LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Abschnitt 9

Korrektheit imperativer Algorithmen

Skript zur Vorlesung
Einführung in die Programmierung
im Wintersemester 2014/15
Ludwig-Maximilians-Universität München

Übersicht

5.1 Einführung

5.2 Hoare-Kalkül:
Beweis der Korrektheit von Java-Programmen

Übersicht

9.1 Einführung

- Algorithmen und Programme sollten das tun, wofür sie entwickelt wurden, d.h. sie sollten *korrekt* sein.

9.1 Einführung

- Algorithmen und Programme sollten das tun, wofür sie entwickelt wurden, d.h. sie sollten *korrekt* sein.

- Zur Erinnerung:
  1) Ein Algorithmus heißt partiell korrekt wenn für alle gültigen Eingaben das Resultat der Spezifikation des Algorithmus entspricht.
  2) Ein Algorithmus heißt (total) korrekt, wenn der Algorithmus partiell korrekt ist, und für alle gültigen Eingaben terminiert.

9.1 Einführung

- Beispiel (Aufgabe 4-2):
  - Aufgabe: Suche nach einer Karteikarte in einem Karteikasten
  - Algorithmus 5:
    - (1) Sie greifen zufällig eine Karte heraus.
    - (2) Trägt diese den gewünschten Titel, so legen Sie die Karte auf einen Ergebnisstapel, ansonsten legen Sie sie auf die Seite.
    - (3) Ist der Kasten leer, so beenden Sie die Suche. (Und nehmen die Karten des Ergebnisstapels auf die Hand)
    - (4) Ansonsten wiederholen Sie das Verfahren ab Schritt 1.
  - partiell korrekt (Ja): Existiert die gesuchte Karte, wird immer diese zurückgegeben. Existiert sie nicht, wird keine zurückgegeben.

9.1 Einführung

- 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.

9.1 Einführung

- 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;
}
```

9.1 Einführung

- 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?

9.1 Einführung

- Welche Art von Aussagen über ein Programm `p` will man nun beweisen?
- Im Beispiel:
  „Wenn x ≥ 0 dann gilt nach Ausführung des Programms p die Bedingung $z = a^2$
- Im Allgemeinen:
  „Wenn für die Eingabewerte des 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)
```

9.1 Einführung

- 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 (die Datentypen haben zwar prinzipiell 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*).

Übersicht

9.2 Hoare-Kalkül

- 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*.

9.2 Hoare-Kalkül

- Es ist also für das Programm, bestehend aus der Anweisungsfolge $(p_1; \ldots ; p_n;)$, 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).
- Im folgenden wird solch eine Beweisverpflichtung (oder Hoare-Tripel) folgenmaßen notiert:

  $$(\text{PRE})\ (p_1; \ldots ; p_n;)\ (\text{POST})$$

  wobei
  - (PRE) die Precondition darstellt,
  - $p_1, \ldots, p_n$ die einzelnen Anweisungen des Programms sind,
  - (POST) die Postcondition darstellt.

9.2 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

9.2 Hoare-Kalkül (Beispiel)

Beispiel:
Zu zeigen: Folgendes Programm soll den Wert einer Variable x mit dem initialen Wert 0 auf 7 setzen.

```
// Vorbedingung: x=0

x=x+2;
x=x+5;

// Nachbedingung: x=7 (Programmspezifikation)
```

9.2 Hoare-Kalkül (Beispiel)

Beispiel:
Zu zeigen: Folgendes Programm soll den Wert einer Variable x mit dem initialen Wert 0 auf 7 setzen.

// Vorbedingung: x=0

x=x+2; ← p_1
x=x+5;

// Nachbedingung: x=7 (Programmspezifikation)

9.2 Hoare-Kalkül (Beispiel)

Beispiel:
Zu zeigen: Folgendes Programm soll den Wert einer Variable x mit dem initialen Wert 0 auf 7 setzen.

// Vorbedingung: x=0

x=x+2; ← p_1
x=x+5; ← p_2

// Nachbedingung: x=7 (Programmspezifikation)

9.2 Hoare-Kalkül (Beispiel)

Beispiel:
Zu zeigen: Folgendes Programm soll den Wert einer Variable x mit dem initialen Wert 0 auf 7 setzen.

// Vorbedingung: x=0

x=x+2; ← p_1
x=x+5; ← $p_2 = p_n$

// Nachbedingung: x=7 (Programmspezifikation)

9.2 Hoare-Kalkül (Beispiel)

Beispiel:
Zu zeigen: Folgendes Programm soll den Wert einer Variable x mit dem initialen Wert 0 auf 7 setzen.

