Algorithmen und Programmieren III

- Vorlesung:
  - Di. 16 – 18
  - Do. 12 – 14 jeweils c.t.
- Dozent etc:
  - H. Schweppe, hs@inf.fu-berlin.de
  - Sprechstunde: Mi. 14-15 und nach Vereinbarung
- Übungsbetreuung
  - Zara Kanaeva, kanaeva@inf.fu-berlin.de
  - Sprechstunde
- Tutorien:
  - Nach Vereinbarung -> Eintrag in elektronische Listen
  - Tutor(inn)en:
    - ...

Algorithmen und Programmieren III : Tutor(inn)en

- Alex Jäger (Mi. 12-14, SR 051)
- Alex Jäger (Fr. 12-14, SR 049)
- Claudia Klost (Mo. 14-16, SR 051)
- Claudia Klost (Fr. 10-12, SR 051)
- Jemea Ntuba (Mo. 14-16, SR 007/008, pi-Gebäude)
- Jemea Ntuba (Mi. 14-16, SR 053)
- Sarah Renkl (Di. 12-14, SR 051)
- Sarah Renkl (Mi. 10-12, SR 053)
- Lars Wolter (Mo. 12-14, SR 051)
- Lars Wolter (Fr. 12-14, SR 051)

Algorithmen und Programmieren III

- Scheinkriterien
  - Übungen:
    - Regelmäßige Abgabe (Zweiergruppen), n-1 von n Blättern
    - Mindestens 50% aller Punkte und
  - Mitarbeit in Tutorien:
    - Jeder namentlich auf Übungszettel genannte muss richtig gelöste Übung vortragen können. Zufällige Auswahl, maximal 2 Fehlversuche und
  - Klausur:
    - Mindestens 50 % der erreichbaren Klausurpunkte
    - Unterlagen sind in der Klausur erlaubt
  - Scheine sind benotet
    - Klausurergebnis entscheidet
    - Nur auf Wunsch (bis 15.11.01) unbenotet

Algorithmen und Programmieren III

Inhalt

1. Spezifikation und Verifikation
2. Datenabstraktion
3. Effiziente Implementierung von Mengen und Relationen
   - Bäume
   - Hashverfahren
   - Graphen
4. Objektorientierte Entwicklung mit UML und Programmiermuster

Literatur

- **John Guttag, Barbara Liskov:** Program Development in Java: Abstraction, Specification, and Object-Oriented Design, Addison-Wesley (ISBN:0201657686), 2000.
- **G. Saake, K. Sattler:** Algorithmen und Datenstrukturen. Eine Einführung mit Java, dpunkt-Verlag, Heidelberg (ISBN: 3898641228), 2001.
- **Güting:** Datenstrukturen und Algorithmen, Teubner, 1992.
- **T. Cormen, C. Leiserson, R. Rivest**: Introduction to Algorithms, MIT Press, 1990.
- **Standish:** Data Structures in Java, Addison-Wesley, 1998.
- **S. Thompson:** HASKELL - The Craft of Functional Programming, Addison-Wesley, 2. Auflage, 1999.
- **R. Bird**: Introduction to Functional Programming using Haskell, Prentice Hall Series in Computer Science, 1998.
- **E. Gamma, R. Helm, R. Johnson, J. Vlissides:** Design Patterns – Elements of Reusable Software, Addison-Wesley, 1995
- **J. Seemann, J. v. Gudenberg**: Software-Entwurf mit UML, Springer Verlag 1999

1 Spezifikation und Verifikation

1.1 Einführendes Beispiel: Ägyptisch Multiplizieren

Am Beispiel von 2807 ´ 7 im Papyrus Rhind, 16. Jh. v. Chr.

Auch bekannt als „Methode der russischen Bauern“

Gegeben zwei natürliche Zahlen a, b > 0
Halbiere b, verdopple a mit dem Ergebnis (a',b') solange, bis b' = 1
Summiere alle a', für die b' ungerade
Ergebnis: a x b

Wenn ihr nur duplieren und halbieren könnet, so könnet ihr das übrige ohne das Eins mal Eins multipliciren.
Christian von Wolff, 1679- 1754
(Philosoph und Mathematiker, Univ. Halle)

a=17	b=11
17	11
34	5
68	2
136	1

a x b = 136+34+17 = 187

8	9
16	4
32	2
64	1

Zwei Java - Algorithmen

```
public static int mul1(int a, int b)
     { int akk = 0;
       while (b > 1)
         { if (b%2 != 0) akk=akk+a;
           a=a*2;
           b=b/2;  // hier wird nur der
                   //ganzzahlige Anteil verwendet
         }
       return akk+a;
     }
```

```
public static int mul2 (int a, int b)
     { int akk = 0;
       while (b > 0)
         { if (b%2 == 0) b=b/2;
             else {b=(b-1)/2; akk=akk+a;}
           a=a*2;
         }
       return akk;
     }
```

leistet die Methode dasselbe?

