie bereits oben diskutiert, ist Testen *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 bei Schleifen sinnvoll, die so komplex sind, dass beim Programmieren eine Art "Restunsicherheit" bezüglich ihrer Korrektheit bestehen bleibt.
  - 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.

Überblick

6. Korrektheit von imperativen Programmen
6.1 Einführung
6.2 Testen der Korrektheit in Java
6.3 Beweis der Korrektheit von (imperativen) Java-Programmen

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 angepasst werden kann.
- Um das Grundprinzip zu verstehen und kleine Beispiele rechnen zu können, 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 der Beweis der Terminierung oft nicht einfach ist und nicht vom Hoare-Kalkül unterstützt wird, beschränken wir uns hier auf den Nachweis der partiellen Korrektheit.

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).

Hoare-Kalkül

- Das Grundprinzip des Korrektheitsbeweises mit dem Hoare-Kalkül ist wie folgt:
- **Schritt 1**
  - Finde eine Zwischenbedingung `Z_1` für `p_n` und spalte den Beweis in

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

    (Der Fall `(PRE) {} (Z_1)` wird in Schritt 2 behandelt).
  - Die Weiterverarbeitung von `(Z_1) {p_n;} (POST)` hängt von der Art der Anweisung `p_n` ab. Für jede Anweisungsart benötigt man eine extra Regel.
- **Schritt 2**
  - Nach dem gleichen Schema werden die Anweisungen `p_1; ... ; p_(n-1);` behandelt, 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.

Hoare-Kalkül

- Beispiel:
  `(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.
- Im Folgenden schauen wir uns die einzelnen Verifikationsregeln für drei ausgewählte Arten von Anweisungen (Zuweisung, **`if`**-Anweisung, **`while`**-Schleife) und deren Beweisverpflichtungen genauer an.

Die Zuweisungsregel

- Ausgangssituation:
  `(PRE) {p_1; ...; p_(n-1); x = t;}(POST)`
  wobei `x` eine Variable und `t` ein Ausdruck ist.
- Die *Zuweisungsregel* transformiert dieses Problem in
  `(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.
- Dies spiegelt genau die Bedeutung der Zuweisung wieder: nach ihrer Ausführung sind `x` und `t` identisch.
- Die Zuweisungsregel erlaubt, die Zwischenbedingung `Z_1` explizit zu berechnen, nämlich
  `Z_1 = POST[x/t].`

Die Zuweisungsregel

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)
```


Die `if`-Regel

- Ausgangssituation:

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

  wobei `B` die Testbedingung ist und `p` und `q` Programmstücke sind.
- Die **`if`**-*Regel* transformiert dieses Problem in vier einzelne Probleme:
  - 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).`

Die `if`-Regel

Beispiel:

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

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

Die `if`-Regel

Die vier Schritte der **`if`**-Regel sind:

- Zwischenbedingung wird keine benötigt, d.h. `Z_1 = PRE = true`.
- Der **`true`**-Zweig:
  `(true && x>0) {y = x;} (y == |x|)`
  Daraus wird mit der Zuweisungsregel
  `(true && x>0) {} (x == |x|)`
  und daraus wiederum
  `(true && x>0) => (x == |x|)` und das stimmt.
- Der **`false`**-Zweig
  `(true && !(x>0)) {y = -x;} (y == |x|)`
  Daraus wird mit der Zuweisungsregel
  `(true && !(x>0)) {} (-x == |x|)`
  und daraus wiederum
  `(true && x<=0) => (-x == |x|)` und das stimmt ebenfalls.
- Der Rest vor der **`if`**-Anweisung ist leer, d.h. man wäre fertig.

Die `while`-Regel

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:

Die `while`-Regel

- Die fünf einzelnen Probleme:
  - 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.
  - 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.
  - Verifiziere den Erhalt der Schleifeninvariante

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

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

Die `while`-Regel

- Die fünf einzelnen Probleme (cont.):
  - 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.
  - Mache weiter mit den restlichen Anweisungen vor der Schleife für die Nachbedingung `Z_1`

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

Die `while`-Regel

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.

Die `while`-Regel

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`

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind (cont.):

- 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))`

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind (cont.):

- 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)`

Die `while`-Regel

Die fünf Schritte der **`while`**-Regel sind (cont.):

- 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.