// Vorbedingung: x=0

x=x+2; ← p_1
x=x+5; ← $p_2 = p_n$

// Nachbedingung: x=7 (Programmspezifikation)

Folgende Beweisverpflichtung ist zu zeigen:

(x=0) (x=x+2; x=x+5;) (x=7)
(pre) (p_1,p_2) (post)

9.2 Hoare-Kalkül

- Dieses Grundprinzip des Korrektheitsbeweises mit dem Hoare-Kalkül ist wie folgt umgesetzt:
  - 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)

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

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

    läuft analog (rekursiv).
  - Der Beweis von

    2. (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).

9.2 Hoare-Kalkül (Beispiel)

Beispiel:

```
// Vorbedingung: x=0
x=x+2;
x=x+5;
// Nachbedingung: x=7 (Programmspezifikation)
```

Wir spalten unsere Beweisverpflichtung auf:

(x=0) (x=x+2; x=x+5;) (x=7)

⇔

(x=0) (x=x+2;) (Z_1) und

(Z_1) (x=x+5;) (x=7)

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

1. $(x=0)\ (x=x+2;)\ (Z_1)$ und
2. $(Z_1)\ (x=x+5;)\ (x=7)$

Wie wählt man eine Zwischenbedingung (hier Z_1)?

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

1. $(x=0)\ (x=x+2;)\ (Z_1)$ und
2. $(Z_1)\ (x=x+5;)\ (x=7)$

Wie wählt man eine Zwischenbedingung (hier Z_1)?

- Wahl der Zwischenbedingung ist der kreative Teil und Benötigt Verständnis des Programms

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

1. $(x=0)\ (x=x+2;)\ (Z_1)$ und
2. $(Z_1)\ (x=x+5;)\ (x=7)$

Wie wählt man eine Zwischenbedingung (hier Z_1)?

- Wahl der Zwischenbedingung ist der kreative Teil und Benötigt Verständnis des Programms

- Die Zwischenbedingung muss schwach genug sein, damit sie aus der vorherigen Zwischenbedingung (hier die Startbedingung x=0) herleitbar ist.

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

1. $(x=0)\ (x=x+2;)\ (Z_1)$ und
2. $(Z_1)\ (x=x+5;)\ (x=7)$

Wie wählt man eine Zwischenbedingung (hier Z_1)?

- Wahl der Zwischenbedingung ist der kreative Teil und Benötigt Verständnis des Programms
- Die Zwischenbedingung muss schwach genug sein, damit sie aus der vorherigen Zwischenbedingung (hier die Startbedingung x=0) herleitbar ist.
- Die Zwischenbedingung muss stark genug sein, damit daraus die nächste Zwischenbedingung (hier die Endbedingung x=7) hergeleitet werden kann.

9.2 Hoare-Kalkül

- 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

$$\text{PRE} \longrightarrow Z_n \xrightarrow{p_1} Z_{n-1} \xrightarrow{p_2} \dots \xrightarrow{p_{n-1}} Z_1 \xrightarrow{p_n} \text{POST}$$

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

$(x=0)\ (x=x+2;)\ (Z_1)$

Wird zu folgenden Beweisverpflichtungen:

$(x=0)\ ()\ (Z_2)$ und
$(Z_2)\ (x=x+2;)\ (Z_1)$

Damit haben wir folgende Beweisverpflichtungen:

$(x=0)\ ()\ (Z_2)$ und
$(Z_2)\ (x=x+2;)\ (Z_1)$ und
$(Z_1)(x=x+5;)(x=7)$

9.2 Hoare-Kalkül

- Die Hoare'sche Methode reduziert also die Verifikation auf rein mathematische Beweisprobleme (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.

9.2 Hoare-Kalkül

- Wertzuweisung (*Zuweisungsregel*):
  - Ausgangssituation:

    $(Z_i)\ (x = t;)(Z_{i-i})$

    wobei x eine Variable und t ein Ausdruck ist.

  - Die Zuweisungsregel erlaubt, die Zwischenbedingung Z_i explizit zu berechnen, nämlich

    $Z_i = Z_{i-1}[x/t],$

    wobei $Z_{i-1}[x/t]$ bedeutet, dass alle Vorkommen der Variablen x in Z_{i-1} durch den Ausdruck t zu ersetzen sind (entspricht der Substitution von t für x im Ausdruck Z_{i-1}).