Testen...

a = 5, b = 7	=>	akk == 35
a = 12, b = 23	=>	akk == 276
a = 5.1, b = 7	=>	Typfehler (Exception)
a = 0 , b = 7	=>	akk == 0
a = 0, b = 0	=>	akk == 0
a = -5, b = 7	=>	akk == -35
a = 7, b = 0	=>	mul1: akk == 7 mul2: akk == 0
a = 5, b = 1	=>	mul1: akk == 5 mul2: akk == 5
a = 5, b = -7	=>	mul1: akk == 5 mul2: akk == 0

Korrektheit?

Was heißt eigentlich "Algorithmus ist korrekt"?

- Sind die Methoden äquivalent? In welchem Sinne?
- Wenn ja: lässt sich das beweisen?
- Sind sie überhaupt korrekt implementiert?
- Unter welchen Annahmen für die Eingaben?
- Was sollen die Methoden überhaupt leisten?

Frage nach Korrektheit sinnlos wenn man nicht klar ist, was ein Algorithmus **genau** leisten soll.

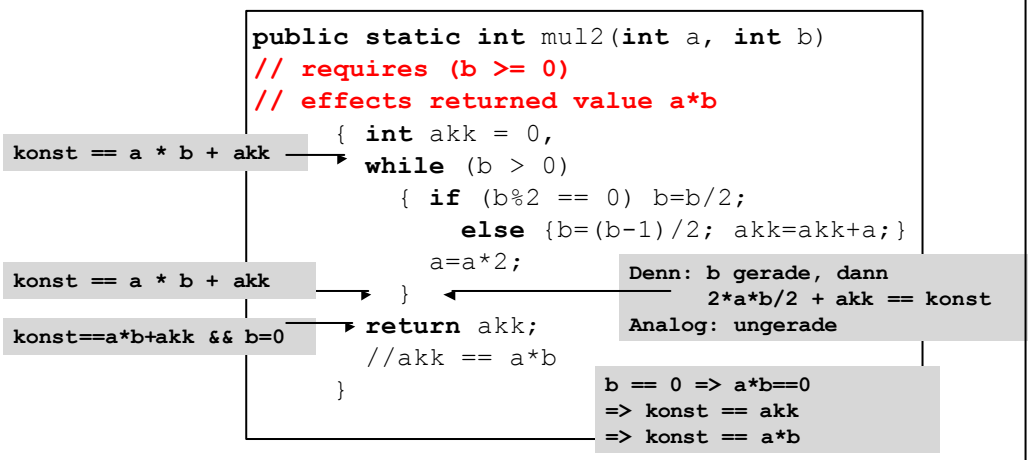
Spezifikation

- Spezifikation (erste Näherung) :
  - Welche Voraussetzungen gelten?
  - Ergebnis: Funktionswert und (Neben)effekte ("Seiteneffekte", Side effects)

~~Schlechter Sprachgebrauch!~~

- Beispiel: "Wert von mul1: Produkt der ganzen Zahlen a und b, keine Effekte"
  - Falsch! Voraussetzung zu schwach!
- Beispiel: "Wert von mul1: Produkt der natürlichen Zahlen a und b, keine sonstigen Effekte"
  - Voraussetzung stärker als nötig: a kann auch negativ sein.
- Beispiel: "Wert von mul1: Produkt der ganzen Zahlen a und b mit $b > 0$, keine Effekte2
  - Scheint korrekt für mul2.
  - Gilt Spezifikation auch für mul1? Lässt sich das beweisen?

Beispiel: Korrektheit durch Invariante beweisen



Bisher: Wenn Programm **return** erreicht, dann hat **akk** den Wert **a*b**. Reicht das?