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

Wir hatten folgende Beweisverpflichtungen:

1. (PRE) () (Z_2) und
2. (Z_2) (x=x+2;) (Z_1) und
3. (Z_1)(x=x+5;)(POST)

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

Wir hatten folgende Beweisverpflichtungen:

1. (PRE) () (Z_2) und
2. (Z_2) (x=x+2;) (Z_1) und
3. (Z_1)(x=x+5;)(POST)

Aus 3. erhalten wir Z_1 mittels Zuweisungsregel:

Z_1=POST[x / x+5] ⇔

Z_1=(x=7)[x / x+5] ⇔

Z_1=(x+5=7) ⇔

Z_1=(x=2)

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

Wir hatten folgende Beweisverpflichtungen:

1. (PRE) () (Z_2) und
2. (Z_2) (x=x+2;) (Z_1) und
3. (Z_1)(x=x+5;)(POST)

Aus 3. erhalten wir Z_1 mittels Zuweisungsregel:

$Z_1 = POST[x / x+5]$ ⇔
$Z_1 = (x=7)[x / x+5]$ ⇔
$Z_1 = (x+5=7)$ ⇔
$Z_1 = (x=2)$

Aus 2. erhalten wir Z_2 mittels Zuweisungsregel:

$Z_2 = Z_1[x / x+5]$ ⇔
$Z_2 = (x=2)[x / x+2]$ ⇔
$Z_2 = (x+2=2)$ ⇔
$Z_2 = (x=0)$

9.2 Hoare-Kalkül (Beispiel)

Beispiel (cont.):

Wir hatten folgende Beweisverpflichtungen:

- 1. (PRE) () (Z_2) und
- 2. (Z_2) (x=x+2;) (Z_1) und
- 3. (Z_1)(x=x+5;)(POST)

Aus 2. erhalten wir Z_2 mittels Zuweisungsregel:

$Z_2 = Z_1[x / x+5]$ ⇔
$Z_2 = (x=2)[x / x+2]$ ⇔
$Z_2 = (x+2=2)$ ⇔
$Z_2 = (x=0)$

Aus 1. erhalten wir:

(PRE) () (Z_2) ⇔
(x=0) () (x=0) ⇔
(x=0) ⟶ (x=0) ⇔
true

9.2 Hoare-Kalkül

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

    `(`Z_i`) (if(B) p else q) (`Z_{i-1}`),`

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

  - Die **`if`**-Regel transformiert dieses Problem in drei Teilprobleme, davon 2 Beweisverpflichtungen:
    - Finde eine geeignete Zwischenbedingung Z_i als neue Vorbedingung für die if-Anweisung
    - Beweise den **`true`**-Zweig: `(`Z_i` && B) (p) (POST).`
    - Beweise den **`false`**-Zweig: `(`Z_i` && !B) (q) (POST).`

9.2 Hoare-Kalkül

- 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.

9.2 Hoare-Kalkül

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

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

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

9.2 Hoare-Kalkül

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

Beispiel:

```
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;
}
```

9.2 Hoare-Kalkül

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

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

  - Ausgangssituation:

    (Z_i) `(while(B) p)` (Z_{i-1})

    wobei `B` die Testbedingung ist und `p` ein Programmstück.
  - Die **`while`**-*Regel* transformiert dieses Problem in vier einzelne Probleme (davon 2 Beweisverpflichtungen):

9.2 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_i als neue Vorbedingung für die **`while`**-Anweisung, so dass gilt:

   Z_i `=> INV.`

   Z_i muss generell (stark) genug sein, um daraus `INV` abzuleiten
3. Verifiziere den Erhalt der Schleifeninvariante

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

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

9.2 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.

9.2 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.

9.2 Hoare-Kalkül

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

- Schleifeninvariante `INV`:

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

- Zwischenbedingung Z_i:

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

  Dies impliziert offensichtlich

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

9.2 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))
```

9.2 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)
```

9.2 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`).

9.2 Hoare-Kalkül: Zusammenfassung

- Zusammenfassung
  - Hoare-Kalkül zum Beweis partieller Korrektheit
    - Kann keine totale Korrektheit zeigen
    - Kann auch keine partielle Korrektheit widerlegen (besser: Gegenbeispiel)
  - „Rückwärtsrechnen“: Von Hinten nach Vorn
  - Was muss an jeder Stelle des Programms (vor jeder Anweisung) gelten, damit am Ende POST (die Semantik des Programms) gelten muss?
    - Finden von Zwischenbedingungen vor jeder Anweisung
  - Drei Arten von Anweisungen hier behandelt
    - Zuweisungen
      - Regel für neue Zwischenbedingung
    - Verzweigung
      - If-Fall und else-Fall getrennt betrachten. Gemeinsame Zwischenbedingung finden
    - Iteration
      - Invariante: Was macht die Schleife eigentlich? Was gilt in jeder Iteration?
      - Zwischenbedingung: Was muss vor der Schleife gelten?
      - Abbruch: Folge aus Abbruchbedingung und Invariante die nächste Zwischenbedingung?