Kritik

- Unbefriedigend:
  - Effekte der Anweisungen in der Schleife, die für Invarianzeigenschaft wichtig sind, wurden durch scharfes Hinsehen gefunden, denn wir wissen, welchen Effekt eine Zuweisung wie a=2*a hat
  - Für ein automatisches Beweisverfahren unzureichend. Nötig: Formale Beschreibung der Effekte von elementaren Anweisungen
  - Notation: Unterscheidung des Variablenwertes vor und nach einer Veränderung wünschenswert
    Beachte: in funktionalen Sprachen kein Thema. In einer Funktionsdefinition bezeichnet a immer denselben Wert.
    (Eigenschaft heißt: Referenzielle Transparenz)
  - Bisher im Beispiel gezeigt: Wenn Programm **return** erreicht, dann hat **akk** den Wert **a*b**. Aber ist die Voraussetzung irgendwann erfüllt?

Endlosschleifen

- Wenn die Bedingung einer Schleife sich nicht durch Anweisungen im Körper verändert, terminiert sie nicht (Endlosschleife)

Aber selbst wenn: entscheidend ist, dass Bedingung irgendwann falsch wird.

Konstante **`true`** wird nie **`false`**

```
while (true);
```

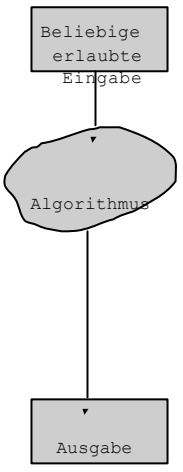
leere Anweisung

```
x = -1;
while (x!=0)
{ x = (x-1)/2;
System.out.println(x);
}
```

Immer noch einfach zu erkennen, aber allgemein sogar **unentscheidbar**, ob beliebige Schleife terminiert!

Partielle und totale Korrektheit

Idee: Trennung der Eigenschaften "Korrektes Ergebnis" und "Termination"



Algorithmus A partiell korrekt:
wenn A terminiert, dann Ergebnis korrekt

Algorithmus A total korrekt:
A terminiert und Ergebnis korrekt

Terminierung des Algorithmus

Behauptung: Der Algorithmus hält für b>0 nat. Zahlen, a ganz, als Eingabe.
Beweisargument :
Fortgesetztes "ganzzahliges" Halbieren von b führt schließlich auf b <= 0.
(1) Für alle geraden Zahlen b >= 2 (!) gilt b/ 2 < b.
Beweis: 1 < 2
==> 1*b < 2*b (da b > 0)
==> b/2 < b
(2) Für alle ungeraden Zahlen b >= 1 gilt (b- 1)/ 2 < b.
-1 < 0
==> b - 1 < b
==> (b - 1)/ 2 < b (da b - 1 >= 0)

Wieso ?
Zeige dazu

In jedem Schleifen-durchlauf kommen wir der Bedingung:
b<= 0 eins näher!

Streng monoton fallende Folge natürlicher Zahlen ist endlich,
==> Terminierung!

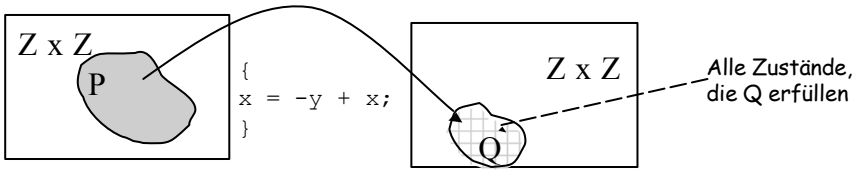
Zusammenfassung des einführenden Beispiels

- Spezifikation des Algorithmus durch Voraussetzung und Effekt
- Beweis, dass Algorithmus den gewünschten Effekt hat, z.B. anhand von Schleifeninvarianten und Kenntnis der Effekte von Anweisungen
- Terminierung des Algorithmus beweisen (gelingt nicht immer! "Halteproblem")

1.2 Abstraktion und Spezifikation

Imperative Algorithmen: Variablen definieren Zustandsraum Z
Algorithmus S: Abbildung des Zustandsraum in sich

P : x > 0 ? y < x S Q : x == x + |y|



P und Q : Prädikate für Teilmengen des Zustandsraums

- Konventionen und Beispiele
  - Schreibweise: {P} S {Q}
    "Wenn P vor Ausführung von S gilt, dann gilt Q nach Ausführung"
    {P} S {Q} heißt Zusicherung (assertion) für S
  - Oft nützlich, Werte von Variablen VOR Ausführung von S anzugeben.
    Konvention: Wert einer Variablen x in Voraussetzung wird mit X bezeichnet
    (andere gelegentlich verwendete Konvention: x' bezeichnet Wert von x)

    Beispiele : {x==X ? y==Y } z=x; x=y; y=z {x==Y ? y==X}

    auch korrekt: {x==X ? y==Y } z=x; x=y; y=z {x==Y }

    falsch: {x==X ? y==Y } y=x; x=y; {x==Y ? y==X}

Spezifikation und Implementierung

- Bemerkungen
  - Zu einer Anweisung S gibt es meist viele Zusicherungen.
    Gesucht sind solche, die Voraussetzung und Effekte "möglichst gut" charakterisieren.
  - es gibt viele Anweisungen S, die eine gegebene Voraussetzung P und mit Effekten Q erfüllen.
  - (P,Q) ist eine Spezifikation. Jede Implementierung S muss (P,Q) erfüllen.

Spezifikation, Voraussetzungen, Effekte

Spezifikation (P, Q)

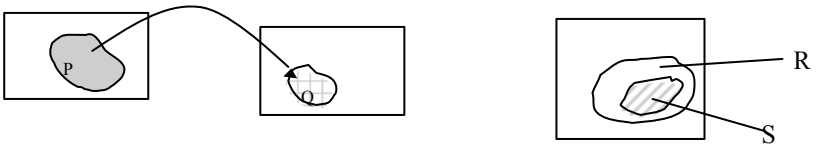
P bestimmt die erlaubten Zustände vor Ausführung eines Algorithmus S, der die Spezifikation erfüllt

Q bezeichnet die Effekte, die S nach sich zieht

P heißt Voraussetzung (precondition, Vorbedingung)
Q heißt Effekt (postcondition, Konsequenz, Nachbedingung)
(P,Q) heißt (abstrakte) Spezifikation

Schwache Voraussetzungen, starke Effekte

- Was heißt: Implementierung S soll "möglichst gut" charakterisiert werden?



Wünschenswert:
möglichst schwache Voraussetzungen
- sollen für viele Zustände zutreffen

möglichst starke Effekte
- Zielzustände sollen möglichst genau charakterisiert werden

S(x) ? R(x)

Schwächste Voraussetzung wäre `true`: jeder Zustand erlaubt

Schwache Voraussetzungen, starke Effekte

Beispiel: {P} S {Q} = {x==3} x=x+1 {x > 0}

S = {x=x+1} erfüllt auch P' = {x ? 0} , Q = {x > 0}
oder P = {x=3}, Q' = {x==4}

Terminologie der math. Logik:
P' schwächer als P wenn P ? ?P'
Q stärker als Q' wenn Q ? Q'

Gesucht: möglichst schwache Voraussetzungen,
möglichst starke Konsequenzen (Effekte)

Schwache und starke Spezifikationen

- Ordnungsrelation auf Spezifikationen

  Spezifikation (P, Q) stärker als (P‘,Q‘) genau dann wenn
  P‘ ? ?P ? ?Q ? ?Q‘ (d.h. P‘ stärker als P und Q stärker als Q‘)

  Schreibweise: (P, Q) ? ?(P‘, Q‘)

  Beispiel:
  P ' = {x >= 0 ? y >= 0 }
  Q' = {y = Y + X}

  P = true
  Q = {y = Y + X ? x = Y*Y ? ?x >= 0 }

- Spezifikation abstrahiert von Implementierung
  im allgemeinen gibt es viele Algorithmen, die einer Spezifikation genügen.

Abstraktion von Implementierungen

- Abstraktion durch Spezifikation

P : true
alle Zustände x,y ? `int` erlaubt

S

Q : (y = Y + X ? x = Y*Y ? ?x >= 0)

Abstraktion

```
S': {
  y = y + x;
  x = -x + y;
  x = x * x
 }
```

```
S'': { int z = x;
       x = y * y;
       y = y + z
 }
```

Spezifikation in natürlicher Sprache legitim:
Wenn x den Wert X und y den Wert Y vor Ausführung von S haben und nach Ausführung gilt: $x==Y^2$ und y == X+Y und x >=0, dann erfüllt S die Spezifikation.

Implizite Voraussetzungen

1. In getypter Sprache muss noch Typkorrektheit gefordert werden,
   Beispiel: `true ? ?Int (x) ?`
   Implizite Voraussetzung: Typkorrektheit
   Typprüfung meist zur Übersetzungszeit (statisch typisierte Sprachen)

2. Keine arithmetischen Ausnahmesituationen
   Beispiel: a * b <= größte darstellbare Zahl
   Implizite Voraussetzung: alle arithmetischen Operationen liefern darstellbare Zahl
   Explizite Formulierung grundsätzlich kein Problem, nur etwas umständlich.

Eigenschaften einiger Spezifikationen

`true` bedeutet "von jedem Zustand z erfüllt"
`false` bedeutet "nicht erfüllbar"
`(P, P)`: ohne Wirkung (bzgl P !)
`(false, Q)`: nicht anwendbar - es gibt keinen Zustand, der die Voraussetzung erfüllt
`(true, Q)`: Effekt erfüllt Q ohne Voraussetzung
`(P, true)`: was beliebt ist erlaubt: Effekt spielt keine Rolle
`(P, false)`: terminiert nicht, da es keine Anweisung und keinen gültigen Zustand gibt, der die Spezifikation erfüllt

x++ erfüllt Spezifikation (x>0, x>0)

Jedes Programm erfüllt diese Spezifikation

Frage: Wie kann man `{P} S {Q}` beweisen, das heißt dass Anweisung `S` Spezifikation `(P,Q)` erfüllt ?

Abstraktion von Implementierungen

- Vorteile
  - Änderbarkeit: Implementierung kann ohne Nebenwirkungen geändert werden (im Gegensatz zur Spec !)
  - Lokalität: Implementierung ist unabhängig vom Kontext, in dem sie ausgeführt wird
- Andere Art der Abstraktion
  - Datenabstraktion: Datentypen durch darauf zulässige Operationen definieren. Siehe z.B. Haskell-Klassen
  - Hauptthema von ALP3

Spezifikation...

- Prozedurale Abstraktion:
  Parametrisierung eines Ausdrucks liefert
  Prozedur: (3 + 5) / 2 ✍ ?x?y ((x+y)/2)
- Spezifikation mit unterschiedlichem Abstraktionsniveau
  Beispiel:
  (x==X >0 ? ?y==Y>0 , result ==X+Y div 2)

  Prädikat für Funktionswert

  Hier kein großer Unterschied zum Algorithmus, aber hier:

  (x=X>0, result: die Zahl, die mit sich selbst multipliziert X ergibt)

Spezifikation

Spezifikation....

- beschreibt das sichtbare Verhalten eines Programm
  „von außen" , im Detail
- soll „hohes" Sprachniveau haben
  - eindeutig, präzise
  - kein Interpretationsspielraum
  - keine Redundanz
- Natürliche Sprache zulässig, wenn präzise.
- Formale Sprache ermöglicht Korrektheitsbeweise

Gibt es stärkere Spezifikation?

Java-Notation

Notation für Spezifikation

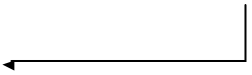
```
<Ergebnistyp> <bezeichner> ( .......)
 // requires:  Prädikat, das erlaubten Zustand vor
 //  Ausführung beschreibt
 // effects: definiert das Verhalten
 //           - berechneter Wert,
 //           - ausführungsbedingte Änderungen
 //               * Änderungen von Argumenten
//                 * nichtlokale Zustandsänderungen
//                    (Seiteneffekte)
```

Beispiel: Suchproblem

„Suche einen Wert x in dem Feld a"

- was wird als Wert erwartet? x? true/false? Ein x zugeordneter Wert?
- ... und wenn x nicht in a ?
- wenn x mehrfach in a enthalten ist?
- enthält a mindestens ein Element ?
- sind alle a[i] , 0 <= i <= a.length - 1 definiert?

Signatur, hier als Methoden Signatur (Java)



Spezifikation:

```
int search ( int []  a, int x)
// requires: a array of length k > 0, a[i] defined for 0 <= i < k
// ? j ( 0 <= j < k : a[j] == x  ?  ?? ?: 0 <= i < j ) a[i] != x )
//   ? ??j = k  ?  ?? ?: 0 <= i < k ) a[i] != x )
// effects: returns j
```